FWX

FWX Lending and Borrowing Pools, and FWX Membership

Smart Contract Audit Report

Date Issued: 31 Aug 2022

Version: Final v1.0

Public

Table of Contents

Executive Summary

Overview

Valix conducted a smart contract audit to evaluate potential security issues of the **FWX Lending and Borrowing Pools, and FWX Membership features**. This audit report was published on `31 Aug 2022`. The audit scope is limited to the **FWX Lending and Borrowing Pools, and FWX Membership features.** Our security best practices strongly recommend that the **FWX team** conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

About FWX Lending and Borrowing Pools, and FWX Membership

FWX Key Features

FWX offers three main features which are the decentralized derivative exchange (DDEX), the lending and borrowing pools (LBPs), and NFT membership. The three features support each other. FWX DDEX needs the liquidity pools to operate, while the LBPs receive real borrowing demand and thus real profits from the derivative trading orders. However, in this phase, we have audited only LBPs and a part of NFT membership.

FWX Lending and Borrowing Pools Feature

FWX offers lending and borrowing features. The lending yield is from the interest paid by borrowers and protocol may paid interest as FWX with a static amount per block. The borrowing annual percentage rates (APRs) is determined by the borrowing demand and lending supply, borrowing interest will be proportional divided to lenders. To borrow token from liquidity pool, other token is required as collateral. The maximum borrowing amount depends on the amount of collateral provided and Max LTV set.

FWX Membership Feature

Membership takes the form of an NFT, which is necessary for participation on the platform. This membership NFT acts like a bankbook, storing a record of all interactions with the protocol, such as lending tokens and initiating loans. Moreover, owners can enhance their membership NFT tier by staking FWX tokens on the platform, earning further privileges in relation to their tier. The staked tokens will be progressively unlocked for unstaking at a rate of 25% every 7 days.

Scope of Work

The security audit conducted does not replace the full security audit of the overall FWX protocol. The scope is limited to the **FWX Lending and Borrowing Pools, and FWX Membership features** and their related smart contracts.

The security audit covered the components at this specific state:

Item	Description
Components	▪ *FWX Lending and Borrowing Pools smart contracts* ▪ *FWX Membership smart contracts* ▪ *Imported associated smart contracts and libraries*
Git Repository	▪ *https://github.com/Forward-Development/Forward-Defi-Protocol*
Audit Commit	▪ *2cb4217175078e887db74171f3174ad2393d5dae (branch: audit)*
Reassessment Commit	▪ *0b848488327ddf4ae436dd485bc8570178f1d090 (branch: audit-1/freeze-4)*
Audited Files	▪ *./contracts/src/core/APHCore.sol* ▪ *./contracts/src/core/APHCoreProxy.sol* ▪ *./contracts/src/core/CoreBase.sol* ▪ *./contracts/src/core/CoreBaseFunc.sol* ▪ *./contracts/src/core/CoreBorrowing.sol* ▪ *./contracts/src/core/CoreFutureTrading.sol* ▪ *./contracts/src/core/CoreSetting.sol* ▪ *./contracts/src/core/event/CoreBorrowingEvent.sol* ▪ *./contracts/src/core/event/CoreEvent.sol* ▪ *./contracts/src/core/event/CoreFutureTradingEvent.sol* ▪ *./contracts/src/core/event/CoreSettingEvent.sol* ▪ *./contracts/src/governance/Timelock.sol* ▪ *./contracts/src/nft/Membership.sol* ▪ *./contracts/src/pool/APHPool.sol* ▪ *./contracts/src/pool/APHPoolProxy.sol* ▪ *./contracts/src/pool/InterestVault.sol* ▪ *./contracts/src/pool/PoolBase.sol* ▪ *./contracts/src/pool/PoolBaseFunc.sol* ▪ *./contracts/src/pool/PoolBorrowing.sol* ▪ *./contracts/src/pool/PoolLending.sol*

	- ./contracts/src/pool/PoolSetting.sol - ./contracts/src/pool/PoolToken.sol - ./contracts/src/pool/event/InterestVaultEvent.sol - ./contracts/src/pool/event/PoolLendingEvent.sol - ./contracts/src/pool/event/PoolSettingEvent.sol - ./contracts/src/stakepool/StakePool.sol - ./contracts/src/stakepool/StakePoolBase.sol - ./contracts/src/utils/PriceFeed.sol - ./contracts/src/utils/ProxyAdmin.sol - ./contracts/src/utils/TransperantProxy.sol - ./contracts/src/utils/Vault.sol - ./contracts/src/utils/WETHHandler.sol - ./contracts/externalContract/modify/non-upgradeable/AssetHandler.sol - ./contracts/externalContract/modify/non-upgradeable/Manager.sol - ./contracts/externalContract/modify/non-upgradeable/ManagerTimelock.sol - ./contracts/externalContract/modify/non-upgradeable/SelectorPausable.sol - ./contracts/externalContract/modify/upgradeable/AssetHandlerUpgradeable.sol - ./contracts/externalContract/modify/upgradeable/ManagerTimelockUpgradeable.sol - ./contracts/externalContract/modify/upgradeable/ManagerUpgradeable.sol - ./contracts/externalContract/modify/upgradeable/SelectorPausableUpgradeable.sol - ./contracts/externalContract/openzeppelin/non-upgradeable/AccessControl.sol - ./contracts/externalContract/openzeppelin/non-upgradeable/Address.sol - ./contracts/externalContract/openzeppelin/non-upgradeable/AggregatorV2V3Interface.sol - ./contracts/externalContract/openzeppelin/non-upgradeable/Context.sol - ./contracts/externalContract/openzeppelin/non-upgradeable/Counters.sol - ./contracts/externalContract/openzeppelin/non-upgradeable/ERC165.sol

	▪ *./contracts/externalContract/openzeppelin/non-upgradeable/ERC20.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/ERC721.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/ERC721Enumerable.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/ERC721Pausable.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IAccessControl.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IERC165.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IERC20.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IERC20Metadata.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IERC721.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IERC721Enumerable.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IERC721Metadata.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IERC721Receiver.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/IWETH.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/Initializable.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/Math.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/Ownable.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/Pausable.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/ReentrancyGuard.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/SafeERC20.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/Strings.sol* ▪ *./contracts/externalContract/openzeppelin/non-upgradeable/TimelockController.sol*

	▪ *./contracts/externalContract/openzeppelin/upgradeable/AddressUpgradeable.sol* ▪ *./contracts/externalContract/openzeppelin/upgradeable/ContextUpgradeable.sol* ▪ *./contracts/externalContract/openzeppelin/upgradeable/IERC20Upgradeable.sol* ▪ *./contracts/externalContract/openzeppelin/upgradeable/InitializableUpgradeable.sol* ▪ *./contracts/externalContract/openzeppelin/upgradeable/MathUpgradeable.sol* ▪ *./contracts/externalContract/openzeppelin/upgradeable/OwnableUpgradeable.sol* ▪ *./contracts/externalContract/openzeppelin/upgradeable/ReentrancyGuardUpgradeable.sol* ▪ *./contracts/externalContract/openzeppelin/upgradeable/SafeERC20Upgradeable.sol* ▪ *./contracts/interfaces/IAPHCore.sol* ▪ *./contracts/interfaces/IAPHCoreSetting.sol* ▪ *./contracts/interfaces/IAPHPool.sol* ▪ *./contracts/interfaces/IAPHPoolSetting.sol* ▪ *./contracts/interfaces/IInterestVault.sol* ▪ *./contracts/interfaces/IMembership.sol* ▪ *./contracts/interfaces/IPriceFeed.sol* ▪ *./contracts/interfaces/IRouter.sol* ▪ *./contracts/interfaces/IStakePool.sol* ▪ *./contracts/interfaces/IWeth.sol* ▪ *./contracts/interfaces/IWethERC20.sol* ▪ *./contracts/interfaces/IWethERC20Upgradeable.sol* ▪ *./contracts/interfaces/IWethHandler.sol* ▪ *Other imported associated Solidity files*
Excluded Files/Contracts	▪ *./contracts/mock/*.sol* ▪ *./contracts/src/helper/Helper.sol* ▪ *./contracts/src/helper/HelperBase.sol* ▪ *./contracts/src/utils/Faucet.sol* ▪ *./contracts/interfaces/IFaucet.sol* ▪ *./contracts/interfaces/IHelper.sol*

Remark: Our security best practices strongly recommend that the FWX team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.

Auditors

Role	Staff List
Auditors	Phuwanai Thummavet
Authors	Phuwanai Thummavet
Reviewers	Sumedt Jitpukdebodin

Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software has no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an "approval" or "endorsement" of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

Audit Result Summary

From the audit results and the remediation and response from the developer, Valix trusts that the **FWX Lending and Borrowing Pools, and FWX Membership features** have sufficient security protections to be safe for use.

Initially, Valix was able to identify **40 issues** that were categorized from the "Critical" to "Informational" risk level in the given timeframe of the assessment.

For the reassessment, the FWX team fixed 38 issues. There were 2 issues including 1 High-risk and 1 Low-risk marked as acknowledged but the team has prepared their mitigation plans already.

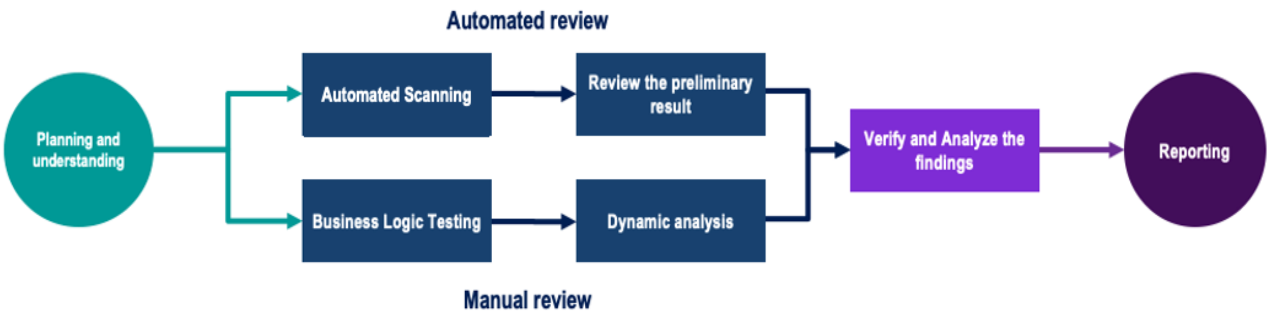
Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

Target	Assessment Result					Reassessment Result				
	C	H	M	L	I	C	H	M	L	I
FWX Lending and Borrowing Pools, and FWX Membership	4	12	16	8	-	0	1	0	1	-

Note: *Risk Rating* ***C*** *Critical,* ***H*** *High,* ***M*** *Medium,* ***L*** *Low,* ***I*** *Informational*

Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



Planning and Understanding

- Determine the scope of testing and understanding of the application's purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

Audit Items

We perform the audit according to the following categories and test names.

Category	ID	Test Name
Security Issue	SEC01	*Authorization Through tx.origin*
	SEC02	*Business Logic Flaw*
	SEC03	*Delegatecall to Untrusted Callee*
	SEC04	*DoS With Block Gas Limit*
	SEC05	*DoS with Failed Call*
	SEC06	*Function Default Visibility*
	SEC07	*Hash Collisions With Multiple Variable Length Arguments*
	SEC08	*Incorrect Constructor Name*
	SEC09	*Improper Access Control or Authorization*
	SEC10	*Improper Emergency Response Mechanism*
	SEC11	*Insufficient Validation of Address Length*
	SEC12	*Integer Overflow and Underflow*
	SEC13	*Outdated Compiler Version*
	SEC14	*Outdated Library Version*
	SEC15	*Private Data On-Chain*
	SEC16	*Reentrancy*
	SEC17	*Transaction Order Dependence*
	SEC18	*Unchecked Call Return Value*
	SEC19	*Unexpected Token Balance*
	SEC20	*Unprotected Assignment of Ownership*
	SEC21	*Unprotected SELFDESTRUCT Instruction*
	SEC22	*Unprotected Token Withdrawal*
	SEC23	*Unsafe Type Inference*
	SEC24	*Use of Deprecated Solidity Functions*
	SEC25	*Use of Untrusted Code or Libraries*
	SEC26	*Weak Sources of Randomness from Chain Attributes*
	SEC27	*Write to Arbitrary Storage Location*

Category	ID	Test Name
Functional Issue	**FNC01**	*Arithmetic Precision*
	FNC02	*Permanently Locked Fund*
	FNC03	*Redundant Fallback Function*
	FNC04	*Timestamp Dependence*
Operational Issue	**OPT01**	*Code With No Effects*
	OPT02	*Message Call with Hardcoded Gas Amount*
	OPT03	*The Implementation Contract Flow or Value and the Document is Mismatched*
	OPT04	*The Usage of Excessive Byte Array*
	OPT05	*Unenforced Timelock on An Upgradeable Proxy Contract*
Developmental Issue	**DEV01**	*Assert Violation*
	DEV02	*Other Compilation Warnings*
	DEV03	*Presence of Unused Variables*
	DEV04	*Shadowing State Variables*
	DEV05	*State Variable Default Visibility*
	DEV06	*Typographical Error*
	DEV07	*Uninitialized Storage Pointer*
	DEV08	*Violation of Solidity Coding Convention*
	DEV09	*Violation of Token (ERC20) Standard API*

Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

Risk Level	Definition
Critical	The code implementation does not match the specification, and it could disrupt the platform.
High	The code implementation does not match the specification, or it could result in losing funds for contract owners or users.
Medium	The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control.
Low	The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line.
Informational	Findings in this category are informational and may be further improved by following best practices and guidelines.

The **risk value** of each issue was calculated from the product of the **impact** and **likelihood values**, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

The shading of the matrix visualizes the different risk levels. Based on the acceptance criteria, the risk levels "Critical" and "High" are unacceptable. Any issue obtaining the above levels must be resolved to lower the risk to an acceptable level.

Findings

Review Findings Summary

The table below shows the summary of our assessments.

No.	Issue	Risk	Status	Functionality is in use
1	Uninitialized Implementation Contracts	Critical	Fixed	In use
2	Potential Theft Of Ethers From WETH Pool	Critical	Fixed	In use
3	Unusable Liquidate Function	Critical	Fixed	In use
4	Lack Of Repayment On Liquidated Loan	Critical	Fixed	In use
5	Phishing Attack To Steal Forward Tokens	High	Fixed	In use
6	Insecure Membership Authentication	High	Fixed	In use
7	Implementation Contracts May Not Be Upgradeable	High	Fixed	In use
8	Uninitialized Base Contracts	High	Fixed	In use
9	Transaction Revert On Loan Repayment	High	Fixed	In use
10	Malfunction Of Rollover Function	High	Fixed	In use
11	Potential Loss Of Pool's Asset	High	Acknowledged	In use
12	Loss Of Collateral Asset During Price Feeding System's Pause	High	Fixed	In use
13	Setting New Router May Halt Pool Token Swap	High	Fixed	In use
14	Contract Upgradeable Without Time Delay	High	Fixed	In use
15	Inaccurate Calculation For Liquidation Point	High	Fixed	In use
16	Flash Loan-Based Price Manipulation Attack On Liquidated Loan	High	Fixed	In use
17	Removal Recommendation For Mock Function	Medium	Fixed	In use
18	Reentrancy Attack to Steal All Forward Tokens From Distributor	Medium	Fixed	In use
19	No Allowlist For Collateral Tokens	Medium	Fixed	In use
20	Misplaced Transfer Approval For Forward Distributor	Medium	Fixed	In use

21	Incorrect Calculation For Bounty Reward	Medium	Fixed	In use
22	Lack Of Sanitization Checks On Loan Config Parameters	Medium	Fixed	In use
23	Underflow On Getting More Loan	Medium	Fixed	In use
24	Incorrect Calculations For Loan Repayment	Medium	Fixed	In use
25	Unchecking Price Feeding System's Pause	Medium	Fixed	In use
26	Inaccurate Interest Calculation For Liquidated Loan	Medium	Fixed	In use
27	Potential Loss Of Collateral Asset For Loan Borrower	Medium	Fixed	In use
28	Potential Lock Of Ethers	Medium	Fixed	In use
29	Incorrectly Updating Membership NFT Rank	Medium	Fixed	In use
30	Possibly Incorrect Calculation For Lending Forward Interest	Medium	Fixed	In use
31	Lack Of Stale Price Detection Mechanism	Medium	Fixed	In use
32	Usage Of Unsafe Functions	Medium	Fixed	In use
33	Liquidator May Receive Zero Bounty Reward	Low	Acknowledged	In use
34	Inaccurate Calculation For Current LTV	Low	Fixed	In use
35	Improperly Getting Membership NFT Rank	Low	Fixed	In use
36	Spamming On Minting Membership NFTs	Low	Fixed	In use
37	Rejection On Getting Active Loans	Low	Fixed	In use
38	Rejection On Getting Pool List	Low	Fixed	In use
39	Compiler May Be Susceptible To Publicly Disclosed Bugs	Low	Fixed	In use
40	Recommended Event Emissions For Transparency	Low	Fixed	In use

The statuses of the issues are defined as follows:

Fixed: The issue has been completely resolved and has no further complications.

Partially Fixed: The issue has been partially resolved.

Acknowledged: The issue's risk has been reported and acknowledged.

Detailed Result

This section provides all issues that we found in detail.

No. 1	Uninitialized Implementation Contracts		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/pool/APHPool.sol* *./contracts/src/pool/PoolSetting.sol* *./contracts/src/pool/APHPoolProxy.sol* *./contracts/src/core/APHCore.sol* *./contracts/src/core/CoreSetting.sol* *./contracts/src/core/APHCoreProxy.sol*		
Locations	*APHPool.sol L: 12 - 39* *PoolSetting.sol L: 66 - 71 and 73 - 78* *APHPoolProxy.sol L: 8 - 19, 21 - 40, 42 - 56, 58 - 69, 71 - 85, 87 - 101, and 103 - 127* *APHCore.sol L: 11 - 33* *CoreSetting.sol L: 38 - 43* *APHCoreProxy.sol L: 9 - 39, 41 - 63, 65 - 87, 89 - 103, and 105 - 127*		

Detailed Issue

The *APHPool* and *APHCore* are designed to be implementation contracts supporting an upgradeable feature. That is, these implementation contracts will be the logic contracts for their proxy contracts.

We found that both the *APHPool* and *APHCore* implementation contracts would be left uninitialized when they are deployed resulting in being taken over by an attacker. As a result, the attacker can perform a denial-of-service attack rendering the proxy contracts unusable.

To understand this issue, consider the following attack scenario of the *APHPool* implementation contract.

1. The *APHPool* implementation and proxy contracts are deployed and set up by a developer.
2. An attacker discovers the *APHPool* implementation contract uninitialized. He takes over the implementation contract by calling the *initialize* function (code snippet 1.1). As a result, the *manager* state variable is set to the *attacker address* (L23).

3. The attacker deploys a *Rogue* contract implementing a *(mock) activateRank* function.
4. The attacker makes a call to the *APHPool*'s *setPoolLendingAddress* function to set the *poolLendingAddress* state variable to the previously deployed *Rogue* contract address (L68 in code snippet 1.2).
5. The attacker executes the *APHPool*'s *activateRank* function which would make a *delegatecall* to the *(mock) activateRank* function of the *Rogue* contract pointed by the *poolLendingAddress* (L9 in code snippet 1.3).
6. The *(mock) activateRank* function invokes the *selfdestruct* instruction resulting in removing the contract code from the *APHPool* implementation contract address.
7. The *APHPool* proxy contract becomes unusable since its implementation contract was destroyed.

We consider this issue critical since suddenly after the *APHPool* and *APHCore* implementation contracts are destroyed, their proxy contracts would no longer operate, leaving all protocol's assets and users' assets frozen.

APHPool.sol

```
function initialize(
    address _tokenAddress,
    address _coreAddress,
    address _membershipAddress
) external virtual initializer {
    require(_tokenAddress != address(0), "APHPool/initialize/tokenAddress-zero-address");
    require(_coreAddress != address(0), "APHPool/initialize/coreAddress-zero-address");
    require(_membershipAddress != address(0), "APHPool/initialize/membership-zero-address");
    tokenAddress = _tokenAddress;
    coreAddress = _coreAddress;
    membershipAddress = _membershipAddress;
    manager = msg.sender;

    forwAddress = 0xAf0244ddcD9EaDA973b28b86BF2F18BCeea1D78f;
    interestVaultAddress = address(
        new InterestVault(tokenAddress, forwAddress, coreAddress, manager)
    );

    WEI_UNIT = 10**18;
    WEI_PERCENT_UNIT = 10**20;
    BLOCK_TIME = 3;
    initialItpPrice = WEI_UNIT;
    initialIfpPrice = WEI_UNIT;
    lambda = 1 ether / 100;

    emit Initialize(manager, coreAddress, interestVaultAddress, membershipAddress);
```

```
    emit TransferManager(address(0), manager);
}
```

Listing 1.1 The *APHPool* implementation contract's *initialize* function allows an attacker to become a contract manager

PoolSetting.sol

```
function setPoolLendingAddress(address _address) external onlyManager {
    address oldAddress = poolLendingAddress;
    poolLendingAddress = _address;

    emit SetPoolLendingAddress(msg.sender, oldAddress, _address);
}
```

Listing 1.2 The *setPoolLendingAddress* function allows an attacker to set the *poolLendingAddress*

APHPoolProxy.sol

```
function activateRank(uint256 nftId) external returns (uint8 newRank) {
    (bool success, bytes memory data) = poolLendingAddress.delegatecall(
        abi.encodeWithSignature("activateRank(uint256)", nftId)
    );
    if (!success) {
        if (data.length == 0) revert();
        assembly {
            revert(add(32, data), mload(data))
        }
    }
    newRank = abi.decode(data, (uint8));
}
```

Listing 1.3 The *activateRank*, one of the functions that can make a *delegatecall* to a *Rogue* contract pointed by the *poolLendingAddress*

Recommendations

To address this issue, we recommend adding the *constructor* like the code snippet below to both the *APHPool* and *APHCore* implementation contracts.

The added *constructor* guarantees that the implementation contract would be automatically initialized during its deployment, closing the room for an attacker to take over the implementation contract anymore.

APHPool.sol

```
contract APHPool is PoolBaseFunc, APHPoolProxy, PoolSetting {
    constructor() initializer {}

    function initialize(
        address _tokenAddress,
        address _coreAddress,
        address _membershipAddress
    ) external virtual initializer {

        // (...SNIPPED...)

    }

    // (...SNIPPED...)
}
```

Listing 1.4 The improved *APHPool* implementation contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our recommendation.

No. 2	Potential Theft Of Ethers From WETH Pool		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/pool/PoolSetting.sol*		
Locations	*PoolSetting.sol L: 80 - 82*		

Detailed Issue

We found a broken authorization issue on the *setWETHHandler* function (code snippet 2.1) in the *PoolSetting* contract that allows anyone to configure the *wethHandler* with any arbitrary address.

We also found that the *_transferFromOut* and *_transferOut* functions (L34 - 49 and L51 - 65 in code snippet 2.2) in the *AssetHandler* contract employ the *WETHHandler* contract indicated by the associated *wethHandler* to unwrap WETH tokens to Ethers (native coin) and then transfer the unwrapped Ethers to a destination address.

Both the *_transferFromOut* and *_transferOut* functions are being utilized by several functions. The following lists only the functions affected by the issue.

1. ***withdraw* function** (*L69 - 98 in PoolLending.sol*)
2. ***claimAllInterest* function** (*L103 - 132 in PoolLending.sol*)
3. ***claimTokenInterest* function** (*L138 - 154 in PoolLending.sol*)
4. ***repay* function** (*L46 - 87 in CoreBorrowing.sol*)
5. ***adjustCollateral* function** (*L94 - 117 in CoreBorrowing.sol*)
6. ***liquidate* function** (*L146 - 162 in CoreBorrowing.sol*)
7. ***borrow* function** (*L16 - 37 in PoolBorrowing.sol*)

The code snippet 2.3 shows the *borrow* function, one of the functions that transfer Ethers out of the *WETH Pool*. With the broken authorization issue on the *setWETHHandler* function, an attacker can easily mock the *WETHHandler* contract to steal all Ethers transferred out from the *WETH Pool* by configuring the *wethHandler* to point to the mock contract.

PoolSetting.sol

```
function setWETHHandler(address _address) external {
    wethHandler = _address;
}
```

Listing 2.1 The *setWETHHandler* function for configuring the *wethHandler*

AssetHandler.sol

```
function _transferFromOut(
    address from,
    address to,
    address token,
    uint256 amount
) internal {
    if (amount == 0) {
        return;
    }
    if (token == wethAddress) {
        IWethERC20(wethAddress).transferFrom(from, wethHandler, amount);
        WETHHandler(payable(wethHandler)).withdrawETH(to, amount);
    } else {
        IERC20(token).transferFrom(from, to, amount);
    }
}

function _transferOut(
    address to,
    address token,
    uint256 amount
) internal {
    if (amount == 0) {
        return;
    }
    if (token == wethAddress) {
        IWethERC20(wethAddress).transfer(wethHandler, amount);
        WETHHandler(payable(wethHandler)).withdrawETH(to, amount);
    } else {
        IERC20(token).transfer(to, amount);
    }
}
```

Listing 2.2 The *_transferFromOut* and *_transferOut* functions that hire the *wethHandler* to transfer Ethers to a destination (*to*) address

PoolBorrowing.sol

```
function borrow(
    uint256 loanId,
    uint256 nftId,
    uint256 borrowAmount,
    uint256 collateralSentAmount,
    address collateralTokenAddress
) external payable nonReentrant whenFuncNotPaused(msg.sig) returns (CoreBase.Loan memory) {
    nftId = _getUsableToken(nftId);

    if (collateralSentAmount != 0) {
        _transferFromIn(tx.origin, coreAddress, collateralTokenAddress, collateralSentAmount);
    }
    CoreBase.Loan memory loan = _borrow(
        loanId,
        nftId,
        borrowAmount,
        collateralSentAmount,
        collateralTokenAddress
    );
    _transferOut(tx.origin, tokenAddress, borrowAmount);
    return loan;
}
```

Listing 2.3 The *borrow* function is one of the functions that transfer Ethers out of the *WETH Pool*

Recommendations

To address this issue, we recommend applying the *onlyManager* modifier to the *setWETHHandler* function as shown in the code snippet below. This allows only a platform manager to configure the *wethHandler*.

PoolSetting.sol

```
function setWETHHandler(address _address) external onlyManager {
    wethHandler = _address;
}
```

Listing 2.4 The resolved *setWETHHandler* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue as per our suggestion.

No. 3	Unusable Liquidate Function		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 146 - 162*		

Detailed Issue

We found the *liquidate* function sending a wrong token for a bounty reward as shown in L160 in the code snippet below. This always makes the *liquidate* function revert a transaction. Consequently, the protocol cannot liquidate loans that reach the liquidation point.

CoreBorrowing.sol

```
146 function liquidate(uint256 loanId, uint256 nftId)
147     external
148     whenFuncNotPaused(msg.sig)
149     nonReentrant
150     returns (
151         uint256 repayBorrow,
152         uint256 repayInterest,
153         uint256 bountyReward,
154         uint256 leftOverCollateral
155     )
156 {
157     Loan storage loan = loans[nftId][loanId];
158     (repayBorrow, repayInterest, bountyReward, leftOverCollateral) =
    _liquidate(loanId, nftId);
159
160     _transferOut(msg.sender, loan.borrowTokenAddress, bountyReward);
161     _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress,
    leftOverCollateral);
162 }
```

Listing 3.1 The *liquidate* function sending a wrong token

Recommendations

We recommend changing the associated token to ***loan.collateralTokenAddress*** instead like L160 in the code snippet below.

CoreBorrowing.sol

```
146 function liquidate(uint256 loanId, uint256 nftId)
147     external
148     whenFuncNotPaused(msg.sig)
149     nonReentrant
150     returns (
151         uint256 repayBorrow,
152         uint256 repayInterest,
153         uint256 bountyReward,
154         uint256 leftOverCollateral
155     )
156 {
157     Loan storage loan = loans[nftId][loanId];
158     (repayBorrow, repayInterest, bountyReward, leftOverCollateral) =
_liquidate(loanId, nftId);
159 
160     _transferOut(msg.sender, loan.collateralTokenAddress, bountyReward);
161     _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress,
leftOverCollateral);
162 }
```

Listing 3.2 The improved *liquidate* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our recommendation.

No. 4	Lack Of Repayment On Liquidated Loan		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 146 - 162*		

Detailed Issue

We found that the *liquidate* function does not repay the borrowed asset and borrowing interest back to its pool (as shown in the code snippet below). This makes the borrowed asset and the borrowing interest locked in the *APHCore* contract, resulting in the loss of the pool's assets.

CoreBorrowing.sol

```
146 function liquidate(uint256 loanId, uint256 nftId)
147     external
148     whenFuncNotPaused(msg.sig)
149     nonReentrant
150     returns (
151         uint256 repayBorrow,
152         uint256 repayInterest,
153         uint256 bountyReward,
154         uint256 leftOverCollateral
155     )
156 {
157     Loan storage loan = loans[nftId][loanId];
158     (repayBorrow, repayInterest, bountyReward, leftOverCollateral) =
    _liquidate(loanId, nftId);
159 
160     _transferOut(msg.sender, loan.borrowTokenAddress, bountyReward);
161     _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress,
    leftOverCollateral);
162 }
```

Listing 4.1 The *liquidate* function that does not repay the borrowed asset and borrowing interest back to its pool

Recommendations

We recommend updating the *liquidate* function to repay the borrowed asset (L160 - 163) and the borrowing interest (L164 - 167) back to the corresponding pool as shown in the code snippet below.

CoreBorrowing.sol

```
146 function liquidate(uint256 loanId, uint256 nftId)
147     external
148     whenFuncNotPaused(msg.sig)
149     nonReentrant
150     returns (
151         uint256 repayBorrow,
152         uint256 repayInterest,
153         uint256 bountyReward,
154         uint256 leftOverCollateral
155     )
156 {
157     Loan storage loan = loans[nftId][loanId];
158     (repayBorrow, repayInterest, bountyReward, leftOverCollateral) =
_liquidate(loanId, nftId);
159
160     IERC20(loan.borrowTokenAddress).safeTransfer(
161         assetToPool[loan.borrowTokenAddress],
162         repayBorrow
163     );
164     IERC20(loan.borrowTokenAddress).safeTransfer(
165         IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(),
166         repayInterest
167     );
168
169     _transferOut(msg.sender, loan.borrowTokenAddress, bountyReward);
170     _transferOut(_getTokenOwnership(nftId), loan.collateralTokenAddress,
leftOverCollateral);
171 }
```

Listing 4.2 The improved *liquidate* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our suggestion.

No. 5	Phishing Attack To Steal Forward Tokens		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/stakepool/StakePool.sol* *./contracts/src/nft/Membership.sol*		
Locations	*StakePool.sol L: 140 - 148 and 207 - 222* *Membership.sol L: 128 - 130 and 154 - 162*		

Detailed Issue

We found potential phishing attacks on the *unstake* function of the *StakePool* contract (code snippet 5.1), leading to the stealing of the *staker*'s claimable *Forward* tokens.

Specifically, the *unstake* function firstly calls the *usableTokenId* function of the *Membership* contract (L146) to authenticate and prove ownership of the specified *nftId* and then receive the legitimate (proved) *nftId*. After that, the *unstake* function invokes the *_unstake* function to perform the unstaking process (L147).

Code snippet 5.2 presents the *usableTokenId* function which calls another internal function *_usableTokenId* (L129). The root cause of this issue resides in the *_usableTokenId* function (L154 - 162) in which the function authenticates ownership of the *given nftId* with *tx.origin* (L156 and L159).

With the *tx.origin*, an attacker can make a phishing campaign to act as a *Forward staker* to execute the *_unstake* function in L207 - 222 in code snippet 5.3. In L219, all *claimable Forward* tokens owned by the *phished staker* (victim) would be transferred to the attacker.

StakePool.sol

```
140 function unstake(uint256 nftId, uint256 amount)
141     external
142     nonReentrant
143     whenFuncNotPaused(msg.sig)
144     returns (StakeInfo memory)
145 {
146     nftId = IMembership(membershipAddress).usableTokenId(nftId);
147     return _unstake(nftId, amount);
148 }
```

Listing 5.1 The *unstake* function of the *StakePool* contract

Membership.sol

```
function usableTokenId(uint256 tokenId) external view returns (uint256) {
    return _usableTokenId(tokenId);
}

// (...SNIPPED...)

function _usableTokenId(uint256 tokenId) internal view returns (uint256) {
    if (tokenId == 0) {
        tokenId = _defaultMembership[tx.origin];
        require(tokenId != 0, "Membership/do-not-owned-any-membership-card");
    } else {
        require(ownerOf(tokenId) == tx.origin,
"Membership/caller-is-not-card-owner");
    }
    return tokenId;
}
```

Listing 5.2 The *usableTokenId* and *_usableTokenId* functions
for authenticating and proving ownership of the specified *nftId*

StakePool.sol

```
function _unstake(uint256 nftId, uint256 amount) internal returns (StakeInfo
memory) {
    StakeInfo storage nftStakeInfo = stakeInfos[nftId];
    _settle(nftStakeInfo);

    require(nftStakeInfo.stakeBalance >= amount,
"StakePool/unstake-balance-is-insufficient");
    if (nftStakeInfo.claimableAmount < amount) {
        amount = nftStakeInfo.claimableAmount;
    }
    nftStakeInfo.stakeBalance -= amount;
    nftStakeInfo.claimableAmount -= amount;

    _updateNFTRank(nftId);
    _transferFromOut(stakeVaultAddress, msg.sender, forwAddress, amount);
    emit UnStake(msg.sender, nftId, amount);
    return nftStakeInfo;
}
```

Listing 5.3 The *_unstake* function transfers *claimable Forward* tokens
to a caller who is an attacker in an event of phishing attack

Recommendations

We recommend improving the *_usableTokenId* function like the code snippet below. The improved function guarantees that only the EOA (Externally Owned Account) users would be able to authenticate and prove ownership of the Membership NFTs (L155) as well as preventing the phishing attacks previously discussed (L157 and L160).

Membership.sol

```
154 function _usableTokenId(uint256 tokenId) internal view returns (uint256) {
155     require(msg.sender == tx.origin,
    "Membership/do-not-support-smart-contract");
156     if (tokenId == 0) {
157         tokenId = _defaultMembership[msg.sender];
158         require(tokenId != 0, "Membership/do-not-owned-any-membership-card");
159     } else {
160         require(ownerOf(tokenId) == msg.sender,
    "Membership/caller-is-not-card-owner");
161     }
162     return tokenId;
163 }
```

Listing 5.4 The improved *_usableTokenId* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed by improving the *_usableTokenId* function according to our recommendation.

No. 6	Insecure Membership Authentication		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol* *./contracts/src/pool/PoolBorrowing.sol* *./contracts/src/pool/PoolLending.sol* *./contracts/src/stakepool/StakePool.sol* *./contracts/src/nft/Membership.sol*		
Locations	*CoreBorrowing.sol L: 46 - 87, 97 - 117, and 127 - 135* *PoolBorrowing.sol L: 16 - 37* *PoolLending.sol L: 15 - 35, 43 - 59, 69 - 98, 103 - 132, 138 - 154, and 160 - 179* *StakePool.sol L: 125 - 133 and 140 - 148* *Membership.sol L: 154 - 162*		

Detailed Issue

We found an insecure authentication issue on the *_usableTokenId* function of the *Membership* contract (code snippet 6.1). This function uses *tx.origin* to authenticate and prove ownership of the specified Membership NFT *tokenId* (L156 and L159).

At this point, we found an insecure use of *tx.origin* in which an attacker can make a phishing campaign to act as a *user* (victim) to invoke the *Forward* platform's functions without the victim's consent.

Membership.sol

```
154 function _usableTokenId(uint256 tokenId) internal view returns (uint256) {
155     if (tokenId == 0) {
156         tokenId = _defaultMembership[tx.origin];
157         require(tokenId != 0, "Membership/do-not-owned-any-membership-card");
158     } else {
159         require(ownerOf(tokenId) == tx.origin,
"Membership/caller-is-not-card-owner");
160     }
161     return tokenId;
162 }
```

Listing 6.1 The *_usableTokenId* function for authenticating and proving ownership of the specified *tokenId*

The following lists *all affected functions* calling the *insecure _usableTokenId* function.

1. ***repay* function** (*L46 - 87 in CoreBorrowing.sol*)
2. ***adjustCollateral* function** (*L97 - 117 in CoreBorrowing.sol*)
3. ***rollover* function** (*L127 - 135 in CoreBorrowing.sol*)
4. ***borrow* function** (*L16 - 37 in PoolBorrowing.sol*)
5. ***activateRank* function** (*L15 - 35 in PoolLending.sol*)
6. ***deposit* function** (*L43 - 59 in PoolLending.sol*)
7. ***withdraw* function** (*L69 - 98 in PoolLending.sol*)
8. ***claimAllInterest* function** (*L103 - 132 in PoolLending.sol*)
9. ***claimTokenInterest* function** (*L138 - 154 in PoolLending.sol*)
10. ***claimForwInterest* function** (*L160 - 179 in PoolLending.sol*)
11. ***stake* function** (*L125 - 133 in StakePool.sol*)
12. ***unstake* function** (*L140 - 148 in StakePool.sol*) – we also found potential phishing attacks for stealing *Forward* tokens (refer to issue no. 5 for details)

Code snippet 6.2 shows the *adjustCollateral* function (one of the affected functions) that eventually executes the *insecure _usableTokenId* function (L100). Subsequently, an attacker can make a phishing attack to adjust any loans' collateral assets belonging to a phished user without their consent. Hence, this can harm the *Forward* platform users' assets.

Furthermore, we also found the insecure use of *tx.origin* on all the affected functions. For instance, the *adjustCollateral* function is making use of the *insecure tx.origin* to refer to a loan owner (L107 and L114) that is prone to be phished.

CoreBorrowing.sol

```
94  function adjustCollateral(
95      uint256 loanId,
96      uint256 nftId,
97      uint256 collateralAdjustAmount,
98      bool isAdd
99  ) external payable whenFuncNotPaused(msg.sig) nonReentrant returns (Loan memory)
    {
100     nftId = _getUsableToken(nftId);
101     Loan storage loan = loans[nftId][loanId];
102 
103     Loan memory loanData = _adjustCollateral(loanId, nftId,
    collateralAdjustAmount, isAdd);
104     if (isAdd) {
105         // add colla to core
106         _transferFromIn(
107             tx.origin,
108             address(this),
109             loan.collateralTokenAddress,
```

```
110             collateralAdjustAmount
111         );
112     } else {
113         // withdraw colla to user
114         _transferOut(tx.origin, loan.collateralTokenAddress,
collateralAdjustAmount);
115     }
116     return loanData;
117 }
```

Listing 6.2 The *adjustCollateral*, one of the affected functions
that make use of the *insecure _usableTokenId* function as well as *insecure tx.origin*

Recommendations

We recommend updating the *_usableTokenId* function like the code snippet 6.3. The improved function guarantees that only the EOA (Externally Owned Account) users would be able to authenticate and prove ownership of the Membership NFTs (L155) as well as preventing the phishing attacks previously discussed (L157 and L160).

Membership.sol

```
154 function _usableTokenId(uint256 tokenId) internal view returns (uint256) {
155     require(msg.sender == tx.origin,
"Membership/do-not-support-smart-contract");
156     if (tokenId == 0) {
157         tokenId = _defaultMembership[msg.sender];
158         require(tokenId != 0, "Membership/do-not-owned-any-membership-card");
159     } else {
160         require(ownerOf(tokenId) == msg.sender,
"Membership/caller-is-not-card-owner");
161     }
162     return tokenId;
163 }
```

Listing 6.3 The improved *_usableTokenId* function

Furthermore, we also recommend updating all the affected functions (including the *adjustCollateral* function) that are making use of the *insecure tx.origin* like the code snippet 6.4. Specifically, the *adjustCollateral* function is improved by using the *msg.sender* instead of the *tx.origin* (L107 and L114). The *msg.sender* always guarantees that we are referring to the function caller, preventing phishing attacks.

CoreBorrowing.sol

```
function adjustCollateral(
    uint256 loanId,
    uint256 nftId,
    uint256 collateralAdjustAmount,
    bool isAdd
) external payable whenFuncNotPaused(msg.sig) nonReentrant returns (Loan memory)
{
    nftId = _getUsableToken(nftId);
    Loan storage loan = loans[nftId][loanId];

    Loan memory loanData = _adjustCollateral(loanId, nftId, collateralAdjustAmount, isAdd);
    if (isAdd) {
        // add colla to core
        _transferFromIn(
            msg.sender,
            address(this),
            loan.collateralTokenAddress,
            collateralAdjustAmount
        );
    } else {
        // withdraw colla to user
        _transferOut(msg.sender, loan.collateralTokenAddress, collateralAdjustAmount);
    }
    return loanData;
}
```

Listing 6.4 The improved *adjustCollateral* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue by updating the *_usableTokenId* function as well as all the affected functions as per our recommendation.

No. 7	Implementation Contracts May Not Be Upgradeable		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*All Solidity files directly or indirectly used by the APHPool and APHCore contracts*		
Locations	*Not specific*		

Detailed Issue

The *APHPool* and *APHCore* are designed to be implementation contracts supporting an upgradeable feature. However, we found some conflict coding practices which may impede the contracts from upgrading.

1. **Both implementation contracts inherit from non-upgradeable base contracts**
   For example, the *PoolBase* contract inherits from non-upgradeable base contracts (L5 - 13 in code snippet 7.1) such as *AssetHandler*, *Manager*, *ReentrancyGuard*, *Initializable*, *SelectorPausable*, etc.

   The following lists all contracts that need to support upgradeable.
   - ./contracts/src/pool/APHPool.sol
   - ./contracts/src/pool/APHPoolProxy.sol
   - ./contracts/src/pool/PoolBase.sol
   - ./contracts/src/pool/PoolBaseFunc.sol
   - ./contracts/src/pool/PoolBorrowing.sol
   - ./contracts/src/pool/PoolLending.sol
   - ./contracts/src/pool/PoolSetting.sol
   - ./contracts/src/pool/PoolToken.sol
   - ./contracts/src/core/APHCore.sol
   - ./contracts/src/core/APHCoreProxy.sol
   - ./contracts/src/core/CoreBase.sol
   - ./contracts/src/core/CoreBaseFunc.sol
   - ./contracts/src/core/CoreBorrowing.sol
   - ./contracts/src/core/CoreFutureTrading.sol
   - ./contracts/src/core/CoreSetting.sol
   - ./contracts/src/utils/Manager.sol
   - ./contracts/src/utils/AssetHandler.sol
   - ./contracts/externalContract/openzeppelin/Math.sol

- ./contracts/externalContract/openzeppelin/Context.sol
- ./contracts/externalContract/modify/SelectorPausable.sol
- ./contracts/externalContract/openzeppelin/Initializable.sol
- ./contracts/externalContract/openzeppelin/ReentrancyGuard.sol
- ./contracts/externalContract/openzeppelin/Address.sol
- And all their base contracts

2. **Some base contracts define state variables without allocating the reserved storage slots (__*gaps*)**

   As you can see in code snippet 7.1, the *PoolBase* contract defines state variables but does not allocate the reserved storage slots (__*gaps*) which might not support contract upgrade in case there might be some state variables need to be added in the future version of the contract.

   The following lists the contracts that might need to allocate the reserved storage slots.
   - ./contracts/src/core/CoreBase.sol
   - ./contracts/src/pool/PoolBase.sol
   - ./contracts/src/pool/PoolToken.sol
   - ./contracts/src/utils/AssetHandler.sol
   - ./contracts/src/utils/Manager.sol
   - ./contracts/externalContract/modify/SelectorPausable.sol
   - ./contracts/externalContract/openzeppelin/ReentrancyGuard.sol

3. **Some base contracts initialize state variables in field declarations or constructors**

   Some base contracts such as *AssetHandler* (L11 and L15 in code snippet 7.2) initialize state variables in field declarations or constructors which would be effective on the implementation contracts only, not on the proxy contracts. Thus, the state variables would be left uninitialized on the proxy contracts.

   The following lists the contracts that initialize state variables in field declarations or constructors.
   - ./contracts/src/utils/AssetHandler.sol
   - ./contracts/externalContract/openzeppelin/ReentrancyGuard.sol

PoolBase.sol

```
pragma solidity 0.8.7;

import "../../externalContract/openzeppelin/Address.sol";
import "../../externalContract/openzeppelin/ReentrancyGuard.sol";
import "../../externalContract/openzeppelin/Initializable.sol";
import "../../externalContract/modify/SelectorPausable.sol";

import "../utils/AssetHandler.sol";
import "../utils/Manager.sol";

contract PoolBase is AssetHandler, Manager, ReentrancyGuard, Initializable, SelectorPausable {
    struct Lend {
        uint8 rank;
        uint64 updatedTimestamp;
    }

    struct WithdrawResult {
        uint256 principle;
        uint256 tokenInterest;
        uint256 forwInterest;
        uint256 pTokenBurn;
        uint256 itpTokenBurn;
        uint256 ifpTokenBurn;
        uint256 tokenInterestBonus;
        uint256 forwInterestBonus;
    }

    uint256 internal WEI_UNIT; //               // 1e18
    uint256 internal WEI_PERCENT_UNIT; //       // 1e20 (100*1e18 for calculating percent)
    uint256 public BLOCK_TIME; //               // time between each block in seconds

    address public poolLendingAddress; //       // address of pool lending logic contract
    address public poolBorrowingAddress; //     // address of pool borrowing logic contract
    address public forwAddress; //              // forw token's address
    address public membershipAddress; //        // address of membership contract
    address public interestVaultAddress; //     // address of interestVault contract
    address public tokenAddress; //             // address of token which pool allows to lend
    address public coreAddress; //              // address of APHCore contract
    mapping(uint256 => Lend) lenders; //        // map nftId => rank

    uint256 internal initialItpPrice;
```

```
    uint256 internal initialIfpPrice;

    // borrowing interest params
    uint256 public lambda; //                         // constant use for weight forw token in iftPrice

    uint256 public targetSupply; //                   // weighting factor to proportional reduce utilOptimse vaule if total lending is less than targetSupply

    uint256[10] public rates; //                      // list of target interest rate at each util
    uint256[10] public utils; //                      // list of utilization rate to which each rate reached
    uint256 public utilsLen; //                       // length of current active rates and utils (both must be equl)
}
```

Listing 7.1 The *PoolBase* contract that does not support upgradeable

AssetHandler.sol

```
contract AssetHandler {
    address public wethAddress = 0xae13d989daC2f0dEbFf460aC112a837C89BAa7cd;

    //address public constant wethToken = 0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c  // bsc (Wrapped BNB)

    address public wethHandler = 0x64493B5B3419e116F9fbE3ec41cF2E65Ef15cAB6;

    function _transferFromIn(
        address from,
        address to,
        address token,
        uint256 amount
    ) internal {

        // (...SNIPPED...)

    }

    function _transferFromOut(
        address from,
        address to,
        address token,
        uint256 amount
    ) internal {

        // (...SNIPPED...)

    }
```

```

    function _transferOut(
        address to,
        address token,
        uint256 amount
    ) internal {

        // (...SNIPPED...)

    }
}
```

Listing 7.2 The *AssetHandler* contract that initializes state variables in field declaration

Recommendations

We recommend updating both the *APHPool* and *APHCore* implementation contracts to ensure that the contracts support the future upgrade as planned.

Consider the code snippets 7.3 and 7.4 below for example.

1. The *PoolBase* contract inherits from upgradeable base contracts only (L5 - 13 in code snippet 7.3).

   Note: Some base contracts are inherited by both upgradeable and non-upgradeable contracts. Our recommendation is to separate base contracts into two versions.

2. The *PoolBase* and *AssetHandler* contracts allocate the *__gaps* variables (L56 in code snippet 7.3 and L76 in code snippet 7.4 respectively) for the reserved storage slots.

3. The *AssetHandler* contract also initializes the *wethAddress* and *wethHandler* state variables using the internal *__AssetHandler_init_unchained* function (L17 - 23 in code snippet 7.4) instead of the field declaration or constructor.

PoolBase.sol

```
pragma solidity 0.8.7;

import "../../externalContract/openzeppelin-contracts/AddressUpgradeable.sol";
import "../../externalContract/openzeppelin-contracts/ReentrancyGuardUpgradeable.sol";
import "../../externalContract/openzeppelin-contracts/Initializable.sol";
import "../../externalContract/modify/SelectorPausableUpgradeable.sol";

import "../utils/AssetHandlerUpgradeable.sol";
import "../utils/ManagerUpgradeable.sol";

contract PoolBase is AssetHandlerUpgradeable, ManagerUpgradeable, ReentrancyGuardUpgradeable, Initializable, SelectorPausableUpgradeable {
    struct Lend {
        uint8 rank;
        uint64 updatedTimestamp;
    }

    struct WithdrawResult {
        uint256 principle;
        uint256 tokenInterest;
        uint256 forwInterest;
        uint256 pTokenBurn;
        uint256 itpTokenBurn;
        uint256 ifpTokenBurn;
        uint256 tokenInterestBonus;
        uint256 forwInterestBonus;
    }

    uint256 internal WEI_UNIT; //               // 1e18
    uint256 internal WEI_PERCENT_UNIT; //       // 1e20 (100*1e18 for calculating percent)
    uint256 public BLOCK_TIME; //               // time between each block in seconds

    address public poolLendingAddress; //       // address of pool lending logic contract
    address public poolBorrowingAddress; //     // address of pool borrowing logic contract
    address public forwAddress; //              // forw token's address
    address public membershipAddress; //        // address of membership contract
    address public interestVaultAddress; //     // address of interestVault contract
    address public tokenAddress; //             // address of token which pool allows to lend
    address public coreAddress; //              // address of APHCore contract
    mapping(uint256 => Lend) lenders; //        // map nftId => rank

```

```
    uint256 internal initialItpPrice;
    uint256 internal initialIfpPrice;

    // borrowing interest params
    uint256 public lambda; //                   // constant use for weight forw token in iftPrice

    uint256 public targetSupply; //             // weighting factor to proportional reduce utilOptimse vaule if total lending is less than targetSupply

    uint256[10] public rates; //                // list of target interest rate at each util
    uint256[10] public utils; //                // list of utilization rate to which each rate reached
    uint256 public utilsLen; //                 // length of current active rates and utils (both must be equl)

    // Allocating __gap or not is up to the developer's decision
    uint256[50] private __gap;
}
```

Listing 7.3 The improved *PoolBase* contract

AssetHandler.sol

```
contract AssetHandler is Initializable {
    address public wethAddress;

    //address public constant wethToken = 0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c  // bsc (Wrapped BNB)

    address public wethHandler;

    function __AssetHandler_init_unchained(
        address _wethAddress,
        address _wethHandler
    ) internal onlyInitializing {
        wethAddress = _wethAddress;
        wethHandler = _wethHandler;
    }

    function _transferFromIn(
        address from,
        address to,
        address token,
        uint256 amount
    ) internal {

        // (...SNIPPED...)
```

```
    }

    function _transferFromOut(
        address from,
        address to,
        address token,
        uint256 amount
    ) internal {

        // (...SNIPPED...)

    }

    function _transferOut(
        address to,
        address token,
        uint256 amount
    ) internal {

        // (...SNIPPED...)

    }

    // Allocating __gap or not is up to the developer's decision
    uint256[50] private __gap;
}
```

Listing 7.4 The improved *AssetHandler* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our recommendation.

No. 8	Uninitialized Base Contracts		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/APHCore.sol* *./contracts/src/pool/APHPool.sol* *./contracts/src/utils/AssetHandler.sol* *./contracts/externalContract/openzeppelin/ReentrancyGuard.sol*		
Locations	*APHCore.sol L: 11 - 33* *APHPool.sol L: 12 - 39* *AssetHandler.sol L: 11 and 15* *ReentrancyGuard.sol L: 40*		

Detailed Issue

We found that the *APHCore* and *APHPool* implementation contracts do not initialize their base contracts' state variables. The base contracts in question include *AssetHandler* and *ReentrancyGuard*.

The root cause of this issue is that both the *AssetHandler* and *ReentrancyGuard* base contracts do not support an upgradeable feature. Therefore, initializing state variables using the field declaration (L11 and L15 in code snippet 8.1) or constructor (L40 in code snippet 8.2) would not be effective on the proxy contracts.

Consequently, the resulting uninitialized state variables can render the proxy contracts unusable.

AssetHandler.sol

```
10 contract AssetHandler {
11     address public wethAddress = 0xae13d989daC2f0dEbFf460aC112a837C89BAa7cd;
12 
13     //address public constant wethToken =
   0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c  // bsc (Wrapped BNB)
14 
15     address public wethHandler = 0x64493B5B3419e116F9fbE3ec41cF2E65Ef15cAB6;

       // (...SNIPPED...)

66 }
```

Listing 8.1 The *AssetHandler* contract that initializes state variables in field declaration

ReentrancyGuard.sol

```
abstract contract ReentrancyGuard {

    // (...SNIPPED...)

    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    constructor() {
        _status = _NOT_ENTERED;
    }

    // (...SNIPPED...)

}
```

Listing 8.2 The *ReentrancyGuard* contract that initializes a state variable using the *constructor*

Recommendations

To remediate this issue, we recommend updating the *AssetHandler* and *ReentrancyGuard* base contracts to support an upgradeable feature and initializing their state variables using *initialize* functions.

For example, the *AssetHandler* contract can initialize its state variables using the *__AssetHandler_init_unchained* function (L17 - 23 in the code snippet below). Whereas, the *ReentrancyGuard* can be upgraded to be the *ReentrancyGuardUpgradeable*. For more details, please refer to https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/security/ReentrancyGuardUpgradeable.sol.

AssetHandler.sol

```
10 contract AssetHandler is Initializable {
11     address public wethAddress;
12 
13     //address public constant wethToken = 
0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c  // bsc (Wrapped BNB)
14 
15     address public wethHandler;
16 
17     function __AssetHandler_init_unchained(
18         address _wethAddress,
19         address _wethHandler
20     ) internal onlyInitializing {
21         wethAddress = _wethAddress;
22         wethHandler = _wethHandler;
```

```
    }
    // (...SNIPPED...)
}
```

Listing 8.3 The improved *AssetHandler* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our recommendation.

No. 9	Transaction Revert On Loan Repayment		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol* *./contracts/src/utils/AssetHandler.sol*		
Locations	*CoreBorrowing.sol L: 46 - 87* *AssetHandler.sol L: 17 - 32*		

Detailed Issue

We found transaction revert issues on the *repay* function (code snippet 9.1) of the *CoreBorrowing* contract. During the repayment process, if the *loan*'s *borrowing* token is the *WETH*, the transaction can be reverted when the function executes the *_transferFromIn* function in order to transfer *Ethers* (native coin) from the function caller to the corresponding *APHPool* (L70 - 75) and *APHPool*'s *interest vault* (L77 - 82).

The root cause of the transaction reverts is because the *_transferFromIn* function strictly checks the number of *Ethers* sent from the function caller must equal the given *amount* (L26 in code snippet 9.2).

CoreBorrowing.sol

```
46 function repay(
47     uint256 loanId,
48     uint256 nftId,
49     uint256 repayAmount,
50     bool isOnlyInterest
51 )
52     external
53     payable
54     whenFuncNotPaused(msg.sig)
55     nonReentrant
56     returns (uint256 borrowPaid, uint256 interestPaid)
57 {
58     nftId = _getUsableToken(nftId);
59     Loan storage loan = loans[nftId][loanId];
60     bool isLoanClosed;
61     uint256 tmpCollateralAmount = loan.collateralAmount;
62     (borrowPaid, interestPaid, isLoanClosed) = _repay(
63         loanId,
64         nftId,
```

```
        repayAmount,
        isOnlyInterest
    );

    if (borrowPaid > 0) {
        _transferFromIn(
            tx.origin,
            assetToPool[loan.borrowTokenAddress],
            loan.borrowTokenAddress,
            borrowPaid
        );
    }
    _transferFromIn(
        tx.origin,
        IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(),
        loan.borrowTokenAddress,
        interestPaid
    );
    if (isLoanClosed) {
        _transferOut(tx.origin, loan.collateralTokenAddress,
tmpCollateralAmount);
    }
    return (borrowPaid, interestPaid);
}
```

Listing 9.1 The *repay* function of the *CoreBorrowing* contract

AssetHandler.sol

```
function _transferFromIn(
    address from,
    address to,
    address token,
    uint256 amount
) internal {
    require(amount != 0, "AssetHandler/amount-is-zero");

    if (token == wethAddress) {
        require(amount == msg.value, "AssetHandler/value-not-matched");
        IWethERC20(wethAddress).deposit{value: amount}();
        IWethERC20(wethAddress).transfer(to, amount);
    } else {
        IERC20(token).transferFrom(from, to, amount);
    }
}
```

Listing 9.2 The *_transferFromIn* function of the *AssetHandler* contract

Recommendations

We recommend improving the *repay* function like the below code snippet. The improved function separates the logic for handling the loan's *borrowing token* into two parts. The first part handles the case of the *borrowing token* is *WETH* (L69 - 97). The second part handles the case of the *borrowing token* is *non-WETH* (L98 - 113).

CoreBorrowing.sol

```
46 function repay(
47     uint256 loanId,
48     uint256 nftId,
49     uint256 repayAmount,
50     bool isOnlyInterest
51 )
52     external
53     payable
54     whenFuncNotPaused(msg.sig)
55     nonReentrant
56     returns (uint256 borrowPaid, uint256 interestPaid)
57 {
58     nftId = _getUsableToken(nftId);
59     Loan storage loan = loans[nftId][loanId];
60     bool isLoanClosed;
61     uint256 tmpCollateralAmount = loan.collateralAmount;
62     (borrowPaid, interestPaid, isLoanClosed) = _repay(
63         loanId,
64         nftId,
65         repayAmount,
66         isOnlyInterest
67     );
68
69     if (loan.borrowTokenAddress == wethAddress) {
70         require(msg.value >= borrowPaid + interestPaid,
   "CoreBorrowing/insufficient-ether-amount");
71
72         // Ether -> WETH
73         _transferFromIn(
74             msg.sender,
75             address(this),
76             wethAddress,
77             msg.value
78         );
79
80         if (borrowPaid > 0) {
81             IERC20(wethAddress).safeTransfer(
82                 assetToPool[wethAddress],
83                 borrowPaid
84             );
```

```
        }
        IERC20(wethAddress).safeTransfer(
            IAPHPool(assetToPool[wethAddress]).interestVaultAddress(),
            interestPaid
        );

        // Return the remaining Ethers
        _transferOut(
            msg.sender,
            wethAddress,
            msg.value - (borrowPaid + interestPaid)
        );
    }
    else { // loan.borrowTokenAddress == non-WETH token
        if (borrowPaid > 0) {
            _transferFromIn(
                tx.origin,
                assetToPool[loan.borrowTokenAddress],
                loan.borrowTokenAddress,
                borrowPaid
            );
        }
        _transferFromIn(
            tx.origin,
          IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress(),
            loan.borrowTokenAddress,
            interestPaid
        );
    }

    if (isLoanClosed) {
        _transferOut(tx.origin, loan.collateralTokenAddress,
tmpCollateralAmount);
    }
    return (borrowPaid, interestPaid);
}
```

Listing 9.3 The improved *repay* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue in accordance with our recommendation.

No. 10	Malfunction Of Rollover Function		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 119 - 135*		

Detailed Issue

We found that the *rollover* function (the code snippet below) does not function as expected.

1. **Wrong function description**
   The *rollover* function is intended to be called by anyone and the function caller will get a *bounty reward* as an incentive.

   However, we found that the function description (L124 - 125) is incorrect as it states that the *delay fee* would be an incentive, not the *bounty reward*.

2. **Other users cannot call the function**
   We found that the function calls the *_getUsableToken* function (L133) to get a usable *nftId*. Since the *_getUsableToken* function is intended to authenticate and prove that the function caller is the owner of the inputted *nftId*.

   Therefore, the *rollover* function would not be able to be executed by other users, except the loan's owner.

3. **No bounty reward for a function caller**
   The *rollover* function does not send a bounty reward to the function caller.

4. **Wrong function argument**
   The *rollover* function passes "***address(this)***" as a caller into the internal function *_rollover* which is a wrong argument (L134).

CoreBorrowing.sol

```
119 /**
120     @dev Function to rollover loan with the given loanId and nftId.
121         Rollover is similar to close and open loan again to change loan's
interest rate.
122         If loan opened longer than 28 days, the interest from extended duration
is calculated
123         with delay fees (ex: 5%)
124         This function can be call by anyone, non-owner who rollver overdue loan
receives
125         delay fees as an incentive.
126     */
127 function rollover(uint256 loanId, uint256 nftId)
128     external
129     whenFuncNotPaused(msg.sig)
130     nonReentrant
131     returns (uint256, uint256)
132 {
133     nftId = _getUsableToken(nftId);
134     return _rollover(loanId, nftId, address(this));
135 }
```

Listing 10.1 The malfunctioning function *rollover*

Recommendations

We recommend updating the *rollover* function to function as expected. ***The code snippet below presents an idea of improving the function only. However, the function should be updated according to its functional design.***

The improved *rollover* function can be described as follows.

1. **Correct function description**
   The function description was corrected in L125.

2. **Anyone can call the function**
   The function was updated to allow anyone to execute (L134).

3. **Bounty reward for a function caller (excepting the loan's owner)**
   The function was updated according to its description. In other words, it would send a bounty reward to a function caller, except the loan's owner (L136 - 139).

4. **Correct function argument**
   The _*rollover*'s function argument was updated by passing the ***msg.sender*** instead in L134.

CoreBorrowing.sol

```
/**
    @dev Function to rollover loan with the given loanId and nftId.
        Rollover is similar to close and open loan again to change loan's interest rate.
        If loan opened longer than 28 days, the interest from extended duration is calculated
        with delay fees (ex: 5%)
        This function can be called by anyone, non-owner who rollvers overdue loan receives
        a bounty reward as an incentive.
    */
function rollover(uint256 loanId, uint256 nftId)
    external
    whenFuncNotPaused(msg.sig)
    nonReentrant
    returns (uint256, uint256)
{
    Loan storage loan = loans[nftId][loanId];
    (uint256 delayInterest, uint256 bountyReward) = _rollover(loanId, nftId, msg.sender);

    // Only user who is not a loan owner will get a bounty reward
    if (_getTokenOwnership(nftId) != msg.sender) {
        _transferOut(msg.sender, loan.collateralTokenAddress, bountyReward);
    }

    return (delayInterest, bountyReward);
}
```

Listing 10.2 The improved *rollover* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue by reworking the *rollover* function. The function would be executable by the loan's owner only, and the owner has to pay for both the delay interest and the bounty reward in terms of the loan's borrowing interest that would eventually be rewarded to all lenders in the pool.

No. 11	Potential Loss Of Pool's Asset		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 479 - 573*		

Detailed Issue

We found that the *_liquidate* function does not handle the critical case in which a liquidated loan cannot be closed a position as shown in the below code snippet in L551 - 555. If this critical case is left unhandled, the affected pool may gradually lose its asset (borrowing token).

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
487 {
        // (...SNIPPED...)

545         uint256 borrowTokenAmountSwap = amounts[amounts.length - 1];
546
547         leftOverCollateral = loan.collateralAmount - amounts[0];
548
549         (repayBorrow, repayInterest, ) = _repay(loanId, nftId,
borrowTokenAmountSwap, false);
550
551         if (loanExts[nftId][loanId].active == true) {
552             // TODO (future work): handle with ciritical condition, this part
must add pool subsidisation for pool loss
553             // Ciritical condition, protocol loss
554             // transfer int or sth else to pool
555         } else {
556             bountyReward = (leftOverCollateral * loanConfig.bountyFeeRate) /
WEI_PERCENT_UNIT;
```

```
            leftOverCollateral -= bountyReward;
        }

    // (...SNIPPED...)
}
```

Listing 11.1 The *_liquidate* function does not handle the critical case in which a liquidated loan cannot be closed a position

Recommendations

We recommend updating the *_liquidate* function to handle the critical case or implementing a monitoring system to keep track of the asset balance of each pool and fill up the pool with its corresponding asset (borrowing token) to cover up the pool's loss (for a middle-term plan).

Reassessment

The *FWX team* acknowledged this issue. For the short-term and middle-term plans, the *FWX team* will implement an off-chain monitoring system to address the pools' loss. For the long-term plan, the team will upgrade the *APHCore* contract to handle the associated critical case.

No. 12	Loss Of Collateral Asset During Price Feeding System's Pause		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol* *./contracts/src/utils/PriceFeed.sol*		
Locations	*CoreBorrowing.sol L: 479 - 573* *PriceFeed.sol L: 45 - 56*		

Detailed Issue

The *_liquidate* function queries the maximum swappable amount (*numberArray[2]*) by calling the *queryReturn* function (L520 - 524 in code snippet 12.1). Then, the maximum swappable amount will be used to determine two liquidation conditions (L527 - 535 for a normal condition and L537 - 543 for a critical condition).

The execution flow will enter the critical condition (L537 - 543) if the calculated maximum swappable amount is less than or equal to the loan's total debt (L526).

We found that the *queryReturn* function would always return zero (0) if the price feeding system is paused (L50 - 52 in code snippet 12.2). As a result, the execution flow would be forced to enter the critical condition (L537 - 543 in code snippet 12.1) regardless of the (real) value of the collateral asset.

Subsequently, the total loan's collateral asset would be forced to swap for a borrowing token to repay the liquidated loan (L549 in code snippet 12.1). Since the swapped borrowing token amount is overabundant, the leftover borrowing tokens would be locked in the *APHCore* contract and never be returned to the loan borrower.

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
```

```
{
    // (...SNIPPED...)

        address[] memory path_data = new address[](2);
        path_data[0] = loan.collateralTokenAddress;
        path_data[1] = loan.borrowTokenAddress;
        uint256[] memory amounts;

        numberArray[2] = IPriceFeed(priceFeedAddress).queryReturn(
            loan.collateralTokenAddress,
            loan.borrowTokenAddress,
            loan.collateralAmount
        );

        if (numberArray[2] > loan.borrowAmount + loan.interestOwed) {
            numberArray[2] = loan.borrowAmount + loan.interestOwed;
            // Normal condition, leftover collateral is exists
            amounts = IRouter(routerAddress).swapTokensForExactTokens(
                numberArray[2], //          // amountOut
                loan.collateralAmount, //   // amountInMax
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        } else {
            amounts = IRouter(routerAddress).swapExactTokensForTokens(
                loan.collateralAmount, //   // amountIn
                0, //                       // amountOutMin
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        }
        uint256 borrowTokenAmountSwap = amounts[amounts.length - 1];

        leftOverCollateral = loan.collateralAmount - amounts[0];

        (repayBorrow, repayInterest, ) = _repay(loanId, nftId,
borrowTokenAmountSwap, false);

    // (...SNIPPED...)
}
```

Listing 12.1 The *_liquidate* function of the *CoreBorrowing* contract

PriceFeed.sol

```
function queryReturn(
    address sourceToken,
    address destToken,
    uint256 sourceAmount
) public view returns (uint256 destAmount) {
    if (globalPricingPaused) {
        return 0;
    }
    (uint256 rate, uint256 precision) = _queryRate(sourceToken, destToken);

    destAmount = (sourceAmount * rate) / precision;
}
```

Listing 12.2 The *queryReturn* function of the *PriceFeed* contract

Recommendations

We recommend updating the *queryReturn* function to revert transactions during the pause of the price feeding system like L50 in the code snippet below.

PriceFeed.sol

```
45 function queryReturn(
46     address sourceToken,
47     address destToken,
48     uint256 sourceAmount
49 ) public view returns (uint256 destAmount) {
50     require(!globalPricingPaused, "PriceFeed/pricing-is-paused");
51 
52     (uint256 rate, uint256 precision) = _queryRate(sourceToken, destToken);
53 
54     destAmount = (sourceAmount * rate) / precision;
55 }
```

Listing 12.3 The improved *queryReturn* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed by reverting transactions during the pause of the price feeding system as suggested.

No. 13	Setting New Router May Halt Pool Token Swap		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreSetting.sol*		
Locations	*CoreSetting.sol L: 31 - 36 and 75 - 96*		

Detailed Issue

The *registerNewPool* function approves a *router* for transferring the *APHPool*'s corresponding token (L84 in code snippet 13.1). This approval would be triggered once a protocol manager registers a new pool.

However, we found that if a manager sets a new *router* via the *setRouterAddress* function (code snippet 13.2), the new *router* would not be able to transfer tokens of existing pools (i.e., approved for the old *router*) for a swap and there is no approach for the manager to approve the new *router* for those tokens.

CoreSetting.sol

```
75  function registerNewPool(
76      address _poolAddress,
77      uint256 _amount,
78      uint256 _targetBlock
79  ) external onlyManager {
80      require(poolToAsset[_poolAddress] == address(0),
    "CoreSetting/pool-is-already-exist");
81
82      address assetAddress = IAPHPool(_poolAddress).tokenAddress();
83      IERC20(forwAddress).approve(forwDistributorAddress, type(uint256).max);
84      IERC20(assetAddress).approve(routerAddress, type(uint256).max);
85
86      poolToAsset[_poolAddress] = assetAddress;
87      assetToPool[assetAddress] = _poolAddress;
88      swapableToken[assetAddress] = true;
89      poolList.push(_poolAddress);
90
91      lastSettleForw[_poolAddress] = block.number;
92
93      _setForwDisPerBlock(_poolAddress, _amount, _targetBlock);
94
95      emit RegisterNewPool(msg.sender, _poolAddress);
```

```
}
```

Listing 13.1 The *registerNewPool* function

CoreSetting.sol

```
function setRouterAddress(address _address) external onlyManager {
    address oldAddress = routerAddress;
    routerAddress = _address;

    emit SetRouterAddress(msg.sender, oldAddress, _address);
}
```

Listing 13.2 The *setRouterAddress* function

Recommendations

We recommend implementing the new functions *approveForRouter* and *_approveForRouter* as shown in code snippet 13.3. For the external function *approveForRouter* (L99 - 104), a manager can approve a specific token for the *router* directly.

Meanwhile, the internal function *_approveForRouter* (L106 - 112) can be called by the *registerNewPool* function (L85 in code snippet 13.4) to approve the new pool's token automatically.

CoreSetting.sol

```
99  function approveForRouter(
100     address _assetAddress
101 ) external onlyManager {
102     require(assetToPool[_assetAddress] != address(0),
    "CoreSetting/unsupported-asset");
103     _approveForRouter(_assetAddress);
104 }
105
106 function _approveForRouter(
107     address _assetAddress
108 ) internal {
109     IERC20(_assetAddress).safeApprove(routerAddress, type(uint256).max);
110
111     emit ApprovedForRouter(msg.sender, _assetAddress, routerAddress);
112 }
```

Listing 13.3 The new *approveForRouter* and *_approveForRouter* functions

CoreSetting.sol

```
function registerNewPool(
    address _poolAddress,
    uint256 _amount,
    uint256 _targetBlock
) external onlyManager {
    require(poolToAsset[_poolAddress] == address(0),
"CoreSetting/pool-is-already-exist");

    address assetAddress = IAPHPool(_poolAddress).tokenAddress();
    IERC20(forwAddress).approve(forwDistributorAddress, type(uint256).max);

    _approveForRouter(assetAddress);

    poolToAsset[_poolAddress] = assetAddress;
    assetToPool[assetAddress] = _poolAddress;
    swapableToken[assetAddress] = true;
    poolList.push(_poolAddress);

    lastSettleForw[_poolAddress] = block.number;

    _setForwDisPerBlock(_poolAddress, _amount, _targetBlock);

    emit RegisterNewPool(msg.sender, _poolAddress);
}
```

Listing 13.4 The improved *registerNewPool* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our recommendation.

No. 14	Contract Upgradeable Without Time Delay		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	***All Solidity files regarding the following smart contracts and modules:*** *APHCore contract,* *APHPool contract,* *Core borrowing module,* *Pool lending module,* *and Pool borrowing module*		
Locations	*Not specific*		

Detailed Issue

The *APHCore* and *APHPool* are upgradeable smart contracts. Furthermore, the *APHCore* contract allows the core borrowing module (via the *setCoreBorrowingAddress* function in code snippet 14.1) to upgrade its internal logic without upgrading the *main APHCore* itself.

Also, the *APHPool* contract allows the pool lending module (via the *setPoolLendingAddress* function in code snippet 14.2) and the pool borrowing module (via the *setPoolBorrowingAddress* function in code snippet 14.2) to upgrade their internal logic without upgrading the *main APHPool* itself.

We found that the upgrade mechanism is not bound to any time delay. This may raise concerns for users since the contract upgrade may contain malicious code to exploit the users' assets.

Moreover, imagine the case that a developer account is being compromised. An attacker can upgrade the contract with malicious code. Without the time delay, the attacker can steal all assets on the platform suddenly.

CoreSetting.sol

```
function setCoreBorrowingAddress(address _address) external onlyManager {
    address oldAddress = coreBorrowingAddress;
    coreBorrowingAddress = _address;

    emit SetCoreBorrowingAddress(msg.sender, oldAddress, _address);
}
```

Listing 14.1 The *setCoreBorrowingAddress* function

PoolSetting.sol

```
function setPoolLendingAddress(address _address) external onlyManager {
    address oldAddress = poolLendingAddress;
    poolLendingAddress = _address;

    emit SetPoolLendingAddress(msg.sender, oldAddress, _address);
}

function setPoolBorrowingAddress(address _address) external onlyManager {
    address oldAddress = poolBorrowingAddress;
    poolBorrowingAddress = _address;

    emit SetPoolBorrowingAddress(msg.sender, oldAddress, _address);
}
```

Listing 14.2 The *setPoolLendingAddress* and *setPoolBorrowingAddress* functions

Recommendations

We recommend applying the *Timelock* contract to the upgrade mechanism as follows:

Developer -> Timelock -> ProxyAdmin -> Proxy -> Logic (Implementation)

With the *Timelock* contract, every time a developer upgrades the *Logic* contract, the upgrade transaction will be deferred by the *Timelock* for some waiting period (e.g., 48 hours) configured by the developer. This enables users to examine the source code of the upgrading contract before it is effective, providing transparency.

The adoption of the *Timelock* also makes the contract upgrade more secure in case the developer finds some bugs during the upgrade; the developer can cancel the upgrade transaction by invoking the *Timelock*.

Since the *Forward protocol* has several complex features, and each feature may require different *Timelock* configurations, using a single *Timelock* instance to handle multiple time delays for all features may be cumbersome and can lead to transparency issues. The following figure is our suggested design (one of the possible designs) that may be suitable for the *Forward protocol*.

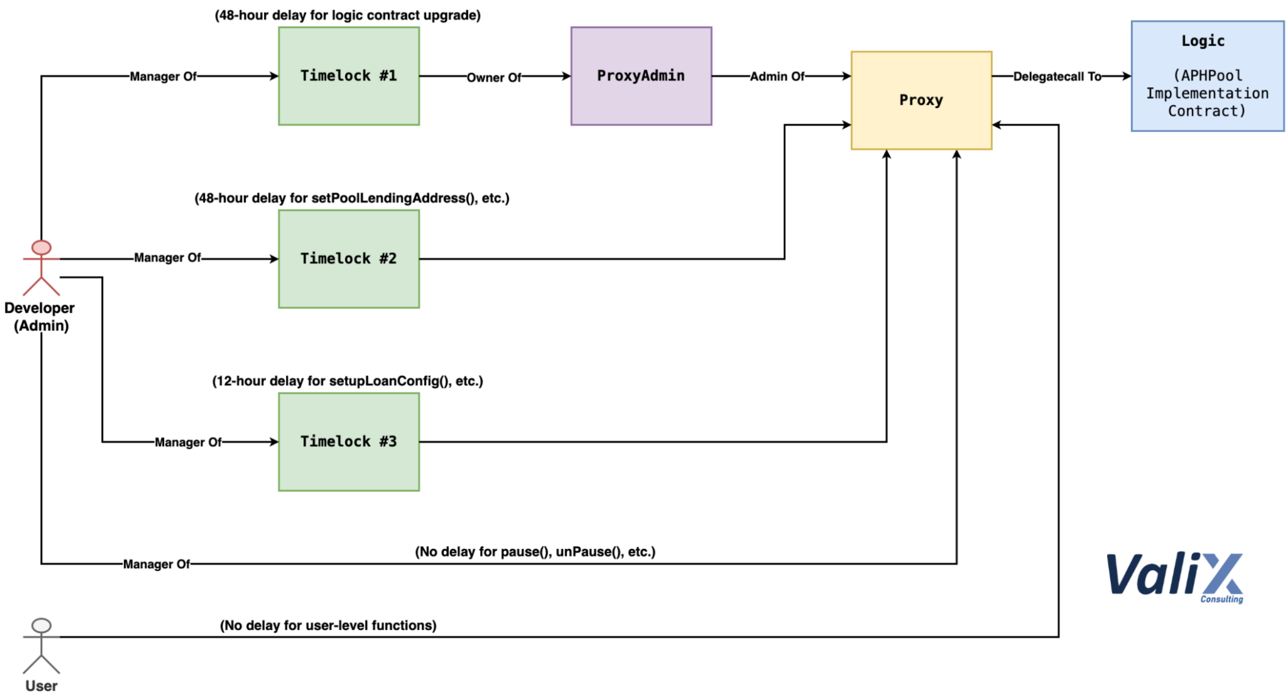


Figure 14.1 Recommended design of using different *Timelock* instances
to handle several features with multiple time delays

There are three *Timelock* instances:

1. **48-hour *Timelock* instance** for controlling the upgrade mechanism of the *Logic* contract (using the *ProxyAdmin* as a managing contract for the *Proxy* contract).

2. **48-hour *Timelock* instance** for managing critical administrative functions such as *setPoolLendingAddress*, etc.

3. **12-hour *Timelock* instance** for handling lower administrative functions such as *setupLoanConfig*, etc.

For the *pause* and *unPause* functions, we consider them the kill-switch functions that should not be under any *Timelock*. Hence, the developer would take a *manager role* to trigger these functions with no time delay. Also, a user can execute any *user-level functions* without time constraints.

The above-recommended design provides the concept of how to remediate this issue only. The design should be adjusted accordingly.

Reassessment

The *FWX team* adopted our suggested design to fix this issue.

No. 15	Inaccurate Calculation For Liquidation Point		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/APHCore.sol*		
Locations	*APHCore.sol L: 137 - 155*		

Detailed Issue

We found that the *isLoanLiquidable* function determines the liquidation point for a given loan inaccurately. Specifically, the function does not include the unsettled (pending) interest in the calculation (L150 in the code snippet below). In addition, the function does not take the loan's minimum interest (*loan.minInterest*) into account as well.

These make the *isLoanLiquidable* function calculate the liquidation point incorrectly (the loan's LTV value will be less than the real value).

We consider this issue high risk because this function would be typically called by liquidators. Thus, the inaccurate results from this function would lead to the loss of users' assets as well as protocol's assets.

APHCore.sol

```
137 function isLoanLiquidable(uint256 nftId, uint256 loanId) external view returns
    (bool) {
138     Loan storage loan = loans[nftId][loanId];
139     (uint256 rate, uint256 precision) = _queryRate(
140         loan.collateralTokenAddress,
141         loan.borrowTokenAddress
142     );
143     LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
144         loan.collateralTokenAddress
145     ];
146     return
147         _isLoanLTVExceedTargetLTV(
148             loan.borrowAmount,
149             loan.collateralAmount,
150             loan.interestOwed,
151             loanConfig.liquidationLTV,
152             rate,
```

```
            precision
        );
}
```

Listing 15.1 The *isLoanLiquidable* function

Recommendations

We recommend updating the *isLoanLiquidable* function to calculate an accurate liquidation point like the code snippet below.

APHCore.sol

```
function isLoanLiquidable(uint256 nftId, uint256 loanId) external view returns (bool) {
    Loan storage loan = loans[nftId][loanId];
    (uint256 rate, uint256 precision) = _queryRate(
        loan.collateralTokenAddress,
        loan.borrowTokenAddress
    );

    if (loan.collateralAmount == 0 || rate == 0) {
        return false;
    }

    LoanConfig storage loanConfig = loanConfigs[loan.borrowTokenAddress][
        loan.collateralTokenAddress
    ];

    uint64 settleTimestamp = uint64(Math.min(block.timestamp, loan.rolloverTimestamp));

    uint256 totalInterest = loan.interestOwed;
    if (settleTimestamp > loan.lastSettleTimestamp) {
        totalInterest += ((loan.owedPerDay * (settleTimestamp - loan.lastSettleTimestamp)) / 1 days);
    }
    totalInterest = Math.max(loan.minInterest, totalInterest);

    return
        _isLoanLTVExceedTargetLTV(
            loan.borrowAmount,
            loan.collateralAmount,
            totalInterest,
            loanConfig.liquidationLTV,
            rate,
            precision
```

```
        );
}
```

Listing 15.2 The improved *isLoanLiquidable* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our suggestion.

No. 16	Flash Loan-Based Price Manipulation Attack On Liquidated Loan		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 479 - 573*		

Detailed Issue

We found that the *_liquidate* function can be attacked by price manipulation using a flash loan. This issue can happen on a liquidated loan that cannot be closed a position due to an insufficiency of the loan's collateral amount.

Specifically, if the loan position cannot be closed, the execution flow of the *_liquidate* function would be forced to swap all loan's collateral amount for a borrowing token in L537 - 543 in the code snippet below.

At this point, we found that the execution of the *swapExactTokensForTokens* function does not specify a proper minimum swapped-out amount for the borrowing token (*amountOutMin* parameter). In a word, the *amountOutMin* parameter is currently set to zero (L539).

With the current setting, the *swapExactTokensForTokens* function would accept every swapped-out amount (even if the *zero* amount). This insecure setting opens room for an attacker to perform flash loan-based price manipulation attacks on the swap pools that the *Forward protocol* is using and take profit from the described insecure swaps.

As a result, this issue can lead to a massive loss of borrowing assets of all pools, affecting the stability of the *Forward protocol*.

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
```

```
{
    // (...SNIPPED...)

    if (
        _isLoanLTVExceedTargetLTV(
            loan.borrowAmount,
            loan.collateralAmount,
            Math.max(loan.interestOwed, loan.minInterest),
            loanConfig.liquidationLTV,
            numberArray[0],
            numberArray[1]
        )
    ) {
        address[] memory path_data = new address[](2);
        path_data[0] = loan.collateralTokenAddress;
        path_data[1] = loan.borrowTokenAddress;
        uint256[] memory amounts;

        numberArray[2] = IPriceFeed(priceFeedAddress).queryReturn(
            loan.collateralTokenAddress,
            loan.borrowTokenAddress,
            loan.collateralAmount
        );

        if (numberArray[2] > loan.borrowAmount + loan.interestOwed) {
            numberArray[2] = loan.borrowAmount + loan.interestOwed;
            // Normal condition, leftover collateral is exists
            amounts = IRouter(routerAddress).swapTokensForExactTokens(
                numberArray[2], //          // amountOut
                loan.collateralAmount, //   // amountInMax
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        } else {
            amounts = IRouter(routerAddress).swapExactTokensForTokens(
                loan.collateralAmount, //   // amountIn
                0, //                       // amountOutMin
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        }
        uint256 borrowTokenAmountSwap = amounts[amounts.length - 1];

        leftOverCollateral = loan.collateralAmount - amounts[0];

        (repayBorrow, repayInterest, ) = _repay(loanId, nftId,
borrowTokenAmountSwap, false);

    // (...SNIPPED...)
```

```
}
```

Listing 16.1 The *_liquidate* function that can be attacked by the price manipulation using a flash loan

Recommendations

We recommend configuring the *amountOutMin* parameter properly for the *swapExactTokensForTokens* function like L537 and L540 in the code snippet below.

The *amountOutMin* parameter would be calculated based on the following formula:

numberArray[2] * (WEI_PERCENT_UNIT - percentDiffAcceptable) / WEI_PERCENT_UNIT

Where

- **numberArray[2]** represents a maximum swappable amount for a borrowing token
- **WEI_PERCENT_UNIT** represents a constant value of 100%
- **percentDiffAcceptable** represents an acceptable slippage value in percentage **(percentDiffAcceptable < WEI_PERCENT_UNIT)**

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
487 {
        // (...SNIPPED...)

505     if (
506         _isLoanLTVExceedTargetLTV(
507             loan.borrowAmount,
508             loan.collateralAmount,
509             Math.max(loan.interestOwed, loan.minInterest),
510             loanConfig.liquidationLTV,
511             numberArray[0],
512             numberArray[1]
513         )
514     ) {
515         address[] memory path_data = new address[](2);
```

```
        path_data[0] = loan.collateralTokenAddress;
        path_data[1] = loan.borrowTokenAddress;
        uint256[] memory amounts;

        numberArray[2] = IPriceFeed(priceFeedAddress).queryReturn(
            loan.collateralTokenAddress,
            loan.borrowTokenAddress,
            loan.collateralAmount
        );

        if (numberArray[2] > loan.borrowAmount + loan.interestOwed) {
            numberArray[2] = loan.borrowAmount + loan.interestOwed;
            // Normal condition, leftover collateral is exists
            amounts = IRouter(routerAddress).swapTokensForExactTokens(
                numberArray[2], //          // amountOut
                loan.collateralAmount, //   // amountInMax
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        } else {
            uint256 amountOutMin = numberArray[2] * (WEI_PERCENT_UNIT - percentDiffAcceptable) / WEI_PERCENT_UNIT;
            amounts = IRouter(routerAddress).swapExactTokensForTokens(
                loan.collateralAmount, //   // amountIn
                amountOutMin, //            // amountOutMin
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        }
        uint256 borrowTokenAmountSwap = amounts[amounts.length - 1];

        leftOverCollateral = loan.collateralAmount - amounts[0];

        (repayBorrow, repayInterest, ) = _repay(loanId, nftId, borrowTokenAmountSwap, false);

    // (...SNIPPED...)
}
```

Listing 16.2 The improved *_liquidate* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue as per our recommendation.

No. 17	Removal Recommendation For Mock Function		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/pool/InterestVault.sol*		
Locations	*InterestVault.sol L: 51 - 54*		

Detailed Issue

We found the mock function named *approveInterestVault* (the code snippet below) that should not be put in production. This mock function allows a manager to approve unlimited *Forward* token transfers from an InterestVault contract to any arbitrary address.

InterestVault.sol

```
51 // TODO: need to make it testable
52 function approveInterestVault(address _pool) external onlyManager {
53     IERC20(forw).approve(_pool, type(uint256).max);
54 }
```

Listing 17.1 The mock function *approveInterestVault*

Recommendations

We recommend removing the mock function *approveInterestVault* from the InterestVault contract.

Reassessment

This issue was fixed by removing the *approveInterestVault* function in accordance with our recommendation.

No. 18	Reentrancy Attack to Steal All Forward Tokens From Distributor		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBaseFunc.sol* *./contracts/src/core/APHCore.sol*		
Locations	*CoreBaseFunc.sol L: 26 - 61* *APHCore.sol L: 40 - 49*		

Detailed Issue

The *settleForwInterest* function is typically called by pools to settle *Forward interest* to the pool's interest vault as shown in code snippet 18.1. The *settleForwInterest* function calls the internal function *_settleForwInterest* (L42) to calculate an amount of *Forward interest* to settle (L47).

We found that if an attacker is able to manage to deploy a mock pool contract somehow (e.g., by phishing attacks). The attacker can invoke a reentrancy attack on the *settleForwInterest* function to steal all *Forward* tokens from the *Forward distributor*.

The root cause of this issue resides in L60 in the *_settleForwInterest* function (code snippet 18.2). Specifically, the *_settleForwInterest* function makes a call (L53 - 59) to an external contract (i.e., the attacker's contract) before updating the mapping *lastSettleForw* (L60). This coding pattern enables the attacker to execute a reentrancy attack.

APHCore.sol

```
function settleForwInterest() external {
    require(poolToAsset[msg.sender] != address(0),
"APHCore/caller-is-not-pool");
    uint256 forwAmount = _settleForwInterest();
    _transferFromOut(
        forwDistributorAddress,
        IAPHPool(msg.sender).interestVaultAddress(),
        forwAddress,
        forwAmount
    );
}
```

Listing 18.1 The external *settleForwInterest* function of the *APHCore* contract

CoreBaseFunc.sol

```
function _settleForwInterest() internal returns (uint256 forwAmount) {
    if (lastSettleForw[msg.sender] != 0) {
        uint256 targetBlock = nextForwDisPerBlock[msg.sender].targetBlock;
        uint256 newForwDisPerBlock = nextForwDisPerBlock[msg.sender].amount;

        if (targetBlock != 0) {
            if (targetBlock >= block.number) {
                forwAmount =
                    (block.number - lastSettleForw[msg.sender]) *
                    forwDisPerBlock[msg.sender];
            } else {
                forwAmount =
                    ((targetBlock - lastSettleForw[msg.sender]) *
forwDisPerBlock[msg.sender]) +
                    ((block.number - targetBlock) * newForwDisPerBlock);
            }

            if (targetBlock <= block.number) {
                forwDisPerBlock[msg.sender] = newForwDisPerBlock;
                nextForwDisPerBlock[msg.sender] = NextForwDisPerBlock(0, 0);
            }
        } else {
            forwAmount =
                (block.number - lastSettleForw[msg.sender]) *
                forwDisPerBlock[msg.sender];
        }
    }

    if (forwAmount != 0) {
        IInterestVault(IAPHPool(msg.sender).interestVaultAddress()).
settleInterest(
            0,
```

```
                0,
                forwAmount
            );
        }
        lastSettleForw[msg.sender] = block.number;
}
```

Listing 18.2 The internal *_settleForwInterest* function of the *CoreBaseFunc* contract

Recommendations

We recommend updating the *_settleForwInterest* function according to the code snippet below. That is, the function would update the mapping *lastSettleForw* (L53) before making a call to an external contract (L55 - 61).

CoreBaseFunc.sol

```
26 function _settleForwInterest() internal returns (uint256 forwAmount) {
27     if (lastSettleForw[msg.sender] != 0) {
28         uint256 targetBlock = nextForwDisPerBlock[msg.sender].targetBlock;
29         uint256 newForwDisPerBlock = nextForwDisPerBlock[msg.sender].amount;
30 
31         if (targetBlock != 0) {
32             if (targetBlock >= block.number) {
33                 forwAmount =
34                     (block.number - lastSettleForw[msg.sender]) *
35                     forwDisPerBlock[msg.sender];
36             } else {
37                 forwAmount =
38                     ((targetBlock - lastSettleForw[msg.sender]) *
forwDisPerBlock[msg.sender]) +
39                     ((block.number - targetBlock) * newForwDisPerBlock);
40             }
41 
42             if (targetBlock <= block.number) {
43                 forwDisPerBlock[msg.sender] = newForwDisPerBlock;
44                 nextForwDisPerBlock[msg.sender] = NextForwDisPerBlock(0, 0);
45             }
46         } else {
47             forwAmount =
48                 (block.number - lastSettleForw[msg.sender]) *
49                 forwDisPerBlock[msg.sender];
50         }
51     }
52 
53     lastSettleForw[msg.sender] = block.number;
54 
55     if (forwAmount != 0) {
```

```
        IInterestVault(IAPHPool(msg.sender).interestVaultAddress()).
settleInterest(
            0,
            0,
            forwAmount
        );
    }
}
```

Listing 18.3 The improved *_settleForwInterest* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue in accordance with our recommendation.

No. 19	No Allowlist For Collateral Tokens		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol* *./contracts/src/pool/PoolBorrowing.sol*		
Locations	*Not specific*		

Detailed Issue

The *Forward protocol* has an allowlist for borrowing tokens in which a protocol manager has to grant and register all borrowing tokens supported. However, we found that the protocol does not control an allowlist for collateral tokens.

Since the protocol feeds the prices of tokens through the *Chainlink* protocol, only ERC-20 tokens supported by *Chainlink* can be used as collateral tokens. However, we consider that relying on the security protection mechanisms of other systems is not the best idea for smart contract security design.

Consider the case that an attacker can somehow manage to feed their token to the protocol. The attacker's managed token may be a low-liquidity or unstable token. Hence, this could open room for an attacker to exploit the *Forward protocol* by using the managed token as loan collateral.

Recommendations

We recommend adding an allowlist for collateral tokens. The allowlist not only improves the security layer of the *Forward protocol* but also increases control flexibility for a protocol manager. In other words, a manager can even allow appropriate tokens as collateral for any specific borrowing tokens.

Reassessment

The *FWX team* fixed this issue by implementing an allowlist check for collateral tokens in the *_borrow* function as presented in L219 - 222 in the code snippet below. Thus, the protocol would accept only tokens allowed for lending as collateral tokens.

CoreBorrowing.sol

```
203 // internal function
204 function _borrow(
205     uint256 loanId,
206     uint256 nftId,
207     uint256 borrowAmount,
208     address borrowTokenAddress,
209     uint256 collateralSentAmount,
210     address collateralTokenAddress,
211     uint256 newOwedPerDay,
212     uint256 interestRate
213 ) internal returns (Loan memory) {
214     require(
215         msg.sender == assetToPool[borrowTokenAddress],
216         "CoreBorrowing/permission-denied-for-borrow"
217     );
218 
219     require(
220         assetToPool[collateralTokenAddress] != address(0),
221         "CoreBorrowing/collateral-token-address-is-not-allowed"
222     );

        // (...SNIPPED...)
330 }
```

Listing 19.1 The revised *_borrow* function applying an allowlist for collateral tokens

No. 20	Misplaced Transfer Approval For Forward Distributor		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreSetting.sol*		
Locations	*CoreSetting.sol L: 24 - 29 and 75 - 96*		

Detailed Issue

The *registerNewPool* function approves a *Forward distributor* account for transferring *Forward* token (L83 in code snippet 20.1). This approval would be triggered once a protocol manager registers a new pool.

However, we found that if a manager sets a new *Forward distributor* account via the *setForwDistributorAddress* function (code snippet 20.2), the new *distributor* account would not be approved automatically. The only way for the new *distributor* account to get approval is that the manager has to invoke the *registerNewPool* function which is not a practical approach.

CoreSetting.sol

```
75  function registerNewPool(
76      address _poolAddress,
77      uint256 _amount,
78      uint256 _targetBlock
79  ) external onlyManager {
80      require(poolToAsset[_poolAddress] == address(0),
    "CoreSetting/pool-is-already-exist");
81
82      address assetAddress = IAPHPool(_poolAddress).tokenAddress();
83      IERC20(forwAddress).approve(forwDistributorAddress, type(uint256).max);
84      IERC20(assetAddress).approve(routerAddress, type(uint256).max);
85
86      poolToAsset[_poolAddress] = assetAddress;
87      assetToPool[assetAddress] = _poolAddress;
88      swapableToken[assetAddress] = true;
89      poolList.push(_poolAddress);
90
91      lastSettleForw[_poolAddress] = block.number;
92
93      _setForwDisPerBlock(_poolAddress, _amount, _targetBlock);
94
```

```
    emit RegisterNewPool(msg.sender, _poolAddress);
}
```

Listing 20.1 The *registerNewPool* function

CoreSetting.sol

```
function setForwDistributorAddress(address _address) external onlyManager {
    address oldAddress = forwDistributorAddress;
    forwDistributorAddress = _address;

    emit SetForwDistributorAddress(msg.sender, oldAddress, _address);
}
```

Listing 20.2 The *setForwDistributorAddress* function

Recommendations

We recommend moving the approval logic from the *registerNewPool* function to the *setForwDistributorAddress* function as shown in the code snippet below.

In L28, the function resets a transfer allowance from the old *distributor* account and approves the transfer to the new *distributor* account in L29. Furthermore, we also recommend using the standard *SafeERC20*'s *safeApprove* function instead of the *ERC20*'s *approve* function for better security.

CoreSetting.sol

```
24 function setForwDistributorAddress(address _address) external onlyManager {
25     address oldAddress = forwDistributorAddress;
26     forwDistributorAddress = _address;
27 
28     IERC20(forwAddress).safeApprove(oldAddress, 0);
29     IERC20(forwAddress).safeApprove(forwDistributorAddress, type(uint256).max);
30 
31     emit SetForwDistributorAddress(msg.sender, oldAddress, _address);
32 }
```

Listing 20.3 The improved *setForwDistributorAddress* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue as per our suggestion.

No. 21	Incorrect Calculation For Bounty Reward		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/pool/PoolSetting.sol* *./contracts/src/core/CoreSetting.sol* *./contracts/src/core/CoreBorrowing.sol*		
Locations	*PoolSetting.sol L: 35 - 64* *CoreSetting.sol L: 128 - 169* *CoreBorrowing.sol L: 479 - 573*		

Detailed Issue

The *bountyFeeRate* parameter can be set by a protocol manager via two external functions: *setupLoanConfig* functions of the *PoolSetting* (code snippet 21.1) and the *CoreSetting* (code snippet 21.2) contracts.

We found that both functions do not perform sanitization checks on the *bountyFeeRate* parameter. The *bountyFeeRate* parameter is used in the *_liquidate* function to calculate the *bountyReward* in L556 in code snippet 21.3.

At this point, if the *bountyFeeRate* is set to be more than the *WEI_PERCENT_UNIT* parameter, the resulting *bountyReward* would become an incorrect value and would make the transaction be reverted due to the *underflow error* in L557 while calculating the *leftOverCollateral*.

PoolSetting.sol

```
35 function setupLoanConfig(
36     address _collateralTokenAddress,
37     uint256 _safeLTV,
38     uint256 _maxLTV,
39     uint256 _liqLTV,
40     uint256 _bountyFeeRate
41 ) external onlyManager {
42     require(
43         _safeLTV < _maxLTV && _maxLTV < _liqLTV && _liqLTV < WEI_PERCENT_UNIT,
44         "PoolSetting/invalid-loan-config"
45     );
```

```

    IAPHCoreSetting(coreAddress).setupLoanConfig(
        tokenAddress,
        _collateralTokenAddress,
        _safeLTV,
        _maxLTV,
        _liqLTV,
        _bountyFeeRate
    );

    emit SetLoanConfig(
        msg.sender,
        _collateralTokenAddress,
        _safeLTV,
        _maxLTV,
        _liqLTV,
        _bountyFeeRate
    );
}
```

Listing 21.1 The *setupLoanConfig* function of the *PoolSetting* contract

CoreSetting.sol

```
function setupLoanConfig(
    address _borrowTokenAddress,
    address _collateralTokenAddress,
    uint256 _safeLTV,
    uint256 _maxLTV,
    uint256 _liquidationLTV,
    uint256 _bountyFeeRate
) external {
    require(
        poolToAsset[msg.sender] != address(0) || msg.sender == manager,
        "CoreSetting/permission-denied-for-setup-loan-config"
    );
    require(
        _borrowTokenAddress != _collateralTokenAddress &&
            assetToPool[_borrowTokenAddress] != address(0) &&
            assetToPool[_collateralTokenAddress] != address(0),
        "CoreSetting/_borrowTokenAddress-is-not-registered-yet"
    );

    LoanConfig memory configOld =
loanConfigs[_borrowTokenAddress][_collateralTokenAddress];
    LoanConfig storage config =
loanConfigs[_borrowTokenAddress][_collateralTokenAddress];
    config.borrowTokenAddress = _borrowTokenAddress;
    config.collateralTokenAddress = _collateralTokenAddress;
    config.safeLTV = _safeLTV;
```

```
    config.maxLTV = _maxLTV;
    config.liquidationLTV = _liquidationLTV;
    config.bountyFeeRate = _bountyFeeRate;

    emit SetupLoanConfig(
        msg.sender,
        _borrowTokenAddress,
        _collateralTokenAddress,
        configOld.safeLTV,
        configOld.maxLTV,
        configOld.liquidationLTV,
        configOld.bountyFeeRate,
        config.safeLTV,
        config.maxLTV,
        config.liquidationLTV,
        config.bountyFeeRate
    );
}
```

Listing 21.2 The *setupLoanConfig* function of the *CoreSetting* contract

CoreBorrowing.sol

```
function _liquidate(uint256 loanId, uint256 nftId)
    internal
    returns (
        uint256 repayBorrow,
        uint256 repayInterest,
        uint256 bountyReward,
        uint256 leftOverCollateral
    )
{

    // (...SNIPPED...)

        (repayBorrow, repayInterest, ) = _repay(loanId, nftId,
borrowTokenAmountSwap, false);

        if (loanExts[nftId][loanId].active == true) {
            // TODO (future work): handle with ciritical condition, this part
must add pool subsidisation for pool loss
            // Ciritical condition, protocol loss
            // transfer int or sth else to pool
        } else {
            bountyReward = (leftOverCollateral * loanConfig.bountyFeeRate) /
WEI_PERCENT_UNIT;
            leftOverCollateral -= bountyReward;
        }

    // (...SNIPPED...)
```

```
}
```

Listing 21.3 The *_liquidate* function of the *CoreBorrowing* contract

Recommendations

We recommend adding sanitization checks on the *bountyFeeRate* parameter, **making sure that the value would not be greater than the *WEI_PERCENT_UNIT* parameter or any appropriate value**, on both the *setupLoanConfig* functions of the *PoolSetting* and the *CoreSetting* contracts.

Reassessment

The *FWX team* fixed this issue by adding sanitization checks on the *bountyFeeRate* parameter to make sure that its value would not be greater than the *WEI_PERCENT_UNIT* parameter.

No. 22	Lack Of Sanitization Checks On Loan Config Parameters		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	./contracts/src/pool/PoolSetting.sol ./contracts/src/core/CoreSetting.sol		
Locations	PoolSetting.sol L: 35 - 64 CoreSetting.sol L: 128 - 169		

Detailed Issue

The loan config parameters can be set by a protocol manager by way of calling the *setupLoanConfig* functions of the *PoolSetting* and *CoreSetting* contracts. Nonetheless, we found some input parameters on those functions are not performed sanitization checks which can lead to incorrect calculations of the *CoreBorrowing* module such as calculating a bounty reward or loan liquidation.

The following lists input parameters left unchecked.

- ***_bountyFeeRate*** in the *setupLoanConfig* function of the *PoolSetting* contract (L40 in code snippet 22.1)
- ***_safeLTV*** in the *setupLoanConfig* function of the *CoreSetting* contract (L131 in code snippet 22.2)
- ***_maxLTV*** in the *setupLoanConfig* function of the *CoreSetting* contract (L132 in code snippet 22.2)
- ***_liquidationLTV*** in the *setupLoanConfig* function of the *CoreSetting* contract (L133 in code snippet 22.2)
- ***_bountyFeeRate*** in the *setupLoanConfig* function of the *CoreSetting* contract (L134 in code snippet 22.2)

PoolSetting.sol

```
35 function setupLoanConfig(
36     address _collateralTokenAddress,
37     uint256 _safeLTV,
38     uint256 _maxLTV,
39     uint256 _liqLTV,
40     uint256 _bountyFeeRate
41 ) external onlyManager {
42     require(
43         _safeLTV < _maxLTV && _maxLTV < _liqLTV && _liqLTV < WEI_PERCENT_UNIT,
```

```
            "PoolSetting/invalid-loan-config"
        );

        IAPHCoreSetting(coreAddress).setupLoanConfig(
            tokenAddress,
            _collateralTokenAddress,
            _safeLTV,
            _maxLTV,
            _liqLTV,
            _bountyFeeRate
        );

        emit SetLoanConfig(
            msg.sender,
            _collateralTokenAddress,
            _safeLTV,
            _maxLTV,
            _liqLTV,
            _bountyFeeRate
        );
}
```

Listing 22.1 The *setupLoanConfig* function of the *PoolSetting* contract

CoreSetting.sol

```
function setupLoanConfig(
    address _borrowTokenAddress,
    address _collateralTokenAddress,
    uint256 _safeLTV,
    uint256 _maxLTV,
    uint256 _liquidationLTV,
    uint256 _bountyFeeRate
) external {
    require(
        poolToAsset[msg.sender] != address(0) || msg.sender == manager,
        "CoreSetting/permission-denied-for-setup-loan-config"
    );
    require(
        _borrowTokenAddress != _collateralTokenAddress &&
            assetToPool[_borrowTokenAddress] != address(0) &&
            assetToPool[_collateralTokenAddress] != address(0),
        "CoreSetting/_borrowTokenAddress-is-not-registered-yet"
    );

    LoanConfig memory configOld =
loanConfigs[_borrowTokenAddress][_collateralTokenAddress];
    LoanConfig storage config =
loanConfigs[_borrowTokenAddress][_collateralTokenAddress];
    config.borrowTokenAddress = _borrowTokenAddress;
```

```
    config.collateralTokenAddress = _collateralTokenAddress;
    config.safeLTV = _safeLTV;
    config.maxLTV = _maxLTV;
    config.liquidationLTV = _liquidationLTV;
    config.bountyFeeRate = _bountyFeeRate;

    emit SetupLoanConfig(
        msg.sender,
        _borrowTokenAddress,
        _collateralTokenAddress,
        configOld.safeLTV,
        configOld.maxLTV,
        configOld.liquidationLTV,
        configOld.bountyFeeRate,
        config.safeLTV,
        config.maxLTV,
        config.liquidationLTV,
        config.bountyFeeRate
    );
}
```

Listing 22.2 The *setupLoanConfig* function of the *CoreSetting* contract

Recommendations

We recommend adding sanitization checks on all the associated input parameters on both the *setupLoanConfig* functions of the *PoolSetting* and *CoreSetting* contracts.

Be sure to validate the value of the ***_bountyFeeRate*** **parameter not to be greater than the** ***WEI_PERCENT_UNIT*** **parameter or any appropriate value** (refer to issue no. 21 for more details).

And, the relationship between LTV parameters should be according to this formula:

```
_safeLTV < _maxLTV < _liquidationLTV < WEI_PERCENT_UNIT
```

Reassessment

The *FWX team* fixed this issue by adding sanitization checks on all the associated input parameters as recommended.

No. 23	Underflow On Getting More Loan		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 224 and 228*		

Detailed Issue

We found an integer underflow on the *_borrow* function of the *CoreBorrowing* contract. The underflow occurs (in L224 and L228 in the code snippet below) when a borrower sends a transaction to borrow more on an existing loan that is overdue.

More specifically, on the overdue loan, the *loan.rolloverTimestamp* would be less than the current *block.timestamp*. Subsequently, the underflow error would occur when the function computes the expression *loan.rolloverTimestamp - block.timestamp*.

CoreBorrowing.sol

```
165 function _borrow(
166     uint256 loanId,
167     uint256 nftId,
168     uint256 borrowAmount,
169     address borrowTokenAddress,
170     uint256 collateralSentAmount,
171     address collateralTokenAddress,
172     uint256 newOwedPerDay,
173     uint256 interestRate
174 ) internal returns (Loan memory) {

        // (...SNIPPED...)

199     if (numberArray[0] == 1) {
200         loan.borrowTokenAddress = borrowTokenAddress;
201         loan.collateralTokenAddress = collateralTokenAddress;
202         loan.owedPerDay = newOwedPerDay;
203         loan.lastSettleTimestamp = uint64(block.timestamp);
204
205         loanExt.initialBorrowTokenPrice = _queryRateUSD(borrowTokenAddress);
206         loanExt.initialCollateralTokenPrice =
```

```
_queryRateUSD(collateralTokenAddress);
            loanExt.active = true;
            loanExt.startTimestamp = uint64(block.timestamp);

            poolStat.borrowInterestOwedPerDay += newOwedPerDay;
        } else {
            require(loanExt.active == true, "CoreBorrowing/loan-is-closed");

            require(
                loan.collateralTokenAddress == collateralTokenAddress,
                "CoreBorrowing/collateral-token-not-matched"
            );

            _settleBorrowInterest(loan);

            numberArray[1] = loan.owedPerDay;
            // owedPerDay = [(r1/365 * (ld-now) * p1) + (r2/365 * ld * p2) + (r2/365 * (leftover) * p1)] / ld
            loan.owedPerDay =
                ((loan.owedPerDay * (loan.rolloverTimestamp - block.timestamp)) +
                    (newOwedPerDay * loanDuration) +
                    ((interestRate *
                        loan.borrowAmount *
                        (loanDuration - ((loan.rolloverTimestamp - block.timestamp)))) /
                        (365 * WEI_PERCENT_UNIT))) /
                loanDuration;

            poolStat.borrowInterestOwedPerDay =
                poolStat.borrowInterestOwedPerDay +
                loan.owedPerDay -
                numberArray[1];
        }

        // (...SNIPPED...)

}
```

Listing 23.1 The *_borrow* function of the *CoreBorrowing* contract

Recommendations

We recommend handling (or refusing) the case when overdue loans are requested to get more loan to remediate the underflow error.

Reassessment

The *FWX team* fixed this issue by revising the *_borrow* function like L265 - 267 in the code snippet below. In the case of the overdue loan, the *_borrow* function will roll over the loan before recalculating borrowing parameters.

CoreBorrowing.sol

```
204 function _borrow(
205     uint256 loanId,
206     uint256 nftId,
207     uint256 borrowAmount,
208     address borrowTokenAddress,
209     uint256 collateralSentAmount,
210     address collateralTokenAddress,
211     uint256 newOwedPerDay,
212     uint256 interestRate
213 ) internal returns (Loan memory) {

        // (...SNIPPED...)

243     if (numberArray[0] == 1) {
244         loan.borrowTokenAddress = borrowTokenAddress;
245         loan.collateralTokenAddress = collateralTokenAddress;
246         loan.owedPerDay = newOwedPerDay;
247         loan.lastSettleTimestamp = uint64(block.timestamp);
248
249         loanExt.initialBorrowTokenPrice = _queryRateUSD(borrowTokenAddress);
250         loanExt.initialCollateralTokenPrice =
    _queryRateUSD(collateralTokenAddress);
251         loanExt.active = true;
252         loanExt.startTimestamp = uint64(block.timestamp);
253
254         poolStat.borrowInterestOwedPerDay += newOwedPerDay;
255     } else {
256         require(loanExt.active == true, "CoreBorrowing/loan-is-closed");
257
258         require(
259             loan.collateralTokenAddress == collateralTokenAddress,
260             "CoreBorrowing/collateral-token-not-matched"
261         );
262
263         _settleBorrowInterest(loan);
264
265         if (loan.rolloverTimestamp < block.timestamp) {
266             _rollover(loanId, nftId, msg.sender);
267         }
268
269         numberArray[1] = loan.owedPerDay;
270         // owedPerDay = [(r1/365 * (ld-now) * p1) + (r2/365 * ld * p2) + (r2/365
```

```
* (leftover) * p1)] / ld
        loan.owedPerDay =
            ((loan.owedPerDay * (loan.rolloverTimestamp - block.timestamp)) +
                (newOwedPerDay * loanDuration) +
                ((interestRate *
                    loan.borrowAmount *
                    (loanDuration - ((loan.rolloverTimestamp -
block.timestamp)))) /
                    (365 * WEI_PERCENT_UNIT))) /
            loanDuration;

        poolStat.borrowInterestOwedPerDay =
            poolStat.borrowInterestOwedPerDay +
            loan.owedPerDay -
            numberArray[1];
    }

    // (...SNIPPED...)

}
```

Listing 23.2 The revised _*borrow* function

No. 24	Incorrect Calculations For Loan Repayment		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 284 - 380*		

Detailed Issue

We found some incorrect calculations for loan repayment on the *_repay* function (code snippet below) of the *CoreBorrowing* contract. There are three incorrect calculation issues as follows.

1. **Underflow on calculating the *borrowPaid* (L344)**

   This issue occurs if:

   (*repayAmount* > *loan.interestOwed*) and

   (*repayAmount* < *loan.minInterest*) and

   (*loan.minInterest* > *loan.interestOwed*)

   Then:

   ***interestPaid* = max(*loan.minInterest, loan.interestOwed*)**

   = *loan.minInterest*

   Underflow occurs during calculating:

   borrowPaid* = *repayAmount* - *interestPaid

   = *repayAmount* - *loan.minInterest*

   => *underflow error* (since *repayAmount* < *loan.minInterest*)

   To understand this issue easier, let's say we have the following parameters:

   repayAmount = 500, ***loan.interestOwed*** = 300, and ***loan.minInterest*** = 600

   Thus:

   ***interestPaid* = max(*loan.minInterest, loan.interestOwed*)**

   = max(600, 300)

   = 600

   borrowPaid* = *repayAmount* - *interestPaid

   = 500 - 600 (*underflow error*)

2. **The calculated *minInterest* always returns zero (L352 - 356)**

   The result of the ***loan.minInterest* always returns zero (0)** after processing the *if - else* statement in L352 - 356.

3. **Underflow on calculating the *interestOwed* (L358)**

   This issue occurs if:

   loan.minInterest* > *loan.interestOwed

   Then:

   ***interestPaid* = max(*loan.minInterest*, *loan.interestOwed*)**

   = *loan.minInterest*

   Underflow occurs during calculating:

   loan.interestOwed* -= *interestPaid

   -= *loan.minInterest*

   => *underflow error* (since *loan.interestOwed* < *loan.minInterest*)

CoreBorrowing.sol

```
284 function _repay(
285     uint256 loanId,
286     uint256 nftId,
287     uint256 repayAmount,
288     bool isOnlyInterest
289 )
290     internal
291     returns (
292         uint256 borrowPaid,
293         uint256 interestPaid,
294         bool isLoanClosed
295     )
296 {
297     Loan storage loan = loans[nftId][loanId];
298     PoolStat storage poolStat = poolStats[assetToPool[loan.borrowTokenAddress]];
299
300     require(loanExts[nftId][loanId].active == true,
"CoreBorrowing/loan-is-closed");
301
302     _settleBorrowInterest(loan);
303
304     uint256 collateralAmountWithdraw = 0;
305     // pay only interest
306     if (isOnlyInterest || repayAmount <= loan.interestOwed) {
307         interestPaid = Math.min(repayAmount, loan.interestOwed);
308         loan.interestOwed -= interestPaid;
309         loan.interestPaid += interestPaid;
```

```

        if (loan.minInterest > interestPaid) {
            loan.minInterest -= interestPaid;
        } else {
            loan.minInterest = 0;
        }

        poolStat.totalInterestPaid += interestPaid;
    } else {
        interestPaid = Math.max(loan.minInterest, loan.interestOwed);
        if (repayAmount >= (loan.borrowAmount + interestPaid)) {
            // close loan
            poolStat.totalInterestPaid += interestPaid;
            poolStat.totalBorrowAmount -= loan.borrowAmount;
            poolStat.borrowInterestOwedPerDay -= loan.owedPerDay;

            collateralAmountWithdraw = loan.collateralAmount;

            totalCollateralHold[loan.collateralTokenAddress] -= collateralAmountWithdraw;

            borrowPaid = loan.borrowAmount;
            loan.minInterest = 0;
            loan.interestOwed = 0;
            loan.owedPerDay = 0;
            loan.borrowAmount = 0;
            loan.collateralAmount = 0;
            loan.interestPaid += interestPaid;

            isLoanClosed = true;
            loanExts[nftId][loanId].active = false;
        } else {
            // pay int and some of principal
            uint256 oldBorrowAmount = loan.borrowAmount;
            loan.interestPaid += interestPaid;
            borrowPaid = repayAmount - interestPaid;
            loan.borrowAmount -= borrowPaid;
            poolStat.borrowInterestOwedPerDay -= loan.owedPerDay;

            loan.owedPerDay = (loan.owedPerDay * loan.borrowAmount) / oldBorrowAmount;

            poolStat.borrowInterestOwedPerDay += loan.owedPerDay;

            if (loan.minInterest > loan.interestOwed) {
                loan.minInterest -= interestPaid;
            } else {
                loan.minInterest = 0;
            }

            loan.interestOwed -= interestPaid;
```

```
                poolStat.totalInterestPaid += interestPaid;
                poolStat.totalBorrowAmount -= borrowPaid;
            }
        }

IInterestVault(IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddre
ss())
            .settleInterest(
                (interestPaid * (WEI_PERCENT_UNIT - feeSpread)) / WEI_PERCENT_UNIT,
                (interestPaid * feeSpread) / WEI_PERCENT_UNIT,
                0
            );

        emit Repay(
            tx.origin,
            nftId,
            loanId,
            collateralAmountWithdraw > 0,
            borrowPaid,
            interestPaid,
            collateralAmountWithdraw
        );
}
```

Listing 24.1 The *_repay* function of the *CoreBorrowing* contract

Recommendations

We recommend revising the associated *_repay* function to correct all issues and performing the unit testing on all possible edge cases to make sure that the function would perform correctly in accordance with its functional design.

Reassessment

The *FWX team* fixed this issue by revising the *_repay* function as the code snippet below.

CoreBorrowing.sol

```
function _repay(
    uint256 loanId,
    uint256 nftId,
    uint256 repayAmount,
    bool isOnlyInterest
)
    internal
    returns (
        uint256 borrowPaid,
```

```
        uint256 interestPaid,
        bool isLoanClosed
    )
{
    Loan storage loan = loans[nftId][loanId];
    PoolStat storage poolStat = poolStats[assetToPool[loan.borrowTokenAddress]];

    require(loanExts[nftId][loanId].active == true,
"CoreBorrowing/loan-is-closed");

    _settleBorrowInterest(loan);

    uint256 collateralAmountWithdraw = 0;

    // pay only interest
    if (isOnlyInterest || repayAmount <= loan.interestOwed) {
        interestPaid = MathUpgradeable.min(repayAmount, loan.interestOwed);
        loan.interestOwed -= interestPaid;
        loan.interestPaid += interestPaid;

        if (loan.minInterest > interestPaid) {
            loan.minInterest -= interestPaid;
        } else {
            loan.minInterest = 0;
        }

        poolStat.totalInterestPaid += interestPaid;
    } else {
        interestPaid = MathUpgradeable.max(loan.minInterest, loan.interestOwed);
        if (repayAmount >= (loan.borrowAmount + interestPaid)) {
            // close loan
            poolStat.totalInterestPaid += interestPaid;
            poolStat.totalBorrowAmount -= loan.borrowAmount;
            poolStat.borrowInterestOwedPerDay -= loan.owedPerDay;

            collateralAmountWithdraw = loan.collateralAmount;

            totalCollateralHold[loan.collateralTokenAddress] -=
collateralAmountWithdraw;

            borrowPaid = loan.borrowAmount;
            loan.minInterest = 0;
            loan.interestOwed = 0;
            loan.owedPerDay = 0;
            loan.borrowAmount = 0;
            loan.collateralAmount = 0;
            loan.interestPaid += interestPaid;

            isLoanClosed = true;
            loanExts[nftId][loanId].active = false;
        } else {
```

```
            // pay int and some of principal
            uint256 oldBorrowAmount = loan.borrowAmount;

            interestPaid = MathUpgradeable.min(interestPaid, loan.interestOwed);
            loan.interestPaid += interestPaid;

            borrowPaid = MathUpgradeable.min(repayAmount - interestPaid, loan.borrowAmount);
            loan.borrowAmount -= borrowPaid;

            poolStat.borrowInterestOwedPerDay -= loan.owedPerDay;

            // set new owedPerDat
            loan.owedPerDay = (loan.owedPerDay * loan.borrowAmount) / oldBorrowAmount;
            poolStat.borrowInterestOwedPerDay += loan.owedPerDay;

            if (loan.minInterest > loan.interestOwed) {
                loan.minInterest -= interestPaid;
            } else {
                loan.minInterest = 0;
            }

            loan.interestOwed -= interestPaid;
            poolStat.totalInterestPaid += interestPaid;
            poolStat.totalBorrowAmount -= borrowPaid;
        }
    }

IInterestVault(IAPHPool(assetToPool[loan.borrowTokenAddress]).interestVaultAddress())
        .settleInterest(
            (interestPaid * (WEI_PERCENT_UNIT - feeSpread)) / WEI_PERCENT_UNIT,
            (interestPaid * feeSpread) / WEI_PERCENT_UNIT,
            0
        );

    emit Repay(
        msg.sender,
        nftId,
        loanId,
        collateralAmountWithdraw > 0,
        borrowPaid,
        interestPaid,
        collateralAmountWithdraw
    );
}
```

Listing 24.2 The revised _*repay* function

No. 25	Unchecking Price Feeding System's Pause		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/utils/PriceFeed.sol*		
Locations	*PriceFeed.sol L: 148 - 150*		

Detailed Issue

The *queryRateUSD* function (code snippet 25.1) returns the token rate in *USD* without checking the price feeding system's pause status (the state variable *globalPricingPaused*). Therefore, the function would operate normally even if a protocol manager pauses the price feeding system.

PriceFeed.sol

```
148 function queryRateUSD(address token) external view returns (uint256 rate) {
149     rate = _queryRateUSD(token);
150 }
```

Listing 25.1 The *queryRateUSD* function that does not check the price feeding system's pause status

Recommendations

We recommend improving the *queryRateUSD* function like the below code snippet by checking the state variable *globalPricingPaused* (L149).

PriceFeed.sol

```
148 function queryRateUSD(address token) external view returns (uint256 rate) {
149     require(!globalPricingPaused, "PriceFeed/pricing-is-paused");
150     rate = _queryRateUSD(token);
151 }
```

Listing 25.2 The improved *queryRateUSD* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue by checking the price feeding system's pause status as suggested.

No. 26	Inaccurate Interest Calculation For Liquidated Loan		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 479 - 573*		

Detailed Issue

The *_liquidate* function would liquidate a loan in the normal condition (L527 - 535 in the code snippet below) if the maximum swappable amount is more than the sum of the loan's borrowing amount and the loan's borrowing interest (***numberArray[2] > loan.borrowAmount + loan.interestOwed***) in L526.

However, we found that the *_liquidate* function does not cover the case that the loan's minimum interest (***loan.minInterest***) is more than the loan's borrowing interest (***loan.interestOwed***). In that case, the loan's minimum interest (***loan.minInterest***) should be used in the condition check instead.

If not, the calculation of the borrowing token amount used for repaying the liquidated loan would be less than the expected amount.

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
487 {
        // (...SNIPPED...)

520         numberArray[2] = IPriceFeed(priceFeedAddress).queryReturn(
521             loan.collateralTokenAddress,
522             loan.borrowTokenAddress,
523             loan.collateralAmount
524         );
525
```

```
526         if (numberArray[2] > loan.borrowAmount + loan.interestOwed) {
527             numberArray[2] = loan.borrowAmount + loan.interestOwed;
528             // Normal condition, leftover collateral is exists
529             amounts = IRouter(routerAddress).swapTokensForExactTokens(
530                 numberArray[2], //          // amountOut
531                 loan.collateralAmount, //   // amountInMax
532                 path_data,
533                 address(this),
534                 1 hours + block.timestamp
535             );
536         } else {
537             amounts = IRouter(routerAddress).swapExactTokensForTokens(
538                 loan.collateralAmount, //   // amountIn
539                 0, //                       // amountOutMin
540                 path_data,
541                 address(this),
542                 1 hours + block.timestamp
543             );
544         }

        // (...SNIPPED...)
573 }
```

Listing 26.1 The *_liquidate* function

Recommendations

We recommend updating the *_liquidate* function like L526 and L527 in the code snippet below to get the *accurate loan's interest*.

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
487 {
        // (...SNIPPED...)

520         numberArray[2] = IPriceFeed(priceFeedAddress).queryReturn(
521             loan.collateralTokenAddress,
522             loan.borrowTokenAddress,
523             loan.collateralAmount
524         );
```

```

        if (numberArray[2] > loan.borrowAmount + Math.max(loan.interestOwed, loan.minInterest)) {
            numberArray[2] = loan.borrowAmount + Math.max(loan.interestOwed, loan.minInterest);
            // Normal condition, leftover collateral is exists
            amounts = IRouter(routerAddress).swapTokensForExactTokens(
                numberArray[2], //          // amountOut
                loan.collateralAmount, //   // amountInMax
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        } else {
            amounts = IRouter(routerAddress).swapExactTokensForTokens(
                loan.collateralAmount, //   // amountIn
                0, //                       // amountOutMin
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        }

    // (...SNIPPED...)
}
```

Listing 26.2 The improved *_liquidate* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our suggestion.

No. 27	Potential Loss Of Collateral Asset For Loan Borrower		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 479 - 573*		

Detailed Issue

In L549 in the code snippet below, the *_liquidate* function invokes the *_repay* function by passing the *borrowTokenAmountSwap* as an argument. The *borrowTokenAmountSwap* is a maximum repayment amount used to calculate the returned parameters *repayBorrow* and *repayInterest*.

We found that if the maximum repayment amount (*borrowTokenAmountSwap*) is more than the loan's total debt (i.e., borrowed asset + interest), the sum of the calculated *repayBorrow* and *repayInterest* would be less than the maximum repayment amount (*borrowTokenAmountSwap*).

In other words, there will be some borrowing tokens left unused and this unused amount will be locked in the *APHCore* contract, resulting in the loss of some part of a collateral asset for the loan borrower.

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
487 {
        // (...SNIPPED...)

515         address[] memory path_data = new address[](2);
516         path_data[0] = loan.collateralTokenAddress;
517         path_data[1] = loan.borrowTokenAddress;
518         uint256[] memory amounts;
519
520         numberArray[2] = IPriceFeed(priceFeedAddress).queryReturn(
```

```
            loan.collateralTokenAddress,
            loan.borrowTokenAddress,
            loan.collateralAmount
        );

        if (numberArray[2] > loan.borrowAmount + loan.interestOwed) {
            numberArray[2] = loan.borrowAmount + loan.interestOwed;
            // Normal condition, leftover collateral is exists
            amounts = IRouter(routerAddress).swapTokensForExactTokens(
                numberArray[2], //          // amountOut
                loan.collateralAmount, //   // amountInMax
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        } else {
            amounts = IRouter(routerAddress).swapExactTokensForTokens(
                loan.collateralAmount, //   // amountIn
                0, //                       // amountOutMin
                path_data,
                address(this),
                1 hours + block.timestamp
            );
        }
        uint256 borrowTokenAmountSwap = amounts[amounts.length - 1];

        leftOverCollateral = loan.collateralAmount - amounts[0];

        (repayBorrow, repayInterest, ) = _repay(loanId, nftId,
borrowTokenAmountSwap, false);

        if (loanExts[nftId][loanId].active == true) {
            // TODO (future work): handle with ciritical condition, this part
must add pool subsidisation for pool loss
            // Ciritical condition, protocol loss
            // transfer int or sth else to pool
        } else {
            bountyReward = (leftOverCollateral * loanConfig.bountyFeeRate) /
WEI_PERCENT_UNIT;
            leftOverCollateral -= bountyReward;
        }

    // (...SNIPPED...)
}
```

Listing 27.1 The _*liquidate* function

Recommendations

We recommend updating the *_liquidate* function like the code snippet below. The function would swap the leftover borrowing token back to the collateral token (L553 - 566). Then, the function would merge the swapped collateral token with the *leftOverCollateral* variable (L569).

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
487 {
        // (...SNIPPED...)

515         address[] memory path_data = new address[](2);
516         path_data[0] = loan.collateralTokenAddress;
517         path_data[1] = loan.borrowTokenAddress;
518         uint256[] memory amounts;
519
520         numberArray[2] = IPriceFeed(priceFeedAddress).queryReturn(
521             loan.collateralTokenAddress,
522             loan.borrowTokenAddress,
523             loan.collateralAmount
524         );
525
526         if (numberArray[2] > loan.borrowAmount + loan.interestOwed) {
527             numberArray[2] = loan.borrowAmount + loan.interestOwed;
528             // Normal condition, leftover collateral is exists
529             amounts = IRouter(routerAddress).swapTokensForExactTokens(
530                 numberArray[2], //          // amountOut
531                 loan.collateralAmount, //   // amountInMax
532                 path_data,
533                 address(this),
534                 1 hours + block.timestamp
535             );
536         } else {
537             amounts = IRouter(routerAddress).swapExactTokensForTokens(
538                 loan.collateralAmount, //   // amountIn
539                 0, //                       // amountOutMin
540                 path_data,
541                 address(this),
542                 1 hours + block.timestamp
543             );
544         }
545         uint256 borrowTokenAmountSwap = amounts[amounts.length - 1];
```

```

        leftOverCollateral = loan.collateralAmount - amounts[0];

        (repayBorrow, repayInterest, ) = _repay(loanId, nftId, borrowTokenAmountSwap, false);

        uint256 leftoverBorrowToken = borrowTokenAmountSwap - (repayBorrow + repayInterest);
        if (leftoverBorrowToken > 0) {
            // Swap the leftover borrowing token back to the collateral
            path_data[0] = loan.borrowTokenAddress;
            path_data[1] = loan.collateralTokenAddress;
            delete amounts;

            amounts = IRouter(routerAddress).swapExactTokensForTokens(
                leftoverBorrowToken,  // amountIn
                0,                    // amountOutMin
                path_data,
                address(this),
                1 hours + block.timestamp
            );

            uint256 collateralAmountSwap = amounts[amounts.length - 1];

            // Merge the swapped collateral with the leftOverCollateral
            leftOverCollateral += collateralAmountSwap;
        }

        if (loanExts[nftId][loanId].active == true) {
            // TODO (future work): handle with ciritical condition, this part must add pool subsidisation for pool loss
            // Ciritical condition, protocol loss
            // transfer int or sth else to pool
        } else {
            bountyReward = (leftOverCollateral * loanConfig.bountyFeeRate) / WEI_PERCENT_UNIT;
            leftOverCollateral -= bountyReward;
        }

    // (...SNIPPED...)
}
```

Listing 27.2 The improved *_liquidate* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed by the *FWX team* according to our suggestion.

No. 28	Potential Lock Of Ethers		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/pool/PoolLending.sol* *./contracts/src/pool/PoolBorrowing.sol* *./contracts/src/core/CoreBorrowing.sol*		
Locations	*PoolLending.sol L: 43 - 59* *PoolBorrowing.sol L: 16 - 37* *CoreBorrowing.sol L: 46 - 87 and 94 - 117*		

Detailed Issue

The following are *payable* functions that accept (native) *Ethers* sent from a user.

- ***deposit* function** (*L43 - 59 in PoolLending.sol*)
- ***borrow* function** (*L16 - 37 in PoolBorrowing.sol*)
- ***repay* function** (*L46 - 87 in CoreBorrowing.sol*)
- ***adjustCollateral* function** (*L94 - 117 in CoreBorrowing.sol*)

We found that the *Ethers* sent from a user to the above functions can be locked forever in the *APHPool* or *APHCore* contract if the interacting pool does not support Ether.

Code snippet 28.1 presents the *deposit*, one of the affected functions. The function allows a user to deposit a supported ERC-20 token to a single lending pool. For example, we can deposit a USDT token to the USDT lending pool. In this case, the USDT lending pool would not support depositing any other tokens, including Ether.

However, if a user mistakenly sends the (native) *Ethers* to the USDT lending pool, the pool's *deposit* function does not have a mechanism to reject the request. As a result, the *deposited Ethers* will be locked in the pool forever.

PoolLending.sol

```
43 function deposit(uint256 nftId, uint256 depositAmount)
44     external
45     payable
46     checkRank(nftId)
47     nonReentrant
48     whenFuncNotPaused(msg.sig)
49     settleForwInterest
50     returns (
51         uint256 mintedP,
52         uint256 mintedItp,
53         uint256 mintedIfp
54     )
55 {
56     nftId = _getUsableToken(nftId);
57     _transferFromIn(tx.origin, address(this), tokenAddress, depositAmount);
58     (mintedP, mintedItp, mintedIfp) = _deposit(tx.origin, nftId, depositAmount);
59 }
```

Listing 28.1 The *deposit*, one of the affected functions

Recommendations

We recommend adding the *Ether rejection mechanism* to the affected functions as follows.

- **L56 - 58 in code snippet 28.2 for the *deposit* function**
- **L23 - 25 in code snippet 28.3 for the *borrow* function**
- **L58 - 60 in code snippet 28.4 for the *repay* function**
- **L100 - 105 in code snippet 28.5 for the *adjustCollateral* function**

The *rejection mechanism* would accept *Ethers* sent from a user only if the pool supports it.

PoolLending.sol

```
43 function deposit(uint256 nftId, uint256 depositAmount)
44     external
45     payable
46     checkRank(nftId)
47     nonReentrant
48     whenFuncNotPaused(msg.sig)
49     settleForwInterest
50     returns (
51         uint256 mintedP,
52         uint256 mintedItp,
53         uint256 mintedIfp
54     )
55 {
```

```
    if (tokenAddress != wethAddress && msg.value != 0) {
        revert("PoolLending/no-support-transferring-ether-in");
    }

    nftId = _getUsableToken(nftId);
    _transferFromIn(tx.origin, address(this), tokenAddress, depositAmount);
    (mintedP, mintedItp, mintedIfp) = _deposit(tx.origin, nftId, depositAmount);
}
```

Listing 28.2 The improved *deposit* function

PoolBorrowing.sol

```
function borrow(
    uint256 loanId,
    uint256 nftId,
    uint256 borrowAmount,
    uint256 collateralSentAmount,
    address collateralTokenAddress
) external payable nonReentrant whenFuncNotPaused(msg.sig) returns (CoreBase.Loan memory) {
    if (collateralTokenAddress != wethAddress && msg.value != 0) {
        revert("PoolBorrowing/no-support-transferring-ether-in");
    }

    nftId = _getUsableToken(nftId);

    if (collateralSentAmount != 0) {
        _transferFromIn(tx.origin, coreAddress, collateralTokenAddress, collateralSentAmount);
    }
    CoreBase.Loan memory loan = _borrow(
        loanId,
        nftId,
        borrowAmount,
        collateralSentAmount,
        collateralTokenAddress
    );
    _transferOut(tx.origin, tokenAddress, borrowAmount);
    return loan;
}
```

Listing 28.3 The improved *borrow* function

CoreBorrowing.sol

```
function repay(
    uint256 loanId,
    uint256 nftId,
    uint256 repayAmount,
    bool isOnlyInterest
)
    external
    payable
    whenFuncNotPaused(msg.sig)
    nonReentrant
    returns (uint256 borrowPaid, uint256 interestPaid)
{
    if (loan.borrowTokenAddress != wethAddress && msg.value != 0) {
        revert("CoreBorrowing/no-support-transferring-ether-in");
    }

    // (...SNIPPED...)
}
```

Listing 28.4 The improved *repay* function

CoreBorrowing.sol

```
function adjustCollateral(
    uint256 loanId,
    uint256 nftId,
    uint256 collateralAdjustAmount,
    bool isAdd
) external payable whenFuncNotPaused(msg.sig) nonReentrant returns (Loan memory)
{
    if (
        (loan.collateralTokenAddress != wethAddress || !isAdd)
        && msg.value != 0
    ) {
        revert("CoreBorrowing/no-support-transferring-ether-in");
    }

    nftId = _getUsableToken(nftId);
    Loan storage loan = loans[nftId][loanId];

    Loan memory loanData = _adjustCollateral(loanId, nftId,
collateralAdjustAmount, isAdd);
    if (isAdd) {
        // add colla to core
        _transferFromIn(
            tx.origin,
            address(this),
```

```
            loan.collateralTokenAddress,
            collateralAdjustAmount
        );
    } else {
        // withdraw colla to user
        _transferOut(tx.origin, loan.collateralTokenAddress,
collateralAdjustAmount);
    }
    return loanData;
}
```

Listing 28.5 The improved *adjustCollateral* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue by adding the *Ether rejection mechanism* according to our suggestion.

No. 29	Incorrectly Updating Membership NFT Rank		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/stakepool/StakePool.sol*		
Locations	*StakePool.sol L: 259 - 270*		

Detailed Issue

The *_updateNFTRank* function updates a Membership NFT rank for a user (*Forward* staker or unstaker) as shown in the code snippet below. We found some flaws in the function implementation as follows.

1. The *_updateNFTRank* function gets a user's rank by **passing the *msg.sender* to the *getRank* function** (L261) which is incorrect because **the *getRank* function requires the *staking pool address***, not a user address.

2. The *_updateNFTRank* function **does not check the first rank** (L263). Therefore, a staker who stakes *Forward* tokens in the first tier would not get a ranking update.

3. The *_updateNFTRank* function **would update the user's rank even if the rank is unchanged** (L265 - 266).

StakePool.sol

```
259 function _updateNFTRank(uint256 nftId) internal returns (uint8 currentRank) {
260     uint256 stakeBalance = stakeInfos[nftId].stakeBalance;
261     currentRank = IMembership(membershipAddress).getRank(msg.sender, nftId);
262 
263     for (uint8 i = rankLen - 1; i > 0; i--) {
264         if (stakeBalance >= rankInfos[i].minimumStakeAmount) {
265             currentRank = i;
266             IMembership(membershipAddress).updateRank(nftId, currentRank);
267             return currentRank;
268         }
269     }
270 }
```

Listing 29.1 The *_updateNFTRank* function

Recommendations

We recommend updating the *_updateNFTRank* function as the code snippet below.

StakePool.sol

```
function _updateNFTRank(uint256 nftId) internal returns (uint8 currentRank) {
    uint256 stakeBalance = stakeInfos[nftId].stakeBalance;
    currentRank = IMembership(membershipAddress).getRank(address(this), nftId);

    for (uint8 i = rankLen - 1; i >= 0; i--) {
        if (stakeBalance >= rankInfos[i].minimumStakeAmount) {
            if (currentRank != i) {
                currentRank = i;
                IMembership(membershipAddress).updateRank(nftId, currentRank);
            }
            return currentRank;
        }
    }
}
```

Listing 29.2 The improved *_updateNFTRank* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed according to the suggested code.

No. 30	Possibly Incorrect Calculation For Lending Forward Interest		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	./contracts/src/pool/APHPool.sol ./contracts/src/pool/PoolBaseFunc.sol		
Locations	APHPool.sol L: 12 - 39 PoolBaseFunc.sol: L: 77 - 100		

Detailed Issue

The *APHPool* contract defines the state variable *BLOCK_TIME* with a *hardcoded value* (*3*) in L32 in the *initialize* function as shown in code snippet 30.1. The *BLOCK_TIME* is used to calculate the lending *Forward* interest in the *_getNextLendingForwInterest* function (L95 in code snippet 30.2).

Since the *BLOCK_TIME* is a hardcoded value, this value might not represent the (real) block time of the blockchain network that the contract would be deployed on, affecting the incorrect calculation for the lending *Forward* interest.

APHPool.sol

```
12 function initialize(
13     address _tokenAddress,
14     address _coreAddress,
15     address _membershipAddress
16 ) external virtual initializer {
       // (...SNIPPED...)

32     BLOCK_TIME = 3;

       // (...SNIPPED...)
39 }
```

Listing 30.1 The *initialize* function

PoolBaseFunc.sol

```
function _getNextLendingForwInterest(uint256 newDepositAmount)
    internal
    view
    returns (uint256 interestRate)
{
    (uint256 rate, uint256 precision) =
IPriceFeed(IAPHCore(coreAddress).priceFeedAddress())
        .queryRate(tokenAddress, forwAddress);

    uint256 ifpPrice = _getInterestForwPrice();

    uint256 newIfpTokenSupply = ifpTokenTotalSupply +
        ((newDepositAmount * WEI_UNIT) / ifpPrice);

    if (newIfpTokenSupply == 0) {
        interestRate = 0;
    } else {
        interestRate =
            (IAPHCore(coreAddress).forwDisPerBlock(address(this)) *
                (365 days / BLOCK_TIME) *
                rate *
                WEI_UNIT) /
            (newIfpTokenSupply * precision);
    }
}
```

Listing 30.2 The *_getNextLendingForwInterest* function

Recommendations

We recommend updating the *initialize* function to configure the *BLOCK_TIME* with an inputted parameter (L16) during a contract initialization process like L33 - 34 in the code snippet below.

APHPool.sol

```
12 function initialize(
13     address _tokenAddress,
14     address _coreAddress,
15     address _membershipAddress,
16     uint256 _blockTime
17 ) external virtual initializer {
       // (...SNIPPED...)

33     require(_blockTime != 0, "_blockTime cannot be zero");
34     BLOCK_TIME = _blockTime;
```

```
    // (...SNIPPED...)
}
```

Listing 30.3 The improved *initialize* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed as per the recommended code.

No. 31	Lack Of Stale Price Detection Mechanism		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/utils/PriceFeed.sol*		
Locations	*PriceFeed.sol L: 141 - 146*		

Detailed Issue

The *_queryRateUSD* function queries for a token price from the deprecated *Chainlink*'s *latestAnswer* function (L144 in the code snippet below). Even though the current implementation of the *_queryRateUSD* function is performing correctly, the *latestAnswer* function cannot report how long the price has previously been updated by an oracle network. In other words, we cannot detect the stale price using the *latestAnswer* function.

When the protocol utilizes the stale price, as a result, the protocol's assets and users' assets can be at risk unexpectedly.

PriceFeed.sol

```
141 function _queryRateUSD(address token) internal view returns (uint256 rate) {
142     require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
143     AggregatorV2V3Interface _Feed = AggregatorV2V3Interface(pricesFeeds[token]);
144     rate = uint256(_Feed.latestAnswer());
145     require(rate != 0, "PriceFeed/price-error");
146 }
```

Listing 31.1 The *_queryRateUSD* function that
utilizes the deprecated *Chainlink*'s function, *latestAnswer*

Recommendations

We recommend employing the recommended *Chainlink*'s *latestRoundData* function as shown in L144 in the code snippet below. With the *latestRoundData* function, we can implement the stale price detection mechanism (L146 - 149), enhancing the reliability of the price data consumed by the protocol.

PriceFeed.sol

```
141 function _queryRateUSD(address token) internal view returns (uint256 rate) {
142     require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
143     AggregatorV2V3Interface _Feed = AggregatorV2V3Interface(pricesFeeds[token]);
144     (, int256 answer, , uint256 updatedAt, ) = _Feed.latestRoundData();
145     rate = uint256(answer);
146     require(
147         block.timestamp - updatedAt < stalePeriod,
148         "PriceFeed/price-is-stale"
149     );
150 }
```

Listing 31.2 The improved *_queryRateUSD* function that employs the recommended *Chainlink*'s function, *latestRoundData*

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue by adopting our recommended code.

No. 32	Usage Of Unsafe Functions		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/pool/InterestVault.sol* *./contracts/src/utils/AssetHandler.sol* *./contracts/src/utils/Vault.sol* *./contracts/src/nft/Membership.sol*		
Locations	*InterestVault.sol L: 53, 128, 129, and 161* *AssetHandler.sol L: 28, 30, 44, 47, 60, and 63* *Vault.sol L: 22 and 26* *Membership.sol L: 107*		

Detailed Issue

We found some usage of unsafe functions including:

- **Unsafe *ERC20*'s *transfer* function**
  - In *_withdrawActualProfit* function (*L161 in InterestVault.sol*)
  - In *_transferFromIn* function (*L28 in AssetHandler.sol*)
  - In *_transferOut* function (*L60 and L63 in AssetHandler.sol*)

- **Unsafe *ERC20*'s *transferFrom* function**
  - In *_transferFromIn* function (*L30 in AssetHandler.sol*)
  - In *_transferFromOut* function (*L44 and L47 in AssetHandler.sol*)

- **Unsafe *ERC20*'s *approve* function**
  - In *approveInterestVault* function (*L53 in InterestVault.sol*)
  - In *_ownerApprove* function (*L128 - 129 in InterestVault.sol*)
  - In *_ownerApprove* function (*L22 in Vault.sol*)
  - In *approveInterestVault* function (*L26 in Vault.sol*)

- **Unsafe *ERC721*'s *_mint* function**
  - In *mint* function (*L107 in Membership.sol*)

The use of above unsafe functions could lead to unexpected token transfer, approval, or minting errors.

Recommendations

We recommend applying the safer functions as follows.

- *ERC20*'s *transfer* function -> ***SafeERC20*'s *safeTransfer* function**
- *ERC20*'s *transferFrom* function -> ***SafeERC20*'s *safeTransferFrom* function**
- *ERC20*'s *approve* function -> ***SafeERC20*'s *safeApprove* function**
- *ERC721*'s *_mint* function -> ***ERC721*'s *_safeMint* function**

Reassessment

The *FWX team* fixed this issue by applying the recommended safer functions.

No. 33	Liquidator May Receive Zero Bounty Reward		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Acknowledged
Associated Files	*./contracts/src/core/CoreBorrowing.sol*		
Locations	*CoreBorrowing.sol L: 479 - 573*		

Detailed Issue

We found that the *_liquidate* function does not handle the case when a liquidated loan cannot be closed a position as shown in the below code snippet in L551 - 555. This affects a bounty reward for a liquidator to be zero (0).

CoreBorrowing.sol

```
479 function _liquidate(uint256 loanId, uint256 nftId)
480     internal
481     returns (
482         uint256 repayBorrow,
483         uint256 repayInterest,
484         uint256 bountyReward,
485         uint256 leftOverCollateral
486     )
487 {
        // (...SNIPPED...)

545         uint256 borrowTokenAmountSwap = amounts[amounts.length - 1];
546
547         leftOverCollateral = loan.collateralAmount - amounts[0];
548
549         (repayBorrow, repayInterest, ) = _repay(loanId, nftId,
borrowTokenAmountSwap, false);
550
551         if (loanExts[nftId][loanId].active == true) {
552             // TODO (future work): handle with ciritical condition, this part
must add pool subsidisation for pool loss
553             // Ciritical condition, protocol loss
554             // transfer int or sth else to pool
555         } else {
556             bountyReward = (leftOverCollateral * loanConfig.bountyFeeRate) /
WEI_PERCENT_UNIT;
```

```
            leftOverCollateral -= bountyReward;
        }

    // (...SNIPPED...)
}
```

Listing 33.1 The *_liquidate* function does not handle the case when a liquidated loan cannot be closed a position

Recommendations

We recommend updating the *_liquidate* function to calculate a liquidator's bounty reward for the associated case.

Reassessment

The *FWX team* confirmed that in case the liquidated loan cannot be closed the position, the liquidator would receive no bounty reward according to the protocol design.

No. 34	Inaccurate Calculation For Current LTV		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/APHCore.sol*		
Locations	*APHCore.sol L: 107 - 123*		

Detailed Issue

We found some nuance that can make the *getLoanCurrentLTV* function (the code snippet below) inaccurately calculate a current LTV (Loan-To-Value) for a given loan.

This nuance happens when the loan's minimum interest (*loan.minInterest*) is more than the loan's settled interest (*loan.interestOwed*) but the unsettled interest is more than the loan's minimum interest (*loan.minInterest*). This leads to an inaccurate LTV whose value is more than the real value.

APHCore.sol

```
107 function getLoanCurrentLTV(uint256 loanId, uint256 nftId) external view returns
    (uint256 ltv) {
108     Loan memory loan = loans[nftId][loanId];
109     (uint256 rate, uint256 precision) = IPriceFeed(priceFeedAddress).queryRate(
110         loan.collateralTokenAddress,
111         loan.borrowTokenAddress
112     );
113     if (loan.collateralAmount == 0 || rate == 0) {
114         return 0;
115     }
116     ltv = (loan.borrowAmount + Math.max(loan.minInterest, loan.interestOwed));
117     ltv =
118         ltv +
119         ((loan.owedPerDay * (block.timestamp -
    uint256(loan.lastSettleTimestamp)) / 1 days);
120     ltv = (ltv * WEI_PERCENT_UNIT * precision) / (loan.collateralAmount * rate);
121
122     return ltv;
123 }
```

Listing 34.1 The *getLoanCurrentLTV* function

Recommendations

We recommend updating the *getLoanCurrentLTV* function to calculate an accurate LTV like the code snippet below.

APHCore.sol

```
function getLoanCurrentLTV(uint256 loanId, uint256 nftId) external view returns (uint256 ltv) {
    Loan memory loan = loans[nftId][loanId];
    (uint256 rate, uint256 precision) = IPriceFeed(priceFeedAddress).queryRate(
        loan.collateralTokenAddress,
        loan.borrowTokenAddress
    );
    if (loan.collateralAmount == 0 || rate == 0) {
        return 0;
    }

    uint256 totalInterest = loan.interestOwed +
        ((loan.owedPerDay * (block.timestamp - uint256(loan.lastSettleTimestamp))) / 1 days);

    totalInterest = Math.max(loan.minInterest, totalInterest);

    ltv = loan.borrowAmount + totalInterest;
    ltv = (ltv * WEI_PERCENT_UNIT * precision) / (loan.collateralAmount * rate);

    return ltv;
}
```

Listing 34.2 The improved *getLoanCurrentLTV* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our suggestion.

No. 35	Improperly Getting Membership NFT Rank		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/stakepool/StakePool.sol* *./contracts/src/nft/Membership.sol*		
Locations	*StakePool.sol L: 155 - 157* *Membership.sol L: 89 - 91*		

Detailed Issue

The *getMaxLTVBonus* function queries for a max LTV bonus of the specified *nftId*. The function calls the *getRank* function to get an NFT rank as presented in L156 in code snippet 35.1. The called *getRank* function retrieves a rank from the current (newest) staking pool as shown in L90 in code snippet 35.2.

In an event of changing a staking pool, we found that the ranking results retrieved from the *getRank* function will point to the new staking pool. This can affect getting ranks of all stakers who stake *Forward* tokens on the old staking pool.

We also consider that using the implicit retrieving of a user's rank from the newest pool by default like this can lead to mistakes when maintaining the code in the future.

StakePool.sol

```
155 function getMaxLTVBonus(uint256 nftId) external view returns (uint256) {
156     return rankInfos[IMembership(membershipAddress).getRank(nftId)].maxLTVBonus;
157 }
```

Listing 35.1 The *getMaxLTVBonus* function of the *StakePool* contract

Membership.sol

```
89 function getRank(uint256 tokenId) external view returns (uint8) {
90     return _poolMembershipRanks[currentPool][tokenId];
91 }
```

Listing 35.2 The *getRank* function of the *Membership* contract

Recommendations

We recommend updating the *getMaxLTVBonus* function like the code snippet below. Another overloaded *getRank* function is called instead and we must pass the *staking pool address* as the first argument (L156).

StakePool.sol

```
155 function getMaxLTVBonus(uint256 nftId) external view returns (uint256) {
156     return rankInfos[IMembership(membershipAddress).getRank(address(this), nftId)].maxLTVBonus;
157 }
```

Listing 35.3 The improved *getMaxLTVBonus* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was fixed in accordance with our recommendation.

No. 36	Spamming On Minting Membership NFTs		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/nft/Membership.sol*		
Locations	*Membership.sol L: 105 - 111*		

Detailed Issue

We found that the *mint* function allows any caller to mint a *Membership NFT token* to any "***to***" address as presented in the code snippet below. The *mint* function opens room for an attacker to make *spam NFT tokens* to a specific address.

Membership.sol

```
105 function mint(address to) external whenNotPaused returns (uint256) {
106     uint256 tokenId = _tokenIdTracker.current();
107     _mint(to, tokenId);
108     _setFirstOwnedDefaultMembership(to, tokenId);
109     _tokenIdTracker.increment();
110     return tokenId;
111 }
```

Listing 36.1 The *mint* function

Recommendations

We recommend updating the *mint* function as the below code snippet. We add the check (L106 - 109) to allow only an EOA (Externally Owned Account) user to mint the NFT and only the function caller is able to mint NFT tokens to itself (L112 and L113).

Membership.sol

```
105 function mint() external whenNotPaused returns (uint256) {
106     require(
107         msg.sender == tx.origin,
108         "Membership/do-not-support-smart-contract"
109     );
110
111     uint256 tokenId = _tokenIdTracker.current();
112     _mint(msg.sender, tokenId);
113     _setFirstOwnedDefaultMembership(msg.sender, tokenId);
114     _tokenIdTracker.increment();
115     return tokenId;
116 }
```

Listing 36.2 The improved *mint* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* fixed this issue according to our recommendation.

No. 37	Rejection On Getting Active Loans		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/APHCore.sol*		
Locations	*APHCore.sol L: 80 - 92*		

Detailed Issue

The *getActiveLoans* is a view function returning all active loans of the specified *nftId* as presented in the code snippet below. Since the number of loans for a specific *nftId* can grow over time, the *getActiveLoans* function can confront a denial-of-service issue if the number of loans is too large.

The root cause of this issue is that the *getActiveLoans* function iterates over all loans belonging to the specified *nftId* (L85) which might take too long for querying on the EVM node, leading to the rejection of the querying request. Another criterion for the EVM node to reject a query request is the upper-bound gas limit on the block. Even if the querying request would not have to pay gas but the EVM node still counts out the gas being used.

APHCore.sol

```
80 function getActiveLoans(uint256 nftId) external view returns (Loan[] memory) {
81     uint256 loanIndex = currentLoanIndex[nftId];
82     Loan[] memory activeLoans = new Loan[](loanIndex);
83 
84     uint256 count = 0;
85     for (uint256 i = 1; i <= loanIndex; i++) {
86         if (loanExts[nftId][i].active) {
87             activeLoans[count] = loans[nftId][i];
88             count++;
89         }
90     }
91     return activeLoans;
92 }
```

Listing 37.1 The *getActiveLoans* function

Furthermore, we also found that the *getActiveLoans* function would return *trailing-zero array elements* if some loans are *inactive* since the function allocates the returned array as per *the number of all loans* (L82), *not only active loans*.

Recommendations

We recommend re-implementing the *getActiveLoans* function to address the *denial-of-service issue* as well as the *trailing-zero array elements*.

One possible solution is to apply pagination for data querying, in which the large querying data are divided into smaller discrete pages.

The code snippet below presents an idea of the pagination version of the *getActiveLoans* function which addresses both the *denial-of-service* and the *trailing-zero array elements* issues.

APHCore.sol

```
80  function getActiveLoans(
81      uint256 nftId,
82      uint256 cursor,
83      uint256 resultsPerPage
84  )
85      external
86      view
87      returns (Loan[] memory activeLoans, uint256 newCursor)
88  {
89      uint256 loanLength = currentLoanIndex[nftId];
90      require(cursor > 0, "APHCore/cursor-must-be-greater-than-zero");
91      require(cursor <= loanLength, "APHCore/cursor-out-of-range");
92      require(resultsPerPage > 0, "resultsPerPage-cannot-be-zero");
93
94      uint256 index;
95      uint256 count;
96      for (index = cursor; index <= loanLength && count < resultsPerPage; index++)
    {
97          if (loanExts[nftId][index].active) {
98              count++;
99          }
100     }
101
102     activeLoans = new Loan[](count);
103     count = 0;
104     for (index = cursor; index <= loanLength && count < resultsPerPage; index++)
    {
105         if (loanExts[nftId][index].active) {
106             activeLoans[count] = loans[nftId][index];
107             count++;
```

```
        }
    }

    return (activeLoans, index);
}
```

Listing 37.2 The improved *getActiveLoans* function with the pagination

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX team* adopted our suggested code to fix this issue.

No. 38	Rejection On Getting Pool List		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/APHCore.sol*		
Locations	*APHCore.sol L: 97 - 99*		

Detailed Issue

The *getPoolList* is a view function returning all registered pools as presented in the code snippet below. Since the number of registered pools can grow over time, the *getPoolList* function can confront a denial-of-service issue if the number of pools is too large.

The root cause of this issue is that the *getPoolList* function returns all pools, which might reach the upper-bound gas limit or take too long for querying on the EVM node, leading to the rejection of the querying request.

Note: even if the querying request would not have to pay gas but the EVM node still counts out the gas being used.

APHCore.sol

```
97  function getPoolList() external view returns (address[] memory) {
98      return poolList;
99  }
```

Listing 38.1 The *getPoolList* function

Recommendations

We recommend re-implementing the *getPoolList* function to address the *denial-of-service issue*.

One possible solution is to apply pagination for data querying, in which the large querying data are divided into smaller discrete pages.

Reassessment

The *FWX team* confirmed that **their pool length would not be more than 100 pools**. Therefore, this issue is considered not to be the case anymore.

No. 39	Compiler May Be Susceptible To Publicly Disclosed Bugs		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/*.sol*		
Locations	*./contracts/*.sol*		

Detailed Issue

The *Forward protocol*'s smart contracts use an outdated Solidity compiler version which may be susceptible to publicly disclosed vulnerabilities. The currently used compiler version is v0.8.7, which contains the list of known bugs as the following links:

https://docs.soliditylang.org/en/v0.8.15/bugs.html

The known bugs may not directly lead to the vulnerability, but it may increase an opportunity to trigger some attacks further.

An example contract that does not use the latest patch version is shown below.

CoreBase.sol

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity 0.8.7;
```

Listing 39.1 Example contract that does not use the latest patch version (v0.8.15)

Recommendations

We recommend using the latest patch version, v0.8.15, which fixes all known bugs.

Reassessment

The *FWX team* fixed this issue by using the Solidity version v0.8.15.

No. 40	Recommended Event Emissions For Transparency		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*./contracts/src/core/APHCore.sol* *./contracts/src/pool/InterestVault.sol* *./contracts/src/pool/PoolSetting.sol* *./contracts/src/utils/PriceFeed.sol*		
Locations	*APHCore.sol L: 40 - 49* *InterestVault.sol L: 27 - 38, 52 - 54, 56 - 58, 60 - 62, 66 - 68, 73 - 75, 82 - 88, 94 - 100, 106 - 108, and 113 - 115* *PoolSetting.sol L: 80 - 82* *PriceFeed.sol L: 100 - 105 and 107 - 111*		

Detailed Issue

The following functions change important states but do not emit events, affecting transparency and traceability for the *Forward protocol*.

1. ***settleForwInterest* function** (*L40 - 49 in APHCore.sol*)
2. ***constructor*** (*L27 - 38 in InterestVault.sol*)
3. ***approveInterestVault* function** (*L52 - 54 in InterestVault.sol*)
4. ***setForwAddress* function** (*L56 - 58 in InterestVault.sol*)
5. ***setTokenAddress* function** (*L60 - 62 in InterestVault.sol*)
6. ***setProtocolAddress* function** (*L66 - 68 in InterestVault.sol*)
7. ***ownerApprove* function** (*L73 - 75 in InterestVault.sol*)
8. ***settleInterest* function** (*L82 - 88 in InterestVault.sol*)
9. ***withdrawTokenInterest* function** (*L94 - 100 in InterestVault.sol*)
10. ***withdrawForwInterest* function** (*L106 - 108 in InterestVault.sol*)
11. ***withdrawActualProfit* function** (*L113 - 115 in InterestVault.sol*)
12. ***setWETHHandler* function** (*L80 - 82 in PoolSetting.sol*)
13. ***setPriceFeed* function** (*L100 - 105 in PriceFeed.sol*)
14. ***setDecimals* function** (*L107 - 111 in PriceFeed.sol*)

Recommendations

We recommend emitting relevant events on the associated functions to improve transparency and traceability for the *Forward protocol*.

Reassessment

The *FWX team* fixed this issue by emitting relevant events on all associated functions.

Appendix

About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in the areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

Contact Information

info@valix.io

https://www.facebook.com/ValixConsulting

https://twitter.com/ValixConsulting

https://medium.com/valixconsulting

References

Title	Link
OWASP Risk Rating Methodology	https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
Smart Contract Weakness Classification and Test Cases	https://swcregistry.io/