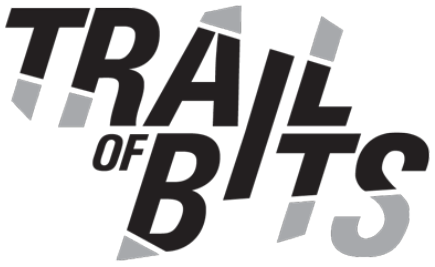

Advanced Blockchain

Security Assessment

April 19, 2021

Prepared For:
Michael Geike | *Advanced Blockchain*
geike@advancedblockchain.com

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

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

Executive Summary

From March 29 to April 16, 2021, Advanced Blockchain engaged Trail of Bits to review the security of the Solidity smart contracts used in several affiliated projects. Trail of Bits conducted this assessment over six person-weeks, with two engineers working from the commits and repositories listed in the Project Dashboard.

During the first week of the engagement, we reviewed the strategies in the Polkastrategies repository and began reviewing the EQLC stablecoin, which is a fork of the Maker Protocol. During the second week, we finished reviewing EQLC and began to review ForceDAO, which is forked from Harvest Finance, and the Quads strategies. In the final week, we assessed Blacksmith. We conducted a manual review to identify any vulnerabilities introduced in the forked projects and to detect any contract logic concerns or common Solidity issues. We also used Slither, our Solidity static analysis tool, to provide broad coverage of the projects and to detect common issues.

Trail of Bits identified 27 issues ranging from high to informational severity. These issues stemmed from a failure to adhere to safe smart contract security practices, including data validation processes; for example, the code assumed the success of external contract calls, which could result in a loss of user funds. Additionally, the code did not showcase development best practices: a few issues derived directly from copy-paste errors in the code, which introduced deviations in the behavior of contracts that should behave similarly. Thorough system testing could have detected these issues; however, the codebase lacked concise documentation and proper tests.

Appendix C details a Slither script meant to help ensure that all functions are protected by the `onlyOwner` modifier when necessary. We discuss code quality concerns not related to any particular security issue in Appendix D. Appendix E provides recommendations for maintaining user security in interactions with Blacksmith lending pairs, and Appendix F includes guidance for interacting with arbitrary tokens.

The projects under review are works in progress; most had minimal test coverage (or none whatsoever) and limited documentation beyond code comments, which generally hindered the verification of code correctness. While most projects appeared to be feature-complete, the project teams should address the concerns detailed in this report, including those around documentation and testing, before deployment. We would also suggest conducting an additional security review, especially of the non-forked projects, after addressing the issues and code quality concerns identified in this report.

Project Dashboard

Application Summary

Name	Advanced Blockchain
Versions	Cyclical (CyclicalFinance) ae97a62ed3b1c828641653bc1080cecb8d51a9ef Deployment (ComposableFinance) 135f7414f1f549cdd1f0b8a4d3f7ac99f727293a EQLC (ComposableFinance) 6b22ad7395e99182b78e3317f2e6b40bd87c34f9 airdrop (ForceDAO) 440e9075d711f5d28b6d51349843322732dcadfe blacksmith (cryptotechmaker) B682d9965752551302d8b68f5162855068533ceb c2aff1c7be8189efee7eb5a1f6a7fa6419eb16b3 contracts (ForceDAO) 9f3900bd1944647923ddcebee841d2f413d2a032 polkastrategies (cryptotechmaker) d8351204638943ed6b98782efa62997ddc4ebfb4 token (ForceDAO) 12324c69ea666bd0a72d35d6b56bc13e2c47a161
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	March 29 – April 16, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	11	■■■■■■■■■■■
Total Medium-Severity Issues	1	■
Total Low-Severity Issues	3	■■■
Total Informational-Severity Issues	11	■■■■■■■■■■■
Total Undetermined-Severity Issues	1	■
Total	27	

Category Breakdown

Access Controls	1	■
Auditing and Logging	1	■
Authentication	1	■
Configuration	4	■■■■
Data Validation	10	■■■■■■■■■■
Documentation	1	■
Patching	5	■■■■■
Testing	1	■
Undefined Behavior	3	■■■
Total	27	

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 code relied heavily on privileged operations. Roles were not split between different addresses or clearly explained in user documentation. Additionally, the codebase did not consistently include access control tests.
Arithmetic	**Moderate.** SafeMath and Solidity v0.8.0 arithmetic operations were used throughout the system but were not tested with automated analysis tools. Additionally, the formulas in the code would benefit from documentation (TOB-ADV-016). We also identified issues in the calculation of user rewards; specifically, the function that calculates a user's share of a reward fails to track balance changes in `polkastrategies` (TOB-ADV-001).
Assembly Use/Low-Level Calls	**Weak.** The contracts did not use assembly, but the lack of a contract existence check in the proxy could allow a call to fail silently (TOB-ADV-010). The codebase also lacked a few other return checks (TOB-ADV-007, TOB-ADV-012).
Centralization	**Moderate.** Throughout the system, it was easy to identify the number of privileged actors. However, each set of contracts had different admin controls; some roles were initiated at deployment and remained unchanged thereafter, and some could be transferred to a different address. All privileged roles were controlled by the contract owner. However, no user documentation on deployment or centralization risks was provided.
Code Stability	**Weak.** The code was undergoing significant changes during the audit and will likely continue to evolve before reaching its final version.
Contract Upgradeability	**Weak.** Each repository used a different contract upgradeability strategy. Because many of these upgradeability mechanisms did not undergo testing, the state variable layout must be verified manually. In addition, there was no continuous integration pipeline to which `slither-check-upgradeability` could be added.
Function Composition	**Weak.** While some functions were documented inline, a significant amount of logic was duplicated throughout the contracts (TOB-ADV-004). There was little external documentation on these

	functions, which also lacked simple unit tests.
Front-Running	**Moderate.** An attacker could front-run the initialization process (TOB-ADV-009) and EIP-2612 `permit` calls (TOB-ADV-014).
Monitoring	**Satisfactory**. Most functions emitted events for critical operations, effectively monitoring off-chain activity; however, a few did not (TOB-ADV-026). Additionally, we were not provided with an incident response plan or information on the use of off-chain components in behavior monitoring.
Specification	**Missing.** We received documentation on certain repositories but were not provided clear specifications or detailed documentation on user flow (TOB-ADV-016).
Testing & Verification	**Weak.** The provided repositories lacked testing across all aspects of the system. Statement and branch coverage was also very limited (TOB-ADV-005). The test suite would benefit from automated function verification and from continuous integration across all branches, which would allow it to pre-detect code that could introduce unexpected behavior.

Engagement Goals

The engagement was scoped to provide a security assessment of the Solidity smart contracts in the Polkastrategies, EQLC, ForceDAO, Quads, and Blacksmith systems. A list of the repositories and commits is provided in the Project Dashboard.

Specifically, we sought to answer the following questions:

- Is it possible for participants to steal or lose tokens?
- Are there appropriate access controls on the system components?
- Can participants perform denial-of-service or phishing attacks against any of the system components?
- Can flash loans be used to exploit the protocol?
- Is receipt token bookkeeping performed properly?
- Are rewards properly calculated and distributed to users?
- Are there any issues in how the projects interface with third-party systems such as Uniswap?

Coverage

Polkastrategies. The Polkastrategies project comprises yield-farming strategies into which users can deposit tokens to earn rewards and raise funds for Polkadot parachain auctions. We reviewed these contracts to ensure that users would receive the appropriate number of tokens when executing a deposit or withdrawal, that rewards would be calculated and distributed equitably, and that the strategies would interact properly with the underlying third-party systems.

EQLC. The EQLC stablecoin project is a fork of the Maker Protocol. We reviewed the changes made after the fork to ensure that they did not introduce any vulnerabilities or undefined behavior.

ForceDAO. ForceDAO is an investment strategy system forked from Harvest Finance. We reviewed the changes introduced after the fork to ensure that accounting and interactions with the underlying third-party protocols were sound and that user funds could not be locked in the contract or accessed by an attacker.

Blacksmith. Blacksmith is an isolated-pair lending platform. We reviewed the system to ensure that vault balances were tracked properly and that custodied funds had adequate safeguards. We reviewed the lending pair code to verify that collateral and debt positions were tracked accurately and liquidations were carried out properly. We also assessed the work-in-progress code for the Uniswap v3 liquidity provider position adapter.

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

❑ **Update the reward calculations such that they account for the length of time that a deposit has been held by the strategy.** TOB-ADV-001

❑ **Add in the missing negation to ensure that receipt tokens are correctly burned during withdrawals.** TOB-ADV-002

❑ **Include a `lockTime` check in the withdraw function instead of the deposit function.** This check will prevent users from withdrawing their funds before the lock time has passed. TOB-ADV-003

❑ **Use inheritance to allow code to be reused across contracts.** Changes to one inherited contract will be applied to all files without requiring developers to copy and paste them. TOB-ADV-004

❑ **Ensure that the unit tests cover all public functions at least once, as well as all known corner cases.** TOB-ADV-005

❑ **Measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.** TOB-ADV-006

❑ **Either wrap the `transferFrom` call in a require statement or use a `safeTransfer` function.** Taking either step will ensure that if a transfer fails, the transaction will also fail. TOB-ADV-007

❑ **Add the missing input validation to the `setProfitSharingNumerator` function in the `ProfitNotifier` contracts.** TOB-ADV-008

❑ **Use a factory pattern that will prevent front-running. Alternatively, ensure that the deployment scripts have robust protections against front-running attacks.** That way, if front-running of the initialization functions does occur, the development teams will be alerted to it. TOB-ADV-009

❑ **Implement a contract existence check before a `delegatecall`.** Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions. TOB-ADV-010

❑ **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. TOB-ADV-011

❑ **Document the fact that a target contract call can succeed even if no code has been executed.** TOB-ADV-012

❑ **To prevent post-deployment forks from affecting calls to `permit`, detect chainID changes and regenerate the domain separator when necessary, or add the chainID opcode to the signature schema.** TOB-ADV-013

❑ **Develop user documentation on edge cases in which the signature-forwarding process can be front-run or an attacker can steal a user's tokens via phishing.** TOB-ADV-014

❑ **Fix the comment documentation and remove duplicated code to prevent future refactoring mistakes.** TOB-ADV-015

❑ **Review and properly document all aspects of the protocol to clarify the code's expected behavior.** TOB-ADV-016

❑ **Add zero-value checks on all function arguments to ensure users can't accidentally set incorrect values, misconfiguring the system.** TOB-ADV-017

❑ **Use neither `ABIEncoderV2` nor any other experimental Solidity feature.** Refactor the code such that structs do not need to be passed to or returned from functions. TOB-ADV-018

❑ **Prevent contract names from being reused or change the compilation framework.** TOB-ADV-019

❑ **Document corner cases that could result from interacting with non-ERC20-compliant tokens used by Balancer proxies. That way, users will be aware of the risks.** TOB-ADV-020

❑ **Review the codebase and document the source and version of each dependency.** Include third-party sources as submodules in your Git repository to maintain internal path consistency and ensure that dependencies are updated periodically. TOB-ADV-021

❑ **Analyze the effects of a deviation from the actual number of blocks mined annually in borrow rate calculations and document the associated risks.** TOB-ADV-022

❑ **Use a two-step process for all non-recoverable critical operations to prevent irrevocable mistakes.** TOB-ADV-023

❑ **Warn users about the risks associated with malicious `LendingPair` contracts.** TOB-ADV-024

❑ **Introduce lower and upper bounds for all configurable parameters in the system to limit privileged users' abilities.** TOB-ADV-025

❑ **Add events for all critical operations.** Events aid in contract monitoring and the detection of suspicious behavior. TOB-ADV-026

❑ **Set addresses when contracts are created rather than using hard-coded contract values.** This will facilitate testing and prevent the execution of calls with incorrect hard-coded addresses. TOB-ADV-027

Long Term

❑ **Thoroughly document and test every function.** Testing of the contracts' behavior would have caught many of the issues in this report. TOB-ADV-001, TOB-ADV-002, TOB-ADV-003, TOB-ADV-015

❑ **Minimize the amount of manual copying and pasting required to apply changes made to one file to other files.** TOB-ADV-004

❑ **Integrate coverage analysis tools into the development process and regularly review the coverage.** TOB-ADV-005

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** TOB-ADV-006

❑ **Always carry out input validation when passing variables to functions, especially to set system parameters.** Additionally, using Echidna or Manticore, test system invariants to ensure that all properties hold. TOB-ADV-008, TOB-ADV-025

❑ **Carefully review the Solidity documentation, especially the "Warnings" section, as well as the pitfalls of using the `delegatecall` proxy pattern.** TOB-ADV-009, TOB-ADV-010

❑ **Consider integrating automated dependency auditing into the development workflow.** If dependencies cannot be updated when a vulnerability is disclosed, ensure that the codebase does not use and is not affected by the vulnerable functionality of the dependency. TOB-ADV-011

❑ **Identify and document all areas of the code that can be affected by the lack of return value checks.** TOB-ADV-012

❑ **Identify and document the risks associated with having forks of multiple chains and develop related mitigation strategies.** TOB-ADV-013

❑ **Document best practices for EQLC and Blacksmith users.** That way, users will know how to respond to requests for signatures. TOB-ADV-014

❑ **Consider writing a formal specification of the protocol.** TOB-ADV-016

❑ **Use Slither, which will catch functions that lack zero checks, as well as unsafe pragmas, duplicated contract names, and unused return values.** TOB-ADV-007, TOB-ADV-017, TOB-ADV-018, TOB-ADV-019

❑ **Analyze all third-party integrations and document the risks that arise when users interact with them.** TOB-ADV-020

❑ **Use an Ethereum development environment and NPM to manage packages in the project.** TOB-ADV-021

❑ **Identify all variables that are affected by external factors, and document the risks associated with deviations from their true values.** TOB-ADV-022

❑ **Identify and document all possible actions that can be taken by privileged accounts and their associated risks.** This will facilitate reviews of the codebase and prevent future mistakes. TOB-ADV-023

❑ **Document the contracts' interactions and the ways in which assets are transferred between the components.** TOB-ADV-024

❑ **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. TOB-ADV-026

❑ **To ensure that contracts can be tested and reused across networks, avoid using hard-coded parameters.** TOB-ADV-027

Automated Testing and Verification

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

- 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.

Echidna Properties

ID	Property	Script
1	The numerator in `ProfitNotifier` cannot be larger than the denominator.	**FAILED (TOB-ADV-008)**

Slither Scripts

ID	Property	Script
1	In the Polkastrategies codebase, all functions have `onlyOwner` modifiers when necessary.	APPENDIX C

Polkastrategies Findings Summary

#	Title	Type	Severity
1	Reward share calculations do not reflect balance changes	Data Validation	High
2	Receipt tokens are not burned when a user withdraws all tokens	Data Validation	Informational
3	Lack of `lockTime` check enables withdrawals during vesting period	Data Validation	Medium

1. Reward share calculations do not reflect balance changes

Severity: High | Difficulty: Low
Type: Data Validation | Finding ID: TOB-ADV-001
Target: `polkastrategies/contracts/strategies/{ForceSC.sol, ForceSLP.sol, HarvestDAI.sol}`

Description
To withdraw ETH from a strategy, the contract calls `getPendingRewards` to calculate the amount of rewards owed to the user. This calculation is based on the current value of the user's net deposits and the time since the user's first deposit.

```
/**
  * @notice View function to see pending rewards for account.
  * @param account user account to check
  * @return pending rewards
  */
 function getPendingRewards(address account) public view returns (uint256) {
     UserInfo storage user = userInfo[account];
     if (user.amountfDai == 0) {
         return 0;
     }
     uint256 rewardPerBlock = 0;
     uint256 balance = IERC20(farmToken).balanceOf(address(this));
     uint256 diff = block.timestamp.sub(firstDepositTimestamp);
     if (diff == 0) {
         rewardPerBlock = balance;
     } else {
         rewardPerBlock = balance.div(diff);
     }
     uint256 rewardPerBlockUser =
         rewardPerBlock.mul(block.timestamp.sub(user.joinTimestamp));
     uint256 ratio = _getRatio(user.amountfDai, totalDeposits, 18);
     return (rewardPerBlockUser.mul(ratio)).div(10**18);
 }
```

Figure 1.1: polkastrategies/contracts/strategies/HarvestDAI.sol#L699-L721

This method of calculating the reward share could provide an honest user who makes multiple deposits into a strategy with more rewards than the user actually earned, and a malicious user could drain a contract of all rewards.

Exploit Scenario

An attacker deposits a small number of tokens into the `HarvestDAI` strategy. After the lockup period, the attacker deposits a much larger amount of ETH before withdrawing all of his tokens. The attacker receives more rewards than his deposit entitles him to and then repeatedly makes deposits and executes withdrawals to drain all of the rewards from the contract.

Recommendations
Short term, update the reward calculations such that they account for the length of time that a deposit has been held by the strategy.

Long term, improve the test coverage to ensure that the correct amount of rewards is returned in various scenarios.

2. Receipt tokens are not burned when a user withdraws all tokens

Severity: Informational | Difficulty: Low
Type: Data Validation | Finding ID: TOB-ADV-002
Target: `polkastrategies/contracts/strategies/{ForceSC.sol, ForceSLP.sol, HarvestDAI.sol}`

Description
Because blacklisted users do not receive receipt tokens for their deposits, the contract checks whether a user is included on the blacklist when executing withdrawals, before attempting to burn the appropriate number of receipt tokens. However, the blacklist check is missing a negation and does not burn the tokens of users attempting to withdraw their entire token balances (or more). Because of the additional bookkeeping performed by the strategy, this flaw does not appear to be exploitable; however, it could be confusing to users.

```
      if (results.obtainedfDai < user.amountfDai) {
           user.amountfDai = user.amountfDai.sub(results.obtainedfDai);
           if(!user.wasUserBlacklisted){
               user.amountReceiptToken = user.amountReceiptToken.sub(results.obtainedfDai);
               receiptToken.burn(msg.sender,results.obtainedfDai);
               emit ReceiptBurned(msg.sender, results.obtainedfDai);
           }
       } else {
           user.amountfDai = 0;
           exit = true;
           if(user.wasUserBlacklisted){
               receiptToken.burn(msg.sender, user.amountReceiptToken);
               emit ReceiptBurned(msg.sender, user.amountReceiptToken);
               user.amountReceiptToken = 0;
           }
       }
```

Figure 2.1: `polkastrategies/contracts/strategies/HarvestDAI.sol#L521-L536`

Exploit Scenario
Alice has deposited tokens into the HarvestDAI strategy. She withdraws all of her tokens and rewards. Because of the missing negation, Alice's receipt tokens are not burned as part of the withdrawal. Alice notices that she still has receipt tokens and becomes confused when she attempts to make another withdrawal but does not receive any more ETH.

Recommendations
Short term, add in the missing negation to ensure that receipt tokens are correctly burned during withdrawals.

Long term, thoroughly document and test every function. This issue would have been caught by a test verifying that receipt token balances remain consistent with the strategy's bookkeeping.

3. Lack of `lockTime` check enables withdrawals during vesting period

Severity: Medium | Difficulty: Low
Type: Data Validation | Finding ID: TOB-ADV-003
Target: `polkastrategies/contracts/strategies/SushiETHSLP.sol`

Description
The `polkastrategies` contracts are supposed to lock user deposits for 120 days. However, `SushiETHSLP` fails to check the lock time on withdrawal, allowing users to withdraw assets immediately after depositing them.

The `ForceSLP` contract checks that the lock time has passed before it allows a user to withdraw tokens and rewards:

```
    /**
     * @notice Withdraw tokens and claim rewards
     * @param deadline Number of blocks until transaction expires
     * @return Amount of ETH obtained
     */
    function withdraw(uint256 amount, uint256 deadline)
        public
        nonReentrant
        returns (uint256)
    {
        // -----
        // validation
        // -----
        uint256 receiptBalance = receiptToken.balanceOf(msg.sender);

        _defend();
        require(deadline >= block.timestamp, "DEADLINE_ERROR");
        require(amount > 0, "AMOUNT_0");
        UserInfo storage user = userInfo[msg.sender];
        require(user.amountdSlp >= amount, "AMOUNT_GREATER_THAN_BALANCE");
        if(!user.wasUserBlacklisted){
            require(receiptBalance >= user.amountReceiptToken, "RECEIPT_AMOUNT");
        }
        if(lockTime>0){
            require(user.timestamp.add(lockTime)<= block.timestamp,"LOCK_TIME");
        }
```

Figure 3.1: `polkastrategies/contracts/strategies/ForceSLP.sol#L511-L536`

The withdraw function in `SushiETHSLP` lacks such a check:

```
    /**
     * @notice Withdraw tokens and claim rewards
     * @param deadline Number of blocks until transaction expires
     * @return Amount of ETH obtained
     */
    function withdraw(uint256 amount, uint256 deadline)
        public
        nonReentrant
        returns (uint256)
    {
        // -----
        // validation
        // -----
        uint256 receiptBalance = receiptToken.balanceOf(msg.sender);

        _defend();
        require(deadline >= block.timestamp, "DEADLINE_ERROR");
        require(amount > 0, "AMOUNT_0");
        UserInfo storage user = userInfo[msg.sender];
        require(user.amount >= amount, "AMOUNT_GREATER_THAN_BALANCE");
        if (!user.wasUserBlacklisted) {
            require(receiptBalance >= user.amount, "RECEIPT_AMOUNT");
        }
```

Figure 3.2: polkastrategies/contracts/strategies/SushiETHSLP.sol#L551-L573

Instead, the `lockTime` check was erroneously included in the deposit function in the `SushiETHSLP` contract:

```
    /**
     * @notice Deposit to this strategy for rewards
     * @param deadline Number of blocks until transaction expires
     * @return Amount of LPs
     */
    function deposit(uint256 deadline)
        public
        payable
        nonReentrant
        returns (uint256)
```

```
{
    // -----
    // validate
    // -----
    _defend();
    require(msg.value > 0, "ETH_0");
    require(deadline >= block.timestamp, "DEADLINE_ERROR");
    require(totalEth.add(msg.value) <= cap, "CAP_REACHED");

    uint256 prevEthBalance = address(this).balance.sub(msg.value);
    uint256 prevTokenBalance = IERC20(token).balanceOf(address(this));

    DepositData memory results;
    UserInfo storage user = userInfo[msg.sender];
    if (lockTime > 0) {
        require(
            user.timestamp.add(lockTime) <= block.timestamp,
            "LOCK_TIME"
        );
    }
```

Figure 3.3: polkastrategies/contracts/strategies/SushiETHSLP.sol#L414-443

Exploit Scenario

Eve deposits ETH into the `SushiETHSLP` contract. As the lock time is not checked in the `withdraw` function, she can bypass the vesting period and withdraw her funds immediately.

Recommendations

Short term, include a `lockTime` check in the `withdraw` function instead of the deposit function. This check will prevent users from withdrawing their funds before the lock time has passed.

Long term, thoroughly document and test every function. Testing of the `withdraw` function's behavior would have caught this issue.

EQLC Findings Summary

#	Title	Type	Severity
12	Lack of return value check on `DssChangeRatesSpell` can lead to unexpected results	Data Validation	High
15	Duplicated calls to file a pip in spotter contract	Configuration	Low

12. Lack of return value check in `DssChangeRatesSpell` can lead to unexpected results

Severity: High
Type: Data Validation
Target: `EQLC/contracts/DssChangeRatesSpell`

Difficulty: High
Finding ID: TOB-ADV-012

Description
The `cast` function in the `DssChangeRatesSpell` contract does not check the return value of `exec`. Without this check, the EQLC system may exhibit unexpected behavior.

To change parameter values, the `cast` function calls `exec` to execute a spell:

```
    function cast() public {
        require(!done, "spell-already-cast");
        done = true;
        PauseLike(pause).exec(action, tag, sig, eta);
    }
```

Figure 12.1: `EQLC/contracts/DSSChangeRatesSpell.sol#L76-L80`

However, according to the MakerDao documentation, the spell will be marked as “done” only if the call succeeds. If the target contract returns false upon failing, the execution will fail silently.

```
Note that the spell is only marked as "done" if the CALL it makes succeeds, meaning it did
not end in an exceptional condition and it did not revert. Conversely, contracts that use
return values instead of exceptions to signal errors could be successfully called without
having the effect you might desire. "Approving" spells to take action on a system after the
spell is deployed generally requires the system to use exception-based error handling to
avoid griefing.
```

Figure 12.2: MakerDao documentation, spell "Gotchas" section

Exploit Scenario
Alice calls the `DssChangeRatesSpell` contract to cast a spell intended to change a contract parameter. The target contract returns false instead of reverting. The transaction is successful, so the spell is marked as cast. However, because the call silently fails, the target parameter does not actually change, even though the target was set properly.

Recommendations
Short term, document the fact that a target contract call can succeed even if no code has been executed.

Long term, identify and document all areas of the code that can be affected by the lack of return value checks.

15. Duplicated calls to file a pip in spotter contract

Severity: Low
Type: Configuration
Target: `EQLC/contracts/DssAddIlkSpell.sol`

Difficulty: Medium
Finding ID: TOB-ADV-015

Description
The `DSSAddIlkSpell` contract twice files a pip in the spotter contract.

The `deploy` function takes a list of addresses to call and arguments to execute and change. However, the two config lines highlighted in figure 15.1 execute the same call to `spotter.file`:

```
    function deploy(bytes32 ilk_, address[8] calldata addrs, uint[5] calldata values)
external {
        // addrs[0] = vat
        // addrs[1] = cat
        // addrs[2] = jug
        // addrs[3] = spotter
        // addrs[4] = end
        // addrs[5] = join
        // addrs[6] = pip
        // addrs[7] = flip
        // values[0] = line
        // values[1] = mat
        // values[2] = duty
        // values[3] = chop
        // values[4] = dunk

        ConfigLike(addrs[3]).file(ilk_, "pip", address(addrs[6])); // vat.file(ilk_, "pip",
pip);
        [...]
        // //new added
        ConfigLike(addrs[3]).file(ilk_, "pip", addrs[6]); // spotter.file(ilk_, "pip", pip);
        }
```

Figure 15.1: `EQLC/contracts/DssAddIlkSpell.sol#L26-L40`

Additionally, the comment on the first call to `spotter.file(ilk)` is incorrect, as there are no calls to `vat.file()` in this config.

Exploit Scenario
Bob, a member of the development team, decides to refactor calls in the contracts. He changes the first instance of the spotter call but not the second. Because the second statement is still present, his code does not reflect the update.

Recommendations
Short term, fix the comment documentation and remove duplicated code to prevent future refactoring mistakes.

Long term, thoroughly document and test every function to detect duplicated statements that could cause unintended behavior.

ForceDAO Findings Summary

#	Title	Type	Severity
7	Lack of return value check in `notifyPools` can lead to unexpected results	Data Validation	High
8	`ProfitNotifier` doesn't validate new numerators	Data Validation	High
10	Lack of contract existence check on `delegatecall` will result in unexpected behavior	Data Validation	High
27	Use of hard-coded addresses may cause errors	Patching	Informational

7. Lack of return value check in `notifyPools` can lead to unexpected results

Severity: High
Type: Data Validation
Target: `ForceDAO/contracts/NotifyHelper.sol`

Difficulty: High
Finding ID: TOB-ADV-007

Description
The `NotifyHelper` contract does not check the return value of a call to transfer tokens. Without this check, the ForceDAO system may exhibit unexpected behavior.

The `notifyPools` function calls ERC20's `transferFrom` function to transfer tokens to pools:

```
/**
 * Notifies all the pools, safe guarding the notification amount.
 */
function notifyPools(
    uint256[] memory amounts,
    address[] memory pools,
    uint256 sum
) public onlyOwner {
    require(
        amounts.length == pools.length,
        "Amounts and pools lengths mismatch"
    );

    uint256 check = 0;
    for (uint256 i = 0; i < pools.length; i++) {
        require(amounts[i] > 0, "Notify zero");
        IRewardPool pool = IRewardPool(pools[i]);
        IERC20 token = IERC20(pool.rewardToken());
        token.transferFrom(msg.sender, pools[i], amounts[i]);
        IRewardPool(pools[i]).notifyRewardAmount(amounts[i]);
        check = check.add(amounts[i]);
    }
    require(sum == check, "Wrong check sum");
}
```

Figure 7.1: ForceDAO/contracts/NotifyHelper.sol#L10-L33

As a result, if the target token implementation returns false instead of reverting, the `notifyPools` function will not detect the failed `transferFrom` call. The `notifyPools` function may return true despite its failure to transfer tokens to their respective pools.

Exploit Scenario
Alice, the owner of a contract, calls the `notifyPools` function. The contract interacts with a token that returns false instead of reverting, such as BAT. Alice forgets to approve the token's transfer to the contract, causing the call to `transferFrom` to fail. As a result, when Alice invokes the contract, the tokens are not sent, but the transaction succeeds.

Recommendations
Short term, either wrap the `transferFrom` call in a require statement or use a `safeTransfer` function. Taking either step will ensure that if a transfer fails, the transaction will also fail.

Long term, integrate Slither into the continuous integration pipeline to catch missing return value checks.

8. `ProfitNotifier` doesn't validate new numerators

Severity: High
Type: Data Validation
Target: `ForceDAO/contracts/strategies/{ProfitNotifier, RewardTokenProfitNotifier}.sol`

Difficulty: High
Finding ID: TOB-ADV-008

Description
A configurable portion of the profits in a strategy is distributed as a reward to be shared by users. The percentage of the profits to be shared is initialized to 0 in the constructor, which validates the value to ensure that it is less than 100%.

```
  constructor(address _storage, address _rewardToken)
       public
       Controllable(_storage)
   {
       rewardToken = _rewardToken;
       profitSharingNumerator = 0;
       profitSharingDenominator = 100;
       require(
           profitSharingNumerator < profitSharingDenominator,
           "invalid profit share"
       );
   }
```

Figure 8.1: `ForceDAO/contracts/strategies/RewardTokenNotifier.sol#L17-L28`

After deployment, the governance system can update this percentage through calls to `setProfitSharingNumerator`. Because this function, unlike the constructor, does not validate the new value, the portion of rewards to be distributed could be set to more than 100%, which would drain the contract.

```
  function setProfitSharingNumerator(uint256 _profitSharingNumerator)
       external
       onlyGovernance
   {
       profitSharingNumerator = _profitSharingNumerator;
   }
```

Figure 8.2: `ForceDAO/contracts/strategies/RewardTokenNotifier.sol#L52-L57`

Exploit Scenario
After deployment, the ForceDAO team updates the profit-sharing percentage to 30% (in accordance with the code comments). Because of a typo made by the team member preparing the transaction, 300% of the profits are transferred as rewards on each call to `forceUnleashed`, draining user deposits from the system.

Recommendations
Short term, add the missing input validation to the `setProfitSharingNumerator` functions in the `ProfitNotifier` contracts.

Long term, always carry out input validation when passing variables to functions, especially to set system parameters. Additionally, using Echidna or Manticore, test system invariants to ensure that all properties hold.

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

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-ADV-010
Target: `ForceDAO/contracts/contracts/VaultProxy.sol`

Description
The `VaultProxy` contract uses the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or is self-destructed, the proxy may not detect failed executions.

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

```
function _delegate(address implementation) internal {
  assembly {
    // Copy msg.data. We take full control of memory in this inline assembly
    // block because it will not return to Solidity code. We overwrite the
    // Solidity scratch pad at memory position 0.
    calldatacopy(0, 0, calldatasize)

    // Call the implementation.
    // out and outsize are 0 because we don't know the size yet.
    let result := delegatecall(gas, implementation, 0, calldatasize, 0, 0)

    // Copy the returned data.
    returndatacopy(0, 0, returndatasize)

    switch result
    // delegatecall returns 0 on error.
    case 0 { revert(0, returndatasize) }
    default { return(0, returndatasize) }
  }
}

/**
 * @dev Function that is run as the first thing in the fallback function.
 * Can be redefined in derived contracts to add functionality.
 * Redefinitions must call super._willFallback().
```

```
 */
function _willFallback() internal {
}

/**
 * @dev fallback implementation.
 * Extracted to enable manual triggering.
 */
function _fallback() internal {
  _willFallback();
  _delegate(_implementation());
}
```

Figure 10.1: `Proxy.sol#L30-L66`

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 10.2: 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, each `delegatecall` returns success without changing the state or executing code. Eve uses this failing to scam users.

Recommendations
Short term, implement a contract existence check 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

27. Use of hard-coded addresses may cause errors

Severity: Informational
Type: Patching
Target: `ForceDAO/contracts/*`

Difficulty: Low
Finding ID: TOB-ADV-027

Description
Each contract needs contract addresses in order to be integrated into other protocols and systems. These addresses are currently hard-coded, which may cause errors and result in the codebase's deployment with an incorrect asset.

```
    address public constant usdc =
        address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    address public constant usdt =
        address(0xdAC17F958D2ee523a2206206994597C13D831ec7);
    address public constant dai =
        address(0x6B175474E89094C44Da98b954EedeAC495271d0F);

    address public constant wbtc =
        address(0x2260FAC5E5542a773Aa44fBCfeDf7C193bc2C599);
    address public constant renBTC =
        address(0xEB4C2781e4ebA804CE9a9803C67d0893436bB27D);
    address public constant sushi =
        address(0x6B3595068778DD592e39A122f4f5a5cF09C90fE2);
    address public constant dego =
        address(0x88EF27e69108B2633F8E1C184CC37940A075cC02);
    address public constant uni =
        address(0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984);
    address public constant comp =
        address(0xc00e94Cb662C3520282E6f5717214004A7f26888);
    address public constant crv =
        address(0xD533a949740bb3306d119CC777fa900bA034cd52);

    address public constant idx =
        address(0x0954906da0Bf32d5479e25f46056d22f08464cab);
    address public constant idle =
        address(0x875773784Af8135eA0ef43b5a374AaD105c5D39e);

    address public constant ycrv =
        address(0xdF5e0e81Dff6FAF3A7e52BA697820c5e32D806A8);

    address public constant weth =
        address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
```

Figure 27.1: `ForceDAO/contracts/contracts/FeeRewardForwarder.sol##L16-L47`

Hard-coded addresses are also used in the following contracts:

- `ForceDao/contracts/strategies/1inch/*`
- `ForceDAO/contracts/strategies/curve/*`
- `ForceDAO/contracts/strategies/idle/*`
- `ForceDAO/contracts/strategies/pancake/*`
- `ForceDAO/contracts/Vault.sol`

Using hard-coded values instead of deployer-provided values makes these contracts incredibly difficult to test.

Exploit Scenario
`FeeRewardForwarder` is deployed, but the address for `uni` is set incorrectly. As a result, the contract executes calls on the wrong address.

Recommendations
Short term, set addresses when contracts are created rather than using hard-coded values. This practice will facilitate testing.

Long term, to ensure that contracts can be tested and reused across networks, avoid using hard-coded parameters.

Quads Findings Summary

#	Title	Type	Severity
20	Use of non-compliant tokens with Balancer may cause integration failure	Undefined Behavior	High

20. Use of non-compliant tokens with Balancer may cause integration failure

Severity: High
Type: Undefined Behavior
Target: `Cyclical`

Difficulty: Medium
Finding ID: TOB-ADV-020

Description
The Quads codebase integrates into Balancer proxies that contain assumptions about how tokens work.

According to the following Balancer documentation, anyone can set up a proxy and choose to create a custom token. The Balancer codebase *assumes* that these tokens will support transfers of 0 and that each transfer will return a boolean.

```
IMPORTANT: make sure that the custom token you are adding complies with the ERC20 standard.
For example it has to allow 0 value transfers and the transfer function must return a
boolean. You can check if the token you are adding is on any of these two lists that gather
many tokens that are not ERC20-compliant:
```

Figure 20.1: The Balancer documentation on ERC20-compliant tokens

Interacting with a Balancer proxy that uses non-ERC20-compliant tokens may result in a loss of funds or unexpected behavior. The risks associated with these interactions should be analyzed and clearly detailed in user documentation.

Recommendations
Short term, document corner cases that could result from interacting with non-ERC20-compliant tokens used by Balancer proxies. That way, users will be aware of the risks.

Long term, analyze all third-party integrations and document the risks that arise when users interact with them.

Blacksmith Findings Summary

#	Title	Type	Severity
22	Borrow rate depends on approximation of blocks per year	Configuration	Informational
24	Malicious pairs can steal approved or deposited tokens	Access Controls	High
25	Flash loan rate lacks bounds and can be set arbitrarily	Data Validation	Low

22. Borrow rate depends on approximation of blocks per year

Severity: Informational | Difficulty: Medium
Type: Configuration | Finding ID: TOB-ADV-022
Target: `Blacksmith/contracts/compound/BaseJumpRateModelV2.sol`

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 base 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 blocks per year that is assumed by the interest
rate model
     */
    uint public constant blocksPerYear = 2102400;

    [...]
    /**
     * @notice Internal function to update the parameters of the interest rate model
     * @param baseRatePerYear The approximate target base APR, as a mantissa (scaled by
1e18)
     * @param multiplierPerYear The rate of increase in interest rate wrt utilization
(scaled by 1e18)
     * @param jumpMultiplierPerYear The multiplierPerBlock after hitting a specified
utilization point
     * @param kink_ The utilization point at which the jump multiplier is applied
     */
    function updateJumpRateModelInternal(uint baseRatePerYear, uint multiplierPerYear, uint
jumpMultiplierPerYear, uint kink_) internal {
        baseRatePerBlock = baseRatePerYear / blocksPerYear;
        multiplierPerBlock = (multiplierPerYear * 1e18) /  (blocksPerYear * kink_);
        jumpMultiplierPerBlock = jumpMultiplierPerYear / blocksPerYear;
        kink = kink_;

        emit NewInterestParams(baseRatePerBlock, multiplierPerBlock, jumpMultiplierPerBlock,
kink);
    }
```

Figure 22.1: `blacksmith/contracts/compound/BaseJumpRateModelV2.sol#L18-L21`

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

Recommendations
Short term, analyze the effects of a deviation from the actual number of blocks mined annually in borrow rate calculations and document the associated risks.

Long term, identify all variables that are affected by external factors, and document the risks associated with deviations from their true values.

References
- Ethereum Average Block Time Chart

24. Malicious pairs can steal approved or deposited tokens

Severity: High | Difficulty: Medium
Type: Access Controls | Finding ID: TOB-ADV-024
Target: `blacksmith/contracts/Vault.sol, blacksmith/contracts/LendingPair.sol`

Description
The `LendingPair` token flow may allow malicious pools to steal users' tokens when users are interacting with tokens deposited into the vault.

To add funds from the vault, a user must deposit funds and then approve the `LendingPair` contract with which the user wants to interact.

```
    modifier allowed(address _from) {
        require(msg.sender == _from || userApprovedContracts[_from][msg.sender] == true,
"ONLY_ALLOWED");
        _;
    }

    [...]

    /// @notice approve a contract to enable the contract to withdraw
    function approveContract(address _contract, bool _status) external  {
        require(_contract != address(0), "Vault: Cannot approve 0");
        userApprovedContracts[msg.sender][_contract] = _status;
        emit Approval(msg.sender, _contract, _status);
    }
```

Figure 24.1: `contracts/vault/PoolRegistry.sol#L32-L270`

This approval enables the `LendingPair` contract to deposit, withdraw, and transfer funds on behalf of the user. For instance, a token holder could choose to use his deposits in the vault to collateralize his position:

```
    /// @notice deposit allows a user to deposit underlying collateral from vault
    /// @param _vaultShareAmount is the amount of user vault shares being collateralized
    function depositCollateral(address _tokenReceipeint, uint256 _vaultShareAmount) external
override {
        vault.transfer(collateralAsset, msg.sender, address(this), _vaultShareAmount);
        // mint receipient vault share amount
```

```
        wrappedCollateralAsset.mint(_tokenReceipeint, _vaultShareAmount);
        emit Deposit(address(this), address(collateralAsset), _tokenReceipeint, msg.sender,
_vaultShareAmount);
    }
```

Figure 24.2: blacksmith/contracts/LendingPair.sol#L154-L161

This `transfer` function will update the internal `Vault` balances as long as the `LendingPool` contract has received user approval:

```
    /// @notice Transfer share of `token` to another account
    /// @param _token The ERC-20 token to transfer.
    /// @param _from which user to pull the tokens.
    /// @param _to which user to push the tokens.
    /// @param _shares of shares to transfer
    function transfer(
        IERC20 _token,
        address _from,
        address _to,
        uint256 _shares
    ) external override whenNotPaused allowed(_from) {
        _transfer(_token, _from, _to, _shares);
    }

    function _transfer(
        IERC20 _token,
        address _from,
        address _to,
        uint256 _shares
    ) internal {
        require(_to != address(0), "VAULT: INVALID_TO_ADDRESS");
        // Effects
        balanceOf[_token][_from] = balanceOf[_token][_from] - _shares;
        balanceOf[_token][_to] = balanceOf[_token][_to] + _shares;
        emit Transfer(_token, _from, _to, _shares);
    }
```

Figure 24.3: blacksmith/contracts/Vault.sol#L136-L161

However, this schema does not validate that the token transfer was initiated by the user. As a result, any `LendingPair` contract approved by a user could transfer and steal approved or deposited tokens. Appendix E provides guidance on precautions that users should take

to reduce the chance of a phishing attack when using `LendingPair` contracts.

Exploit Scenario
Alice creates a `LendingPair` contract with tokens A and B. Eve creates a malicious `LendingPair` pool with tokens C and D. Bob joins both lending pairs and allows his vault to transfer all of its tokens. Eve's pool steals all of Bob's A and B tokens.

Recommendations
Short term, warn users about the risks associated with malicious `LendingPair` contracts.

Long term, document the contracts' interactions and the ways in which assets are transferred between the components.

25. Flash loan rate lacks bounds and can be set arbitrarily

Severity: Low
Type: Data Validation
Target: `blacksmith/contracts/Vault.sol`

Difficulty: Low
Finding ID: TOB-ADV-025

Description
There are no lower or upper bounds on the flash loan rate implemented in the contract. The Blacksmith team could therefore set an arbitrarily high flash loan rate to secure higher fees.

The Blacksmith team sets the `_flashLoanRate` when the `Vault` is first initialized.

```
    function initialize(uint256 _flashLoanRate, address _blackSmithTeam)
        external
        initializer
    {
        require(_blackSmithTeam != address(0), "INVALID_TEAM");
        flashLoanRate = _flashLoanRate;
        blackSmithTeam = _blackSmithTeam;
    }
```

Figure 25.1: `blacksmith/contracts/Vault.sol#L41-L48`

The `blackSmithTeam` address can then update this value by calling `updateFlashloanRate`:

```
/// @dev Update the flashloan rate charged, only blacksmith team can call
/// @param _newRate The ERC-20 token.
function updateFlashloanRate(uint256 _newRate) external onlyBlacksmithTeam {
    flashLoanRate = _newRate;
    emit UpdateFlashLoanRate(_newRate);
}
```

Figure 25.2: `blacksmith/contracts/Vault.sol#L226-L231`

However, because there is no check on either setter function, the flash loan rate can be set arbitrarily. A very high rate could enable the Blacksmith team to steal vault deposits.

Exploit Scenario
Bob, a member of the development team, tries to set the fee to 1%. However, he converts the rate to an incorrect normalized number, setting the flash loan rate to 1,000%. As a result, a user is forced to pay a higher-than-expected flash loan fee.

Recommendations
Short term, introduce lower and upper bounds for all configurable parameters in the system to limit privileged users' abilities.

Long term, identify all incoming parameters in the system as well as the financial implications of large and small corner-case values. Additionally, use Echidna or Manticore to ensure that system invariants hold.

General Findings Summary

#	Title	Type	Severity
4	Logic duplicated across code	Patching	Informational
5	Insufficient testing	Testing	Informational
6	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
9	Initialization functions can be front-run	Configuration	High
11	Project dependencies contain vulnerabilities	Patching	Low
13	Lack of chainID validation allows reuse of signatures across forks	Authentication	High
14	Risks associated with EIP 2612	Configuration	Informational
16	Lack of contract documentation makes codebase difficult to understand	Documentation	Informational
17	Lack of zero check on functions	Data Validation	Informational
18	ABIEncoderV2 is not production-ready	Patching	Undetermined
19	Contract name duplication hinders third-party integrations	Undefined Behavior	Informational
21	Contracts used as dependencies do not track upstream changes	Patching	Low
23	Lack of two-step procedure for critical operations	Data Validation	High
26	No events for critical operations	Auditing and Logging	Informational

4. Logic duplicated across code

Severity: Informational | Difficulty: Low
Type: Patching | Finding ID: TOB-ADV-004
Target: `polkastrategies/contracts, EQLC/contracts, Cyclical/contracts`

Description
The logic in the repositories provided to Trail of Bits contains a significant amount of duplicated code. This development practice increases the risk that new bugs will be introduced into the system, as bug fixes must be copied and pasted into files across the system.

In the `polkastrategies` repository, the calculation and setter functions for contract variables contain duplicated logic. As such, logic changes or bug fixes implemented in one section must be manually copied and pasted into the corresponding code in all other files.

For example, the function that calculates the ratio used to determine pending rewards (shown in figure 4.1) must be copied and pasted into `HarvestDAI`, `ForceSLP`, and `SushiETHSLP`. Any changes to the logic used to calculate pending rewards would need to be manually implemented in the three other contracts.

```
    function _getRatio(
        uint256 numerator,
        uint256 denominator,
        uint256 precision
    ) private pure returns (uint256) {
        uint256 _numerator = numerator * 10**(precision + 1);
        uint256 _quotient = ((_numerator / denominator) + 5) / 10;
        return (_quotient);
    }
```

Figure 4.1: `polkastrategies/contracts/strategies/ForceSC.sol#L704-L712`

Similarly, the `DSSAddIlkSpell` and `DssChangeRatesSpell` contracts in `EQLC` use interfaces to execute contract calls, as shown below:

```
interface PauseLike {
    function delay() external returns (uint);
    function exec(address, bytes32, bytes calldata, uint256) external;
    function plot(address, bytes32, bytes calldata, uint256) external;
}

interface ConfigLike {
```

```
    function init(bytes32) external;
    function file(bytes32, bytes32, address) external;
    function file(bytes32, bytes32, uint) external;
    function rely(address) external;
}

interface JugLike {
    function drip(bytes32) external;
    function file(bytes32, bytes32, uint256) external;
}

interface PotLike {
    function drip() external;
    function file(bytes32, uint256) external;
}
```

Figure 4.2: EQLC/contracts/DssChangeRatesSpell.sol#L5-L30

These interfaces are copied and pasted into the two spells. Any changes to one of the interfaces would need to be made in all of the `Spell` contracts. A failure to do so would result in unexpected behavior.

Duplicated code is also used throughout the Quads contracts, including in fee calculation, modifier, and helper functions. These functions should be put into a helper contract so that their logic will be shared across contracts:

```
 function _defend() private view returns (bool) {
     require(
         approved[msg.sender] || msg.sender == tx.origin,
         "access_denied"
     );
 }

 function _existsInArray(address[] memory arr, address searchTerm)
     private
     pure
     returns (bool)
 {
     for (uint256 i = 0; i < arr.length; i++) {
         if (arr[i] == searchTerm) {
             return true;
         }
```

```
    }
    return false;
}

function _calculateFee(uint256 amount) private view returns (uint256) {
    return (amount.mul(fee)).div(feeFactor);
}
```

Figure 4.3: Cyclical/contracts/QuadsSingleLP.sol#L359-L383

When this development practice is used, it is incredibly easy to copy and paste code into the wrong file or function. Additionally, the lack of tests makes it much more difficult to detect bugs caused by copy-paste errors (TOB-ADV-005).

Exploit Scenario

Alice, a member of the development team, discovers a bug in the calculation of pending rewards. To address the error, she needs to copy and paste the fixes into the other strategy contracts. She accidentally copies the code into the wrong function, introducing an exploitable security issue.

Recommendations

Short term, use inheritance to allow code to be reused across contracts. Changes to one inherited contract will be applied to all files without requiring developers to copy and paste them.

Long term, minimize the amount of manual copying and pasting required to apply changes made to one file to other files.

5. Insufficient testing

Severity: Informational
Type: Testing
Target: `polkastrategies, EQLC, ForceDAO, Cyclical`

Difficulty: Low
Finding ID: TOB-ADV-005

Description
The repositories under review lack appropriate testing, which increases the likelihood of errors in the development process and makes the code more difficult to review.

The `polkastrategies` repository manages the bookkeeping of users' principals and the rewards they earn on deposits made into other applications. This portion of the code must be implemented correctly, as a bug would allow an attacker to steal funds from the contract by withdrawing excess funds or undeserved rewards. However, despite the importance of the withdrawal, deposit, and reward flows, the tests of those flows fail:

```
  0 passing (1s)
  9 failing
  [...]
  9) Contract: SushiSLP
       should test yUSD-WETH SLP
         should test positive scenario:
     Error: Invalid address passed to SushiETHSLP.at(): undefined
      at Function.at
(~/.nvm/versions/node/v12.18.3/lib/node_modules/truffle/build/webpack:/packages/contract/lib
/contract/constructorMethods.js:67:1)
      at execute (test/strategies/sushiSLP.ts:53:40)
      at Context.<anonymous> (test/strategies/sushiSLP.ts:128:19)
      at processImmediate (internal/timers.js:456:21)
```

Figure 5.1: The terminal output when `polkastrategies` tests are run

The `EQLC` codebase builds off of an MKR-like stablecoin, requiring users to lock in funds as collateral to mint EQLC. Running this command during the audit resulted in compilation errors.

The `ForceDAO` repository contains tests for the setter functions and time lock. However, the contracts lack tests for the core deposit-withdrawal flow.

The `Cyclical` repository contains some tests; however, the deployment scripts fail and throw errors.

Exploit Scenario

The development team modifies the implementation of one of the contracts, introducing a security issue that remains when the contracts are deployed.

Recommendations
Short term, ensure that the unit tests cover all public functions at least once, as well as all known corner cases.

Long term, integrate coverage analysis tools into the development process and regularly review the coverage.

6. Solidity compiler optimizations can be problematic

Severity: Informational | Difficulty: Low
Type: Undefined Behavior | Finding ID: TOB-ADV-006
Target: `polkastrategies/truffle-config.js, ForceDAO/contracts/truffle-config.js`

Description
`polkastrategies` has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the `polkastrategies` contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

9. Initialization functions can be front-run

Severity: High | Difficulty: High
Type: Configuration | Finding ID: TOB-ADV-009
Target: `ForceDAO/contracts/*, blacksmith/contracts/*`

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

Due to the use of the `delegatecall` proxy pattern, many of the contracts cannot be initialized with a constructor, and have initializer functions:

```
function initializeStrategy(
    address _storage,
    address _underlying,
    address _vault,
    address _rewardPool,
    address _rewardToken,
    uint256 _poolID,
    uint256 _profitSharingNumerator,
    uint256 _profitSharingDenominator
) public initializer {
    BaseUpgradeableStrategy.initialize(
        _storage,
        _underlying,
        _vault,
        _rewardPool,
        _rewardToken,
        _profitSharingNumerator,
        _profitSharingDenominator,
        true, // sell
        1e18, // sell floor
        12 hours // implementation change delay
    );
```

Figure 9.1: ForceDAO/contracts/strategies/sushiswap/SushiMasterChefLPStrategy.sol#L79-L100

An attacker could front-run these functions and initialize the contracts with malicious values.

This issue was found in several other functions in the ForceDAO codebase:

- `GovernableInit – initialize`
- `Vault – initializeVault`
- `VaultStorage – initialize`
- `BaseUpgradeableStrategy – initialize`
- `SushiMasterChefLPStrategy – initializeStrategy`
- `PancakeBaseUpgradeableStrategy – initialize`
- `PancakeMasterChefLPStrategy – initializeStrategy`

This issue is also present in the Blacksmith contracts:

- `ERC20 – initializeERC20`
- `ERC20Permit – initializeERC20Permit`
- `UUPSProxy – initializeProxy`
- `Owner – initializeOwner`
- `DebtToken – initialize`
- `LendingPair – initialize`
- `Vault – initialize`
- `WrapperToken – initialize`

Exploit Scenario

Bob deploys the `SushiMasterChefLPStrategy` contract. Eve front-runs the contract initialization and sets her own address for the `_rewardPool` address. As a result, she receives all of the funds meant to be directed to the reward pool.

Recommendations

Short term, either use a factory pattern that will prevent front-running of the initialization, or ensure that the deployment scripts have robust protections against front-running attacks.

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

11. Project dependencies contain vulnerabilities

Severity: Low | Difficulty: Low
Type: Patching | Finding ID: TOB-ADV-011
Target: `ForceDAO/*, polkastrategies/*, mstable12/*, blacksmith/*, Cyclical/*`

Description
Although dependency scans did not yield a direct threat to the projects under review, `yarn audit` 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 repositories under review. The below output details these issues:

NPM Advisory	Description	Dependency	Repository
1547	Signature Malleability	`elliptic`	`forceDAO, mstable12`
1594	Server-Side Request Forgery	`axios`	`forceDAO, mstable12, polkastrategies`
1631	Regular Expression Denial of Service	`diff`	`mstable12`
1654	Prototype Pollution	`y18n`	`blacksmith, Cyclical`

Table 11.1: NPM advisories affecting several projects' dependencies

Exploit Scenario
Alice installs the dependencies for one of the in-scope repositories 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.

Recommendations
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 codebase does not use and is not affected by the vulnerable functionality of the dependency.

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

Severity: High
Type: Authentication
Target: `EQLC, Blacksmith`

Difficulty: High
Finding ID: TOB-ADV-013

Description
The EQLC stablecoin token contracts and Blacksmith 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 schema. However, this chainID is fixed at the time of deployment.

```
   constructor(uint256 chainId_) public {
        wards[msg.sender] = 1;
        DOMAIN_SEPARATOR = keccak256(abi.encode(
            keccak256("EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)"),
            keccak256(bytes(name)),
            keccak256(bytes(version)),
            chainId_,
            address(this)
        ));
    }
```

Figure 13.1: `EQLC/contracts/dss/dai.sol#L59-L68`

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. If a change in the chainID is detected, the domain separator can be cached and regenerated. Alternatively, instead of regenerating the entire domain separator, the chainID can be included in the schema of the signature passed to the `permit` function.

Figure 13.2 shows the `permit` function using `ecrecover`:

```
   function permit(address holder, address spender, uint256 nonce, uint256 expiry,
                    bool allowed, uint8 v, bytes32 r, bytes32 s) external
    {
        bytes32 digest =
            keccak256(abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH,
```

```
                                holder,
                                spender,
                                nonce,
                                expiry,
                                allowed))
        ));

        require(holder != address(0), "Dai/invalid-address-0");
        require(holder == ecrecover(digest, v, r, s), "Dai/invalid-permit");
        require(expiry == 0 || now <= expiry, "Dai/permit-expired");
        require(nonce == nonces[holder]++, "Dai/invalid-nonce");
        uint wad = allowed ? uint(-1) : 0;
        allowance[holder][spender] = wad;
        emit Approval(holder, spender, wad);
    }
```

Figure 13.2: EQLC/contracts/dss/dai.sol#L120-142

The signature schema does not account for the contract's chain. 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 Ethereum. Bob has submitted a signature to permit Eve to spend those tokens on his behalf. When the London hard fork is executed, a subset of the community declines to implement the upgrade. As a result, there are two parallel chains with the same chainID, and Eve can use Bob's signature to transfer funds on both chains.

Recommendations
Short term, to prevent post-deployment forks from affecting calls to `permit`, detect chainID changes and regenerate the domain separator when necessary, or add the chainID opcode to the signature schema.

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

14. Risks associated with EIP 2612

Severity: Informational
Type: Configuration
Target: `EQLC, Blacksmith`

Difficulty: High
Finding ID: TOB-ADV-014

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

EIP 2612 uses signatures as an alternative to the traditional `approve` and `transferFrom` flow. 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. 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 have been transferred). 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 through phishing by asking for signatures in a context unrelated to the Advanced Blockchain contracts. The hash message may look benign and random to users.

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, develop user documentation on edge cases in which the signature-forwarding process can be front-run or an attacker can steal a user's tokens via phishing.

Long term, document best practices for EQLC and Blacksmith users. In addition to taking other precautions, users must do the following:

- Be extremely careful when signing a message
- Avoid signing messages from suspicious sources
- Always require hashing schemes to be public.

References
[EIP 2612 Security Considerations]

16. Lack of contract documentation makes codebase difficult to understand

Severity: Informational
Type: Documentation
Target: `All projects`

Difficulty: Low
Finding ID: TOB-ADV-016

Description
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 detail. In addition to general documentation, a list of specific points to clarify for each project is provided below.

- Polkastrategies
  - Formulaic derivations of reward calculations
  - The purpose of the blacklist and the distribution of reward tokens
- EQLC
  - The end-to-end execution of a deployment script
- Quads
  - Formulaic derivations of fee calculations and the reasons that certain constant factor values are chosen
  - The differences between the user flows in `Quads`, `QuadsLP`, and `QuadsSingleLP`
- Blacksmith
  - The user flow in interactions with `LendingPair` and `Vault`
  - The assumptions about `LendingPair` contracts

As a general recommendation, developing a list of logic changes implemented for each fork would facilitate change monitoring.

The documentation should include all expected properties and assumptions relevant to the abovementioned aspects of the codebase.

Recommendations
Short term, review and properly document the abovementioned aspects of the codebase.

Long term, consider writing a formal specification of the protocol.

17. Lack of zero check on functions

Severity: Informational
Type: Data Validation
Target: `EQLC/*, ForceDAO/*, Quads/*, blacksmith/*`

Difficulty: High
Finding ID: TOB-ADV-017

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

For example, the constructor in Blacksmith's `LendingPairFactory` contract sets the admin arguments, enabling the address to perform the privileged operations of updating implementations and tokens:

```
constructor(
    address _admin,
    address _pairLogic,
    address _collateralWrapperLogic,
    address _debtTokenLogic,
    address _borrowAssetWrapperLogic
) {
    admin = _admin;
    lendingPairImplementation = _pairLogic;
    collateralWrapperImplementation = _collateralWrapperLogic;
    debtTokenImplementation = _debtTokenLogic;
    borrowAssetWrapperImplementation = _borrowAssetWrapperLogic;
}
```

Figure 17.1: `blacksmith/contracts/LendingPairFactory.sol#L33-L46`

Once this address has been set to the zero address, the contract admin can no longer be changed, and functions to update the state can no longer be invoked. Changing the contract requires the deployment of a new contract.

This issue is also prevalent in the following contracts:

- `EQLC`
  - `UNIV2LPOracle.change() – _src`
  - `DssCHangeRatesSpell.constructor() – pause_`
  - `DssAddIlkSpell.constructor() – pause_`
  - `Jug.file – vow`
  - `Pot.file – vow`
- `ForceDAO`
  - `Timelock.constructor – admin_`

    - `Timelock.setPendingAdmin – pendingAdmin__`
    - `Timelock.executeTransaction – target`
    - `StaticsHelper.setPriceFeed – token`
    - `StaticsHelper.setLpSubToken – token`
    - `StaticsHelper.setRewardPool – vault, rewardPool`
    - `TokenPool.constructor – _token`
    - `TokenGeyser.constructor – _storage`
    - `BaseUpgradeabilityStrategyStorage.initialize – _storage, _underlying, _vault, _rewardPool, _rewardToken`
- `Quads`
    - `Quads.constructor – _balancerProxy, _badgerProxy, _deltaProxy, _uniswapProxy, _sushiswapProxy, _ffeeAddress, _wfeeAddress`
- `Blacksmith`
    - `BaseJumpRateModelV2.constructor – owner`
    - `PriceOracleAggregator.updateOracleForAsset – _asset`
    - `ChainlinkUSDAdapter.constructor – _asset, _aggregator`
    - `Ownable.initializeOwner – __owner`
    - `DebtToken.initialize – _underlying`
    - `LendingPairFactory.constructor – _admin, _pairLogic, _collateralWrapperLogic, _debtTokenLogic, _borrowAssetWrapperLogic`
    - `LendingPairFactory.updatePairImpl – _newLogicContract`
    - `LendingPairFactory.updateCollateralWrapperImpl – _newLogicContract`
    - `LendingPairFactory.updateDebtTokenImpl – _newLogicContract`
    - `LendingPairFactory.updateBorrowAssetWrapperImpl - _newLogicContract`
    - `LendingPairFactory.createIR – _team`
    - `LendingPairFactory.createLendingPairWithProxy – _team, _oracle, _vault, _collateralAsset`
    - `LendingPairFactory.initWrapperTokensWithProxy – implementation, pair, assetDetails`
    - `LendingPairFactory.initializeWrapperTokens – _pair, _wrapperToken, _assetDetails`
    - `WrapperToken.initialize – _underlying`

Exploit Scenario
Alice deploys a new version of the `LendingPairFactory` contract. When she deploys the contract, she accidentally provides `address(0)` as the admin. As a result, the contract implementation cannot be updated, and a new contract must be deployed.

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

Long term, use Slither, which will catch functions that do not have zero checks.

18. ABIEncoderV2 is not production-ready

Severity: Undetermined | Difficulty: Low
Type: Patching | Finding ID: TOB-ADV-018
Target: `polkastrategies/contracts/{IMasterChef.sol, SushiETHSLP.sol}`, `Cyclical/contracts/*, blacksmith/contracts/*`

Description
The contracts use the new Solidity ABI encoder, `ABIEncoderV2`. This experimental encoder is not ready for production.

More than 3% of all GitHub issues for the Solidity compiler are related to experimental features, primarily `ABIEncoderV2`. Several issues and bug reports are still open and unresolved. `ABIEncoderV2` has been associated with more than 20 high-severity bugs, some of which are so recent that they have not yet been included in a Solidity release.

For example, in March 2019 a severe bug introduced in Solidity 0.5.5 was found in the encoder.

Exploit Scenario
The Quads contracts are deployed. After the deployment, a bug is found in the encoder, which means that the contracts are broken and can all be exploited in the same way.

Recommendations
Short term, use neither `ABIEncoderV2` nor any other experimental Solidity feature. Refactor the code such that structs do not need to be passed to or returned from functions.

Long term, integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.

19. Contract name duplication hinders third-party integrations

Severity: Informational
Type: Undefined Behavior
Target: `ForceDAO, EQLC, Cyclical`

Difficulty: Low
Finding ID: TOB-ADV-019

Description
The codebase has two contracts that share a name. This causes the frameworks to generate incorrect `json` artifacts, preventing third parties from using their tools correctly.

`SafeMath` is defined in `RewardPool`.sol:

```
library SafeMath {
   [...]
}
```

Figure 19.1: `ForceDAO/contracts/RewardPool.sol#L94-L#246`

A contract with an identical name (specifically `"@openzeppelin/contracts/math/SafeMath.sol";`) is imported into the codebase.

These frameworks do not properly support codebases with duplicated contract names. The compilation overwrites the compilation artifacts and prevents the use of third-party tools such as Slither.

This issue is also present in the `EQLC` and `Cyclical` repositories.

Recommendations
Short term, prevent contract names from being reused or change the compilation framework.

Long term, use Slither to detect duplicate contract names.

References
- https://github.com/trufflesuite/truffle/issues/3124 (details a similar issue in Truffle)

21. Contracts used as dependencies do not track upstream changes

Severity: Low | Difficulty: Low
Type: Patching | Finding ID: TOB-ADV-021
Target: `polkastrategies/*, Cyclical/contracts/*, Blacksmith/contracts/*`

Description
Several third-party contracts have been copied and pasted into the repositories under review, including `BalancerConstants`, `BalancerSafeMath`, `ReentrancyGuard`, and `SafeApprove` (for Quads). Moreover, the code documentation does not specify the exact revision that was made or whether it was modified. As such, the contracts may not reliably reflect updates or security fixes implemented in their dependencies, as those changes must be manually integrated into the contracts.

This issue is also present in the following places:

- `polkastrategies/contracts/libraries/*`
- `Cyclical/contracts/libraries/*`
- `Cyclical/contracts/utils/*`
- `blacksmith/contracts/compound/*`
- `blacksmith/contracts/token/*`
- `blacksmith/contracts/upgradeability/*`
- `blacksmith/contracts/util/*`

Additionally, `Cyclical/contracts/libraries/` contains `SafeApprove`, which is imported into the `BalancerProxy` contract. However, the `SafeApprove` function is not used in the codebase.

Exploit Scenario
A third-party contract used in Quads receives an update with a critical fix for a vulnerability. An attacker detects the use of a vulnerable contract and exploits the vulnerability against one of the contracts.

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 to maintain internal path consistency and ensure that dependencies are updated periodically.

Long term, use an Ethereum development environment and NPM to manage packages in the project.

23. Lack of two-step process for critical operations

Severity: High | Difficulty: High
Type: Data Validation | Finding ID: TOB-ADV-023
Target: `blacksmith/contracts/*, polkastrategies/contracts/*`

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

For example, the `blackSmithTeam` variable is set on the deployment of the `Vault` contract. This address can perform the privileged operations of pausing the `Vault` and changing flash loan fees.

```
    function initialize(uint256 _flashLoanRate, address _blackSmithTeam)
        external
        initializer
    {
        require(_blackSmithTeam != address(0), "INVALID_TEAM");
        flashLoanRate = _flashLoanRate;
        blackSmithTeam = _blackSmithTeam;
    }
```

Figure 23.1: `blacksmith/contracts/Vault.sol#L41-L48`

To change the address of the `blackSmithTeam` contract owner, the owner must call `transferToNewTeam`:

```
    /// @notice Transfer control from current team address to another
    /// @param _newTeam The new team
    function transferToNewTeam(address _newTeam) external onlyBlacksmithTeam {
        require(_newTeam != address(0), "INVALID_NEW_TEAM");
        blackSmithTeam = _newTeam;
        emit TransferControl(_newTeam);
    }
```

Figure 23.2: `blacksmith/contracts/Vault.sol#L271-L277`

If the address is incorrect, the new address will immediately take on the functionality of the new role.

However, a two-step process would be similar to the `approve-transferFrom` functionality: the contract would approve a new address for a new role, and the new address would acquire the role by calling the contract.

The following functions would benefit from this two-step process:

- `blacksmith/contracts/util/Ownable.sol – transferOwnership`
- `polkastrategies/contracts/strategies/{ForceSC.sol, ForceSLP.sol, HarvestDAI.sol, SushiETHSLP.sol} – setTreasury`

Exploit Scenario

Alice deploys a new version of a multisig wallet for the Blacksmith team address. When she invokes the `transferToNewTeam` function to replace the address, she accidentally enters the wrong address. The new address is granted immediate access to the role, and it is too late to revert the action.

Recommendations

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 that can be taken by privileged accounts and their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

26. No events for critical operations

Severity: Informational
Type: Auditing and Logging
Target: `ForceDAO/contracts/*, Cyclical/contracts/*`

Difficulty: Low
Finding ID: TOB-ADV-026

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

For instance, in the `Quads` contract, the `RemovedLiquidity` event is emitted after funds have been removed from SushiSwap:

```
(amountA, amountB, pair) = sushiswapProxy.removeLiquidity(
    results.tokenA,
    results.tokenB,
    results.firstLpAmount,
    uint256(0),
    uint256(0),
    params.deadline
);

emit RemovedLiquidity(
    msg.sender,
    results.tokenA,
    results.tokenB,
    results.firstLpAmount
);

results.finalAmountA = results.finalAmountA.add(amountA);
results.finalAmountB = results.finalAmountB.add(amountB);
```

Figure 26.1: `Cyclical/contracts/Quads.sol#L1069-L1086`

However, this event is not emitted when the contract interacts with Uniswap:

```
(amountA, amountB, pair) = uniswapProxy.removeLiquidity(
    results.tokenA,
    results.tokenB,
    results.firstLpAmount,
    uint256(0),
    uint256(0),
    params.deadline
);
results.finalAmountA = results.finalAmountA.add(amountA);
```

```
results.finalAmountB = results.finalAmountB.add(amountB);
```

Figure 26.2: Cyclical/contracts/Quads.sol#L1007-L1016

The following critical operations would benefit from triggering events:

- `Cyclical/contracts/Quads.sol` – setter functions
- `ForceDAO/contracts/strategies/curve/CRVStrategyYCRV.sol` – `yCurveFromDai`

Without events, users and blockchain-monitoring systems cannot easily detect suspicious behavior.

Exploit Scenario

Eve compromises the `Quads` contract and sets her own fee address and fee amount. Because no events are emitted, Bob does not notice the compromise.

Recommendations

Short term, add events for all critical operations. Events aid in contract monitoring and the detection of suspicious behavior.

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.

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 exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation 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
Code Stability	Related to the recent frequency of code updates
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. Detecting Functions Missing `onlyOwner` Modifiers in polkastrategies

The `polkastrategies` codebase has many functions that are meant to be callable only by owners. As a result, most of the functions are protected by an `onlyOwner` modifier. We used Slither to identify and review functions that are not protected by this modifier.

The script (included below) follows an approach in which every reachable function must be either protected by an `onlyOwner` modifier or included on the `ACCEPTED_LIST`.

```
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
{modifiers_access_controls} modifier")
                no_bug_found = False
    if no_bug_found:
        print("\t- No bug found")

safe_functions = ["deposit","withdraw","receive"]

_check_access_controls(
    contracts.get_contract_from_name("ForceSC"),
    ["onlyOwner"],
    safe_functions
)
_check_access_controls(
    contracts.get_contract_from_name("ForceSLP"),
    ["onlyOwner"],
    safe_functions
)
_check_access_controls(
    contracts.get_contract_from_name("HarvestDAI"),
    ["onlyOwner"],
    safe_functions
```

```
)
_check_access_controls(
    contracts.get_contract_from_name("SushiETHSLP"),
    ["onlyOwner"],
    safe_functions
)
```

C.1: check-only-owner.py

D. 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.

EQLC

- **Ensure that the contract name matches the file name.** This will reduce the likelihood of mistakes and typos in the tests.
  - `EQLC/contracts/pips/ERC20Pip.sol#L90 – ERC20PIp`
- **Refactor variable names to ensure that they are sufficiently dissimilar.** This will help prevent errors caused by typos.
  - `pips/UNIV2LPOracle.sol#105 – normalizer0, normalizer1`
- **Split up the deployment scripts and contract address variables to enhance readability.**

ForceDAO

- **Fix typos in the codebase to ensure clarity.**
  - `ForceProfitSharing.sol#L58 – require(force.transfer(msg.sender, what), "Trabsfer failed");`
- **Remove unused variables to enhance readability.** These variables can be re-added in the future if necessary.
  - `contracts/StaticsHelper.sol – rewardPools`

Quads

- **Use self-explanatory variable names to make the purpose of each variable clear.**
  - `contracts/Quads.sol – ffeeAddress, wfeeAddress`
- **Remove `params.deadline` from `QuadsSingleLP` if the variable is not meant to be used.** This will increase the contract code's readability.
- **Clearly identify functions that use the Balancer version of SafeMath, as this library's behavior in `bmul` and `bdiv` is slightly different than the behavior of OpenZeppelin's SafeMath in equivalent functions.**

Blacksmith

- **Fix typos in the codebase to ensure clarity.**
  - `contracts/LendingPair.sol – wrappedCollateralAsset.mint(_tokenReceipeint, _vaultShareAmount);`

- **Ensure that the error messages for reverted transactions are meaningful so that users will know how to respond to a reversion.**
    - `contracts/LendingPair.sol – require(_tokenReceipeint != address(0), "IDB");`
- **Use a standardized term for the contracts' privileged users**. The contracts currently use `owner`, `admin`, and `blackSmithTeam` interchangeably to refer to privileged users.

General Recommendations

- **Refactor the code to use libraries or inherited contracts for shared code**. That way, every bug fix will be applied throughout the codebase.
- **Use either yarn or NPM to track dependencies in the ForceDAO and Cyclical repositories.** Tracking dependencies will ensure that the correct version is used.
- **Consider adopting a pricing scheme that doesn't require an owner to manually update the price in a contract for the UI.** That type of pricing scheme will enable the owner to reduce their gas costs and will enhance readability.

```
uint256 public ethPrice; //for UI; to be updated from a script

/**
 * @notice Set ETH price
 * @dev Can only be called by the owner
 * @param _price ETH price
 */
function setEthPrice(uint256 _price) external onlyOwner {
    require(_price >0,"PRICE_0");
    ethPrice = _price;
}
```

D.1: polkastrategies/contracts/ForceSC.sol#L300-308

```
function setPrice(address token, uint256 value) public onlyOwner {
    require(token != address(0), "0x0");
    require(value > 0, "0");
    prices[token] = value;
}
```

D.2: Cyclical/contracts/QuadsSingleLP.sol#L171-L175

E. Risks Associated with Arbitrary LendingPairs in Blacksmith

The Blacksmith codebase allows the deployment of arbitrary `LendingPair` contracts. As a result, we recommend that users carefully review the pool code to ensure that it behaves as expected. For instance, pools should meet the following criteria:

- **They should not be upgradeable**. Bad actors can stealthily introduce new functionalities into upgradeable pools to steal users' funds, even if the pools were deployed by their developers as expected, without malicious intent.
- **They should be incapable of self-destructing.** Destructible pools can be subject to malicious upgrades through the use of `CREATE2` and have other inherent risks.
- **Pools should have a clear fee schema and should not allow users to steal funds using high fees.** Users must be informed of the amount of fees that pools earn on and take from transactions. Users should also check that those fees are consistent with those set in the contract.
- **Pools should have documented parameters**. Each `LendingPair` should be documented and visible to users.
- **They should favor immutable parameters**. Immutable/constant parameters reduce the risks associated with privileged users.

Users should carefully check the vault calls and all arguments. Users should do their due diligence in ensuring that the relevant `LendingPair` is transferring and withdrawing funds correctly.

F. Token Integration Checklist

The following checklist provides recommendations for interactions 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 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 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` or `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.