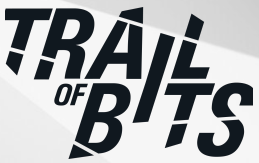

# Futureswap DOS

## Security Assessment

**February 21, 2023**

*Prepared for:*
**Derek Alia**
Futureswap

*Prepared by:* **Alexander Remie, Michael Colburn, Elvis Skozdopolj, and Sam Alws**

# About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

**Trail of Bits, Inc.**
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

# Notices and Remarks

## Copyright and Distribution



## Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

# Table of Contents

# Executive Summary

## Engagement Overview

Futureswap engaged Trail of Bits to review the security of its DeFi Operating System (DOS) smart contracts. From January 16 to February 6, 2023, a team of four consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

## Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and documentation. We performed static and dynamic automated and manual testing of the target system.

## Summary of Findings

The audit uncovered significant flaws that could impact system integrity or availability. A summary of the findings and details on notable findings are provided below.

### EXPOSURE ANALYSIS

| *Severity* | *Count* |
|---|---|
| High | 4 |
| Medium | 4 |
| Low | 1 |
| Informational | 5 |
| Undetermined | 2 |

### CATEGORY BREAKDOWN

| *Category* | *Count* |
|---|---|
| Access Controls | 1 |
| Auditing and Logging | 1 |
| Data Validation | 9 |
| Timing | 1 |
| Undefined Behavior | 4 |

## Notable Findings

Significant flaws that impact system integrity or availability are listed below.

- **Risk of arbitrary code execution due to missing access controls in onApprovalReceived (TOB-DOS-5)**
  The `DSafeProxy` contract's `onApprovalReceived` function fails to perform access control checks to ensure that the correct user is calling it. Consequently, an attacker could call this function to execute arbitrary code on behalf of other users' dSafes and steal users' funds.

- **Risk of permanent loss of funds due to incorrect dSafe version validation (TOB-DOS-11)**
  When users upgrade their dSafes, they need to specify the version string to upgrade them to. However, this string is not checked, so users can upgrade their dSafes to nonexistent versions, preventing them from accessing their accounts.

# Project Summary

## Contact Information

The following managers were associated with this project:


**Dan Guido**, Account Manager
dan@trailofbits.com

**Mary O'Brien**, Project Manager
mary.obrien@trailofbits.com


The following engineers were associated with this project:


**Alexander Remie**, Consultant
alexander.remie@trailofbits.com

**Michael Colburn**, Consultant
michael.colburn@trailofbits.com

**Elvis Skozdopolj**, Consultant
elvis.skozdopolj@trailofbits.com

**Sam Alws**, Consultant
sam.alws@trailofbits.com


## Project Timeline

The significant events and milestones of the project are listed below.

| Date | Event |
|---|---|
| **January 13, 2023** | Pre-project kickoff call |
| **January 24, 2023** | Status update meeting #1 |
| **January 31, 2023** | Status update meeting #2 |
| **February 7, 2023** | Delivery of report draft |
| **February 7, 2023** | Report readout meeting |
| **February 21, 2023** | Delivery of final report |

# Summary of Recommendations

The Futureswap DOS project is a work in progress with multiple planned iterations. Trail of Bits recommends that Futureswap address the findings detailed in this report and take the following additional steps prior to deployment:

- Increase the breadth of the unit tests to achieve full coverage on every file and to test for corner cases.
- Simplify the implementation according to the recommendations outlined in appendix C.
- Identify additional global system invariants and continue to extend the Echidna end-to-end test suite to ensure that it captures all functions. See appendix D for a list of invariants we identified for the DOS system.
- Implement fixes for all issues outlined in this report, and perform another security review on a release candidate prior to production deployment.

## Project Goals

The engagement was scoped to provide a security assessment of the Futureswap DOS project. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is interest always properly accounted for during borrowing and lending operations?
- Are liquidation operations executed as expected? Does the system always remain properly collateralized?
- Is NFT pricing and accounting handled properly?
- Can the system be accessed with unapproved assets?
- Can anybody gain unintended access to user funds?
- Is there proper logical separation between dSafes and dAccounts?
- Can funds become locked in the protocol?
- Does the use of upgradeability introduce any vulnerabilities?
- Are there appropriate access controls on all functions?

# Project Targets

The engagement involved a review and testing of the following target.

### DOS

| | |
|---|---|
| Repository | https://github.com/futureswap/DOS |
| Versions | `4293375f4231af2693263389d3f94f13fdcf7910`, `52f7431c90b999c51dff0e656da05ad3a25f6e7b` |
| Type | Solidity |
| Platform | EVM |

# Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

**`contracts/dos/`:** This folder contains the core contracts of the DOS system, the libraries that operate on dSafes, the dSafe proxy and logic contracts, and a contract that implements an alternative liquidation mechanism. Additionally, this folder contains the `TransferAndCall2` contract, which allows the `transferAndCall` function to be used with any existing token (similar to the `Permit2` contract's extension of the ERC-2612 standard). We performed static analysis and a manual review on these contracts. Additionally, we set up an end-to-end Echidna testing harness to test system-wide DOS invariants. During our manual review of these contracts, we looked for vulnerabilities such as flaws in the internal accounting of dSafes and dAccounts, denial-of-service vulnerabilities in the `DOS` contract, incorrect interest calculations, incorrect or missing access controls, missing events, unsafe ownership transfers, arithmetic vulnerabilities, reentrancy issues, front-running issues, flaws in the liquidation functions, flaws related to upgradeable contracts, incorrect handling of ERC-20 tokens (with nonstandard behavior), incorrect handling of NFTs, flaws in the `withdraw` and `deposit` functions, and flaws in the `TransferAndCall2` contract that attackers could use to steal tokens.

**`contracts/duoswapV2/`:** This folder contains a sample implementation of a UniswapV2-like pair contract that directly integrates with the DOS system's internal accounting. We manually reviewed the folder to look for common flaws related to UniswapV2 integrations.

**`contracts/external/`:** This folder contains four interfaces of external contracts that the DOS system interacts with, such as `Permit2`.

**`contracts/oracles/`:** This folder contains three oracles that are used to retrieve the prices of tokens. We performed static analysis and a manual review to look for vulnerabilities such as common flaws in operations for retrieving token prices from UniswapV2/V3 pools, missing validation in the Chainlink oracle, and flaws that attackers could use to manipulate the prices that the oracles report.

**`contracts/governance/`:** This folder contains the contracts that make up the decentralized governance system used to control and configure the DOS system. We performed static analysis and a manual review to look for vulnerabilities such as missing or incorrect access controls, flaws in the use of the proxy contracts, flaws that attackers could use to manipulate the voting outcome, and flaws in the `executeBatch` mechanism.

**`contracts/lib/`:** This folder contains libraries and contracts that are used in the core contracts. We performed static analysis, a manual review, and property-based testing

during our review of these files. For several of the functions in the `FsMath` library, we wrote property tests that we ran with Echidna.

## Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **DuoswapV2:** Futureswap indicated that this component was of a lower priority than the other components under audit. Although we did briefly look at this component, a more in-depth analysis of this component is required in the long term.

# Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

## Test Harness Configuration

We used the following tools in the automated testing phase of this project:

| Tool | Description |
|---|---|
| Slither | A static analysis framework that can statically verify algebraic relationships between Solidity variables |
| Echidna | A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation |

## Test Results

The results of this focused testing are detailed below.

**`FsMath`.** This library contains various arithmetic functions.

| Property | Tool | Result |
|---|---|---|
| The `exp` function returns results in the expected range: `exp(x > 0) > 1` and `0 < exp(x < 0) < 1`. | Echidna | TOB-DOS-9 |
| The `pow` function returns results with the expected sign. | Echidna | **Passed** |
| The `sqrt` function returns results in the expected range: `sqrt(x > 1) < x` and `sqrt(x < 1) > x`. | Echidna | **Passed** |

**`End-to-end DOS`.** This is the main contract that users interact with.

| Property | Tool | Result |
|---|---|---|
| Calls to the `depositERC20(amount)` function never revert (assuming that the given `amount` of ERC-20 tokens is owned by the caller) and always increase the given account balance by `amount`. | Echidna | **Passed** |
| Calls to the `depositERC20(amount)` function followed by calls to the `withdrawERC20(amount)` function never revert (assuming that the given `amount` of ERC-20 tokens is owned by the caller) and result in no change to the given account balance. | Echidna | **Passed** |
| Third-party calls never change the DOS contract's owner. | Echidna | **Passed** |
| Calls to the DOS contract leave the contract solvent after they complete. | Echidna | **Passed** |
| Calls to the `executeBatch` function never cause DOS assertions to fail. | Echidna | **WIP*** |

*For the last property ("Calls to the `executeBatch` function never cause DOS assertions to fail"), the test fails, but this failure is a false positive: assertions in DOS are sometimes used for input validation (e.g., in the `approveAndCall` function) rather than to indicate that a property should always hold.

# Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

| Category | Summary | Result |
|---|---|---|
| Arithmetic | Complicated calculations using many numerical methods and optimizations are performed by the `FsMath` library. Although this complexity is often necessary, this code would benefit from documentation of its methods and assumptions, as well as simplification where possible. We found one issue in `FsMath` caused by an incorrect assumption (TOB-DOS-9).<br><br>Outside of `FsMath`, the arithmetic is less complicated, but it contains a division-by-zero vulnerability (TOB-DOS-8) and a rounding issue (TOB-DOS-4). The code would benefit from more unit tests and increased scrutiny about behavior around edge cases. | **Moderate** |
| Auditing | The functions in the DOS contracts emit sufficient events to help Futureswap detect unexpected behavior. We did identify three functions that do not emit events directly; however, events for these functions' actions are emitted from the governance contract that calls them (TOB-DOS-1). | **Satisfactory** |
| Authentication / Access Controls | The system would benefit from additional documentation regarding access controls of the functions throughout the system. Additionally, we identified a function whose missing access controls could allow an attacker to steal funds (TOB-DOS-5). | **Moderate** |
| Complexity Management | The current implementation suffers from a large amount of complexity that could be reduced. For our recommendations for simplifying the codebase, see | **Weak** |

| | | |
|---|---|---|
| | appendix C. | |
| Decentralization | The current system is fully decentralized. Configuration variables are updated through a voting contract in which users, depending on the amount of FST they own at a specific block, can vote on proposals. Users are also able to vote on pausing the contracts (which blocks both deposits and withdrawals). dSafe implementations are upgradeable but can be upgraded only by dSafe owners.<br><br>However, we recommend developing additional user-facing documentation on how users can participate in the decentralized governance system and safely upgrade their dSafe implementations. | **Satisfactory** |
| Documentation | The consistency of comments throughout the codebase can be greatly improved. Some files have almost no NatSpec/inline comments, whereas others do but only for half of the functions. Consider adding comments in a consistent manner to all Solidity files. In some places, the NatSpec comments do not fully align with the implementation (TOB-DOS-13). User-facing documentation is in place but contains errors and could be greatly improved with more detailed information. | **Moderate** |
| Front-Running Resistance | In general, front-running is not an issue. The only exception we found is the front-running risk in the liquidation functions (TOB-DOS-2), which the user-facing documentation does not mention. We recommend updating the documentation to include the front-running risks associated with these functions. | **Satisfactory** |
| Low-Level Manipulation | The use of bitwise operations to encode NFT IDs adds complexity, although these operations are necessary due to storage requirements. Additionally, the `Proofs` library uses bitwise operations and assembly to manually control the memory. Although this might be more efficient, it adds a lot of complexity. Finally, in `FsMath`, bitwise manipulations could result in an unexpected integer overflow (TOB-DOS-9). | **Moderate** |

| | | |
|---|---|---|
| Testing and Verification | The current unit test suite lacks depth and breadth. Several issues in the report are the result of edge cases that could have been caught by more in-depth unit tests. Additionally, we recommend using Echidna for the `FsMath` library and also setting up an end-to-end Echidna test suite that tests system-wide DOS invariants. | **Moderate** |

# Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

| ID | Title | Type | Severity |
|---|---|---|---|
| 1 | Insufficient event generation | Auditing and Logging | Informational |
| 2 | Front-running risk in liquidate function | Timing | Informational |
| 3 | Wrong value removed in extractNFT | Undefined Behavior | High |
| 4 | Rounding issue in insertPosition could allow users to withdraw a negligible amount of assets for free | Data Validation | Medium |
| 5 | Risk of arbitrary code execution due to missing access controls in onApprovalReceived | Access Controls | High |
| 6 | Missing Chainlink price feed safety checks | Data Validation | Medium |
| 7 | Incorrect accounting of nonstandard ERC-20 tokens could result in loss of user funds | Data Validation | Medium |
| 8 | Risk of denial-of-service attacks due to rounding error in computeInterestRate | Data Validation | Medium |
| 9 | Risk of overflow in exp, causing it to return negative values | Data Validation | Informational |
| 10 | Risk of LP token value manipulation | Data Validation | Informational |
| 11 | Risk of permanent loss of funds due to incorrect dSafe version validation | Data Validation | High |

| | | | |
|---|---|---|---|
| 12 | Lack of two-step process for dSafe ownership changes | Data Validation | **High** |
| 13 | Inconsistency between approveAndCall's implementation and NatSpec comments | Data Validation | **Undetermined** |
| 14 | NFT forwarding cannot be set via executeBatch | Undefined Behavior | **Informational** |
| 15 | Unreachable code path in onTransferReceived2 | Undefined Behavior | **Undetermined** |
| 16 | Risk of out-of-gas error due to loops in isSolvent | Undefined Behavior | **Low** |

# Detailed Findings

| 1. Insufficient event generation | |
|---|---|
| Severity: **Informational** | Difficulty: **Low** |
| Type: Auditing and Logging | Finding ID: TOB-DOS-1 |
| Target: DOS protocol | |

### Description
Multiple critical operations do not emit events. As a result, it will be difficult to review the correct behavior of the contracts once they have been deployed.

Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions, including malfunctioning contracts or attacks.

The following operations should trigger events:

- `GovernanceProxy.setMaxSupportedGasCost`
- `UniV3Oracle.setCollateralFactor`
- `ERC20ChainlinkValueOracle.setRiskFactors`

### Exploit Scenario
An attacker discovers a vulnerability in the `UniV3Oracle` contract and modifies its execution. Because these actions generate no events, the behavior goes unnoticed until there is follow-on damage, such as financial loss.

### Recommendations
Short term, add events for all operations that could contribute to a higher level of monitoring and alerting.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

### 2. Front-running risk in liquidate function

| Severity: **Informational** | Difficulty: **High** |
| --- | --- |
| Type: Timing | Finding ID: TOB-DOS-2 |
| Target: `DOS.sol` | |

**Description**

An attacker (such as a malicious miner) can front-run calls to the `liquidate` function to take liquidation rewards for himself.

In the DOS contract, the `liquidate` function is defined as follows:

```
function liquidate(address dSafe) external override onlyDSafe whenNotPaused
dSafeExists(dSafe) {
    (int256 totalValue, int256 collateral, int256 debt) =
getRiskAdjustedPositionValues(dSafe);
    require(collateral < debt, "DSafe is not liquidatable");
    uint16[] memory dSafeERC20s = dSafes[dSafe].getERC20s();
    for (uint256 i = 0; i < dSafeERC20s.length; i++) {
        uint16 erc20Idx = dSafeERC20s[i];
        _transferAllERC20(erc20Idx, dSafe, msg.sender);
    }
    while (dSafes[dSafe].nfts.length > 0) {
        _transferNFT(dSafes[dSafe].nfts[dSafes[dSafe].nfts.length - 1], dSafe,
msg.sender);
    }
    if (totalValue > 0) {
        // totalValue of the liquidated dSafe is split between liquidatable and
liquidator:
        // totalValue * (1 - liqFraction) - reward of the liquidator, and
        // totalValue * liqFraction - change, liquidator is sending back to
liquidatable
        int256 percentUnderwater = (collateral * 1 ether) / debt;
        int256 leftover = ((totalValue * config.liqFraction * percentUnderwater) / 1
ether) /
            1 ether;
        _transferERC20(
            IERC20(erc20Infos[K_NUMERAIRE_IDX].erc20Contract),
            msg.sender,
            dSafe,
            leftover
        );
    }
    emit IDOSCore.SafeLiquidated(dSafe, msg.sender);
}
```

*Figure 2.1: The `liquidate` function in `DOS.sol`*

The reward is sent to the address that called the function (`msg.sender`), but the function has no restrictions on who can call it. Because anyone can call `liquidate`, the function is vulnerable to front-running. For example, a miner could choose not to add a liquidation transaction to the block and then replace it with his own transaction to liquidate the same account; similarly, a third party could front-run a liquidation transaction by submitting the same transaction with a higher gas price.

This issue is only of informational severity; in the event of a front-running attack, the account would still be liquidated, preserving the solvency of the DOS contract. However, front-running attacks may deter users from liquidating accounts.

### Exploit Scenario

Alice is a Futureswap user who has more debt than collateral. Bob attempts to liquidate Alice's account and claim the liquidation reward. Eve, a miner, sees Bob's transaction and replaces it with her own transaction to liquidate Alice's account. Eve gets the liquidation reward instead of Bob.

### Recommendations

Short term, add information about front-running risks to the documentation so that users are aware of them.

Long term, consider changing the liquidation reward system so that long-time users get more liquidation rewards than non-users. This will decrease the incentive to perform this attack.

| 3. Wrong value removed in extractNFT | |
|---|---|
| Severity: **High** | Difficulty: **Low** |
| Type: Undefined Behavior | Finding ID: TOB-DOS-3 |
| Target: `DOS.sol` | |

### Description

Because of an implementation mistake in the `extractNFT` function, users may not be able to withdraw their NFTs.

In `extractNFT`, a swap-and-pop pattern is used to remove an NFT (represented by `dSafe.nfts[idx]`) from the list containing the given dSafe's NFTs:

```
NFTId lastNFTId = dSafe.nfts[dSafe.nfts.length - 1];
map[lastNFTId].dSafeIdx = idx;
dSafe.nfts.pop();
```

*Figure 3.1: The attempted swap-and-pop in `extractNFT` in `DOS.sol`*

However, `dSafe.nfts[idx]` is never replaced with the final NFT in the list; instead, `dSafe.nfts[idx]` is left in the list, and `dSafe.nfts[dSafe.nfts.length - 1]` is removed. This results in an incorrect list of NFTs owned by the dSafe, which will cause later calls to the `isSolvent` function to revert, making the withdrawal of this NFT impossible.

### Exploit scenario

Alice, who owns a dSafe with three NFTs, tries to withdraw the second NFT. Because of the implementation mistake described above, her transaction reverts and she cannot withdraw her NFT.

### Recommendations

Short term, change the code in `extractNFT` so that `dSafe.nfts[idx]` is overwritten with `lastNFTId`.

Long term, add more extensive tests that can catch similar accounting errors.

### 4. Rounding issue in insertPosition could allow users to withdraw a negligible amount of assets for free

| Severity: **Medium** | Difficulty: **Low** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-4 |
| Target: `DOS.sol` | |

#### Description

The `insertPosition` function contains a rounding issue that could allow users to take out a negligible amount of debt without having to repay it.

The following code in the DOS contract's `insertPosition` function calculates the shares given to a user who is depositing collateral or taking out debt:

```
int256 shares;
if (pool.shares == 0) {
    FsUtils.Assert(pool.tokens == 0);
    shares = tokens;
} else {
    shares = (pool.shares * tokens) / pool.tokens;
}
```

*Figure 4.1: Code for calculating shares in `DOS.sol`*

However, the expression `(pool.shares * tokens) / pool.tokens` rounds toward zero. As a result, when `pool.tokens` is larger than `pool.shares` (which will be the case when interest accrues) and `tokens` equals 1 (i.e., the user takes out one token of debt), the user will receive zero shares and will never have to pay back the debt.

Note that the maximum number of tokens that users could take out for free is `ceil(pool.tokens / pool.shares) - 1`.

#### Exploit Scenario

The DOS's pool of ETH debt is 15e18 tokens (15 ETH) and 10e18 shares. Eve opens up an account and withdraws one wei without putting in any collateral. She receives zero shares of debt, so her account is still considered solvent and she will never have to pay back the one wei she took out.

#### Recommendations

Short term, replace `insertPosition`'s current share calculation with a division function that rounds toward negative infinity instead of zero.

Long term, examine the codebase for rounding and division errors and implement fuzzing to catch similar issues.

### 5. Risk of arbitrary code execution due to missing access controls in onApprovalReceived

| Severity: **High** | Difficulty: **Low** |
| --- | --- |
| Type: Access Controls | Finding ID: TOB-DOS-5 |
| Target: `contracts/dos/DSafeProxy.sol` | |

#### Description

Because the `onApprovalReceived` function does not have the proper access controls, any user could call the function, execute arbitrary code on another user's dSafe and dAccount, and possibly drain that user's assets.

Users call the `createDSafe` function on the `DOS` contract to create dSafe smart wallets. With these wallets, users can hold ERC-20 and ERC-721 assets in their dSafes and deposit and borrow assets through their associated dAccounts.

Through the `DOS` contract, a user could approve a contract to spend the ERC-20 tokens stored in their dSafe and execute the approved contract's code through the `onApprovalReceived` callback function; only the `DOS` contract should be able to call this function. After the `onApprovalReceived` function is executed, the approved `sender`, `amount`, and `callData` variables are reset to their previous values. When called on a dSafe, the `onApprovalReceived` callback function executes the following code:

```
270    function onApprovalReceived(
271       address sender,
272       uint256 amount,
273       Call memory call
274    ) external returns (bytes4) {
275       if (call.callData.length == 0) {
276           revert("PL: INVALID_DATA");
277       }
278       emit TokensApproved(sender, amount, call.callData);
279
280       Call[] memory calls = new Call[](1);
281       calls[0] = call;
282
283       dos.executeBatch(calls);
284
285       return this.onApprovalReceived.selector;
286    }
```

*Figure 5.1: `contracts/dos/DSafeProxy.sol`*

This code executes a function call defined in the `call` parameter on behalf of the dSafe.

However, although `onApprovalReceived` is meant to be called only by the DOS contract in a callback, the current implementation allows any user to call the function and execute code on behalf of another user's dSafe.

### Exploit Scenario
Alice deploys a dSafe contract and deposits 1,000 USDC. Bob sees Alice's newly funded dSafe and calls `onApprovalReceived` with a USDC transfer call as the `call` parameter. The DOS contract executes the call on behalf of Alice's dSafe, transferring the 1,000 USDC to Bob's account.

### Recommendations
Short term, add an access control mechanism to `onApprovalReceived` that allows only approved callers like the DOS contract to call the `onApprovalReceived` callback function.

Long term, review the access controls for all of the components and write unit tests that ensure each component's access controls are working correctly. Additionally, add access control properties to the Echidna end-to-end test suite.

### 6. Missing Chainlink price feed safety checks

| Severity: **Medium** | Difficulty: **High** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-6 |
| Target: `contracts/oracles/ERC20ChainlinkValueOracle.sol` | |

#### Description
Certain safety checks that should be used to validate data returned from the `latestRoundData` function are missing:

- `require(updatedAt > 0)`: This checks whether the requested round is valid and complete; an example use of the check can be found in the `historical-price-feed-data` project, and a description of the parameter validation can be found in Chainlink's documentation.
- `require(price > 0)`: While `price` is expected to be greater than zero, the code may consume zero or even negative prices, as shown in figure 6.1.

```
54    function calcValue(
55        int256 balance
56    ) external view override returns (int256 value, int256 riskAdjustedValue) {
57        (, int256 price, , , ) = priceOracle.latestRoundData();
58        value = (balance * price) / base;
59        if (balance >= 0) {
60            riskAdjustedValue = (value * collateralFactor) / 1 ether;
61        } else {
62            riskAdjustedValue = (value * 1 ether) / borrowFactor;
63        }
64        return (value, riskAdjustedValue);
65    }
```

*Figure 6.1: `contracts/oracles/ERC20ChainlinkValueOracle.sol`*

#### Exploit Scenario
Because of a bug in Chainlink, `latestRoundData` returns a negative price. The bug is noticed by an attacker who uses it to liquidate a dAccount or drain the protocol.

#### Recommendations
Short term, implement the checks listed above.

Long term, add tests for the Chainlink price feed with various edge cases, possibly with specially crafted random data. Review the specific implementation of Chainlink Aggregator that the system will use and adjust the system to its semantic.

### 7. Incorrect accounting of nonstandard ERC-20 tokens could result in loss of user funds

| Severity: **Medium** | Difficulty: **High** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-7 |
| Target: `contracts/dos/DOS.sol` | |

#### Description

The deposit and withdrawal functions of the DOS contract do not account for nonstandard ERC-20 token implementations that take fees on transfers. These cases could result in incorrect accounting of dAccount balances and interest.

The deposit functions (figures 7.1–7.3) perform balance accounting based on the number of tokens passed to the function or the balance of the sender. This approach assumes that the number of tokens sent will always equal the number of tokens received, which is not the case for many nonstandard ERC-20 tokens.

```
351    function depositERC20ForSafe(
352        address erc20,
353        address to,
354        uint256 amount
355    ) external override dSafeExists(to) whenNotPaused {
356        if (amount == 0) return;
357        (, uint16 erc20Idx) = getERC20Info(IERC20(erc20));
358        int256 signedAmount = FsMath.safeCastToSigned(amount);
359        _dAccountERC20ChangeBy(to, erc20Idx, signedAmount);
360        emit IDOSCore.ERC20BalanceChanged(erc20, to, signedAmount);
361        IERC20(erc20).safeTransferFrom(msg.sender, address(this), amount);
362    }
```

*Figure 7.1: `contracts/dos/DOS.sol`*

```
367    function depositERC20(IERC20 erc20, uint256 amount) external override
onlyDSafe whenNotPaused {
368        if (amount == 0) return;
369        (, uint16 erc20Idx) = getERC20Info(erc20);
370        int256 signedAmount = FsMath.safeCastToSigned(amount);
371        _dAccountERC20ChangeBy(msg.sender, erc20Idx, signedAmount);
372        emit IDOSCore.ERC20BalanceChanged(address(erc20), msg.sender,
```

```
signedAmount);
373         erc20.safeTransferFrom(msg.sender, address(this), amount);
374     }
```

*Figure 7.2: contracts/dos/DOS.sol*

```
389     function depositFull(IERC20[] calldata erc20s) external override onlyDSafe
whenNotPaused {
390         for (uint256 i = 0; i < erc20s.length; i++) {
391             (ERC20Info storage erc20Info, uint16 erc20Idx) =
getERC20Info(erc20s[i]);
392             IERC20 erc20 = IERC20(erc20Info.erc20Contract);
393             uint256 amount = erc20.balanceOf(msg.sender);
394             int256 signedAmount = FsMath.safeCastToSigned(amount);
395             _dAccountERC20ChangeBy(msg.sender, erc20Idx, signedAmount);
396             emit IDOSCore.ERC20BalanceChanged(address(erc20), msg.sender,
signedAmount);
397             erc20.safeTransferFrom(msg.sender, address(this), amount);
398         }
399     }
```

*Figure 7.3: contracts/dos/DOS.sol*

If the given token is a fee-on-transfer token, the accounted balances will be larger than the actual number of tokens received by the contract. If multiple users were to deposit the same token, the last user to withdraw would receive fewer tokens than they should and could lose their entire deposit.

### Exploit Scenario

The protocol has approved USDT to be used as collateral, and the USDT team has decided to impose a 10% fee on all transfers.

Alice deposits 1,000 USDT into the protocol, and the full balance is attributed to her dAccount. The actual amount in the protocol is now 900 USDT. Bob deposits 100,000 USDT into the protocol, and the full balance is attributed to his dAccount. The protocol now holds 90,900 USDT.

After some time has passed, Bob decides to withdraw his USDT, and he is able to withdraw the entire USDT balance of the protocol. When Alice attempts to withdraw, her transaction reverts due to insufficient balance.

Both Alice and Bob can still borrow assets since their dAccount balance is 1,000 USDT and 9,100 USDT, respectively, which leaves the protocol with bad debt.

### Recommendations

Short term, add `balanceOf` checks that are executed before and after transfers to determine the actual number of tokens that was received.

Long term, review the token integration checklist before adding a token to the protocol.

### 8. Risk of denial-of-service attacks due to rounding error in computeInterestRate

| Severity: **Medium** | Difficulty: **Low** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-8 |
| Target: `contracts/dos/DOS.sol` | |

#### Description

The first depositor for any ERC-20 token can execute a permanent denial-of-service attack by depositing a number of tokens that is less than the protocol's `fractionalReserveLeverage`.

The protocol allows any user with a dSafe to deposit protocol-approved assets into their dAccount, which enables them to borrow other assets from the protocol. The protocol uses the `_updateInterest` function to calculate interest on deposited assets and debt on each action that changes the balance of a dAccount, as shown in figure 8.1.

```
861    function _dAccountERC20ChangeBy(address dSafeAddress, uint16 erc20Idx, int256
amount) internal {
862        _updateInterest(erc20Idx);
863        DSafeLib.DSafe storage dSafe = dSafes[dSafeAddress];
864        ERC20Share shares = dSafe.erc20Share[erc20Idx];
865        ERC20Info storage erc20Info = erc20Infos[erc20Idx];
866        int256 currentAmount = _extractPosition(shares, erc20Info);
867        int256 newAmount = currentAmount + amount;
868        dSafe.erc20Share[erc20Idx] = _insertPosition(newAmount, dSafe, erc20Idx);
869    }
870
871    function _dAccountERC20Clear(address dSafeAddress, uint16 erc20Idx) internal
returns (int256) {
872        _updateInterest(erc20Idx);
873        DSafeLib.DSafe storage dSafe = dSafes[dSafeAddress];
874        ERC20Share shares = dSafe.erc20Share[erc20Idx];
875        int256 erc20Amount = _extractPosition(shares, erc20Infos[erc20Idx]);
876        dSafe.erc20Share[erc20Idx] = ERC20Share.wrap(0);
877        dSafe.removeERC20IdxFromDAccount(erc20Idx);
878        return erc20Amount;
879    }
```

*Figure 8.1: contracts/dos/DOS.sol*

On each call to `_updateInterest`, the protocol uses the `computeInterestRate` function to calculate a new interest rate based on the token utilization, as shown in figure 8.2.

```
740    if (poolAssets == 0)
741       utilization = 0; // if there are no assets, utilization is 0
742    else utilization = uint256((debt * 1e18) / ((collateral - debt) / leverage));
```

*Figure 8.2: contracts/dos/DOS.sol*

However, if the first deposit is less than the `fractionalReserveLeverage` of the protocol (`leverage` in figure 8.2), or if the value of `collateral` is sufficiently close to the value of `debt`, a division-by-zero error will occur, causing any further attempts to withdraw or deposit assets to revert.

In such a case, users could resolve this problem by submitting a governance proposal to execute a batch of transactions that would do the following:

1. Reduce the protocol's `fractionalReserveLeverage` variable to be less than or equal to the first depositor's collateral
2. Execute the `depositERC20ForSafe` function to deposit a small number of tokens to the first depositor's dAccount so that the collateral is larger than or equal to the original `fractionalReserveLeverage`
3. Return the `fractionalReserveLeverage` variable to its original value

Although such an issue can be resolved, it would still result in a denial of service of at least two days for the token in question. Additionally, all future ERC-20 tokens added to the protocol would be vulnerable to the same denial-of-service attack.

### Exploit Scenario

The protocol's `fractionalReserveLeverage` is set to 9, and the protocol has approved WETH to be used as collateral.

Alice sees the vulnerability and deposits one wei as the first deposit for WETH. This succeeds, as `poolAssets == 0` on the first deposit.

Bob attempts to deposit 1e18 WETH into the pool, but the transaction reverts because of the division-by-zero error described above. The utilization is equal to the following:

`utilization = 0 * 1e18 / ((1 - 0) / 9)`

```
utilization = 0 / 0
```

All further attempts to withdraw or deposit this asset by any user revert.

### Recommendations

Short term, add a minimum deposit amount that is larger than or equal to the maximum `fractionalReserveLeverage` of the protocol.

Long term, add tests for the deposit, withdrawal, and calculation of interest with various edge cases. Additionally, write invariants for the interest rate and test them using Echidna.

### 9. Risk of overflow in exp, causing it to return negative values

| Severity: **Informational** | Difficulty: **High** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-9 |
| Target: `contracts/lib/FsMath.sol` | |

#### Description

The `FsMath.exp` function returns a negative number on certain large inputs, such as 133.

In `FsMath.exp`, the following check is made to determine whether the input number is too large:

```
require(shiftLeft < (256 - FIXED_POINT_SCALE_BITS), "Exponentiation overflows");
```

*Figure 9.1: `contracts/lib/FsMath.sol`*

At the end of `FsMath.exp`, the following line bit-shifts the result before returning it:

```
return shiftLeft >= 0 ? (prod << uint256(shiftLeft)) : (prod >>
uint256(-shiftLeft));
```

*Figure 9.2: `contracts/lib/FsMath.sol`*

In Solidity, bit-shift operations do not check for overflow, so the check made earlier in the function (figure 9.1) is the only thing preventing an overflow during the bit-shift. However, in certain cases where `shiftLeft` is just short of `(256 - FIXED_POINT_SCALE_BITS)`, `prod` can be large enough that the bit-shift operation overflows, resulting in a negative return value.

Using Echidna, we developed the following test, which detects unexpected results from the exp function:

```
contract Echidna {
  function testExp(int256 xa, int256 xb) public {
    int256 x = xa*FsMath.FIXED_POINT_SCALE+xb;
    int256 result = FsMath.exp(x);


    // if x >= 0, exp(x) >= 1
    // if x <= 0, 0 <= exp(x) <= 1
```

```
    if (x >= 0)
      assert(result >= FsMath.FIXED_POINT_SCALE);
    if (x <= 0)
      assert(0 <= result && result <= FsMath.FIXED_POINT_SCALE);
  }
}
```

*Figure 9.3: The Echidna test on exp*

Figure 9.4 shows Echidna's output when fuzzing this function:

```
Analyzing contract:
/Users/sam/Documents/audit-futureswap-jan2023/contracts/testing/Echidna.sol:Echidna
testExp(int256,int256): failed!💥
  Call sequence:
    testExp(133,-3)

Event sequence: Panic(1)
AssertionFailed(..):  fuzzing (1330222/5000000)

Unique instructions: 503
Unique codehashes: 2
Corpus size: 8
Seed: 5961358604494398468
sam@Sams-MacBook-Pro audit-futureswap-jan2023 % vi echidna.yaml
sam@Sams-MacBook-Pro audit-futureswap-jan2023 % rm -r corpus
sam@Sams-MacBook-Pro audit-futureswap-jan2023 % echidna-test . --config echidna.yaml
--contract Echidna
Loaded total of 0 transactions from corpus/reproducers/
Loaded total of 0 transactions from corpus/coverage/
Analyzing contract:
/Users/sam/Documents/audit-futureswap-jan2023/contracts/testing/Echidna.sol:Echidna
testExp(int256,int256): failed!💥
  Call sequence:
    testExp(133,0)

Event sequence: Panic(1)
AssertionFailed(..):  passed! 🎉

Unique instructions: 503
Unique codehashes: 2
Corpus size: 7
Seed: 5444910436476293501
```

*Figure 9.4: Echidna's output when fuzzing exp*

The exp function is used only for calculating interest in the DOS contract, and very large inputs to exp should never occur, so this issue is only of informational severity.

### Exploit Scenario

A Futureswap developer adds functionality that uses the exp function; the functionality relies on exp to return only positive values and to revert if an overflow occurs. An attacker takes advantage of this incorrect assumption.

### Recommendations

Short term, determine the largest size that prod can be and adjust the overflow check accordingly. Alternatively, in the exp function add a check for potential overflows that occurs directly before the bit-shift operation.

Long term, use Echidna to fuzz all of the FsMath functions to check that the desired properties hold.

### 10. Risk of LP token value manipulation

| Severity: **Informational** | Difficulty: **Low** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-10 |
| Target: `contracts/oracles/UniV2Oracle.sol` | |

#### Description

An attacker could steal protocol assets due to a flaw in the `UniV2Oracle` contract's LP price calculation.

The protocol allows users to borrow assets by depositing collateral in the form of LP tokens and other approved assets. The value of the LP tokens is calculated in the `calcValue` function of the `UniV2Oracle` contract by using the balances of the `DuoswapV2Pair` contract's dAccount as the pool reserves.

Since pool reserves can be influenced by users interacting with the protocol, the price of the LP tokens can be manipulated.

```
 55          uint256 balance0 = uint256(IDOS(dos).getDAccountERC20(dSafe,
IERC20(token0)));
 56          uint256 balance1 = uint256(IDOS(dos).getDAccountERC20(dSafe,
IERC20(token1)));
 57
 58          (int256 price0, int256 adjustedPrice0) =
erc20ValueOracle[token0].calcValue(
 59              FsMath.safeCastToSigned(balance0)
 60          );
 61          (int256 price1, int256 adjustedPrice1) =
erc20ValueOracle[token1].calcValue(
 62              FsMath.safeCastToSigned(balance1)
 63          );
 64
 65          value = ((price0 + price1) * amount) /
FsMath.safeCastToSigned(totalSupply);
 66          riskAdjustedValue =
 67              ((adjustedPrice0 + adjustedPrice1) * amount) /
 68              FsMath.safeCastToSigned(totalSupply);
```

*Figure 10.1: `contracts/oracles/UniV2Oracle.sol`*

The `UniV2Oracle` contract takes the following steps to calculate the value of the provided `amount` of LP tokens:

- Computes the total value of reserve tokens in the given pool based on the balance of the pool's dAccount and the sum of the values calculated in the `erc20ValueOracle[token].calcValue` function calls
- Multiplies the total value computed with the `amount` of LP tokens passed into the function as a parameter
- Divides the result by the total supply of LP tokens in the given pool

This means that the value of a single LP token is determined by the amount of reserves in the pool and the prices of each of the two assets.

The token prices are very difficult to manipulate since the protocol uses Chainlink oracle price feeds; however, the reserves of the tokens can be easily manipulated by simply swapping one asset for the other.

### Exploit Scenario

Assume that the price of WETH is $1,000, the price of USDC is $1, and the `DuoswapV2Pair` contract holds 100,000 USDC and 100 WETH.

The assets in `DuoswapV2Pair` are valued at $200,000, and the total supply of LP tokens is 100,000. Alice holds 1,000 LP tokens of the USDC/WETH pair in her dAccount.

Using the current calculation, ignoring the small reduction in value resulting from the `erc20ValueOracle[token].calcValue` function calls, we get the following results:

`adjustedPrice0` = $100,000

`adjustedPrice1` = $100,000

`value` = $200,000 * 1,000 / 100,000

The value of Alice's 1,000 LP tokens is $2,000.

Alice realizes that she can manipulate the price and takes out a large flash loan of 1,000 WETH, worth $1 million. She deposits 50,000 USDC and 50 WETH into the pool, getting 50,000 LP tokens back. The current reserves in the pool are 150,000 USDC and 150 WETH.

Alice swaps 1,000 WETH worth $1 million for USDC, getting around 149,980 USDC back. The new pool reserves are around 20 USDC and 1,150 WETH. The total value of Alice's 51,000 LP tokens is ($20 + $1,150,000) * 51,000 / 150,000 = $391,006.

### Recommendations

Short term, update the calculation in `calcValue` to derive the LP token value from the constant product invariant *K* based on the Alpha Finance implementation.

Long term, write invariants for the oracle and test them using Echidna.

### 11. Risk of permanent loss of funds due to incorrect dSafe version validation

| Severity: **High** | Difficulty: **Medium** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-11 |
| Target: `contracts/dos/DOS.sol` | |

#### Description

The `upgradeDSafeImplementation` function does not properly check for invalid versions of the dSafe implementation contract passed to it. As a result, if a user passes in an invalid version string, their dSafe implementation will be upgraded to the null address, permanently disabling their dSafe and locking all of the assets in their dSafe and dAccount.

The protocol allows users to upgrade to a new version of the dSafe implementation contract by calling the `upgradeDSafeImplementation` function (figure 11.1). This function verifies that the version is not deprecated and that it does not contain any known bugs.

```
 1    function upgradeDSafeImplementation(
 2       string calldata version
 3    ) external override onlyDSafe whenNotPaused {
 4       (
 5           ,
 6           IVersionManager.Status status,
 7           IVersionManager.BugLevel bugLevel,
 8           address implementation,
 9
10       ) = versionManager.getVersionDetails(version);
11       require(status != IVersionManager.Status.DEPRECATED, "Version is
deprecated");
12       require(bugLevel == IVersionManager.BugLevel.NONE, "Version has bugs");
13       dSafeLogic[msg.sender] = implementation;
14       emit IDOSConfig.DSafeImplementationUpgraded(msg.sender, version,
implementation);
15    }
```

*Figure 11.1: `contracts/dos/DOS.sol`*

To fetch the version information, the `upgradeDSafeImplementation` function calls the `VersionManager.getVersionDetails(version)` function, which uses the version string as a key to fetch the corresponding value held in the `_versions` mapping (as shown in figure 11.3).

```
193     Version storage v = _versions[versionName];
194
195     versionString = v.versionName;
196     status = v.status;
197     bugLevel = v.bugLevel;
198     implementation = v.implementation;
199     dateAdded = v.dateAdded;
200
201     return (versionString, status, bugLevel, implementation, dateAdded);
```

*Figure 11.3: contracts/dos/VersionManager.sol*

However, if the function accesses a nonexistent key-value pair in the mapping, it will not revert; instead, it will return the `Version` struct with all values set to `0`. Since the values of `Status`, `BugLevel`, and `implementation` will be set to `BETA`, `NONE`, and the null address, respectively, all of the checks in the `upgradeDSafeImplementation` function will pass.

### Exploit Scenario

Alice sees that a new version of the dSafe implementation contract is out and chooses to upgrade her dSafe. She passes `2,0.0` instead of `2.0.0` into the `upgradeDSafeImplementation` function by mistake.

Since `2,0.0` is a nonexistent version, all checks pass and her dSafe implementation is upgraded to the null address. Alice has permanently lost access to her dSafe, her dAccount, and all of the assets that they hold.

### Recommendations

Short term, either move the `DEPRECATED` status to the zero index of the enum or add another check to ensure that upgrading to an invalid version is not possible.

Long term, enhance the breadth of the unit tests to test all execution paths of the dSafe upgrade mechanism. Consider writing invariants for the dSafe upgrade mechanism and testing them using Echidna.

### 12. Lack of two-step process for dSafe ownership changes

| Severity: **High** | Difficulty: **High** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-12 |
| Target: `contracts/dos/DOS.sol` | |

#### Description

The `transferDSafeOwnership` function is used to change the owner of a particular dSafe contract. Transferring ownership through one function call is error-prone and could result in irrevocable mistakes.

```
1008    function transferDSafeOwnership(address newOwner) external override
onlyDSafe whenNotPaused {
1009        dSafes[msg.sender].owner = newOwner;
1010        emit IDOSConfig.DSafeOwnershipTransferred(msg.sender, newOwner);
1011    }
```

*Figure 12.1: `contracts/dos/DOS.sol`*

If a user were to mistakenly transfer ownership of their dSafe to the wrong address, they would permanently lose access to their dSafe and all of the assets held within. Additionally, since the `transferDSafeOwnership` function does not validate the provided `newOwner` address, a user could mistakenly transfer ownership of their dSafe to the zero address.

#### Exploit Scenario

Alice believes her account might have been compromised and decides to transfer ownership of her dSafe to a new account. She mistakenly provides the wrong address to the call to `transferDSafeOwnership`, permanently losing access to her dSafe and all of the assets held within.

#### Recommendations

Short term, implement a two-step process to transfer dSafe ownership, in which the owner proposes a new address and then the new address executes a call to accept the role, completing the transfer.

Long term, update the user-facing documentation to explain how the two-step dSafe ownership transfer mechanism works and why it is used.

### 13. Inconsistency between approveAndCall's implementation and NatSpec comments

| Severity: **Undetermined** | Difficulty: **Low** |
| --- | --- |
| Type: Data Validation | Finding ID: TOB-DOS-13 |
| Target: `contracts/dos/DOS.sol` | |

#### Description

The `approveAndCall` function's behavior is not aligned with the NatSpec comments surrounding it, which could result in unexpected behavior.

The function's comments indicate that the `spender` should be a dSafe and that the `onApprovalReceived` callback will be called on the `spender` address; however, the function implementation behaves differently.

```
602     /// @notice Approve an array of tokens and then call `onApprovalReceived` on
spender
603         /// @param approvals An array of ERC20 tokens with amounts, or ERC721
contracts with tokenIds
604         /// @param spender The address of the spender dSafe
605         /// @param data Additional data with no specified format, sent in call to
`spender`
606         function approveAndCall(
607             Approval[] calldata approvals,
608             address spender,
609             bytes calldata data
610         ) external override onlyDSafe whenNotPaused {
                // ...
617             if (!_checkOnApprovalReceived(msg.sender, 0, spender, data)) {
618                 revert WrongDataReturned();
619             }
                // ...
623         }
```

*Figure 13.1: `contracts/dos/DOS.sol`*

```
802     function _checkOnApprovalReceived(
803             address spender, // msg.sender from previous call
804             uint256 amount,
```

```
805             address target,
806             bytes memory data
807         ) internal returns (bool) {
                // ...
814             try IERC1363SpenderExtended(spender).onApprovalReceived(msg.sender,
amount, call) returns (
815                 bytes4 retval
816             ) {/* ... */} catch (bytes memory reason) {
                    /* ... */
824             }
825         }
```

*Figure 13.2: contracts/dos/DOS.sol*

The function does not verify that the `spender` address is a valid dSafe, and it calls the `onApprovalReceived` callback on the caller of the function instead of the address specified in `spender`. Either the implementation of the function or the NatSpec comments are incorrect, which could prevent the function from behaving correctly.

### Exploit Scenario

The Futureswap team decides to implement additional functionality that relies on `approveAndCall`. The developer checks the function's NatSpec comments and implements the new functionality assuming that the function behaves as described. This new functionality breaks protocol assumptions.

### Recommendations

Short term, adjust either the `approveAndCall` function's implementation or NatSpec comments so that the function's behavior and documentation are aligned. Additionally, review all important functions in the codebase to ensure that their implementations and documentation are aligned.

Long term, document the behavior of all important system functions, define system- and function-level invariants, and test them with Echidna.

### 14. NFT forwarding cannot be set via executeBatch

| Severity: **Informational** | Difficulty: **Low** |
| --- | --- |
| Type: Undefined Behavior | Finding ID: TOB-DOS-14 |
| Target: `contracts/dos/DSafeProxy.sol` | |

#### Description

The `executeBatch` function resets the `forwardNFT` variable to its previous value at the end of its execution, meaning that users can set the variable only temporarily during the function's execution.

The protocol allows dSafes to automatically forward NFTs that they receive to the DOS contract to be used as collateral. Users can configure their dSafes to forward NFTs by setting the `forwardNFT` variable in the `executeBatch` function to `true`.

```
1    function executeBatch(Call[] memory calls) external payable onlyOwner {
2       bool saveForwardNFT = forwardNFT;
3       forwardNFT = false;
4       dos.executeBatch(calls);
5       forwardNFT = saveForwardNFT;
6    }
```

*Figure 14.1: `contracts/dos/DSafeProxy.sol`*

However, because the function resets the `forwardNFT` value to its previous value before it finishes executing, the `forwardNFT` state variable cannot be permanently changed by calling this function.

#### Exploit Scenario

Alice owns a dSafe and currently has a debt position in the protocol. To make it easier to quickly deposit collateral, she attempts to set the `forwardNFT` state variable to `true` so that her NFTs can be automatically forwarded to DOS. The call to `executeBatch` passes, but the `forwardNFT` state variable's value is reset to `false` at the end of the function's execution.

Alice's position becomes liquidatable, and she transfers an NFT to her dSafe to improve her position, expecting it to be deposited as collateral. The NFT is not deposited, and Alice's position is liquidated.

### Recommendations

Short term, review the expected behavior of the NFT forwarding mechanism and clearly document it.

Long term, clearly document the behavior of all user-facing functions and make the documentation publicly available.

### 15. Unreachable code path in onTransferReceived2

| Severity: **Undetermined** | Difficulty: **Low** |
| --- | --- |
| Type: Undefined Behavior | Finding ID: TOB-DOS-15 |
| Target: `contracts/dos/DSafeProxy.sol` | |

#### Description

The `onTransferReceived2` function can take various execution paths depending on the value of the data that is passed to it. One of these paths executes a batch of calls on behalf of the given dSafe, but only if the caller of the function is the dSafe owner (line 216 in figure 15.1); however, this path is unreachable because the function has the `onlyTransferAndCall2` modifier, which means that only the `TransferAndCall2` contract can call this function.

```
201    function onTransferReceived2(
202        address operator,
203        address from,
204        ITransferReceiver2.Transfer[] calldata transfers,
205        bytes calldata data
206    ) external override onlyTransferAndCall2 returns (bytes4) {
           // ...
212        if (data.length == 0) {
213            /* just deposit in the proxy, nothing to do */
214        } else if (data[0] == 0x00) {
215            // execute batch
216            require(msg.sender == dos.getDSafeOwner(address(this)), "Not owner");
217            Call[] memory calls = abi.decode(data[1:], (Call[]));
218            dos.executeBatch(calls);
219        } else if (data[0] == 0x01) {
           // ...
268    }
```

*Figure 15.1: `contracts/dos/DSafeProxy.sol`*

#### Exploit Scenario

Alice uses the `TransferAndCall2` contract to transfer tokens to her dSafe and to execute calls on behalf of her dSafe. Because the caller of the `onTransferReceived2` function is the `TransferAndCall2` contract, this call always reverts.

### Recommendations

Short term, review the expected behavior of the `onTransferReceived2` function and make changes to ensure that all execution paths are reachable.

Long term, create documentation that highlights the intended use of the function and all of its execution branches. Add additional tests to check that all execution paths are reachable.

### 16. Risk of out-of-gas error due to loops in isSolvent

| Severity: **Low** | Difficulty: **High** |
| --- | --- |
| Type: Undefined Behavior | Finding ID: TOB-DOS-16 |
| Target: `DOS.sol` | |

#### Description

The loops inside the `isSolvent` function and the `getRiskAdjustedPositionValues` function (called by `isSolvent`), which loop through the given dAccount's debt and collateral, could exhaust the gas paid by the caller if there are too many elements to loop through. This would brick the given dAccount, as any operation that calls `isSolvent` would also exhaust the gas. This would also make it impossible for anybody to liquidate the dAccount.

```
 934    for (uint256 i = 0; i < erc20Infos.length; i++) {
 935        int256 totalDebt = erc20Infos[i].debt.tokens;
 936        int256 reserve = erc20Infos[i].collateral.tokens + totalDebt;
 937        FsUtils.Assert(
 938            IERC20(erc20Infos[i].erc20Contract).balanceOf(address(this)) >=
uint256(reserve)
 939        );
 940        require(reserve >= -totalDebt / leverage, "Not enough reserve for debt");
 941    }
```

*Figure 16.1: The loop inside `isSolvent` in `contracts/dos/DOS.sol`*

```
 664    for (uint256 i = 0; i < erc20Idxs.length; i++) {
 665        uint16 erc20Idx = erc20Idxs[i];
 666        ERC20Info storage erc20Info = erc20Infos[erc20Idx];
 667        int256 balance = getBalance(dSafe.erc20Share[erc20Idx], erc20Info);
 668        (int256 value, int256 riskAdjustedValue) =
erc20Info.valueOracle.calcValue(balance);
 669        totalValue += value;
 670        if (balance >= 0) {
 671            collateral += riskAdjustedValue;
 672        } else {
 673            debt -= riskAdjustedValue;
 674        }
 675    }
 676    for (uint256 i = 0; i < dSafe.nfts.length; i++) {
```

```
677         DSafeLib.NFTId nftId = dSafe.nfts[i];
678         (uint16 erc721Idx, uint256 tokenId) = getNFTData(nftId);
679         ERC721Info storage nftInfo = erc721Infos[erc721Idx];
680         (int256 nftValue, int256 nftRiskAdjustedValue) =
nftInfo.valueOracle.calcValue(tokenId);
681         totalValue += nftValue;
682         collateral += nftRiskAdjustedValue;
683     }
```

*Figure 16.2: The loop inside `getRiskAdjustedPositionValues` in `contracts/dos/DOS.sol`*

If one of these loops runs out of gas and the dAccount is bricked, the dAccount holder could withdraw some assets so that the loops need to perform fewer iterations.

During our testing (in which we did not use any ERC-721 tokens), we found that a dAccount would need to hold around 277 ERC-20 tokens for the loops to exhaust the gas. This number is higher than the current maximum number of ERC-20 tokens that a dAccount can hold (`dAccountErc20Idxs`), which is 256. However, as the comment on the `dAccountErc20Idxs` variable states, this maximum number could be increased during an upgrade. Additionally, if the dAccount also owns ERC-721 tokens, then the out-of-gas error would be possible within the 256 limit.

```
127     struct DSafe {
128         address owner;
129         mapping(uint16 => ERC20Share) erc20Share;
130         NFTId[] nfts;
131         // bitmask of DOS indexes of ERC20 present in a dSafe. `1` can be increased
on updates
132         uint256[1] dAccountErc20Idxs;
133     }
```

*Figure 16.3: The `DSafe` struct declaration in `contracts/dos/DOS.sol`*

### Exploit scenario

Alice, who owns a dSafe with 256 ERC-20 tokens and 10 NFTs, tries to deposit another NFT into her account. Because her account now holds 11 NFTs, the loop in `isSolvent` exhausts the gas and her transaction reverts.

### Recommendations

Short term, use unit testing to determine the actual number of ERC-721 and ERC-20 tokens in a dAccount that would cause the `isSolvent` and `getRiskAdjustedPositionValues` loops to exhaust the gas; update the implementation to enforce a maximum number of tokens below those limits. Additionally, update the user-facing documentation with an explanation of this failure scenario and instructions for users to unbrick their dAccounts.

Long term, consider using a different pattern for determining dAccount information that does not use loops or that requires far fewer iterations. This would completely negate the risk of out-of-gas reverts due to looping.

# A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

| Vulnerability Categories | |
|---|---|
| **Category** | **Description** |
| **Access Controls** | Insufficient authorization or assessment of rights |
| **Auditing and Logging** | Insufficient auditing of actions or logging of problems |
| **Authentication** | Improper identification of users |
| **Configuration** | Misconfigured servers, devices, or software components |
| **Cryptography** | A breach of system confidentiality or integrity |
| **Data Exposure** | Exposure of sensitive information |
| **Data Validation** | Improper reliance on the structure or values of data |
| **Denial of Service** | A system failure with an availability impact |
| **Error Reporting** | Insecure or insufficient reporting of error conditions |
| **Patching** | Use of an outdated software package or library |
| **Session Management** | Improper identification of authenticated users |
| **Testing** | Insufficient test methodology or test coverage |
| **Timing** | Race conditions or other order-of-operations flaws |
| **Undefined Behavior** | Undefined behavior triggered within the system |

| Severity Levels | |
|---|---|
| **Severity** | **Description** |
| **Informational** | The issue does not pose an immediate risk but is relevant to security best practices. |
| **Undetermined** | The extent of the risk was not determined during this engagement. |
| **Low** | The risk is small or is not one the client has indicated is important. |
| **Medium** | User information is at risk; exploitation could pose reputational, legal, or moderate financial risks. |
| **High** | The flaw could affect numerous users and have serious reputational, legal, or financial implications. |

| Difficulty Levels | |
|---|---|
| **Difficulty** | **Description** |
| **Undetermined** | The difficulty of exploitation was not determined during this engagement. |
| **Low** | The flaw is well known; public tools for its exploitation exist or can be scripted. |
| **Medium** | An attacker must write an exploit or will need in-depth knowledge of the system. |
| **High** | An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue. |

# B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

| Code Maturity Categories | |
|---|---|
| **Category** | **Description** |
| **Arithmetic** | The proper use of mathematical operations and semantics |
| **Auditing** | The use of event auditing and logging to support monitoring |
| **Authentication / Access Controls** | The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system |
| **Complexity Management** | The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions |
| **Cryptography and Key Management** | The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution |
| **Decentralization** | The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades |
| **Documentation** | The presence of comprehensive and readable codebase documentation |
| **Front-Running Resistance** | The system's resistance to front-running attacks |
| **Low-Level Manipulation** | The justified use of inline assembly and low-level calls |
| **Testing and Verification** | The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage |

| Rating Criteria | |
|---|---|
| **Rating** | **Description** |
| **Strong** | No issues were found, and the system exceeds industry standards. |
| **Satisfactory** | Minor issues were found, but the system is compliant with best practices. |
| **Moderate** | Some issues that may affect system safety were found. |
| **Weak** | Many issues that affect system safety were found. |
| **Missing** | A required component is missing, significantly affecting system safety. |
| **Not Applicable** | The category is not applicable to this review. |
| **Not Considered** | The category was not considered in this review. |
| **Further Investigation Required** | Further investigation is required to reach a meaningful conclusion. |

# C. Code Simplification Recommendations

During our review of the DOS codebase, we identified various opportunities to simplify the code's complexity. Code that is not overly complex is easier to maintain, easier to audit, and therefore less likely to contain bugs that go undetected. This appendix summarizes our recommendations for simplifying the code in the DOS core contracts.

## Reducing the Complexity of the executeBatch Execution Flow

The `DOS`, `DSafeProxy`, and `DSafeLogic` contracts currently contain logic for executing calls on behalf of a dSafe through the `executeBatch` function. Because `DOS` needs to check the solvency of a dAccount on each `executeBatch` execution, a very complex execution path is used to route the calls:

1. Calls to `executeBatch` on a `DSafeProxy` are routed to the `DSafeLogic.executeBatch` function through `delegatecall`.
2. The `DSafeLogic` contract externally calls the `DOS.executeBatch` function.
3. The `DOS.executeBatch` function externally calls the `DSafeProxy.executeBatch` function, which executes the calls.
4. The solvency of the position is checked by `DOS.executeBatch` at the end of its execution.

This complex execution path made our manual review of the codebase difficult, and its interrelated dependencies delayed our ability to test the codebase with Echidna. We recommend simplifying the execution path of `executeBatch` to avoid unnecessary call routing.

One way to simplify the execution path would be to have calls to `executeBatch` executed directly from the `DSafeProxy` contract:

1. `DSafeProxy` performs a `delegatecall` to `DSafeLogic.executeBatch`, which directly executes the calls.
2. The `DSafeLogic.executeBatch` function checks the solvency of the position by externally calling `DOS.isSolvent`; the `executeBatch` function reverts if the position is not solvent.

```
function executeBatch(Call[] memory calls) external payable onlyOwner {
    bool saveForwardNFT = forwardNFT;
    forwardNFT = false;
    CallLib.executeBatch(calls);
    forwardNFT = saveForwardNFT;
```

```
    if (!dos.isSolvent(address(this)) {
        revert Insolvent();
    }

}
```

*Figure C.1: A refactored `executeBatch` function in `DSafeLogic`*

## Reducing the Difficulty of Initializing New DOS Contracts

Because of the number of contracts that DOS relies on, it is difficult to initialize a new DOS contract from scratch. To set up a new DOS contract, the creator must create and initialize the following contracts:

- `VersionManager`
- `DOSConfig`
- `DSafeLogic`
- `GovernanceProxy`
- `HashNFT`
- `Voting`
- `DOS`

The current documentation does not provide step-by-step guidance on how to do this; instead, developers must learn the process by looking at test cases and contract code. This makes it more likely that developers will make mistakes while initializing new DOS contracts and makes the code more difficult to follow.

One way to ease the process of initializing new DOS contracts is to document the process or to provide example Solidity code that performed the initialization. Another solution is to reduce the number of contracts needed to initialize DOS contracts. For example, `DOS`, `VersionManager`, and `GovernanceProxy` could be merged into one contract since there is no way to update a DOS contract's `VersionManager` or `GovernanceProxy` anyway. A third solution is to make contract constructors initialize their dependencies rather than taking addresses as user input. For example, the `Voting` contract's constructor could initialize a new `HashNFT` instead of accepting it as a parameter.

## Reducing the Difficulty of Calling DOS Contract Functions

When calling DOS functions, users need to manually create their own `Call` structs, using `abi.encodeWithSignature` to create the call data bytes. This is an error-prone process, and it is difficult to verify that a `Call` struct will be correct before running it. One way to

deal with this issue is to write a set of helper functions that automatically construct `Call` structs based on user-provided parameters. Here is an example of what this function could look like:

```
function getTransferERC20Call(address receiver, address erc20, address to, uint256
amount) internal pure returns (Call memory) {
    return Call(receiver,
abi.encodeWithSignature("transferERC20(address,address,uint256)", erc20, to,
amount), 0);
}
```

*Figure C.2: A helper function for calling `getTransferERC20`*

We wrote a full set of these helper functions for our own convenience while creating Echidna tests; this library could be repurposed as a general-use library for DOS users.

In addition, the system of proxies used throughout the codebase is fairly complicated and not well documented. First-time users need to learn that, for example, they should call `DSafeProxy` rather than `DSafeLogic`, and that they should call `DOS` rather than `DOSConfig`, even for configuration-related functions defined in `DOSConfig`. To mitigate this problem, we recommend writing clear documentation on how the proxy system works and which contracts to call under each circumstance.

# D. System Invariants

This appendix lists DOS system properties that should always hold true. This list is nowhere near exhaustive; these properties are simply those on our to-do list that we did not have time to test with Echidna. We recommend extending our Echidna tests to test these invariants. We also recommend that the Futureswap team identify any additional system properties and develop tests for them.

**depositERC20**
The following properties of the `DOS.depositERC20` function should hold:

- `depositERC20` reverts if `amount` is greater than the user's balance.
- `depositERC20` reverts if the token is not allowlisted.

**withdrawFull**
The following properties of the `DOS.withdrawFull` function should hold:

- Withdrawing the entire balance of an asset removes that asset from the user's dAccount.
- Calls to `withdrawFull` never revert under the following preconditions:
    - The asset is in the list of the dAccount's assets.
    - The dAccount's collateral is greater than or equal to its debt.
    - The dAccount's asset balance is greater than or equal to `0`.

**Liquifier**
The following properties related to the `Liquifier` contract should hold:

- The liquidator accrues no debt when liquifying a target dSafe under the following preconditions:
    - The `erc20s` array does not include the `numeraire`.
    - The `erc20s` array contains all of the tokens that the liquidated dSafe has debt or collateral in.
    - The `erc20s` array contains any token that would be obtained by NFT termination.
    - The range of `erc20sAllowedPriceRanges` is not too narrow.

- `Liquifier` operations always revert if the `numeraire` is included in the `erc20s` array.

# E. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `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 --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below 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 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.

## 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 whether 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.

## ERC20 Tokens

### ERC20 Conformity Checks

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.

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.

#### Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`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.

### 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.

## ERC721 Tokens

### ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❑ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❑ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets.
- ❑ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❑ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❑ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❑ **The `ownerOf` function reverts if the `tokenId` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❑ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❑ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

### Common Risks of the ERC721 Standard

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❑ **The `onERC721Received` callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.

- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.

# F. 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.

- **Use consistent naming conventions throughout the codebase.** Throughout the codebase, variables are named with either a prepended underscore, an appended underscore, or no underscore at all. Consider deciding on a specific convention and apply it to each file. This will improve the readability of the implementation.
- **Use custom errors throughout the codebase.** Errors throughout the codebase are not consistent. Some are custom errors, and others are `require` statements with string revert messages. Consider normalizing the codebase's errors by using custom errors in all places.
- **Create a separate `ERC20PoolLib` library.** Currently, the `DSafeLib` library contains functions that do not act on dSafes but on `ERC20Pool` structs. Consider splitting the `DSafeLib` library's functions that operate on `ERC20Pool` structs into a separate `ERC20PoolLib` library. This will improve the readability of the code.
- **Put each contract/library into its own file.** Currently, most files contain multiple contracts. Although in some cases this file organization makes sense, in many cases the readability of the codebase would be improved if the contracts were in separate files.
- **Remove the `DuoswapV2Pair` contract's downcasting of `uint256` timestamps to `uint32`.** This operation will automatically calculate `% 2 ** 32`.

```
113    uint32 blockTimestamp = uint32(block.timestamp % 2 ** 32);
```

*Figure F.1: `contracts/duoswapV2/DuoswapV2Pair.sol`*

- **Replace the `== false` check in the `DOS.addERC721Info` function with an exclamation mark at the beginning of the `if` condition.** This is a simpler way to explicitly check for `false`.

```
1084    if (IERC165(erc721Contract).supportsInterface(type(IERC721).interfaceId) ==
false) {
```

*Figure F.2: `contracts/dos/DOS.sol`*

- **Either remove the internal function `getERC721Info` or update the implementation to use it.** This function is currently unused.

```
306    function getERC721Info(IERC721 erc721) internal view returns (ERC721Info
storage, uint16) {
307        if (infoIdx[address(erc721)].kind != ContractKind.ERC721) {
308            revert NotRegistered(address(erc721));
309        }
310        uint16 idx = infoIdx[address(erc721)].idx;
311        return (erc721Infos[idx], idx);
312    }
```

*Figure F.3: contracts/dos/DOS.sol*

- **Update the `transferFromAndCall2Impl` function so that it returns a value.** It does not currently return a value, but all functions that call it return its currently nonexistent return value.

```
148    function transferFromAndCall2Impl(
149        address from,
150        address receiver,
151        address weth,
152        ITransferReceiver2.Transfer[] calldata transfers,
153        bytes memory data
154    ) internal {
```

*Figure F.4: contracts/dos/TransferAndCall2.sol*

```
63    return transferFromAndCall2Impl(msg.sender, receiver, address(0), transfers,
data);
```

*Figure F.5: An example call to `transferFromAndCall2Impl` in contracts/dos/TransferAndCall2.sol*