



SMART CONTRACT AUDIT REPORT

for

PLEXUS



Prepared By: Shuxiao Wang

PeckShield
February 2, 2021

Document Properties

Client	Plexus
Title	Smart Contract Audit Report
Target	Plexus
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 2, 2021	Xuxian Jiang	Final Release
1.0-rc1	January 30, 2021	Xuxian Jiang	Release Candidate #1
0.3	January 29, 2021	Xuxian Jiang	Additional Findings
0.2	January 22, 2021	Xuxian Jiang	Additional Findings
0.1	January 18, 2021	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1	Introduction	4
1.1	About Plexus	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Wrong Hardcoded Aave AToken Address	11
3.2	Accommodation of approve() Idiosyncrasies	12
3.3	Business Logic Errors in tier2Aave::withdraw()	14
3.4	Improper Handling of ETHs in tier2Farm::deposit()	16
3.5	Loss of Staked Funds With Wrongly Triggered tier2Farm::kill()	17
3.6	Sufficient Allowance Guarantee in tier2Farm::withdraw()	18
3.7	Possible Front-Running For Reduced Return	20
3.8	Incompatibility with Deflationary/Rebasing Tokens	22
3.9	Lack Of Sanity Checks For System Parameters	23
3.10	Removal Of Unused Variables And Code	24
3.11	Safe-Version Replacement With safeTransfer(), safeTransferFrom(), And safeApprove()	25
4	Conclusion	27
	References	28

1 | Introduction

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

1.1 About Plexus

The `Plexus` protocol is a decentralized distribution and aggregation channel for DeFi protocols. In other words, it is a yield farming aggregator and `Plexus` rewards ecosystem. At the protocol layer, `Plexus` is an ecosystem of smart-contracts that provide bridges between protocols to increase capital efficiency. It allows participating users to earn interest from popular lending platforms (e.g., `Aave`) or external yield farms by depositing supported ERC20-compliant tokens into the protocol. In the meantime, the participating users are also rewarded with the `PLX` ERC20 token rewards. It continues the yield-farming paradigm in current DeFi offerings with additional aggregation functionality and improved capital deployment capability to further attract and incentivize users for participation.

The basic information of the `Plexus` protocol is as follows:

Table 1.1: Basic Information of Plexus

Item	Description
Issuer	Plexus
Website	https://plexus.money
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 2, 2021

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

- <https://github.com/stimuluspackage/PlexusContracts.git> (f7e8196)

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

- <https://github.com/stimuluspackage/PlexusContracts.git> (7d34e1e)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

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

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

Table 1.3: The Full List of Check Items

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




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

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

2 | Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	
Low	6	
Informational	2	
Total	11	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 2 informational recommendation.

Table 2.1: Key Audit Findings of Plexus

ID	Severity	Title	Category	Status
PVE-001	Medium	Wrong Hardcoded Aave AToken Address	Business Logic	Resolved
PVE-002	Low	Accommodation of approve() Idiosyncrasies	Business Logic	Resolved
PVE-003	Low	Business Logic Errors in tier2Aave::withdraw()	Business Logic	Resolved
PVE-004	Low	Improper Handling of ETHs in tier2Farm::deposit()	Coding Practices	Resolved
PVE-005	Medium	Loss of Staked Funds With Wrongly Triggered tier2Farm::kill()	Business Logic	Resolved
PVE-006	Low	Sufficient Allowance Guarantee in tier2Farm::withdraw()	Coding Practices	Resolved
PVE-007	Low	Possible Front-Running For Reduced Return	Time and State	Resolved
PVE-008	Informational	Incompatibility with Deflationary/Rebasing Token	Business Logic	Resolved
PVE-009	Low	Lack Of Sanity Checks For System Parameters	Business Logic	Resolved
PVE-010	Informational	Removal Of Unused Variables And Code	Coding Practices	Resolved
PVE-011	Medium	Safe-Version Replacement With safeTransfer(), safeTransferFrom(), And safeApprove()	Coding Practices	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Wrong Hardcoded Aave AToken Address

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: tier2Aave
- Category: Business Logic [6]
- CWE subcategory: N/A

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Plexus protocol is no exception. Specifically, if we examine the constructor of the tier2Aave contract, it has defined a number of system-wide states: `stakingContracts`, `stakingContractsStakingToken`, `tokenToAToken`, and `aTokenToToken`. In the following, we show the related constructor.

```

112  constructor() public payable {
113      stakingContracts["DAI"] = 0x7d2768dE32b0b80b7a3454c06BdAc94A69DDc7A9 ;
114      stakingContracts["ALL"] = 0x7d2768dE32b0b80b7a3454c06BdAc94A69DDc7A9 ;
115      stakingContractsStakingToken["DAI"] = 0
          x25550Cccbd68533Fa04bFD3e3AC4D09f9e00Fc50;
116      tokenToAToken[0x6B175474E89094C44Da98b954EedeAC495271d0F] = 0
          x25550Cccbd68533Fa04bFD3e3AC4D09f9e00Fc50;
117      aTokenToToken[0x25550Cccbd68533Fa04bFD3e3AC4D09f9e00Fc50] = 0
          x6B175474E89094C44Da98b954EedeAC495271d0F;
118      tokenToFarmMapping[stakingContractsStakingToken["DAI"]] = stakingContracts["
          DAI"];
119      owner = msg.sender;
120      admin = msg.sender;
121
122  }
```

Listing 3.1: tier2Aave :: constructor()

It is important to ensure the correctness of these token contracts as they define various important aspects of the protocol operation and need to exercise extra care when configuring or updating it. It comes to our attention that the configured DAI and the associated aDAI mapping is incorrect. A misconfigured DAI/aDAI mapping could potentially result in loss of user funds!

Recommendation Validate these hard-coded token contracts and ensure they are consistent with the mainnet deployment.

Status The issue has been fixed by this commit: 6408424.

3.2 Accommodation of approve() Idiosyncrasies

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In this section, we examine certain non-compliant ERC20 tokens that may exhibit specific idiosyncrasies in their approve() implementation. The respective idiosyncrasies may be present in widely-used token contracts and need to be accommodated for seamless integration and support.

In particular, we use the popular stablecoin, i.e., USDT, as our example. We show the related code snippet below. On its entry of approve(), there is a requirement, i.e., `require(!(_value != 0) && (allowed[msg.sender][_spender] != 0))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (<https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729>).

```

194  /**
195   * @dev Approve the passed address to spend the specified amount of tokens on behalf
      of msg.sender.
196   * @param _spender The address which will spend the funds.
197   * @param _value The amount of tokens to be spent.
198   */
199   function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

201       // To change the approve amount you first have to reduce the addresses'
202       // allowance to zero by calling 'approve(_spender, 0)' if it is not
203       // already 0 to mitigate the race condition described here:
204       // https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205       require(!(_value != 0) && (allowed[msg.sender][_spender] != 0));

```

```

207     allowed[msg.sender][_spender] = _value;
208     Approval(msg.sender, _spender, _value);
209 }

```

Listing 3.2: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use as an example the `deposit` routine from the `tier2Farm` contract. The routine performs the intended investment for later rewards. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (line 150): the first one reduces the allowance to 0; and the second one sets the new allowance.

```

129 function deposit(address tokenAddress, uint256 amount, address onBehalfOf) payable
    onlyOwner public returns (bool){

132     if(tokenAddress == 0x0000000000000000000000000000000000){

134         depositBalances[onBehalfOf][tokenAddress] = depositBalances[onBehalfOf][
            tokenAddress] + msg.value;

136         stake(amount, onBehalfOf, tokenAddress );
137         totalAmountStaked[tokenAddress] = totalAmountStaked[tokenAddress].add(
            amount);
138         emit Deposit(onBehalfOf, amount, tokenAddress);
139         return true;

141     }

143     ERC20 thisToken = ERC20(tokenAddress);
144     require(thisToken.transferFrom(msg.sender, address(this), amount), "Not enough
        tokens to transferFrom or no approval");

146     depositBalances[onBehalfOf][tokenAddress] = depositBalances[onBehalfOf][
        tokenAddress] + amount;

148     uint256 approvedAmount = thisToken.allowance(address(this), tokenToFarmMapping[
        tokenAddress]);
149     if(approvedAmount < amount ){
150         thisToken.approve(tokenToFarmMapping[tokenAddress], amount.mul(10000000));
151     }
152     stake(amount, onBehalfOf, tokenAddress );

154     totalAmountStaked[tokenAddress] = totalAmountStaked[tokenAddress].add(amount);

156     emit Deposit(onBehalfOf, amount, tokenAddress);
157     return true;
158 }

```

Listing 3.3: tier2Farm::deposit()

Recommendation Accommodate the above-mentioned idiosyncrasy of `approve()`.

Status The issue has been fixed by this commit: [c1fb42b](#).

3.3 Business Logic Errors in `tier2Aave::withdraw()`

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `tier2Aave`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

As a decentralized distribution and aggregation channel for DeFi protocols, the `Plexus` protocol is extensible in supporting or aggregating external protocols for additional yields. In the meantime, each external protocol needs to be supported by following the defined interfaces for interaction. In the following, we examine one such interface, i.e., `withdraw()`.

To elaborate, we show below the `withdraw()` routine from the `tier2Aave` contract. As the name indicates, this routine handles the withdraw request from the participating user.

```

205  function withdraw(address tokenAddress, uint256 amount, address payable onBehalfOf)
      onlyOwner payable public returns(bool){
206
207      ERC20 thisToken = ERC20(tokenAddress);
208      //uint256 numberTokensPreWithdrawal = getStakedBalance(address(this), tokenAddress
          );
209
210      if(tokenAddress == 0x0000000000000000000000000000000000000000000000000000000000000000){
211          require(depositBalances[msg.sender][tokenAddress] >= amount, "You didnt
              deposit enough eth");
212
213          totalAmountStaked[tokenAddress] = totalAmountStaked[tokenAddress].sub(
              depositBalances[onBehalfOf][tokenAddress]);
214          depositBalances[onBehalfOf][tokenAddress] = depositBalances[onBehalfOf][
              tokenAddress] - amount;
215          onBehalfOf.send(amount);
216          return true;
217      }
218
219
220
221      require(depositBalances[onBehalfOf][tokenAddress] > 0, "You dont have any tokens
          deposited");
222
223
224

```

```

225     //uint256 numberTokensPostWithdrawal = thisToken.balanceOf(address(this));
226
227     //uint256 usersBalancePercentage = depositBalances[onBehalfOf][tokenAddress].div
        (totalAmountStaked[tokenAddress]);
228
229     uint256 numberTokensPlusRewardsForUser1 = getStakedPoolBalanceByUser(onBehalfOf,
        tokenAddress);
230     uint256 commissionForDAO1 = calculateCommission(numberTokensPlusRewardsForUser1)
        ;
231     uint256 numberTokensPlusRewardsForUserMinusCommission =
        numberTokensPlusRewardsForUser1 - commissionForDAO1;
232
233     unstake(amount, onBehalfOf, tokenAddress);
234
235     //staking platforms only withdraw all for the most part, and for security
        sticking to this
236     totalAmountStaked[tokenAddress] = totalAmountStaked[tokenAddress].sub(
        depositBalances[onBehalfOf][tokenAddress]);
237     ...
238
239 }

```

Listing 3.4: tier2Aave :: withdraw()

It comes to our attention that the above routine does not properly handle certain token addresses. In particular, the lending platform Aave does not directly support ETH. Instead, the wrapped version of ETH, i.e., WETH, is supported. As a result, the code snippet at lines 210 – 218 becomes largely irrelevant and may be removed. Even if it is relevant, the reduction of `totalAmountStaked[tokenAddress]` and `depositBalances[onBehalfOf][tokenAddress]` is not consistent: the former is reduced by `depositBalances[onBehalfOf][tokenAddress]`, while the latter is reduced by `amount`! Note other tier2 contracts, e.g., `tier2Farm`, `tier2Pickle`, and `tier2Aggregator`, share similar issues.

Moreover, the `withdraw()` function takes an `amount` parameter, indicating the amount of balance that is supposed to be withdrawn. However, it turns out partial withdrawal is not allowed. The team has confirmed that each `withdraw` implies a full withdrawal. With that, it is suggested to clarify in the function headers or consider the removal of this parameter.

Recommendation If ETH is intended for support, correct the above logic. Otherwise, remove the irrelevant code. Also document the intended purpose of each parameter by following the Ethereum Natural Language Specification Format (NatSpec) in the function headers.

Status The issue has been fixed by the following commit: 604b3f6.

3.4 Improper Handling of ETHs in tier2Farm::deposit()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: tier2Farm
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

As mentioned in Section 3.3, the Plexus protocol acts as a decentralized distribution and aggregation channel for DeFi protocols and standardizes the interface to interact with external protocols. In the following, we examine another interface, i.e., `deposit()`, from the `tier2Farm` contract.

To elaborate, we show below the implementation of the `deposit()` function. As the name indicates, this function is responsible for performing the investment-related deposit operation that essentially stakes funds into the intended (external) protocol.

```

129  function deposit(address tokenAddress, uint256 amount, address onBehalfOf payable
    onlyOwner public returns (bool)){
130
131
132      if(tokenAddress == 0x0000000000000000000000000000000000000000){
133
134          depositBalances[onBehalfOf][tokenAddress] = depositBalances[onBehalfOf][
            tokenAddress] + msg.value;
135
136          stake(amount, onBehalfOf, tokenAddress );
137          totalAmountStaked[tokenAddress] = totalAmountStaked[tokenAddress].add(
            amount);
138          emit Deposit(onBehalfOf, amount, tokenAddress);
139          return true;
140      }
141
142
143      ERC20 thisToken = ERC20(tokenAddress);
144      require(thisToken.transferFrom(msg.sender, address(this), amount), "Not enough
        tokens to transferFrom or no approval");
145
146      depositBalances[onBehalfOf][tokenAddress] = depositBalances[onBehalfOf][
        tokenAddress] + amount;
147
148      uint256 approvedAmount = thisToken.allowance(address(this), tokenToFarmMapping[
        tokenAddress]);
149      if(approvedAmount < amount ){
150          thisToken.approve(tokenToFarmMapping[tokenAddress], amount.mul(10000000));
151      }
152      stake(amount, onBehalfOf, tokenAddress );
153

```



```

154     totalAmountStaked[tokenAddress] = totalAmountStaked[tokenAddress].add(amount);
155
156     emit Deposit(onBehalfOf, amount, tokenAddress);
157     return true;
158 }

```

Listing 3.5: tier2Farm::deposit()

It comes to our attention that the above routine does not properly handle ETH deposits. In particular, when `tokenAddress == 0x00` (lines 132–141), there is a need to validate that the given `amount` should be the same as `msg.value`. Also, for the intended `stake()` call (line 136), the received ETH needs to transfer to the external protocol in the accepted form. Specifically, if ETH is directly supported by the external DeFi protocol, we can simply send ETH in the `stake()` call. Otherwise, there is a need to wrap ETH into WETH for the `stake()` call.

Note other tier2 contracts, e.g., `tier2Pickle`, and `tier2Aggregator`, share the same issues.

Recommendation Revise the logic to properly handle ETH-related deposits.

Status The issue has been fixed by this commit: [597d0ef](#).

3.5 Loss of Staked Funds With Wrongly Triggered tier2Farm::kill()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In the Plexus, most contracts have been equipped with a built-in `kill()` functionality that allows the owner to explicitly self-destruct the specific contract. However, this capability needs to exercise extra case as these contracts may directly interact with external DeFi protocol. Because of that, these contracts may effectively act as the holders of staked funds in these external DeFi protocols.

To elaborate, we show below the `kill()` routine from the `tier2Farm` contract. A blind call of it makes it unable to further unstake the funds, if any, from the external DeFi protocols.

```

287 function kill() virtual public onlyOwner {
288
289     selfdestruct(owner);
290 }

```

291 }

Listing 3.6: tier2Farm::kill()

A better approach may be to verify there are no assets remaining in current contract and only invoke `selfdestruct()` (line 289) after the successful validation. Note all current tier2 contracts, e.g., tier2Aave, tier2Farm, tier2Pickle, and tier2Aggregator, share the same issue.

Recommendation Revise the `kill()` logic to ensure staked funds are not at risk.

Status The issue has been fixed by this commit: 83a8e36.

3.6 Sufficient Allowance Guarantee in tier2Farm::withdraw()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: tier2Farm
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

As mentioned earlier, the Plexus protocol takes a tier-based approach to facilitate the aggregation of external DeFi protocols. And the tier-2 contracts are modular, each being a proxy that can be dynamically removed or amended with no risk exposure to existing capital. In addition, to accommodate certain DeFi protocols that may support partial withdrawal, a normal `withdraw()` implementation in a tier-2 contract typically unstakes the full balance to meet the user withdrawal request and then stakes back the remaining balance, if any.

To elaborate, we show below the `withdraw()` routine from the tier2Farm contract. It comes to our attention that the staking of the remaining balance (line 238) does not properly check whether there is sufficient allowance to stake. An insufficient allowance may break the re-staking attempt, hence reverting the withdraw operation!

```

190 function withdraw(address tokenAddress, uint256 amount, address payable onBehalfOf)
    onlyOwner payable public returns(bool){
192
193     ERC20 thisToken = ERC20(tokenAddress);
    //uint256 numberTokensPreWithdrawal = getStakedBalance(address(this), tokenAddress
    );
195
196     if(tokenAddress == 0x0000000000000000000000000000000000000000){
        require(depositBalances[msg.sender][tokenAddress] >= amount, "You didnt
        deposit enough eth");
    }

```

```
198         totalAmountStaked[tokenAddress] = totalAmountStaked[tokenAddress].sub(
199             depositBalances[onBehalfOf][tokenAddress]);
200         depositBalances[onBehalfOf][tokenAddress] = depositBalances[onBehalfOf][
201             tokenAddress] - amount;
202         onBehalfOf.send(amount);
203         return true;
204     }
205
206     require(depositBalances[onBehalfOf][tokenAddress] > 0, "You dont have any tokens
207         deposited");
208
209     //uint256 numberTokensPostWithdrawal = thisToken.balanceOf(address(this));
210
211     //uint256 usersBalancePercentage = depositBalances[onBehalfOf][tokenAddress].div
212         (totalAmountStaked[tokenAddress]);
213
214     uint256 numberTokensPlusRewardsForUser1 = getStakedPoolBalanceByUser(onBehalfOf,
215         tokenAddress);
216     uint256 commissionForDAO1 = calculateCommission(numberTokensPlusRewardsForUser1)
217         ;
218     uint256 numberTokensPlusRewardsForUserMinusCommission =
219         numberTokensPlusRewardsForUser1-commissionForDAO1;
220
221     unstake(amount, onBehalfOf, tokenAddress);
222
223     //staking platforms only withdraw all for the most part, and for security
224     //sticking to this
225     totalAmountStaked[tokenAddress] = totalAmountStaked[tokenAddress].sub(
226         depositBalances[onBehalfOf][tokenAddress]);
227
228     depositBalances[onBehalfOf][tokenAddress] = 0;
229     require(numberTokensPlusRewardsForUserMinusCommission >0, "For some reason
230         numberTokensPlusRewardsForUserMinusCommission is zero");
231
232     require(thisToken.transfer(onBehalfOf,
233         numberTokensPlusRewardsForUserMinusCommission), "You dont have enough tokens
234         inside this contract to withdraw from deposits");
235     if(numberTokensPlusRewardsForUserMinusCommission >0){
236         thisToken.transfer(owner, commissionForDAO1);
237     }
238
239     uint256 remainingBalance = thisToken.balanceOf(address(this));
240     if(remainingBalance >0){
```

```

238         stake(remainingBalance, address(this), tokenAddress);
239     }

242     emit Withdrawal(onBehalfOf, amount, tokenAddress);
243     return true;

245 }

```

Listing 3.7: tier2Farm :: withdraw()

Note all current tier2 contracts, e.g., tier2Aave, tier2Farm, tier2Pickle, and tier2Aggregator, share the same issue.

Recommendation Ensure sufficient allowance for a successful stake operation.

Status The issue has been fixed by this commit: 71ca93a.

3.7 Possible Front-Running For Reduced Return

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: WrapAndUnWrap
- Category: Time and State [7]
- CWE subcategory: CWE-682 [3]

Description

As common in various strategies for yield-farming, there is a need to convert from one token to another. The Plexus protocol has included a WrapAndUnWrap contract to facilitates the conversion. To elaborate, we show below the key conversion routine, i.e., conductUniswap(). This routine has been used in various contexts to optimize the asset allocation and deployment.

```

389 function conductUniswap(address sellToken, address buyToken, uint amount) internal
390     returns (uint256 amounts1){
391     if(sellToken == ETH_TOKEN_ADDRESS && buyToken == WETH_TOKEN_ADDRESS){
392         wethToken.deposit{value: msg.value}();
393     }
394     else if(sellToken == address(0x0)){
395
396         // address [] memory addresses = new address[](2);
397         address [] memory addresses = getBestPath(WETH_TOKEN_ADDRESS, buyToken,
398             amount);
399         //addresses[0] = WETH_TOKEN_ADDRESS;
400         //addresses[1] = buyToken;

```

```

400         uniswapExchange.swapExactETHForTokens{value:msg.value}(0, addresses,
401             address(this), 10000000000000000 );
402     }
403
404     else if(sellToken == WETH_TOKEN_ADDRESS){
405         wethToken.withdraw(amount);
406
407         //address [] memory addresses = new address[](2);
408         address [] memory addresses = getBestPath(WETH_TOKEN_ADDRESS, buyToken,
409             amount);
410         //addresses[0] = WETH_TOKEN_ADDRESS;
411         //addresses[1] = buyToken;
412         uniswapExchange.swapExactETHForTokens{value:amount}(0, addresses,
413             address(this), 10000000000000000 );
414     }
415
416     else{
417
418
419         address [] memory addresses = getBestPath(sellToken, buyToken, amount);
420         uint256 [] memory amounts = conductUniswapT4T(addresses, amount );
421         uint256 resultingTokens = amounts[amounts.length-1];
422         return resultingTokens;
423     }
424 }

```

Listing 3.8: WrapAndUnWrap::conductUniswap()

We notice the conversion is routed to `UniswapV2` in order to swap one token to another. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or time-weighted average price of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of farming users.

Status The issue has been addressed by the following commit: `ceb82e4`.

3.8 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Core
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

In Plexus, the `Core` contract is designed to be the main entry for users who want to interact with the protocol. For example, an user can deposit the funds to collect yields or rewards. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets (e.g., `DAI`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```

122     function deposit(string memory tier2ContractName, address tokenAddress, uint256
      amount) nonReentrant() payable public returns (bool){
123
124         ERC20 token;
125         if(tokenAddress==ETH_TOKEN_PLACEHOLDER_ADDRESS){
126             wethToken.deposit{value:msg.value}();
127             tokenAddress=WETH_TOKEN_ADDRESS;
128             token = ERC20(tokenAddress);
129         }
130         else{
131             token = ERC20(tokenAddress);
132             token.transferFrom(msg.sender, address(this), amount);
133         }
134         token.approve(stakingAddress, approvalAmount);
135         bool result = staking.deposit(tier2ContractName, tokenAddress, amount, msg.sender
            );
136         require(result, "There was an issue in core with your deposit request. Please see
            logs");
137         return result;
138
139     }

```

Listing 3.9: Core::deposit()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the protocol. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is widely-adopted USDT.

Status The issue has been addressed by the following commit: `01e8938`.

3.9 Lack Of Sanity Checks For System Parameters

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

As mentioned in Section 3.1, DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Plexus protocol is no exception. Specifically, if we examine the `tier2Pickle` contract, it has defined a system-wide risk parameter: `commission`. In the following, we show the related route that allows for its update.

```

126 function updateCommission(uint amount) public onlyOwner returns(bool){
127     commission = amount;
128     return true;
129 }
```

Listing 3.10: `tier2Pickle :: updateCommission()`

Apparently, the above update logic can be improved by applying a more rigorous range check. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large commission fee parameter (say more than 100%) will revert the `withdraw()` operation, putting staked funds at risk.

Recommendation Validate the given `amount` argument before updating the commission parameter in the system.

Status The issue has been addressed by the following commit: 80a3004.

3.10 Removal Of Unused Variables And Code

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1099 [1]

Description

Plexus makes use of a number of reference libraries and contracts, such as SafeMath, ERC20, and Uniswap, to facilitate the protocol implementation and organization. For instance, the Tier2FarmController smart contract interacts with at least four different external contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the tier2Farm contract, the variables platformToken and tokenStakingContract are not used anywhere. Therefore, these variables can be safely removed.

```

67 contract Tier2FarmController{
68
69     using SafeMath
70         for uint256;
71
72
73     address payable public owner;
74     address public platformToken = 0xa0246c9032bC3A600820415aE600c6388619A14D;
75     address public tokenStakingContract = 0x25550Cccbd68533Fa04bFD3e3AC4D09f9e00Fc50;
76     address ETH_TOKEN_ADDRESS = address(0x0);
77     mapping (string => address) public stakingContracts;
78     mapping (address => address) public tokenToFarmMapping;
79     mapping (string => address) public stakingContractsStakingToken;
80     mapping (address => mapping (address => uint256)) public depositBalances;
81     uint256 public commission = 400; // Default is 4 percent
82
83     ...
84 }
```

Listing 3.11: tier2Farm

In the same vein, we also observe states, e.g., principalPlusRewards, tokensInRewardsReserve, and lpTokensInRewardsReserve, in TokenRewards are not used either. The burnaddress from Oracle can also be removed. For maintenance, their removals are recommended.

Recommendation Remove unnecessary imports of reference contracts and remove unused code.

Status The issue has been addressed by the following commit: [4395e75](#).

3.11 Safe-Version Replacement With `safeTransfer()`, `safeTransferFrom()`, And `safeApprove()`

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Core
- Category: Coding Practices [\[5\]](#)
- CWE subcategory: CWE-1126 [\[2\]](#)

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In Section [3.2](#), we have examined the `approve()` idiosyncrasies. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., ZRX, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *“Transfers `_value` amount of tokens to address `_to`, and MUST fire the Transfer event. The function SHOULD throw if the message caller’s account balance does not have enough tokens to spend.”*

```

64     function transfer(address _to, uint _value) returns (bool) {
65         //Default assumes totalSupply can't be over max (2^256 - 1).
66         if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67             balances[msg.sender] -= _value;
68             balances[_to] += _value;
69             Transfer(msg.sender, _to, _value);
70             return true;
71         } else { return false; }
72     }

74     function transferFrom(address _from, address _to, uint _value) returns (bool) {
75         if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
76             balances[_to] + _value >= balances[_to]) {

```

```

77         balances[_from] -= _value;
78         allowed[_from][msg.sender] -= _value;
79         Transfer(_from, _to, _value);
80         return true;
81     } else { return false; }
82 }

```

Listing 3.12: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`. In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. To use this library you can add a using `SafeERC20` for `IERC20`. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `deposit()` routine in the `Core` contract. If the `USDT` token is supported as `tokenAddress`, the unsafe version of `token.transferFrom(msg.sender, address(this), amount)` (line 132) may revert as there is no return value in the `USDT` token contract's `transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```

122     function deposit(string memory tier2ContractName, address tokenAddress, uint256
123         amount) nonReentrant() payable public returns (bool){
124         ERC20 token;
125         if(tokenAddress==ETH_TOKEN_PLACEHOLDER_ADDRESS){
126             wethToken.deposit{value:msg.value}();
127             tokenAddress=WETH_TOKEN_ADDRESS;
128             token = ERC20(tokenAddress);
129         }
130         else{
131             token = ERC20(tokenAddress);
132             token.transferFrom(msg.sender, address(this), amount);
133         }
134         token.approve(stakingAddress, approvalAmount);
135         bool result = staking.deposit(tier2ContractName, tokenAddress, amount, msg.sender
136             );
137         require(result, "There was an issue in core with your deposit request. Please see
138             logs");
139         return result;

```

Listing 3.13: Core::deposit()

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status The issue has been addressed by the following commit: 7d34e1e.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Plexus` protocol. The audited system presents a new addition to current DeFi offerings by acting as a decentralized distribution and aggregation channel for defi protocols.. The current code base is neatly organized and those identified issues are promptly confirmed and fixed.

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



References

- [1] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. <https://cwe.mitre.org/data/definitions/1099.html>.
- [2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [3] MITRE. CWE-682: Incorrect Calculation. <https://cwe.mitre.org/data/definitions/682.html>.
- [4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. <https://cwe.mitre.org/data/definitions/389.html>.
- [8] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. <https://www.peckshield.com>.

