SMART CONTRACT AUDIT REPORT

for

DXDY Safety Module

Prepared By: Terry Miranda PeckShield

October 23, 2021

Document Properties

Client	DXDY
Title	Smart Contract Audit Report
Target	DXDY Safety Module
Version	1.0
Author	Calvin Morris
Auditors	James Martin, Calvin Morris
Reviewed by	Terry Miranda
Approved by	Calvin Morris
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	October 23, 2021	Calvin Morris	Final Release
1.0-rc2	October 22, 2021	Calvin Morris	Release Candidate
1.0-rc1	October 10, 2021	Calvin Morris	Release Candidate

Contents

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the safety module in the DXDY protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsis- tencies 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 is well-designed and engineered. This document outlines our audit results.

1.1 About DXDY

DXDY is a leading decentralized exchange that currently supports perpetual, margin trading, and spot trading, as well as lending, and borrowing. DXDY runs on smart contracts on the Ethereum blockchain, and allows users to trade with no intermediaries. The audited safety module allows to stake protocol tokens DXDY and earn the same DXDY as rewards. The staked assets can be withdrawn, subject to an epoch schedule with a blackout window, operating the same way as in the liquidity staking module. In addition, the safety module has a slasher role which will be controlled by the governance DAO. The slasher can take funds out of the contract at its discretion, and these losses are applied pro-rata to all stakers.

The basic information of the DXDY Safety Module is as follows:

Table 1.1: Basic Information of DXDY Safety Module

Item	Description
Name	DXDY
Website	https://dxdy.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 23, 2021

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing).

Table 1.2: Vulnerability Severity Classification

Impac		Likelihood		
		High	*Medium*	*Low*
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the 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 con- tracts 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 devel- opment. 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 Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an ex- ploitable 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 safety module in the DXDY protocol. 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	0	
Low	3	🟦🟦🟦
Informational	1	🟦
Total	4	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 low-severity vulnerabili- ties, and 1 informational recommendation.

Table 2.1: Key DXDY Safety Module Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Sanity Checks For System Parameters	Coding Practices	Confirmed
PVE-002	Informational	Suggested Slashing Logic	Business Logic	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Resolved
PVE-004	Low	Redundant nonReentrant Modifier Removal	Coding Practices	Resolved

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

3 | Detailed Results

3.1 Improved Sanity Checks For System Parameters

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SM1EpochSchedule
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The DXDY protocol is no exception. Specifically, if we examine the SM1EpochSchedule contract of the safety module, it has defined a number of protocol-wide risk parameters, e.g., _BLACKOUT_WINDOW_ and _REWARDS_PER_SECOND_. In the following, we show the corresponding routines that allow for their changes.

```
function _setEpochParameters(uint256 interval, uint256 offset) internal
  {SM1Types.EpochParameters memory epochParameters =
    SM1Types.EpochParameters({interval: interval.toUint128(), offset: offset.toUint128()});
  _EPOCH_PARAMETERS_ = epochParameters;
  emit EpochParametersChanged(epochParameters);
}

function _setBlackoutWindow(uint256 blackoutWindow) internal {
  _BLACKOUT_WINDOW_ = blackoutWindow;
  emit BlackoutWindowChanged(blackoutWindow);
}
```

Listing 3.1: A Number of Setters in SM1EpochSchedule

Our result shows the update logic on the above parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of a large _BLACKOUT_WINDOW_ parameter will revert the requestWithdrawal() operation.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status This issue has been confirmed.

3.2 Suggested Slashing Logic

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SM1Slashing
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The DXDY safety module is designed with a slasher role which will be controlled by the governance DAO. The slasher can take funds out of the contract at its discretion, and these losses are applied pro-rata to all stakers.

To elaborate, we show below the slash() function. By design, all funds in the contract, active or inactive, are slash-able. The current logic supports two types of slashes: partial or full. A partial slash is recorded by updating the exchange rate. A full slash has the effect of setting all balances to zero in the timestamp in which the slash occurred. Both types of slashes are implemented by accordingly changing the protocol-wide exchange rate.

```
function slash(
  uint256 requested Amount ,
  address recipient
)
  external only
  Role( SLASHER_ROLE ) non
  Reentrant
  returns ( uint256 )
{
  uint256 underlyingBalance = STAKED_TOKEN.balanceOf(address(this));
  uint256 partialSlashAmount = Math.min(requestedAmount, underlyingBalance);
  uint256 remainingAfterPartialSlash = underlyingBalance.sub(partialSlashAmount);

  if (remainingAfterPartialSlash == 0) {
    return _fullSlash(underlyingBalance, recipient);
  }

  if (
    underlying Balance . div ( remaining AfterPartialSlash ) >=
        MAX_EXCHANGE_RATE_GROWTH_PER_SLASH
  ) {
    return _fullSlash(underlyingBalance, recipient);
  }

  // Partial slash: update the exchange rate.
  //
  // It is unlikely, but possible, for this multiplication to overflow.
  // In such a case, the slasher should request a full slash to reset the exchange rate.
  uint256 newExchangeRate = (
    _EXCHANGE_RATE_.mul(underlyingBalance).div(remainingAfterPartialSlash)
  );
  _EXCHANGE_RATE_ = newExchangeRate;

  // Transfer the slashed token.
  STAKED_TOKEN.safeTransfer(recipient, partialSlashAmount);

  emit Slashed(partialSlashAmount, recipient, newExchangeRate, false);
  return partialSlashAmount;
}
```

Listing 3.2: The Privileged slash() Operation in SM1Slashing

Our analysis shows the slashing logic is equally applied to all funds in the contract, including those that have been requested and ready for immediate withdrawal. In other words, if the user withdraws earlier than the slashing operation, the user will not suffer any loss from slashing. While we understand it is a design choice, it may be worthwhile to explore an alternative in exempting those withdraw requests that are made at least one epoch earlier.

Recommendation Revised the slashing logic in exempting earlier withdraw requests that have been made at least one epoch earlier when a slash occurs.

Status This issue has been resolved as a new slashing logic replaces the current implementation, which removes the “full slash” concept and limits slashing to 95%.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The DXDY safety module supports a number of roles that can be regulated and managed by the one with the designated OWNER_ROLE. As the name indicates, this is a privileged account that plays a critical role in governing and regulating the token-related operations (e.g., assigning other roles). In the following, we show a representative privileged operation with the privileged roles.

```
function slash(
   uint256 requested Amount ,
   address recipient
)
   external only
   Role( SLASHER_ROLE ) non
   Reentrant
   returns ( uint256 )
{
   uint256 underlyingBalance = STAKED_TOKEN.balanceOf(address(this));
   uint256 partialSlashAmount = Math.min(requestedAmount, underlyingBalance);
   uint256 remainingAfterPartialSlash = underlyingBalance.sub(partialSlashAmount);

   if (remainingAfterPartialSlash == 0) {
      return _fullSlash(underlyingBalance, recipient);
   }

   if (
      underlying Balance . div ( remaining AfterPartialSlash ) >=
            MAX_EXCHANGE_RATE_GROWTH_PER_SLASH
   ) {
      return _fullSlash(underlyingBalance, recipient);
   }

   // Partial slash: update the exchange rate.
   //
   // It is unlikely, but possible, for this multiplication to overflow.
   // In such a case , the slasher should request a full slash to reset the exchange
      rate.
   uint256 newExchangeRate = (
      _EXCHANGE_RATE_.mul(underlyingBalance).div(remainingAfterPartialSlash)
   );
   _EXCHANGE_RATE_ = newExchangeRate;

   // Transfer the slashed token .
   STAKED_TOKEN.safeTransfer(recipient, partialSlashAmount);

   emit Slashed(partialSlashAmount, recipient, newExchangeRate, false);
   return partialSlashAmount;
}
```

Listing 3.3: The Privileged slash() Operation in SM1Slashing

We emphasize that the privilege assignment is necessary and consistent with the intended design. However, it is worrisome if the owner is not governed by a DAO-like structure. The discussion with the team has confirmed that this privileged account will be managed by a governance DAO. It should be noted that a compromised owner account would allow the attacker to mess up internal records and claim rewards for others, which directly undermines the assumption of the staking support.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the in- tended trustless nature and high-quality distributed governance.

Status The team confirms that the contract will not be used in production without the owner set to the governance DAO which will be controlled via token governance.

3.4 Redundant nonReentrant Modifier Removal

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: SM1Admin
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The support on the DXDY safety module has an extensive use of the nonReentrant modifier, as a precaution to defend against possible reentrancy risks. However, our analysis shows its use in a number of contracts may not be necessary.

For example, if we examine closely the SM1Admin contract, there are a number of public func- tions, i.e., setEpochParameters(), setBlackoutWindow(), and setRewardsPerSecond(). These functions allow the authorized entities to configure respective protocol-wide risk parameters. Note that these functions are protected with nonReentrant, which is unnecessary and can be safely removed.

```
function setEpochParameters(uint256 interval, uint256 offset)
  external
  onlyRole(EPOCH_PARAMETERS_ROLE)
  nonReentrant
{
  if (!hasEpochZeroStarted()) {
    require(block.timestamp < offset, 'SM1Admin: Cannot jump past start of epoch zero'
        );
    _setEpochParameters(interval, offset);
    return;
  }

  // We must settle the total active balance to ensure the index is recorded at the
      epoch
  // boundary as needed, before we make any changes to the epoch formula.
  _settleTotalActiveBalance();

  // Update the epoch parameters. Require that the current epoch number is unchanged.
```

```
    uint256 originalCurrentEpoch = getCurrentEpoch();
    _setEpochParameters(interval, offset);
    uint256 newCurrentEpoch = getCurrentEpoch();
    require(originalCurrentEpoch == newCurrentEpoch, 'SM1Admin: Cannot jump between
        epochs');
  }

  /**
   * @notice Set the blackout window, during which one cannot request withdrawals of
       staked funds.
   */
  function setBlackoutWindow(uint256 blackoutWindow)
    external
    onlyRole(EPOCH_PARAMETERS_ROLE)
    nonReentrant
  {
    _setBlackoutWindow(blackoutWindow);
  }

  /**
   * @notice Set the emission rate of rewards.
   *
   * @param  emissionPerSecond  The new number of rewards tokens given out per second.
   */
  function setRewardsPerSecond(uint256 emissionPerSecond)
    external
    onlyRole(REWARDS_RATE_ROLE)
    nonReentrant
  {
    uint256 totalStaked = 0;
    if (hasEpochZeroStarted()) {
      // We must settle the total active balance to ensure the index is recorded at the
          epoch
      // boundary as needed, before we make any changes to the emission rate.
      totalStaked = _settleTotalActiveBalance();
    }
    _setRewardsPerSecond(emissionPerSecond, totalStaked);
  }
```

Listing 3.4: SM1Admin::setEpochParameters()/setBlackoutWindow()/setRewardsPerSecond()

Recommendation Consider the removal of the redundant nonReentrant modifier in the above functions.

Status This issue has been resolved as the team intends to using nonReentrant on all external non-view functions to help ensure that the contract code is safe even if it is later adapted in other contexts.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the safety module in the DXDY protocol. The safety module allows to stake protocol tokens DXDY and earn the same DXDY as rewards. It also supports the slashing functionality that can be applied pro-rata to all stakers. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Furthermore, we need to emphasize that smart contracts as a whole are still in an early, but excit- ing 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/ definitions/563.html.

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

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

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/ 1006.html.

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