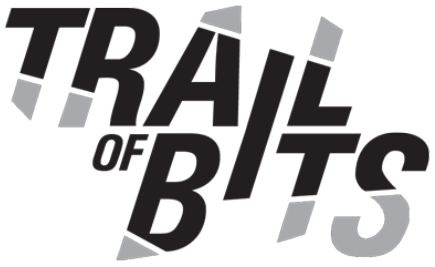

dForce Lending Protocol

Security Assessment

March 22, 2021

Prepared For:
Margaret Yao | *dForce*
myao@dforce.network

Snow Ji | dForce
snow@dforce.network

Prepared By:
Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Natalie Chin | *Trail of Bits*
natalie.chin@trailofbits.com

Executive Summary

From March 1 to March 19, 2021, dForce engaged Trail of Bits to review the security of the dForce Lending Protocol. Trail of Bits conducted this assessment over six person-weeks, with two engineers working from commit hash `419c2a1` from the `dforce-network/LendingContracts` repository.

During the first week of the engagement, we familiarized ourselves with the codebase and began a manual review focused on the lending pools and a tool-assisted review using Slither and Echidna. During the second week, we focused on the MSD stablecoin and lending pool flash loan functionality while continuing to develop Echidna tests. In the final week, we conducted an in-depth review of potential interactions between the stablecoin and lending pool components and addressed the concerns identified in the dForce-provided design document.

We identified 17 issues ranging from high to informational severity. The first high-severity issue concerns the possibility that an attacker could front-run the contract initialization process. Two others concern (respectively) the risks of storing private deployment keys in environment variables and a scenario in which permit signatures could be reused on multiple chains due to a network hard fork. The final high-severity finding describes how the lack of a contract existence check prior to `delegatecalls` could result in unexpected behavior if the implementation contract is set incorrectly.

One medium-severity issue highlights the benefits of a two-step role assignment process, and another concerns a mismatch between the code and specification that allows a larger-than-expected number of tokens to be liquidated. Two low-severity issues are related to project dependencies, and the other concerns poor error handling in the test suite. The informational-severity issues cover topics such as documentation improvements, code and test quality concerns, and the high number of privileges afforded to contract owners.

We discuss code quality concerns not related to any particular security issue in Appendix C. Appendix D introduces a script that can be used to help ensure all functions are protected with `nonReentrant` modifiers. Appendix E contains a script intended to ensure that all functions work as expected if a market is paused. Appendix F lists points to take into account when considering adding support for new assets.

Overall, the dForce codebase, a complex codebase with many interacting components, is a work in progress. There are gaps between the information provided in the documentation and the implementation, and mathematical formulas lack derivations. As a result, the necessity and impact of the code's invariants are often unclear. Additionally, the code has

numerous unused parameters, dead code, and unused dependencies, which hampered the efficiency of the review.

While the system appears to be feature-complete, before further deployment, dForce should address the concerns in this report, including improving the documentation and refactoring to simplify the codebase where possible. Due to the complexity of the system, we would also suggest conducting an additional security review after addressing the issues and code quality concerns identified in this report.

Project Dashboard

Application Summary

Name	dForce Lending Protocol
Version	`419c2a1`
Type	Smart Contracts
Platforms	Solidity

Engagement Summary

Dates	March 1– March 19, 2021
Method	Whitebox
Consultants Engaged	2
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	4	■ ■ ■ ■
Total Medium-Severity Issues	2	■ ■
Total Low-Severity Issues	3	■ ■ ■
Total Informational-Severity Issues	8	■ ■ ■ ■ ■ ■ ■ ■
Total Undetermined-Severity Issues	0	
Total	17	

Category Breakdown

Access Controls	2	■ ■
Authentication	1	■
Configuration	5	■ ■ ■ ■ ■
Data Validation	4	■ ■ ■ ■
Documentation	1	■
Error Reporting	1	■
Patching	2	■ ■
Undefined Behavior	1	■

Total	17	

Code Maturity Evaluation

In the table below, we review the maturity of the codebase and the likelihood of future issues. We rate the maturity of each area of control from strong to weak and briefly explain our reasoning.

Category Name	Description
Access Controls	**Moderate.** The dForce Lending code heavily relied on privileged operations. Roles were not split between different addresses or clearly explained to users in documentation.
Arithmetic	**Moderate.** `SafeMath` and `SafeRatioMath` were used throughout the system. However, these arithmetic operations were not tested with automated analysis tools. The mathematics throughout the system were somewhat documented, but certain sections included significant deviations. Additionally, many functions were undocumented, and their derivations and choice were unclear. (TOB-DFL-009)
Assembly Use/Low Level	**Moderate.** The contracts did not contain assembly usage, but the lack of a contract existence check on the proxy could allow a call to fail silently. (TOB-DFL-011)
Centralization	**Weak.** The dForce Lending architecture is controlled by a single owner through a single private key (TOB-DFL-007). All privileged roles are controlled by this owner (TOB-DFL-010). No user documentation on deployment or centralization risks was provided.
Contract Upgradeability	**Moderate.** The code uses migrations and upgrades to update code logic. For upgrades, the `ProxyAdmin` contract is mostly untested. Additionally, the state variable layout between upgrades must be done manually. `slither-check-upgradeability` is not integrated into a continuous integration pipeline.
Function Composition	**Weak.** While most functions were documented inline, the contract architecture used overly complex inheritance, making it very difficult to understand the code. These functions should be collapsed to increase readability.
Front-Running	**Moderate.** Front-running issues were identified in the initialization process (TOB-DFL-001), and the `permit` call could be front-run by an attacker. (TOB-DFL-005)
Key Management	**Weak.** The owner of the contract is set to a single private key. Key management is a critical point of failure in the system. Exploitation

	of the owner's private key would allow an attacker to steal all funds. (TOB-DFL-007)
Monitoring	**Satisfactory.** To enable effective off-chain monitoring, events are present for all critical operations. However, we were not provided with an incident response plan or information on how off-chain components would be used to track behavior.
Specification	**Moderate.** There were deviations between the codebase specification and the code, but the code and documentation followed naming conventions. (TOB-DFL-009, TOB-DFL-012)
Testing & Verification	**Weak.** The testing suffers due to the system's architecture and bad practices. While there is 99% statement coverage and 96% branch coverage, the tests are not robust, as the tests silently fail (TOB-DFL-014). The test suite would benefit from better testing practices and continuous integration across all branches to pre-detect code that could introduce unexpected behavior. (TOB-DFL-015)

Engagement Goals

The engagement was scoped to provide a security assessment of the smart contracts that form the dForce Lending protocol at commit hash `419c2a1` from the `dforce-network/LendingContracts` repository.

Specifically, we sought to answer the following questions:

- Is it possible for participants to steal or lose tokens?
- Are appropriate access controls set for system components?
- Are there any circumstances under which arithmetic overflow could affect the system?
- Can participants perform denial-of-service or phishing attacks against any of the system components?
- Are interest and asset exchange rates calculated correctly?
- Can flash loans be used to exploit the protocol?
- Are there any negative interactions between the lending pools and the stablecoin?

Coverage

Lending Pools. The lending pools are structured as a central `Controller` contract that connects many `iToken` markets representing different assets. We reviewed the lending pool functionality to ensure markets adhere to their parameters, interest is accrued properly, account equity is calculated correctly, and liquidation logic is sound.

MSD Stablecoin. As the stablecoin is backed by the lending pool, the structure of the contracts borrows heavily from the `iToken` contract. We reviewed the `iMSD` contract to ensure that differences in functionality did not introduce any issues. We also reviewed the `MSD` saving token implementation to check that tokens are minted and burned properly and that interest is accrued correctly.

Arithmetic. The accuracy and soundness of arithmetic is crucial to a lending protocol. We reviewed the interest rate calculations to ensure that borrow, supply, and asset exchange rates are calculated properly and assessed the custom `SafeMath` functionality for correctness.

Access Controls. Most system parameters have setter functions that allow their values to be modified post-deployment. We reviewed this privileged functionality to ensure that only the intended users can modify system and market parameters.

Out of Scope

The `PriceOracle.sol` contract was out of the scope of this assessment.

After the audit, dForce mentioned that portions of its application are formally verified by Certora. These properties and provers were out of the scope of this audit.

Automated Testing and Verification

Trail of Bits used automated testing techniques to enhance coverage of certain areas of the contracts, including:

- Slither, a Solidity static analysis framework.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation.

Automated testing techniques augment our manual security review but do not replace it. Each method has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode; Echidna may not randomly generate an edge case that violates a property.

Automated Testing with Echidna

By reviewing the codebase and design document, we identified various properties or invariants that should hold. Using Echidna, we implemented the following property-based tests:

End-to-End Properties

ID	Property	Result
1	The reserve ratio fee cannot exceed `1e18`.	PASSED
2	The flash loan fee cannot exceed `1e18`.	PASSED
3	The protocol fee cannot exceed `1e18`.	PASSED
4	Zero address should not hold underlying tokens.	PASSED
5	Zero address should not be able to borrow from the `Controller`.	PASSED

Scenario-Based Properties

ID	Property	Result
6	No more than 50% of the loan can be liquidated.	**FAILED** TOB-DFL-012

7	Repaying a loan of the borrowed amount should result in the same principal.	PASSED
8	Repaying a loan of the borrowed amount should zero out totalBorrowsRepay.	PASSED
9	Repaying less than the amount borrowed should result in a lower principal.	PASSED
10	Repaying less than the amount borrowed should decrease `totalBorrowsRepay`.	PASSED
11	Update interest should change only on the first call in a block.	PASSED

Slither Scripts

ID	Property	Script
1	All necessary functions have `nonReentrant` modifiers.	APPENDIX D
2	All necessary functions use market-pausable values.	APPENDIX E

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Either use a factory pattern that will prevent front-running or ensure the deployment scripts are robust in the event of a front-running attack.** That way, dForce will be alerted to front-running of the initialization functions, or front-running will be prevented. TOB-DFL-001

❑ **Use a two-step process for all non-recoverable critical operations to prevent irrevocable mistakes.** TOB-DFL-002

❑ **Ensure that dependencies are up to date.** Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation. TOB-DFL-003

❑ **Review the codebase and document the source and version of each dependency.** Include third-party sources as submodules in your Git repository so that internal path consistency is maintained and dependencies are periodically updated. TOB-DFL-004

❑ **Document the risks of `permit` front-running and of a phishing attack and ensure that external contracts and scripts reflect these possibilities.** This will reduce the likelihood of a successful phishing attempt. TOB-DFL-005

❑ **Add zero-value checks on all function arguments to ensure that users can't accidentally set incorrect values.** These checks will prevent incorrect configuration of the system. TOB-DFL-006

❑ **Replace the owner of the contracts with a multisig wallet requiring M-of-N signatures prior to protocol changes.** This will ensure that the leakage of one secret is unlikely to affect the overarching system. TOB-DFL-007

❑ **Add the chainID opcode to the signature schema to prevent** post-deployment forks from affecting this permit. TOB-DFL-008

❑ **Review and properly document the derivation of the interest rate models and curves, derivation of liquidation formulas, and absorption of dust.** TOB-DFL-009

❑ **Clearly document the functions and implementations the owner can change.** Split privileges to ensure that no one address has excessive ownership of the system. TOB-DFL-010

❑ **Check for contract existence before a `delegatecall`.** Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions. TOB-DFL-011

❑ **Determine what the correct percentage of a loan can be liquidated and update the code or documentation accordingly.** This will help ensure that protocol participants are informed of the risks of liquidation. TOB-DFL-012

❑ **Document the risks of transactions running out of gas so that users know what to do if a transaction fails.** TOB-DFL-013

❑ **Test these operations against a specific error message.** Testing will ensure that errors are never silenced, and the test suite will check that a contract call has reverted for the right reason. TOB-DFL-014

❑ **Remove the non-hardhat network sleeps or document why they are required.** This will make it easier to integrate third-party testing tools into the codebase. TOB-DFL-015

❑ **Analyze the effects of a deviation in this value, and document the risks so that users are aware of them prior to using the system.** TOB-DFL-016

❑ **Introduce a specific flash loan repayment flow into the system, or use a flash loan lock to ensure that certain operations cannot occur when a flash loan is taken out.** TOB-DFL-017

Long Term

❑ **Carefully review the Solidity documentation, especially the "Warnings" section, as well as** the pitfalls of using the `delegatecall` proxy pattern. TOB-DFL-001, TOB-DFL-011

❑ **Identify and document all possible actions and their associated risks for privileged accounts.** Identifying the risks will facilitate codebase reviews and help prevent future mistakes. TOB-DFL-002

❑ **Consider integrating automated dependency auditing into the development workflow.** If dependencies cannot be updated when a vulnerability is disclosed, ensure

that the dForce Lending codebase does not use and is not affected by the vulnerable functionality of the dependency. TOB-DFL-003

❑ **Use an Ethereum development environment and NPM to manage packages as part of your project.** TOB-DFL-004

❑ **Document best practices for dForce users.** Users must be extremely careful when signing a message, avoid signing messages from suspicious sources, and always require hashing schemes to be public, in addition to taking other precautions. TOB-DFL-005

❑ **Use Slither, which will catch functions that do not have zero checks.** TOB-DFL-006

❑ **Use a hardware security module (HSM), which will ensure that the private key never leaves the module.** TOB-DFL-007

❑ **Identify and document the risks associated with having forks of multiple chains, and identify related mitigation strategies.** TOB-DFL-008

❑ **Consider writing an updated formal specification of the protocol.** TOB-DFL-009

❑ **Document the risks associated with privileged users and single points of failure.** Ensure that users are aware of all the risks associated with the system. TOB-DFL-010

❑ **Use Echidna or Manticore to ensure that arithmetic invariants hold.** TOB-DFL-012

❑ **Identify all areas of the system that would be affected by calls in loop and ensure that users know what to do if one occurs.** TOB-DFL-013

❑ **Follow standard testing practices for smart contracts to minimize the number of issues during development.** TOB-DFL-014

❑ **Integrate a better workaround to facilitate access to tooling.** This will ensure that the dForce system can easily be tested with third-party tools such as Echidna. TOB-DFL-015

❑ **Identify all variables that are affected externally, and document the risks associated with deviations.** TOB-DFL-016

❑ **Integrate Slither into a continuous integration pipeline to identify reentrant code.** TOB-DFL-017

Findings Summary

#	Title	Type	Severity
1	Initialization functions can be front-run	Configuration	High
2	Lack of two-step process for critical operations leads to errors	Data Validation	Medium
3	Project dependencies contain vulnerabilities	Patching	Low
4	Contracts used as dependencies do not track upstream changes	Patching	Low
5	Risks associated with EIP 2612	Configuration	Informational
6	Lack of zero check on functions	Data Validation	Informational
7	Insufficient protection on sensitive owner private keys	Configuration	High
8	Lack of chainID validation allows reuse of signatures across forks	Authentication	High
9	Lack of contract documentation makes codebase difficult to understand	Documentation	Informational
10	Contract owner has too many privileges	Access Controls	Informational
11	Lack of contract existence check on `delegatecall` will result in unexpected behavior	Access Controls	High
12	Liquidators can liquidate more than 50% of loan	Data Validation	Medium
13	External calls in loops may result in denial of service	Data Validation	Informational
14	Poor error-handling practices in test suite	Error Reporting	Low
15	Added code in fixtures makes third-party integrations more difficult	Configuration	Informational
16	Borrow rate depends on approximation of blocks per year	Configuration	Informational

17	Removal of nonReentrant modifier on flashLoan may have unintended consequences	Undefined Behavior	Informational

1. Initialization functions can be front-run

Severity: High
Type: Configuration
Target: `contracts/`

Difficulty: High
Finding ID: TOB-DFL-001

Description
Several implementation contracts have initialization functions that can be front-run, which could allow an attacker to incorrectly initialize the contracts.

Due to the use of the `delegatecall` proxy pattern, `MSDController`, `Controller`, `RewardDistributor`, `LendingData`, and `MSDS` cannot be initialized with a constructor and have initializer functions:

```
/**
 * @notice Expects to call only once to initialize the MSD controller.
 */
function initialize() external initializer {
    __Ownable_init();
}
```

Figure 1.1: `contracts/msd/MSDController.sol#L60-L65`

The `init` function allows the caller to be the owner of all of these contracts:

```
/**
 * @dev Initializes the contract setting the deployer as the initial owner.
 */
function __Ownable_init() internal {
    owner = msg.sender;
    emit NewOwner(address(0), msg.sender);
}
```

Figure 1.2: `contracts/library/Ownable.sol#L37-L43`

These functions can be front-run by an attacker, allowing the attacker to initialize the contracts with malicious values.

Exploit Scenario
Bob deploys the `Controller` contract. Eve front-runs the `Controller` initialization and sets her own address for the `owner`. As a result, she has admin control on all functions protected by an `onlyOwner` modifier.

Recommendation
Short term, either use a factory pattern that will prevent front-running of the initialization or ensure the deployment scripts are robust in the event of a front-running attack.

Long term, carefully review the Solidity documentation, especially the “Warnings” section, as well as the pitfalls of using the `delegatecall` proxy pattern.

2. Lack of two-step process for critical operations

Severity: Medium
Type: Data Validation
Target: `contracts/Controller.sol`

Difficulty: High
Finding ID: TOB-DFL-002

Description
Several critical operations are executed in one function call. This schema is error-prone and can lead to irrevocable mistakes.

For example, the `pauseGuardian` variable defines the address that can pause the mint, borrow, liquidate, and transfer processes in case of an emergency. The setter function for this address immediately sets the new guardian address:

```
    /**
     * @notice Sets the pauseGuardian
     * @dev Admin function to set pauseGuardian
     * @param _newPauseGuardian The new pause guardian
     */
    function _setPauseGuardian(address _newPauseGuardian)
        external
        override
        onlyOwner
    {
        address _oldPauseGuardian = pauseGuardian;

        require(
            _newPauseGuardian != address(0) &&
                _newPauseGuardian != _oldPauseGuardian,
            "Pause guardian address invalid"
        );

        pauseGuardian = _newPauseGuardian;

        emit NewPauseGuardian(_oldPauseGuardian, _newPauseGuardian);
    }
```

Figure 2.1: `contracts/Controller.sol#L401-L421`

As a result, if the address is incorrect, critical aspects of the protocol could be changed by an arbitrary address, rather than by the contract's owner.

Exploit Scenario
Alice, a member of the dForce team, sets a new address as the `pauseGuardian`. She accidentally sets the address as that of an attacker, Eve. Alice must immediately revoke access by calling `_setPauseGuardian`; otherwise, Eve could pause the protocol.

Recommendation
Short term, use a two-step process for all non-recoverable critical operations to prevent irrevocable mistakes.

Long term, identify and document all possible actions and their associated risks for privileged accounts. Identifying the risks will facilitate codebase reviews and help prevent future mistakes.

3. Project dependencies contain vulnerabilities

Severity: Low
Type: Patching
Target: `contracts/`

Difficulty: Low
Finding ID: TOB-DFL-003

Description
Although dependency scans did not yield a direct threat to the dForce codebase, `yarn audit` has identified dependencies with known vulnerabilities. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the dForce Lending system as a whole. The following output details these issues:

NPM Advisory	Description	Dependency
877	Insecure Credential Storage	`web3`
1500	Prototype Pollution	`yars-parser`
1556	Denial of Service	`node-fetch`

Table 3.1: NPM advisories affecting dForce's dependencies.

Exploit Scenario
Alice installs the dependencies for dForce Lending on a clean machine. Unbeknownst to Alice, a dependency of the project has become malicious or exploitable. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendation
Short term, ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If dependencies cannot be updated when a vulnerability is disclosed, ensure that the dForce Lending codebase does not use and is not affected by the vulnerable functionality of the dependency.

4. Contracts used as dependencies do not track upstream changes

Severity: Low
Type: Patching
Target: `contracts/library/*`

Difficulty: Low
Finding ID: TOB-DFL-004

Description
Several third-party contracts have been copied and pasted into the dForce Lending repository, including into files in the `library/` directory. The code documentation does not specify the exact revision that was made or whether the code was modified. Contracts will not reliably receive updates and security fixes on these dependencies, as the dependencies must be updated manually.

Exploit Scenario
A third-party contract used in dForce Lending receives an update with a critical fix for a vulnerability. An attacker detects the use of a vulnerable contract and can exploit the vulnerability against any of the contracts in the `library`.

Recommendations
Short term, review the codebase and document the source and version of each dependency. Include third-party sources as submodules in your Git repository so that internal path consistency is maintained and dependencies are updated periodically.

Long term, use an Ethereum development environment and NPM to manage packages as part of your project.

5. Risks associated with EIP 2612

Severity: Informational
Type: Configuration
Target: `contracts/msd/MSD.sol`

Difficulty: High
Finding ID: TOB-DFL-005

Description
The use of EIP 2612 increases the risk of permit function front-running and of phishing attacks.

EIP 2612 replaces the traditional `approve` and `transferFrom` flow with signatures. These signatures allow a third party to transfer tokens on behalf of a user, with verification of a signed message.

An external party can front-run the `permit` function by submitting the signature first:

```
    /**
     * @dev EIP2612 permit function. For more details, please look at here:
     * https://eips.ethereum.org/EIPS/eip-2612
     * @param _owner The owner of the funds.
     * @param _spender The spender.
     * @param _value The amount.
     * @param _deadline The deadline timestamp, type(uint256).max for max deadline.
     * @param _v Signature param.
     * @param _s Signature param.
     * @param _r Signature param.
     */
    function permit(
        address _owner,
        address _spender,
        uint256 _value,
        uint256 _deadline,
        uint8 _v,
        bytes32 _r,
        bytes32 _s
    ) external {
        [...]
    }
```

Figure 5.1: `contracts/msd/MSD.sol#L118-L164`

The use of EIP 2612 makes it possible for a different party to front-run the initial caller's transaction. As a result, the intended caller's transaction will fail (as the signature has already been used and the funds, approved). This may also affect external contracts that rely on a successful `permit()` for execution.

EIP 2612 also makes it easier for an attacker to steal a user's tokens by running a phishing campaign asking for signatures in a context unrelated to the dForce contracts. The hash message may look benign and random to users:

```
        uint256 _currentNonce = nonces[_owner];
        bytes32 _digest =
            keccak256(
                abi.encodePacked(
                    "\x19\x01",
                    DOMAIN_SEPARATOR,
                    keccak256(
                        abi.encode(
                            PERMIT_TYPEHASH,
                            _owner,
                            _spender,
                            _value,
                            _currentNonce,
                            _deadline
                        )
                    )
                )
            );
        address _recoveredAddress = ecrecover(_digest, _v, _r, _s);
        require(
            _recoveredAddress != address(0) && _recoveredAddress == _owner,
            "permit: INVALID_SIGNATURE!"
        );
```

Figure 5.2: `contracts/msd/MSD.sol#L118-L164`

Exploit Scenario

Bob has 1,000 iTokens. Eve creates an ERC20 token with a malicious airdrop called ProofOfSignature. To claim the tokens, the participants must sign a hash. Eve generates a hash to transfer 1,000 iTokens from Bob. Eve asks Bob to sign the hash to get free tokens. Bob signs the hash, and Eve uses it to steal Bob's tokens.

Recommendations

Short term, either

- document the risk of `permit` being front-run and ensure that external contracts and scripts reflect this possibility, or
- reduce the likelihood of a successful phishing campaign by adding user documentation and on-chain mitigations.

Long term, document best practices for dForce users. In addition to taking other precautions, users must

- be extremely careful when signing a message,
- avoid signing messages from suspicious sources, and
- always require hashing schemes to be public.

References

- EIP 2612's Security Implications

6. Lack of zero check on functions

Severity: Informational
Type: Data Validation
Target: `contracts/TokenBase/TokenAdmin`

Difficulty: High
Finding ID: TOB-DFL-006

Description
Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

For example, the `_setController` function in `TokenAdmin` sets the Controller address meant to interact with the tokens to calculate amounts:

```
/**
 * @notice Initializes the contract.
 */
function initialize(Controller _controller) external initializer {
    __Ownable_init();
    controller = _controller;
    paused = true;
}
```

Figure 6.1: `contracts/RewardDistributor.sol#L76-L83`

Immediately after this address has been set to `address(0)`, the admin must reset the value; failure to do so may result in unexpected behavior for the contract.

This issue is also prevalent in the following contracts:

- `RewardDistributor.initialize`: `_controller`.
- `iETH.initialize`: `_controller, _interestRateModel`.
- `iToken.initialize`: `_underlyingToken, _controller, _underlyingToken`.
- `MSDS.initialize`: `_underlying, _interestRateModel, _msdController`.
- `iMSD.initialize`: `_underlyingToken, _lendingController, _interestRateModel, _msdController`.

Exploit Scenario
Alice deploys a new version of the `Controller`. When she invokes the `Controller` address setter to replace the address, she accidentally enters the zero address.

Recommendations
Short term, add zero-value checks on all function arguments to ensure users can't accidentally set incorrect values, causing incorrect configuration of the system.

Long term, add zero-value checks for all user-supplied arguments to ensure that they are not `address(0)` to prevent user mistakes.

7. Insufficient protection on sensitive owner private keys

Severity: High
Type: Configuration
Target: `hardhat.config.js`

Difficulty: Medium
Finding ID: TOB-DFL-007

Description
Sensitive information such as owner private keys and API keys is stored in the process environment. This increases the likelihood of compromise, which would allow an attacker to steal funds from the protocol with owner privileges.

The following portion of the `hardhat.config.js` uses secrets directly from the process environment:

```
const privateKey = process.env.PRIVATE_KEY;
const infuraKey = process.env.INFURA_KEY;
[...]
mainnet: {
   url: `https://mainnet.infura.io/v3/${infuraKey}`,
   accounts: [`0x${privateKey}`],
   gas: 8000000,
   gasPrice: 1000000000, // 1gWei
   timeout: 200000,
 },
  // TODO: there is an unexpected case when tries to verify contracts, so do not use it at
now!!!
  etherscan: {
    apiKey: process.env.ETHERSCAN_KEY,
  },
```

Figure 7.1: `hardhat.config.js#L17-L80`

Exploit Scenario
Alice, a member of the dForce team, has secrets stored in the process environment. Eve, an attacker, gains access to Alice's device and extracts the private key. This gives her owner access to all of the functions in the protocol and enables her to update the protocol contracts to steal all funds.

Recommendation
Short term, replace the owner of the contracts with a multisig wallet requiring M-of-N signatures prior to protocol changes. This will ensure that the leakage of one secret is unlikely to affect the overarching system.

Long term, use a hardware security module (HSM), which will ensure that the private key never leaves the module.

References

- Cryptocurrency hardware wallets

8. Lack of chainID validation allows reuse of signatures across forks

Severity: High | Difficulty: High
Type: Authentication | Finding ID: TOB-DFL-008
Target: `contracts/msd/{MSD.sol, MSDS.sol}, contracts/TokenBase/Base.sol`

Description
The dForce Lending protocol token contracts implement EIP 2612 to provide EIP 712-signed approvals through a `permit` function. A domain separator and the chainID are included in the signature scheme. However, this chainID is fixed at the time of deployment. In the event of a post-deployment chain hard fork, the chainID cannot be updated, and signatures may be replayed across both versions of the chain. As a result, an attacker could reuse signatures to receive user funds on both chains. Explicitly including the chainID in the schema of the signature passed to permit would mitigate this issue and prevent the need to regenerate the entire domain separator.

The following shows the `permit` function using `ecrecover`:

```
    /**
     * @dev EIP2612 permit function. For more details, please look at here:
     * https://eips.ethereum.org/EIPS/eip-2612
     * @param _owner The owner of the funds.
     * @param _spender The spender.
     * @param _value The amount.
     * @param _deadline The deadline timestamp, type(uint256).max for max deadline.
     * @param _v Signature param.
     * @param _s Signature param.
     * @param _r Signature param.
     */
    function permit(
        address _owner,
        address _spender,
        uint256 _value,
        uint256 _deadline,
        uint8 _v,
        bytes32 _r,
        bytes32 _s
    ) external {
        [...]
    }
```

Figure 8.1: `contracts/msd/MSD.sol#L118-L164`

The signature schema does not account for the contract's chain. As a result, if a fork of Ethereum is made after the contract's creation, every signature will be usable in both forks.

Exploit Scenario
Bob holds tokens worth $1,000 on Eth1.0. Bob has submitted a signature to permit Eve to spend these tokens on his behalf. With Eth 2.0, a fork of mainnet is performed. As a result, Eve can use Bob's signature to transfer funds on the new chain and the old chain.

Recommendation
Short term, add the chainID opcode to the signature schema to prevent post-deployment forks from affecting this permit.

Long term, identify and document the risks associated with having forks of multiple chains, and identify related mitigation strategies.

9. Lack of contract documentation makes codebase difficult to understand

Severity: Informational
Type: Documentation
Target: `contracts`

Difficulty: Low
Finding ID: TOB-DFL-009

Description
Overall, the codebase lacks code documentation, high-level descriptions, and examples, making the contracts difficult to review and increasing the likelihood of user mistakes.

The documentation would benefit from more details, including on the derivation of the formula for the interest rate models and the purpose of having multiple curves, as well as the derivation of the formula for liquidations. Also consider detailing the absorption of dust from user transactions into the protocol when a user repays a loan, small amounts of ETH are minted, or other "donations" to the protocol are made.

Additionally, the documentation contains outdated information. Updated information on utilization ratio, borrowing rate, and supply rate formulas would make the code much easier to follow.

The documentation on each of these items should include their expected properties and assumptions.

Recommendation
Short term, review and properly document the above mentioned items.

Long term, consider writing an updated formal specification of the protocol.

10. Contract owner has too many privileges

Severity: Informational
Type: Access Controls
Target: `contracts`

Difficulty: Medium
Finding ID: TOB-DFL-010

Description
The owner of the contracts has too many privileges relative to standard users. Users can lose all of their assets if a contract owner private key is compromised.

The contract owner can do the following:

- Upgrade the system's implementation to steal funds
- Upgrade the token's implementation to act maliciously
- Increase the amount of iTokens for reward distribution to such an extent that rewards cannot be disbursed
- Arbitrarily update the interest model contracts

The concentration of these privileges creates a single point of failure. It increases the likelihood that the owner will be targeted by an attacker, especially given the insufficient protection on sensitive owner private keys (TOB-DFL-007). Additionally, it incentivizes the owner to act maliciously.

Exploit Scenario
Alice deploys these contracts. Over time, the contracts' assets reach a value of $20,000,000. Eve gains access to Alice's machine, upgrades the implementations, and steals all of her funds.

Recommendation
Short term:

- Clearly document the functions and implementations the owner can change.
- Split privileges to ensure that no one address has excessive ownership of the system.

Long term, document the risks associated with privileged users and single points of failure. Ensure that users are aware of all the risks associated with the system.

11. Lack of contract existence check on `delegatecall` will result in unexpected behavior

Severity: High
Difficulty: High
Type: Access Controls
Finding ID: TOB-DFL-011
Target: `contracts/library/ProxyAdmin.sol`

Description
The `upgradeAndCall` function uses the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or is self-destructed, the proxy can cause unexpected behavior.

The `ProxyAdmin` contract uses `TransparentUpgradeableProxy`, which inherits from a chain of Open Zeppelin contracts such as `UpgradeableProxy` and `Proxy`. Eventually, arbitrary calls are executed by the `_fallback` function in the proxy, which lacks a contract existence check:

```
    /**
     * @dev Delegates the current call to the address returned by `_implementation()`.
     *
     * This function does not return to its internall call site, it will return directly to
the external caller.
     */
    function _fallback() internal {
        _beforeFallback();
        _delegate(_implementation());
    }

    /**
     * @dev Fallback function that delegates calls to the address returned by
`_implementation()`. Will run if no other
     * function in the contract matches the call data.
     */
    fallback () external payable {
        _fallback();
    }
```

Figure 11.1: `Proxy.sol#L49-L65`

`TransparentUpgradeableProxy` defines the `_beforeFallback` function, which also lacks an existence check:

```
    /**
     * @dev Makes sure the admin cannot access the fallback function. See
{Proxy-_beforeFallback}.
     */
    function _beforeFallback() internal override virtual {
        require(msg.sender != _admin(), "TransparentUpgradeableProxy: admin cannot fallback
to proxy target");
        super._beforeFallback();
    }
```

Figure 11.2: `TransparentUpgradeProxy.sol#L146-L152`

As a result, a `delegatecall` to a destructed contract will return success as part of the EVM specification. The Solidity documentation includes the following warning:

> *The low-level call, delegatecall and callcode will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.*

Figure 11.3: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`.

The proxy will not throw an error if its implementation is incorrectly set or self-destructed. It will instead return success even though no code was executed.

Exploit Scenario
Eve upgrades the proxy to point to an incorrect new implementation. As a result, all the `delegatecalls` return success without changing the state or executing code. Eve uses this failing to scam users.

Recommendation
Short term, check for contract existence before a `delegatecall`. Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.

Long term, carefully review the Solidity documentation, especially the “Warnings” section, and the pitfalls of using the `delegatecall` proxy pattern.

References
- Contract upgrade anti-patterns

12. Liquidators can liquidate more than 50% of loan

Severity: Medium
Type: Data Validation
Target: `contracts/Controller.sol`

Difficulty: Low
Finding ID: TOB-DFL-012

Description
The documentation states that liquidators should be able to liquidate 50% of a loan at the most, but the implementation allows liquidation of up to 90%.

According to documentation provided by dForce, if a loan is undercollateralized, 50% of it can be liquidated:

```
The liquidationBorrow action allows any external actor to purchase part of a collateral at a
discounted price, a maximum of 50% of the loan can be liquidated.
```

Figure 12.1: dForce Documentation

The `liquidateCalculateSeizeTokens` function determines how many of the borrower's tokens should be seized and sent to the liquidator:

```
    // closeFactorMantissa must be strictly greater than this value
    uint256 internal constant closeFactorMinMantissa = 0.05e18; // 0.05

    // closeFactorMantissa must not exceed this value
    uint256 internal constant closeFactorMaxMantissa = 0.9e18; // 0.9

    function liquidateCalculateSeizeTokens(
        uint256 borrowBalance,
        uint256 closeFactorMantissa
    ) internal returns (uint256) {
        // Only allowed to repay the borrow balance's close factor
        uint256 maxRepay = borrowBalance.rmul(closeFactorMantissa);
        return maxRepay;
    }

    function exploit_liquidate(
        uint256 borrowBalance,
        uint256 mantissa
    ) public  {
        require(mantissa > closeFactorMinMantissa);
        require(mantissa < closeFactorMaxMantissa);
```

```
        uint256 amountToSeize = liquidateCalculateSeizeTokens(
            borrowBalance, mantissa
        );
        assert(amountToSeize <= borrowBalance.div(2));
    }
```

Figure 12.2: `Echidna property`

However, this assertion fails, because the close factor mantissa is bound between [5% and 90%] inclusive.

```
assertion in exploit_liquidate: failed!💥
  Call sequence:
    exploit_liquidate(16076921520843,500180178193676777)
```

Figure 12.3: `Echidna property`

Exploit Scenario
Alice borrows 16,076,921,520,843 iTokens from Eve. However, her account has a shortfall, so her collateral can be liquidated. Eve liquidates more than 50% of the loan at a heavily discounted rate.

Recommendation
Short term, either

- update the documentation to reflect the correct bounds of the mantissa, or
- update the code such that it includes a 50% limit on liquidation in accordance with the documentation.

Long term, use Echidna or Manticore to ensure that arithmetic invariants hold.

13. External calls in loops may result in denial of service

Severity: Informational
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-DFL-013
Target: `contracts/RewardDistributor.sol`

Description
The use of external calls in nested loops and subsequent loops, iterating over lists provided by callers, may result in an out-of-gas failure during execution.

To claim rewards on a token, the code iterates over all tokens and then over all holders to update their reward states. It then performs another iteration over the holders to transfer their funds to users:

```
    /**
     * @notice Claim reward accrued in iTokens by the holders
     * @param _holders The account to claim for
     * @param _iTokens The _iTokens to claim from
     */
    function claimReward(address[] memory _holders, address[] memory _iTokens)
        public
        override
    {
        // Update rewards for all _iTokens for holders
        for (uint256 i = 0; i < _iTokens.length; i++) {
            address _iToken = _iTokens[i];
            _updateDistributionState(_iToken, false);
            _updateDistributionState(_iToken, true);
            for (uint256 j = 0; j < _holders.length; j++) {
                _updateReward(_iToken, _holders[j], false);
                _updateReward(_iToken, _holders[j], true);
            }
        }

        // Withdraw all reward for all holders
        for (uint256 j = 0; j < _holders.length; j++) {
            address _account = _holders[j];
            uint256 _reward = reward[_account];
            if (_reward > 0) {
                reward[_account] = 0;
                IERC20Upgradeable(rewardToken).safeTransfer(_account, _reward);
```

```
            }
        }
```

Figure 13.1: contracts/RewardDistributor.sol#L397-L421

Exploit Scenario
Alice executes a claim reward for many tokens and holders. The execution runs out of gas during computation, and she must resubmit the executions with a smaller list of tokens and holders.

Recommendation
Short term, document the risks so that users know what to do if a transaction fails because it has run out of gas.

Long term, identify all areas of the system that would be affected by calls in loop and ensure that users know what to do if one occurs.

14. Poor error-handling practices in test suite

Severity: Low
Type: Error Reporting
Target: `test/iETH/testiETHPolicy.js`

Difficulty: Low
Finding ID: TOB-DFL-014

Description
The test suite does not properly test expected behavior, as the contracts run in production. Additionally, certain components lack error-handling methods. These deficiencies can cause failed tests to be overlooked.

In particular, the tests fail to properly check error messages. For example, errors are silenced with a try-catch statement:

```
it("Should revert due to user do not have enough ETH", async function () {
  try {
    await iETH
      .connect(minter)
      .mint(minter.address, { value: await minter.getBalance() });
  } catch (error) {
    // console.log(error);
  }
});
```

Figure 14.1: `test/iETH/testiETHPolicy.js#L279-L287`

If this error is silenced, there will be no guarantee that a smart contract call has reverted for the right reason. As a result, if the test suite passes, it will provide no guarantee that the transaction call reverted correctly.

Exploit Scenario
dForce adds new functionality to the codebase and runs tests to ensure uniformity of the behavior. The test property fails due to misconfiguration, and a contract is incorrectly deployed. However, because the error is not checked properly, the test suite runs perfectly, leading dForce to deploy untested code.

Recommendations
Short term, test these operations against a specific error message. Testing will ensure that errors are never silenced, and the test suite will check that a contract call has reverted for the right reason.

Long term, follow standard testing practices for smart contracts to minimize the number of issues during development.

15. Added code in fixtures makes third-party integrations more difficult

Severity: Informational
Type: Configuration
Target: `test/helpers/fixtures.js`

Difficulty: Low
Finding ID: TOB-DFL-015

Description
The test fixtures introduce "if" statements that cause the execution to sleep for 10 seconds between every contract call:

```
  console.log("going to set flashloan fee ratio: ", flashloanFeeRatio);
  tx = await iETH._setNewFlashloanFeeRatio(
    utils.parseEther(flashloanFeeRatio.toString())
  );
  if (network.name != "hardhat") {
    await sleep(sleepTime);
    await tx.wait(blockSleep);
  }
  console.log("finish to set flashloan fee ratio.\n");

  console.log("going to set protocol fee ratio: ", protocolFeeRatio);
  tx = await iETH._setNewProtocolFeeRatio(
    utils.parseEther(protocolFeeRatio.toString())
  );
  if (network.name != "hardhat") {
    await sleep(sleepTime);
    await tx.wait(blockSleep);
```

Figure 15.1: `test/iETH/testiETHPolicy.js#L279-L287`

This period of sleep for every transaction on a non-hardhat network makes the deployment of contracts on different networks very difficult. As a result, integrating third-party tools such as Echidna for property testing is also very difficult.

Exploit Scenario
Due to the code added to the test fixtures, the contracts cannot be deployed on `localhost`. As a result, Echidna cannot be efficiently integrated into the codebase.

Recommendations
Short term, either

- remove the non-hardhat network sleeps, or
- document why they are required.

Long term, integrate a better workaround to facilitate access to tooling. This will ensure that the dForce system can easily be tested with third-party tools such as Echidna.

16. Borrow rate depends on approximation of blocks per year

Severity: Informational
Type: Configuration
Target: `contracts/InterestRateModel`

Difficulty: Medium
Finding ID: TOB-DFL-016

Description
The borrow rate formula uses an approximation of the number of blocks mined annually. This number can change across different blockchains and years. The current value assumes that a new block is mined every 15 seconds, but on Ethereum mainnet, a new block is mined every ~13 seconds.

To calculate the borrow rate, the formula determines the approximate borrow rate over the past year and divides that number by the estimated number of blocks mined per year:

```
/**
* @notice The approximate number of Ethereum blocks produced each year
/
uint256 public constant blocksPerYear = 2102400;

function getBorrowRate(
        uint256 _balance,
        uint256 _borrows,
        uint256 _reserves
    ) external view returns (uint256 _borrowRate) {
        [...]

        // Borrow rate is:
        // 1). 0% <= UR < 5%:   0.02*UR^2 + 0.08*UR^4 + 0.05*UR^7
        // 2). 5% <= UR < 100%: 0.05*UR^2 + 0.08*UR^4 + 0.05*UR^7
        uint256 _factor = _util < _threshold ? 10 : 25;
        uint256 _temp =_util.rpow(2, BASE).mul(_factor);
        _temp = _temp.add(_util.rpow(4, BASE).mul(40));
        _temp = _temp.add(_util.rpow(7, BASE).mul(25));

        uint256 _annualBorrowRateScaled = _temp.mul(2).div(1000);

        // And then divide down by blocks per year.
        _borrowRate = _annualBorrowRateScaled.div(blocksPerYear);
    }
```

Figure 16.1: `contracts/InterestRateModel/StandardInterestRateModel.sol#L16-L20`

However, `blocksPerYear` is an estimated value and may change depending on transaction throughput. Additionally, different blockchains may have different block-settling times, which could also alter this number.

Exploit Scenario
The borrow rate formula uses an incorrect number of blocks per year, resulting in deviations from the actual borrow rate.

Recommendation
Short term, analyze the effects of a deviation from annual number of blocks, and document the risks so that users are aware of them prior to using the system.

Long term, identify all variables that are affected externally, and document the risks associated with a deviation from the true value.

References
- Ethereum Average Block Time Chart

17. Removal of nonReentrant modifier on flashLoan may have unintended consequences

Severity: Informational | Difficulty: Undetermined
Type: Undefined Behavior | Finding ID: TOB-DFL-017
Target: *contracts/TokenBase/Base.sol*

Description
Removal of the `nonReentrant` modifier on the flash loan function may have unintended effects on the dForce Lending system.

The dForce team expressed interest in removing the `nonReentrant` modifier from the flash loan to allow users to call `liquidateBorrow` within a flash loan.

```
Whether there is a risk to remove the the reentrancyGuard from flashloan, therefore
encourage the liquidation as liquidateBorrow can then be called within the flashloan.
```

Figure 17.1: dForce documentation.

However, various risks are associated with the removal of the modifier. For example, it may be possible to exploit the `_flashloanInternal` function, which does not follow a checks-effects-interaction pattern. If that function is exploited, it may be possible for a user to use reentrancy to take out multiple flash loans:

```
    /**
     * @dev Executes flashloan, that is borrowing and repaying are in one transaction.
     * @param _recipient The account to receive the borrowed reserves.
     * @param _loanAmount The amount to request for the flashloan.
     */
    function _flashloanInternal(
        address payable _recipient,
        uint256 _loanAmount,
        bytes memory _data
    ) internal {
        // Records the curent cash before the flashloan.
        uint256 _cashBefore = _getCurrentCash();

        // Calculates flashloan fee.
        uint256 _flashloanFee = _loanAmount.rmul(flashloanFeeRatio);
        require(
            _flashloanFee > 0,
            "_flashloanInternal: Request amount is too small for a flashloan!"
        );

        controller.beforeFlashloan(address(this), _recipient, _loanAmount);
```

```
        // Increases total borrows to keep exchange rate unchanged.
        totalBorrows = totalBorrows.add(_loanAmount);

        // Transfers funds to caller contract, it will fail if `_loanAmount` >
`_getCurrentCash()`.
        _doTransferOut(_recipient, _loanAmount);

        // Execute the hook.
        IFlashloanExecutor(_recipient).executeOperation(
            address(underlying),
            _loanAmount,
            _flashloanFee,
            _data
        );

        // Checks the current cash after repaying flashloan amount with fee.
        require(
            _getCurrentCash().sub(_cashBefore) >= _flashloanFee,
            "_flashloanInternal: Fail to repay borrow with fee!"
        );

        // Restores total borrows.
        totalBorrows = totalBorrows.sub(_loanAmount);

        [...]
    }
```

Figure 17.3: contracts/TokenBase/Base.sol#L465-L523

Moreover, as the internal flash loan function checks only the difference between pre- and post-loan cash balances, a user may be able to repay a loan with funds from the flash loan. Token seizures as part of the liquidation process could also be affected, as the code for seizing tokens and repaying is shared Additionally, any new features will need to be carefully analyzed with respect to the removal of the `nonReentrant` modifier.

Recommendation
Short term, either

- introduce a specific flash loan repayment flow into the system, or
- use a flash loan lock to ensure that certain operations cannot occur when a flash loan is taken out.

Long term, integrate Slither into a continuous integration pipeline to identify reentrant code.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights.
Auditing and Logging	Related to auditing of actions or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to protecting the privacy or integrity of data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing a system failure.
Documentation	Related to documentation errors, omissions, or inaccuracies.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Testing	Related to test methodology or test coverage.
Timing	Related to race conditions, locking, or the order of operations.
Undefined Behavior	Related to undefined behavior triggered by the program.

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is

	important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited public tools exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purposes.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to the use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General

- **Remove unused functions.** Many functions take parameters that are used to suppress compiler warnings, making the code more difficult to understand. They can be re-added in the future if necessary.
- **Use intermediate local variables and standard `if` statements instead of ternary operations.** The use of the ternary operator makes conditional statements more difficult to understand.
- **Use Slither's built-in `slither-check-upgradeability` tool to help verify the storage layout when developing upgradeable contracts.** It can help catch common upgradeability issues.
- **Ensure that functions follow the checks-effects-interactions pattern for events so that off-chain components receive events in order**.

```
- Reentrancy in Controller._addMarket - contracts/Controller.sol#170-223
- Reentrancy in Base._borrowInternal - contracts/TokenBase/Base.sol#250-278
- Reentrancy in Base._flashloanInternal - contracts/TokenBase/Base.sol#464-521
- Reentrancy in Base._liquidateBorrowInternal - contracts/TokenBase/Base.sol#348-415
- Reentrancy in Base._mintInternal - contracts/TokenBase/Base.sol#168-201
- Reentrancy in Base._redeemInternal - contracts/TokenBase/Base.sol#211-243
- Reentrancy in Base._repayInternal - contracts/TokenBase/Base.sol#288-340
- Reentrancy in Base._seizeInternal - contracts/TokenBase/Base.sol#424-457
- Reentrancy in FixedInterestRateModel._setBorrowRate -
  contracts/InterestRateModel/FixedInterestRateModel.sol#95-104
- Reentrancy in MSDS._setInterestRateModel - contracts/msd/MSDS.sol#115-131
- Reentrancy in FixedInterestRateModel._setSupplyRate -
  contracts/InterestRateModel/FixedInterestRateModel.sol#109-118
- Reentrancy in TokenERC20._transferTokens - contracts/TokenBase/TokenERC20.sol#17-39
- Reentrancy in MSDController._withdrawReserves -
  contracts/msd/MSDController.sol#182-204
- Reentrancy in MSDS.mint - contracts/msd/MSDS.sol#223-231
- Reentrancy in MSDS.redeem - contracts/msd/MSDS.sol#238-246
- Reentrancy in MSDS.redeemUnderlying - contracts/msd/MSDS.sol#253-261
```

Figure C.1: `reentrancy-events` Slither findings

Controller

- **Remove unused hook functions (`afterBorrow`, `afterTransfer`, etc.).** This will increase the code's readability. They can be re-added in the future if necessary.
- **Convert before and after hooks to `view` functions.** This will enable the functions to verify arithmetic and their callers to execute state-changing code, enhancing the readability of the code.
- Address the typo in the name of the `AccountEuityLocalVars` struct by changing its name to `AccountEquityLocalVars`.

StandardInterestRateModel

- **For complex arithmetic operations, use intermediate variables with clear names.** The use of consecutive arithmetic operators in a single assignment statement makes the arithmetic difficult to understand.

Base

- **Update the design documentation's description of the interest rate calculation to better reflect the way the calculation is done in the contracts**. This will make it easier to verify the accuracy of interest accrual. Additionally, consider using Echidna or Manticore to verify these properties.

Tests

- **Remove unused test functions.** This will keep the test suite cleaner and aid in readability.

D. Detecting functions missing `nonReentrant` modifier with Slither

The dForce codebase has many functions that interact with external contracts. As a result, most of the functions are protected by a `nonReentrant` modifier to prevent the possibility of reentrancy attacks. We used Slither to identify and review functions that are not protected by this modifier.

The below script follows an approach in which every reachable function must be either

- protected with a `nonReentrant` modifier or
- included in the `ACCEPTED_LIST`.

No issues were found while using the script; however, the script detected functions that are missing `nonreentrant` modifiers (shown below). These functions should be documented appropriately.

```
### Check iToken access controls
        - Base.permit(address,address,uint256,uint256,uint8,bytes32,bytes32) should have an
non re-eentrant modifier
        - TokenAdmin._setController(IControllerInterface) should have an non re-eentrant
modifier
        - TokenAdmin._setInterestRateModel(IInterestRateModelInterface) should have an non
re-eentrant modifier
        - TokenAdmin._setNewReserveRatio(uint256) should have an non re-eentrant modifier
        - TokenAdmin._setNewFlashloanFeeRatio(uint256) should have an non re-eentrant
modifier
        - TokenAdmin._setNewProtocolFeeRatio(uint256) should have an non re-eentrant
modifier
        - TokenAdmin._withdrawReserves(uint256) should have an non re-eentrant modifier
        - ERC20.approve(address,uint256) should have an non re-eentrant modifier
        - ERC20.increaseAllowance(address,uint256) should have an non re-eentrant modifier
        - ERC20.decreaseAllowance(address,uint256) should have an non re-eentrant modifier
        - Ownable._setPendingOwner(address) should have an non re-eentrant modifier
        - Ownable._acceptOwner() should have an non re-eentrant modifier
        - iToken.updateInterest() should have an non re-eentrant modifier
```

D.1: `non-reentrant-checks.py` results

The script is shown below:

```
from slither import Slither
from slither.core.declarations import Contract
from typing import List

contracts = Slither(".", ignore_compile=True)

def _check_access_controls(
    contract: Contract, modifiers_access_controls: List[str], ACCEPTED_LIST: List[str]
):
```

```
    print(f"### Check {contract} access controls")
    no_bug_found = True
    for function in contract.functions_entry_points:
        if function.is_constructor:
            continue
        if function.view:
            continue

        if not function.modifiers or (
            not any((str(x) in modifiers_access_controls) for x in function.modifiers)
        ):
            if not function.name in ACCEPTED_LIST:
                print(f"\t- {function.canonical_name} should have an non re-eentrant
modifier")
                no_bug_found = False
    if no_bug_found:
        print("\t- No bug found")


_check_access_controls(
    contracts.get_contract_from_name("iToken"),
    ["nonReentrant"],
    ["initialize", "balanceOfUnderlying", "exchangeRateCurrent", "totalBorrowsCurrent"],
)
```

D.2: non-reentrant-checks.py

E. Detecting functions using market-pausing values with Slither

The dForce codebase has many functions that can be called if a market has had its minting, borrowing, and redeeming paused. As a result, most functions check to ensure that a market has not been paused before starting execution. We used Slither to identify and review functions that are missing these checks.

The below script flags the following:

- Functions using `mapping_element`
- Functions not in the `ACCEPTED_LIST`

No issues were found while using the script.

The script is shown below:

```
from slither import Slither
from slither.slithir.operations.member import Member
from slither.slithir.operations.solidity_call import SolidityCall
from slither.core.declarations import Contract
from typing import List

contracts = Slither(".", ignore_compile=True)

def _check_paused(contract: Contract, mapping_element: str, ACCEPTED_LIST: List[str]):
    print(f"### Check {contract}'s {mapping_element} access controls")
    no_bug_found = True
    for function in contract.functions:
        if function.name in ACCEPTED_LIST:
            continue
        for ir in function.slithir_operations:
            if isinstance(ir, Member) and ir.variable_right.name == mapping_element:
                no_bug_found = False
                print(
                    f"\t- {ir.node.function.canonical_name}: {ir.node.expression}
({ir.node.source_mapping_str})"
                )
    if no_bug_found:
        print("\t- No bug found")

## check individual market pauses
_check_paused(
    contracts.get_contract_from_name("Controller"),
    "mintPaused",
    ["beforeMint", "_setiTokenPausedInternal", "_setMintPausedInternal"],
)
_check_paused(
    contracts.get_contract_from_name("Controller"),
    "redeemPaused",
    ["beforeRedeem", "_setiTokenPausedInternal", "_setRedeemPausedInternal"],
)
```

```
_check_paused(
    contracts.get_contract_from_name("Controller"),
    "borrowPaused",
    ["beforeBorrow", "_setiTokenPausedInternal", "_setBorrowPausedInternal",
"beforeFlashloan"],
)
```

E.1: check-market-pausable.py

F. Token Integration Checklist

The following checklist provides recommendations for interacting with arbitrary tokens. Every unchecked item should be justified and its associated risks, understood. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow the checklist, you should have this output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of Echidna and
Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **`The name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a

known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` and `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non–token-related functions.** Non–token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., balances[token_address][msg.sender] may not reflect the actual balance).

Owner privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

G. Fix Log

dForce addressed the issues we found in their codebase. Each of the fixes provided was checked by Trail of Bits on the week of March 26.

#	Title	Type	Severity	Status
1	Initialization functions can be front-run	Configuration	High	Not fixed.
2	Lack of two-step process for critical operations	Data Validation	Medium	Not fixed.
3	Project dependencies contain vulnerabilities	Patching	Low	Not fixed.
4	Contracts used as dependencies do not track upstream changes	Patching	Low	Not fixed.
5	Risks associated with EIP 2612	Configuration	Informational	Not fixed.
6	Lack of zero check on functions	Data Validation	Informational	Fixed. (39de05e)
7	Insufficient protection on sensitive owner private keys	Configuration	High	Not fixed.
8	Lack of chainID validation allows reuse of signatures across forks	Authentication	High	Fixed. (a7b8fb0, d659f2b)
9	Lack of contract documentation makes codebase difficult to understand	Documentation	Informational	Not fixed.
10	Contract owner has too many privileges	Access Controls	Informational	Not fixed.
11	Lack of contract existence check on `delegatecall` will result in unexpected behavior	Access Controls	High	Not fixed.
12	Liquidators can liquidate more than 50% of loan	Data Validation	Medium	Not fixed.
13	External calls in loops may result in denial of service	Data Validation	Informational	Not fixed.

14	Poor error-handling practices in test suite	Error Reporting	Low	Not fixed.
15	Added code in fixtures makes third-party integrations more difficult	Configuration	Informational	Not fixed.
16	Borrow rate depends on approximation of blocks per year	Configuration	Informational	Not fixed.
17	Removal of nonReentrant modifier on flashLoan may have unintended consequences	Undefined Behavior	Informational	Not fixed.

1. Initialization functions can be frontrun - Not fixed
dForce team stated:

> *Will adapt the factory pattern in the future deployment.*

2. Lack of two-step process for critical operations leads to errors - Not fixed
dForce team stated:

> *Good to know.*

3. Project dependencies contain vulnerabilities - Not fixed
dForce team stated:

> *Good to know.*

4. Contracts used as dependencies do not track upstream changes - Not fixed
dForce team stated:

> *Will try our best to document those changes.*

5. Risks associated with EIP 2612- Not fixed
dForce team stated:

> *Will bring such attention to our users and external partners in documents.*

7. Insufficient protection on sensitive owner private keys - Not fixed
dForce team stated:

> *The ownership will be transfer to hardware wallet after deployment, and later will be transferred to governance.*

9. Lack of contract documentation makes codebase difficult to understand - Not fixed
dForce team stated:

> *Will keep improving the document.*

10. Contract owner has too many privileges - Not fixed

dForce team stated:

The ownership will be transferred to governance.

11. Lack of contract existence check on delegatecall will result in unexpected behavior - Not fixed

dForce team stated:

we will document this risk for future upgrading

12. Liquidators can liquidate more than 50% of loan - Not fixed

dForce team stated:

Have fixed this in the document.

However, Trail of Bits was not provided with documentation changes associated with this fix.

13. External calls in loops may result in denial of service - Not fixed

dForce team stated:

Will document this for users.

14. Poor error-handling practices in test suite - Not fixed

dForce team stated:

Case removed.

Trail of Bits recommends testing a specific error message on the transactions that are expected to revert instead of removing test cases. A robust test suite should cover positive and negative cases.

15. Added code in fixtures makes third-party integrations more difficult - Not fixed

dForce team stated:

We have fixed this, and will integrate these tools into the project.

However, Trail of Bits was not provided with a commit for this fix.

16. Imprecise calculation of blocks per year may affect result in deviations in getBorrowRate - Not fixed

dForce team stated:

Will document this for users and check them before deploy onto different chains.

17. Removal of nonReentrant modifier on flashLoan may have unintended consequences - Not fixed

dForce team stated:

Have fixed this in the document.