Dear Reviewers,

In this attachment, we present a complementary experiment and some evidence to support our response to the constructive comments: Reviewer#A-Q1, Reviewer#A-Q2, Reviewer#A-Q3, and Reviewer#A-W1, in the rebuttal.

1. Detailed Information for Reviewer#A-Q1

In this section, we aim to elucidate the differences between centralization defects and existing defects. We will discuss several centralization defects that identified by Reviewer A. The discussion will highlight the primary distinctions between centralization defects and existing defects, supplemented with relevant code examples.

1.1 Management without Timelock (MT) & Timestamp Dependency

As shown in Table 1, while Management without Timelock (MT) and Timestamp Dependency are both time-related defects, their definitions differ significantly. The primary issue with Management without Timelock is that it allows administrators to modify smart contracts without a timelock mechanism. In contrast, the main concern with Timestamp Dependency is the inherent insecurity of the *block.timestamp* instruction itself.

Table 1: Definition of similar defects related to time

Defect	Definition
Management without Timelock (MT)	Management functions can be executed without time delay.
Timestamp Dependency (SWC-116)	The contract's logic relies on *block.timestamp* to make critical decisions.

To facilitate understanding, we provide code examples of these two types of defects. As shown in Figure 1, the execution of the function *run* depends on the return value of the function *isSaleFinished*, which relies on the value of *block.timestamp*. Since *block.timestamp* can be manipulated by blockchain miners within a certain range, this allows miners to influence the execution of the *run* function.

```
// code example of Timestamp Dependency
function isSaleFinished() private returns (bool) {
  return block.timestamp >= 1546300800;
}

function run() public {
  if (isSaleFinished()) {
      emit Finished();
  } else {
      emit notFinished();
  }
}
```

Figure 1: Code Example of Timestamp Dependency

As illustrated in Figure 2, the functions *pause* and *unpause* can be used to suspend or resume the execution of certain functions within the contract, thereby managing state changes or business logic. However, the absence of a timelock mechanism in these functions permits the contract owner to arbitrarily modify the contract state without providing users adequate time to respond.

```
// code example of Management without Timelock
function pause() public onlyOwner {
   paused = true;
 }

 function unpause() public onlyOwner {
   paused = false;
 }

 modifier whenNotPaused() {
   require(!paused, "Contract is paused");
   _;
 }
```

Figure 2: Code Example of Management without Timelock

To address Management without Timelock, a timelock mechanism can be incorporated into the management functions, as illustrated in Figure 3.

```
// code example of Timelock Mechanism
modifier onlyBefore(uint256 _time) {
  require(block.timestamp < _time, "Time has already passed");
  _;
}

modifier onlyAfter(uint256 _time) {
  require(block.timestamp > _time, "Time has not arrived");
  _;
}
```

Figure 3: Code Example of Timelock Mechanism

1.2 Selfdestruct with Single Signature (SS) & Unprotected SELFDESTRUCT

As shown in Table 2, although both Selfdestruct with Single Signature (SS) and Unprotected SELFDESTRUCT analyze the *selfdestruct* function, there are notable differences between these two defects. The main distinction lies in who has the permission to invoke the *selfdestruct* function. For Selfdestruct with Single Signature, the *selfdestruct* function should be callable only by a specific node (such as the contract owner). In contrast, for Unprotected SELFDESTRUCT, the *selfdestruct* function can be invoked by any user on the blockchain.

Table 2: Definition of similar defects related to *selfdestruct* function

Defect	Definition
Selfdestruct with Single Signature (SS)	The *selfdestruct* function can be executed by a single-signature address.
Unprotected SELFDESTRUCT (SWC-106)	Due to missing or insufficient access controls, malicious parties can *selfdestruct* the contract.

To facilitate understanding, we provide code examples of these two defects. As illustrated in Figure 4, the *selfdestruct* function within the *destroy* function lacks access control, allowing any user on the blockchain to invoke it directly. Consequently, this vulnerability can result in the destruction of the project and the transfer of all Ether in the contract to the attacker's account.

```
// code example of Unprotected Selfdestruct
function destroy() public  {
  selfdestruct(msg.sender);
}
```

Figure 4: Code Example of Unprotected SELFDESTRUCT

As shown in Figure 5, the *close* function performs identity verification of the caller before executing the *selfdestruct* function to ensure that only the contract owner can invoke it. However, if the contract owner's account is controlled by a single signature node, the loss or theft of this node's private key could result in severe consequences. This defect can be mitigated by incorporating a multi-signature verification mechanism within the function or by transferring the authority to invoke the function to a multi-signature contract.

```
// code example of Selfdestruct with Single Signature
function close() public  {
  require(msg.sender == _owner, "Only the contract owner can call this function");
  selfdestruct(_owner);
}
```

Figure 5: Code Example of Selfdestruct with Single Signature

1.3 Individual Contract Output Reliance (IOR) & Unchecked Call Return Value

As shown in Table 3, although both Individual Contract Output Reliance (IOR) and Unchecked Call Return Value defects involve function calls to external contracts, there are significant differences between these two defects. The critical factor for Unchecked Call Return Value is whether the return value of a message call is checked. The critical factor for Individual Contract Output Reliance is whether key variables or logic depend solely on the output of a single contract, such as a single smart contract oracle.

Table 3: Definition of similar defects related to invocation to external contract

Defect	Definition
Individual Contract Output Reliance (IOR)	Exist logic that relies on the output of an individual external contract.
Unchecked Call Return Value (SWC-104)	The return value of a message call is not checked.

To facilitate understanding, we provide code examples of these two defects. As illustrated in Figure 6, the functions *callchecked* and *callnotchecked* represent scenarios where the return value of a message call is checked and unchecked, respectively.

```
// code example of Unchecked Return Value
function callchecked(address callee) public {
  require(callee.call());
}

function callnotchecked(address callee) public {
  callee.call();
}
```

Figure 6: Code Example of Unchecked Call Return Value

As illustrated in Figure 7, the function *currentTick* directly retrieves the current price tick from a decentralized exchange *pool*. Since the return value of the call to the external contract *pool* is used to calculate the variable *price*, this function does not suffer from Unchecked Call Return Value. However, because the value of *price* solely depends on the output of a single contract *pool*, it exhibits a centralization defect. If the contract *pool* fails or is attacked and outputs an incorrect result, the token will be transferred at an erroneous price.

```
//code example of Individual Contract Output Reliance
contract Contract_Output{
  uint160 sqrtPrice = TickMath.getSqrtRatioAtTick(currentTick());
  uint256 price = FullMath.mulDiv(uint256(sqrtPrice).mul(uint256(sqrtPrice)),PRECISION, 2**(96*2));
  function currentTick() public view returns (int24 tick) {
      (, tick, , , , , ) = pool.slot0();
  }
}
```

Figure 7: Code Example of Individual Contract Output Reliance

2. Evidence for Reviewer#A-Q2

In this section, we refer to the descriptions in the original papers and GitHub repositories to demonstrate that the selected tools are unable to detect centralization defects. Specifically, we examine three tools mentioned by Reviewer #A: Mythril, Slither, and Smartian.

Specifically, we first refer to the papers and documentation of these tools to demonstrate all types of smart contract defects they can detect, along with their corresponding descriptions. Based on these descriptions, we then explain that the defects identified by these tools do not encompass centralization defects. Consequently, we illustrate that these tools are incapable of detecting centralization defects in DeFi projects.

To better determine whether the vulnerabilities detected by these tools fall within the scope of centralization defects, we will first present the definition of centralization defects in DeFi projects. As stated in the second paragraph of Section 1 (Introduction) of the paper, the definition of centralization defects in DeFi projects is as follows:

"A centralization defect in a DeFi project refers to any error, flaw, or fault in a smart contract's design or development stage that results in a single point of failure [2]. This means that specific accounts, users, or addresses could disrupt normal operations, potentially causing project malfunctions or even shutdown."

According to this definition, centralization defects in DeFi projects emphasize the potential impact of actions or failures by a single node with special privileges or functions (such as the contract owner) on the normal operation. Consequently, the potential impact of actions by ordinary users on the blockchain is not considered within the scope of centralization defects.

Next, we will elaborate on various types of smart contract defects that can be detected by the tools Mythril, Slither, and Smartian. Additionally, we will assess their capability to detect centralization defects.

2.1 Mythril

Source: the document for detailed instructions of Mythril on its Github repository (https://mythril-classic.readthedocs.io/en/master/module-list.html).

Modules

Delegate Call To Untrusted Contract

The delegatecall module detects SWC-112 (DELEGATECALL to Untrusted Callee).

Dependence on Predictable Variables

The predictable variables module detects SWC-120 (Weak Randomness) and SWC-116 (Timestamp Dependence).

Ether Thief

The Ether Thief module detects SWC-105 (Unprotected Ether Withdrawal).

Exceptions

The exceptions module detects SWC-110 (Assert Violation).

External Calls

The external calls module warns about SWC-107 (Reentrancy) by detecting calls to external contracts.

Integer

The integer module detects SWC-101 (Integer Overflow and Underflow).

Multiple Sends

The multiple sends module detects SWC-113 (Denial of Service with Failed Call) by checking for multiple calls or sends in a single transaction.

Suicide

The suicide module detects SWC-106 (Unprotected SELFDESTRUCT).

State Change External Calls

The state change external calls module detects SWC-107 (Reentrancy) by detecting state change after calls to an external contract.

Unchecked Retval

The unchecked retval module detects SWC-104 (Unchecked Call Return Value).

User Supplied assertion

The user supplied assertion module detects SWC-110 (Assert Violation) for user-supplied assertions. User supplied assertions should be log messages of the form:
`emit AssertionFailed(string)` .

Arbitrary Storage Write

The arbitrary storage write module detects SWC-124 (Write to Arbitrary Storage Location).

Arbitrary Jump

The arbitrary jump module detects SWC-127 (Arbitrary Jump with Function Type Variable).

Figure 8: All the modules of Mythril

As shown in Figure 8, Mythril consists of 12 modules designed to detect 12 types of defects in smart contracts. We will next evaluate each of these defects based on their definitions to determine whether they fall within the scope of centralization defects.

- Delegatecall To Untrusted Callee (SWC-112):

  "Delegatecall to Untrusted Callee" refers to the use of the *delegatecall* function, which operates similarly to a message call but executes the code at the target address in the context of the calling contract. During this process, *msg.sender* and *msg.value* remain unchanged. Calling untrusted contracts is highly risky, as the code at the target address can modify any storage values of the caller and has complete control over the caller's balance. This defect mainly highlights the improper use of *delegatecall* and does not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

- Weak Randomness (SWC-120):

  "Weak Randomness" refers to the use of insecure sources of randomness in Ethereum. For example, using *block.timestamp* is insecure because a miner can choose any timestamp within a few seconds and still have their block accepted by others. Similarly, using fields like *blockhash* and *block.difficulty* is also insecure, as these are controlled by the miner. This defect primarily concerns the security of the source of randomness and does not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

- Timestamp Dependency (SWC-116):

  "Timestamp Dependency" means that a contract's logic relies on *block.timestamp* to make critical decisions. Since the value of *block.timestamp* can be influenced, albeit within limits, by miners, this reliance can lead to various security issues. As discussed in Section 1.1, this defect does not fall within the scope of centralization defects.

- Unprotected Ether Withdrawal (SWC-105):

  "Unprotected Ether Withdrawal" means that Ethers in the contract can be withdrawn without proper access control or authorization checks, potentially enabling unauthorized users to withdraw funds from the contract. This defect focuses on the access control of the ether withdrawal function and does not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

- Assert Violation (SWC-110):

  "Assert Violation" means that the use of the *assert* statement can lead to unexpected behavior or vulnerabilities. In Solidity, the *assert* statement is used to check for conditions that should never be false during execution. When an *assert* statement fails, it signals a bug in the code and consumes all remaining gas, reverting any changes made during the transaction. This defect mainly focuses on the improper use of assert statements and does

not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

- Reentrancy (SWC-107):

  "Reentrancy" means that an external contract calls back into the calling contract before the initial execution is complete. This can lead to unexpected behavior and potentially allow an attacker to manipulate the contract's state or drain its funds. This defect primarily emphasizes the risks associated with recursive calls and does not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

- Integer Overflow and Underflow (SWC-101):

  "Integer Overflow and Underflow" means that arithmetic operations exceed the maximum or minimum values that can be represented within a given data type, leading to unexpected behavior and potential security risks. This defect primarily emphasizes the range of values resulting from integer arithmetic operations and does not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

- Denial of Service with Failed Call (SWC-113):

  "Denial of Service with Failed Call" occurs when a contract's function call to an external contract fails, and the failure is not properly handled. This can lead to a denial of service (DoS), where the smart contract becomes unusable or its functionality is significantly impaired. This defect primarily emphasizes the unhandled failures in contract calls and does not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

- Unprotected SELFDESTRUCT (SWC-106):

  "Unprotected SELFDESTRUCT" means that the *selfdestruct* operation is accessible to unauthorized users. The SELFDESTRUCT opcode in Ethereum allows a contract to delete itself from the blockchain, sending its remaining Ether balance to a specified address. If this operation is not properly restricted, malicious actors can exploit it to destroy the contract and potentially steal its funds. As discussed in Section 1.2, this defect is not considered within the scope of centralization defects.

- Unchecked Call Return Value (SWC-104):

  "Unchecked Call Return Value" means that the return value of a low-level call (such as *call*, *delegatecall*, or *callcode*) is not checked. If these calls fail and their return values are not properly handled, it can lead to unexpected behavior, security issues, and potential loss of funds. As discussed in Section 1.3, this defect is not considered within the scope of centralization defects.

- Write to Arbitrary Storage Location (SWC-124):

  "Write to Arbitrary Storage Location" means that an attacker can manipulate the storage layout of a contract to write data to arbitrary storage locations. This can lead to unauthorized modifications of the contract's state, potentially causing significant security breaches and financial losses. This defect primarily emphasizes storage manipulation within the contract and does not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

- Arbitrary Jump with Function Type Variable (SWC-127)

  "Arbitrary Jump with Function Type Variable" occurs when a smart contract allows untrusted input to dictate the flow of execution via function type variables or function selectors. This can lead to unintended execution paths, unauthorized code execution, and various security issues. This defect primarily emphasizes the function type variables within the contract and does not relate to the behavior of any specific node within the contract. Therefore, it is not considered within the scope of centralization defects.

2.2 Slither

Source: The original paper (https://arxiv.org/pdf/1908.09878), and their Github repository (https://github.com/crytic/slither).

In the original paper on Slither, only reentrancy vulnerabilities were detected. As mentioned in Section 2.1, reentrancy is out of the scope of centralization defects. Slither is a static analysis framework for Solidity and Vyper. Many subsequent works have been developed based on this framework. Due to space limitations, we will not consider these subsequent works here.

2.3 Smartian

Source: The original paper (https://softsec.kaist.ac.kr/~sangkilc/papers/choi-ase2021.pdf), and their Github repository (https://github.com/SoftSec-KAIST/Smartian).

As illustrated in Figure 9, Smartian can detect 13 types of defects in smart contracts. We will now evaluate whether each of these defects falls within the scope of centralization defects based on their definitions. For defects that overlap with those detected by Mythril (even if the names differ), we will not provide further elaboration.

TABLE I
BUG CLASSES SUPPORTED BY SMARTIAN.

ID	Bug Name	Description
AF	Assertion Failure	The condition of an `assert` statement is not satisfied [2].
AW	Arbitrary Write	An attacker can overwrite arbitrary storage data by accessing a mismanaged array object [12].
BD	Block State Dependency	Block states (e.g. timestamp, number) decide ether transfer of a contract [36], [44].
CH	Control-flow Hijack	An attacker can arbitrarily control the destination of a `JUMP` or `DELEGATECALL` instruction [1], [36].
EL	Ether Leak	A contract allows an arbitrary user to freely retrieve ether from the contract [54].
FE	Freezing Ether†	A contract can receive ether but does not have any means to send out ether [36], [54].
IB	Integer Bug	Integer overflows or underflows occur, and the result becomes an unexpected value.
ME	Mishandled Exception	A contract does not check for an exception when calling external functions or sending ether [36], [44].
MS	Multiple Send	A contract sends out ether multiple times within one transaction. This is a specific case of DoS [5].
RE	Reentrancy	A function in a victim contract is re-entered and leads to a race condition on state variables [44].
RV	Requirement Violation‡	The condition of a `require` statement is not satisfied [8].
SC	Suicidal Contract	An arbitrary user can destroy a victim contract by running a `SELFDESTRUCT` instruction [54].
TO	Tranasaction Origin Use	A contract relies on the origin of a transaction (i.e. `tx.origin`) for user authorization [3].

Figure 9: All the defects supported by Smartian

- Assertion Failure:

  This is similar to Assert Violation (SWC-110) discussed in Section 2.1.

- Arbitrary Write:

  This is similar to Write to Arbitrary Storage Location (SWC-124) discussed in Section 2.1.

- Block State Dependency:

  This is similar to Timestamp Dependency (SWC-116) discussed in Section 2.1.

- Control-flow Hijack:

  This is similar to Delegatecall To Untrusted Callee (SWC-112) discussed in Section 2.1.

- Ether Leak:

  This is similar to Unprotected Ether Withdrawal (SWC-105) discussed in Section 2.1.

- Freezing Ether:

  "Freezing Ether" refers to a situation where Ether (ETH) becomes permanently locked in a contract, rendering it inaccessible and unusable. When Ether is frozen, it cannot be retrieved or utilized, potentially resulting in significant financial losses. This defect primarily highlights the absence of a withdrawal mechanism within the contract and does not relate to the behavior of any specific node within the contract. Consequently, it does not fall within the scope of centralization defects.

- Integer Bug:

  This is similar to Integer Overflow and Underflow (SWC-101) discussed in Section 2.1.

- Mishandled Exception:

  This is similar to Unchecked Call Return Value (SWC-104) discussed in Section 2.1.

- Multiple Send:

  This is similar to Denial of Service with Failed Call (SWC-113) discussed in Section 2.1.

- Reentrancy:

  This is similar to Reentrancy (SWC-107) discussed in Section 2.1.

- Requirement Violation:

  This is similar to Assert Violation (SWC-110) discussed in Section 2.1.

- Suicidal Contract:

  This is similar to Unprotected SELFDESTRUCT (SWC-106) discussed in Section 2.1.

- Transaction Origin Use:

  "Transaction Origin Use", often referred to as the *tx.origin* vulnerability, occurs when a smart contract relies on the global variable *tx.origin* for authorization or access control. This vulnerability can be exploited by attackers to bypass security checks and perform unauthorized actions. This defect primarily highlights the improper use of *tx.origin* and does not relate to the behavior of any specific node within the contract. Consequently, it does not fall within the scope of centralization defects.

In summary, Mythril, Slither, and Smartian are all incapable of detecting centralization defects in DeFi projects.

3. A Case Study for Reviewer#A-Q3

In this section, we would like to provide a case study of coexistence of multiple centralization defects within a single DeFi project.

DeFi project THE9 is an integrated blockchain payment system and innovative KIOSK Platform aim to provide seamless and efficient transactions. The smart contract in this project exhibits three distinct types of centralization defects: Mint Function with Single Signature (MFS), Critical Variables Manipulation with Single Signature (CVS), and Management without Timelock (MT). The subsequent sections will elaborate on the contract code corresponding to these centralization defects.

3.1 Mint Function with Single Signature (MFS)

As illustrated in Figure 10, the *createAmountWithLock* function in the THE9 contract is utilized to create new accounts for beneficiaries joining the project and to deposit a specified amount of funds into these accounts. Since the address of the new beneficiary and the amount to be deposited are provided as parameters to the function, they can be arbitrarily input by the function caller. This implies that the *createAmountWithLock* function can be invoked to mint any amount of tokens to any account. Furthermore, the two modifiers used in this function, *whenNotPaused* and *onlyRole (ORG_ADMIN_ROLE)*, ensure that the function can only be called by a pre-designated admin account when the contract is not paused. However, the address corresponding to the *ORG_ADMIN_ROLE* account in DeFi project THE9 is controlled by a single-signature node, which leads to a centralization defect: Mint Function with Single Signature.

This centralization defect may result in the malfunctioning of certain critical contract functions and could lead to severe financial losses. In the case of THE9, if the private key of the *ORG_ADMIN_ROLE* account is lost, it may prevent the creation of new accounts for new beneficiaries. Conversely, if the private key of the *ORG_ADMIN_ROLE* account is compromised, a hacker could exploit the minting function to generate a large number of tokens, thereby launching an attack that would affect the value and liquidity of THE9's token.

```
function createAmountWithLock(address beneficiary, uint256 amount, uint256 firstReleaseTime, uint256 unlockPercent, uint256 lockCycleDays)
    public
    whenNotPaused
    onlyRole(ORG_ADMIN_ROLE)
{
    if (firstReleaseTime == 0) {
        firstReleaseTime = DEFAULT_RELEASE_TIMESTAMP;
    }
    assert(_beforeSaveAmountWithLock(beneficiary, firstReleaseTime, unlockPercent, lockCycleDays));

    if (_checkExists(beneficiary)) {
        updateAmountWithLock(beneficiary, amount, firstReleaseTime, unlockPercent, lockCycleDays);
    } else {
        _setInfo(
            beneficiary,
            _calcDecimal(amount),
            firstReleaseTime,
            unlockPercent,
            lockCycleDays,
            100,
            _calcDecimal(amount)
        );
        Beneficiaries.push(beneficiary);

        emit CreateAmountWithLock(beneficiary, _getInfo(beneficiary));
    }
}
```

Figure 10: Function *createAmountWithLock* in DeFi project THE9

3.2 Critical Variables Manipulation with Single Signature (CVS)

As shown in Figure 11, the data structure *BeneficiaryInfo* in the THE9 contract is responsible for storing key information such as the amount of assets locked by the beneficiary in the project and the locking period. However, there is a function in the contract called *updateAmountWithLock* (as depicted in Figure 12) which can arbitrarily modify the values of critical variables in *BeneficiaryInfo* by invoking the internal function *_setInfo()*. Similar to the function *createAmountWithLock, updateAmountWithLock* can only be invoked by a single-signature node corresponding to the *ORG_ADMIN_ROLE* account, leading to a centralization vulnerability: Critical Variables Manipulation with Single Signature.

```
struct BeneficiaryInfo {
    uint256 amount;
    uint256 releaseTime;
    uint256 unlockPercent;
    uint256 lockCycleDays;
    uint256 remainPercent;
    uint256 remainAmount;
}
```

Figure 11: Data Structure *BeneficiaryInfo* in DeFi project THE9

```
function updateAmountWithLock(address beneficiary, uint256 amount, uint256 firstReleaseTime, uint256 unlockPercent, uint256 lockCycleDays)
    public
    whenNotPaused
    onlyRole(ORG_ADMIN_ROLE)
{
    require(_checkExists(beneficiary), "THE9: beneficiary not found");

    if (firstReleaseTime == 0) {
        firstReleaseTime = DEFAULT_RELEASE_TIMESTAMP;
    }
    assert(_beforeSaveAmountWithLock(beneficiary, firstReleaseTime, unlockPercent, lockCycleDays));

    BeneficiaryInfo memory beforeInfo = _getInfo(beneficiary);
    if(beforeInfo.remainPercent > 0 && beforeInfo.remainAmount > 0) {
        require(_getInfo(beneficiary).remainPercent >= 100, "THE9: account to which revenue was transferred");
    }

    _setInfo(
        beneficiary,
        _calcDecimal(amount),
        firstReleaseTime,
        unlockPercent,
        lockCycleDays,
        100,
        _calcDecimal(amount)
    );

    emit UpdateAmountWithLock(beneficiary, amount, firstReleaseTime, beforeInfo);
}
```

Figure 12: Function *updateAmountWithLock* in DeFi project THE9

3.3 Management without Timelock (MT)

As shown in Figure 13, the *pause* and *unpause* functions are utilized to manage the contract status or business logic by pausing or unpausing the execution of the contract. It is important to note that these functions can only be invoked by the *PAUSER_ROLE* account, and they do not incorporate a TimeLock mechanism to appropriately delay the effective time of changes in contract status. This delay would give users sufficient time to react to changes, such as withdrawing funds from DeFi projects. If the private key of the *PAUSER_ROLE* account is compromised, a hacker could exploit this authority to pause or unpause the contract at will. In such a scenario, users would have no time to respond to changes in the contract status.

```
/* public functions */
function pause() public onlyRole(PAUSER_ROLE) {
    _pause();
}

function unpause() public onlyRole(PAUSER_ROLE) {
    _unpause();
}
```

Figure 13: Function *pause* and *unpause* in DeFi project THE9

It is worth noting that the audit report from the blockchain security platform Certik identified centralization defects in the THE9 project. However, possibly due to cost considerations or other factors, the THE9 project team has not addressed these defects despite acknowledging them. For the sake of project security, we have opted not to provide a detailed explanation of the centralization defects present in this project within this paper.

The relevant information for this project is as follows:

- Project Website: https://www.the9company.io/

- Contract Address: 0xefd113cea2f656fe7899a25a06243a6e40406e08 (Ethereum)

- Audit Report from Certik: https://skynet.certik.com/zh-CN/projects/the-9

4. A Case Study for Reviewer#A-W1

In this section, we would like to present a real-world case of economic loss caused by a centralization defect in a DeFi project, serving as a motivational example for this work.

The DeFi project MUMI incurred an economic loss of approximately $35,000 due to the presence of a Mint Function with Single Signature (MFS) centralization defect in its contract. Specifically, the MUMI project's contract included a function for minting MUMI tokens that was controlled by a single node. An attacker exploited this function to secretly mint a large number of tokens into their own account. Utilizing these secretly minted tokens, the attacker drained assets from the liquidity pool. Specifically, the attacker used 416,483,104,164,831 MUMI tokens to obtain approximately 9.736 wETH, worth around $35,000. Beyond the financial loss, this issue severely impacted the liquidity of the project's corresponding token in the cryptocurrency market.

For more details on this incident, please refer to https://www.certik.com/zh-CN/resources/blog/the-vanishing-act-how-exit-scammers-mint-new-tokens-undetected.