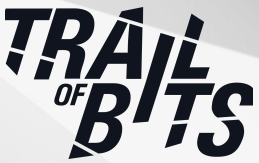

Arkis DeFi Prime Brokerage Protocol

Security Assessment

December 2, 2024

Prepared for:
Oleksandr Proskurin
Arkis

Prepared by: **Benjamin Samuels and Damilola Edwards**

About Trail of Bits

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

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

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

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

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

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

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Jessica Nelson, Project Manager
jessica.nelson@trailofbits.com@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Damilola Edwards, Consultant
damilola.edwards@trailofbits.com

Project Timeline

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

Date	Event
September 30, 2024	Pre-project kickoff call
October 11, 2024	Status update #1
October 18, 2024	Status update #2
October 25, 2024	Status update #3
November 2, 2024	Delivery of report draft
November 4, 2024	Report readout meeting
December 2, 2024	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Arkis engaged Trail of Bits to review the security of Arkis's DeFi Prime Brokerage smart contracts. These contracts implement a leveraged trading system that allows users to interact with supported DeFi protocols using leveraged funds borrowed from a lender. These funds are undercollateralized and liquidated if the value of the user's account plus its collateral drops below a specific safety threshold.

A team of two consultants conducted the review from October 7 to November 1, 2024, for a total of five engineer-weeks of effort. Our testing efforts focused on the methods a malicious user may employ to exfiltrate borrowed funds and ways the system as a whole may be misconfigured. With full access to source code and documentation, we performed static and dynamic testing of the Arkis smart contracts, using automated and manual processes.

Observations and Impact

Over the course of the review, we observed that Arkis contains several complexity management issues that reduce the effectiveness of both automated and manual reviews. Responsibilities between on-chain and off-chain code are not clear, as indicated by issues like TOB-ARK-7. The codebase's complexity has also created compliance logic issues such as TOB-ARK-10 and TOB-ARK-11.

A large proportion of the system's overall security depends on the capabilities of its off-chain components to liquidate positions in a timely manner. We highly recommend obtaining a separate review for these components.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Arkis take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Reduce the complexity of the system by applying the following mitigations:**
    - Convert the system from the diamond proxy standard to a traditional universal upgradeable proxy. The compliance diamond's various validation facets may be called using a switch/case statement for each specific protocol/function selector to upgradeable validator contracts.
    - Avoid storing any kind of state in libraries. Instead, store object-centric state

in higher-order contracts whose behavior can be inherited. Making this change would also allow the Arkis team to cleanly encapsulate complex libraries like `ProtocolsEvaluatorsRepository`, `Config`, `ProtocolsRepository`, and `TokensRepository` in stand-alone or higher-order inherited contracts.

- For each function involved in the setup/closing of accounts, document what values are trusted/untrusted and where that function is called during the setup/closing flow. For unauthenticated functions, consider situations in which functions are called in the wrong order. There may be issues similar to TOB-ARK-7 that can be identified in this way.
- Simplify the `Account` state machine. The same requirements can be fulfilled without using function pointers by using a switch/case statement, allowing static analysis tooling to follow the flow of code. In this construction, changes to the state machine would require a contract upgrade.
- Consolidate similar flows of logic. For example, in the current system, there are two completely separate routes through which a protocol interaction can be validated: the liquidation route and the user command-submission route. Consolidating these two flows into a single validation flow would remove nearly 25% of the system's on-chain code, drastically reducing the public-facing attack surface.
  - A second example of similar logic flows is the `TokensRepository` libraries. There are two libraries called `TokensRepository`—one for the compliance validation flow and one for the liquidation validation flow. One library uses the `WhitelistingController` contract for enforcement, and the other library stores it as library-private state. These two libraries would be better off being consolidated to reduce the amount of duplicated functionality across the project.

- **Improve the system's test suite by applying the following mitigations:**
  - Ensure the entire deployed codebase is thoroughly tested to the same standard as the already tested code. If some code is reachable only from an end-to-end test, then its other components must be mocked for a unit test.
  - Run mutation testing to identify weaknesses in the unit testing suite. Coverage is not a great indicator of test suite quality at this complexity level, and mutation fuzzing can help fill that gap.
  - Add Foundry-based integration tests to ensure each diamond proxy can be composed correctly using the same `Diamond.sol` file that Hardhat will use to deploy and in the end-to-end tests.

- Consider adding a stateful fuzzing test suite. Stateful fuzzing test suites are ideal for testing heavily stateful code such as Arkis's account creation and management flows.

- **Obtain a threat model and code review for off-chain components.** The security of Arkis's smart contracts is critically dependent on the secure operation of its off-chain components. These off-chain components should receive a dedicated threat model and code review.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	3
Low	1
Informational	6
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Code Quality	3
Data Validation	6
Undefined Behavior	5

Project Targets

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

Arkis DeFi Prime Brokerage Smart Contracts

Repository	gitlab.com/primebridge/smart-contracts
Commit Hash	`8a3cf1e2b838e743569c16913b577058a3de0c94`
Type	Solidity
Platform	EVM

Project Coverage

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

- **Compliance contracts:** These contracts are used to apply compliance policies against user transactions. They also control a whitelist of allowed tokens, protocols, and pools that users are allowed to interact with. The Pendle compliance contracts were reviewed to ensure there is no way for a user to submit a request that causes funds to leave the account, and to ensure only whitelisted tokens are interacted with.

- **Agreement contracts:** These contracts are used to specify the nature of a lender/borrower relationship. They specify the collateral token, the lending token, allowed borrowers, and allowed lenders, and they facilitate funding and repayment of funds. The factory portion of the code was reviewed to ensure it is properly initialized, and the agreement contract itself was reviewed to ensure it has adequate access controls, it is properly initialized, and state is tracked correctly.

- **Dispatcher contracts:** These contracts are used to facilitate parts of the account life cycle and provide an entrypoint for the system admin to perform liquidations. The access control for each contract was reviewed, along with any constraints placed on inputs passed to the registration flow. Several libraries that facilitate liquidations are present in this component; these libraries were manually reviewed for logic and consistency issues.

- **Compiler contracts:** These contracts implement a just-in-time (JIT) compiled instruction set that is executed by liquidation programs. Each supported protocol provides its own implementation for each instruction. The compiler itself was manually reviewed for validation errors and issues related to verification of supported tokens and protocols. The Pendle-specific implementation for each instruction was reviewed to ensure each implementation matches the intention behind the instruction.

- **Account contracts:** These contracts are the main interaction point for borrowers during trading operations. They implement a state machine for the account life cycle and handle the recycling of accounts when they are closed. This component also implements the `CommandSafeExecutor` library, which executes user transactions. The account contract's state machine was reviewed, along with its involvement in the account registration, opening, and closing life cycle. The `CommandSafeExecutor` contract was reviewed to identify potential issues allowing the command validation to be bypassed.

- **Library and base contracts:** These contracts implement various utilities and higher-order classes to facilitate access control and the diamond proxy pattern. These contracts were manually reviewed to ensure they are upgrade-safe, do not contain storage slot collisions, correctly use primitive instructions like `CALL` and `DELEGATECALL`, and are correctly initialized; they were reviewed for other miscellaneous logic issues as well.

Coverage Limitations

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

- The system's smart contracts extensively use code patterns that drastically reduce the effectiveness of static analysis and manual review by obscuring component relationships. The use of the diamond proxy standard, function pointers, and tightly coupled internal libraries are the largest contributors. These constructs have materially impacted the review's coverage: it may not be as thorough as it would have been without these constructs.
- At the time of the engagement, validation of liquidation plans was not yet fully implemented on the smart contract side, so this feature's coverage is limited.
- The system's off-chain code for authorizing users, opening accounts, triggering liquidations, and so on is out of the scope of the review.
- Support for protocols other than Pendle is considered out of the scope of the review.
- The Pendle protocol itself is out of the scope of the review.
- The system's staking contract and logic dependent on the staking contract are out of the scope of the review.
- Potential collisions between function selectors were not explored due to time constraints.

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	The in-scope components of Arkis use simple arithmetic for calculating leverage and collateral that is well tested given its complexity.	**Strong**
Auditing	All meaningful state changes are accompanied by event emissions. However, the security of the system is dependent on the integrity and performance characteristics of the off-chain components that trigger liquidations. These components are not in scope, so the mechanism they use to introspect the state of the on-chain contracts could not be reviewed.	**Further Investigation Required**
Authentication / Access Controls	While the contracts enforce role restrictions well, there is no documentation about the intended access level of each role or its permissions. Adding documentation about each role in the system and its potential actions, responsibilities, and trust level would drastically improve this evaluation. In addition, Arkis would benefit from building a threat model of its on-chain contracts. There is a mix of immutable and upgradeable contracts in the project (e.g., `ImmutableBeaconProxy` and `BeaconUpgradeable`), and it is not clear if there should be a trust boundary separating these components.	**Moderate**
Complexity Management	At the time of review, the codebase was in a transitional state, and not all features were implemented; therefore, some abstractions that currently appear to be overengineered may be adequate for the final design specification. Given this uncertainty, we recommend further evaluating the system complexity once the codebase is	**Further Investigation Required**

	feature-complete. The following feedback is for the codebase as it was at the time of the review and may be subject to change as features are completed. Arkis's code is well thought out but contains complexity issues that reduce the effectiveness of manual and automated code review. The code uses several abstractions that prevent static analysis from being used effectively—specifically, the diamond proxy standard and function pointers. Both abstractions prevent static analysis tools from understanding how data flows throughout the system, reducing their effectiveness and forcing more of the system's analysis to be performed using error-prone manual analysis. There are several instances of library usage that effectively "hide" where certain parts of the state are stored, and libraries are tightly coupled to each other in some locations. For example, the `ProtocolsEvaluatorRepository` library is tightly coupled with the `Config` library, as the system assumes contracts using the `ProtocolsEvaluatorRepository` library have configured the `Config` library compliance address. This defeats the purpose of the abstraction—if there are assumptions made about contracts using `ProtocolsEvaluatorsRepository`, then those assumptions need to be encapsulated into a higher-order class. There are also issues around figuring out what inputs are and are not trusted. A large fraction of Arkis's logic is run off-chain, and without knowing how those components work, some parameters must be considered untrusted unless we explicitly trust the off-chain components to behave fairly. This is demonstrated in TOB-ARK-7, where an untrusted agreement address may cause funds to be drained depending on how the off-chain components work. Concrete recommendations for how to address Arkis's complexity issues can be found in the long-term guidance in the Executive Summary.	
Cryptography and Key	There are no notable uses of cryptography or keys.	**Not Applicable**

Management		
Documentation	Arkis has relatively thorough documentation for its interfaces; however, it is lacking in some specific locations (TOB-ARK-14). Given the number of upgradeable components in the system, Arkis should develop internal documentation for how to safely upgrade its different components in a runbook-like format. An upgradeability runbook should contain the following: • Scripts for upgrading each component • A set of validation criteria/scripts to verify that an upgrade was successful ○ One common validation method is to create a shadow fork after the code has been deployed and to run a series of transactions against the fork to verify that the upgrade was successful. • A set of steps/scripts to follow in case an upgrade fails	**Moderate**
Low-Level Manipulation	The Arkis smart contracts do not contain any notable instances of low-level manipulation.	**Strong**
Testing and Verification	Arkis's unit test coverage was measured at 88%; however, multiple critical libraries, including `LiquidityPoolsRepository` and `JitCompiler`, are untested. Well-tested parts of the code undergo excellent happy and unhappy path testing, which is commendable; however, at Arkis's complexity level, that level of testing needs to be applied throughout the project and complemented with test verification techniques like mutation testing along with more thorough state machine testing using stateful fuzzing. Multiple defects were identified during the engagement's mutation testing campaign, which are documented in appendix C. Concrete recommendations for how to improve Arkis's testing maturity can be found in the long-term guidance in	**Weak**

	the Executive Summary.	
Transaction Ordering	The Arkis protocol may be vulnerable to sandwiching attacks, as indicated by TOB-ARK-9. However, there is not enough information available at this time to determine whether this is a systemic issue or one that can be easily addressed while respecting the system's requirements.	**Further Investigation Required**

Summary of Findings

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

ID	Title	Type	Severity
1	AccessControlDS uses AccessControl, which has storage collision risks	Undefined Behavior	Medium
2	EnumerableSet.add/remove return value not checked	Data Validation	Low
3	Proxy storage slots may collide due to lack of a namespacing scheme	Undefined Behavior	Informational
4	Margin accounts can be registered with suspended agreements	Data Validation	Medium
5	BeaconDS can create broken margin accounts via an undeployed Account implementation	Undefined Behavior	Medium
6	ThresholdsVerifier is initialized using the compiler contract instead of the compliance contract	Undefined Behavior	Informational
7	Risk of leveraged fund drain via hostile agreement contract	Data Validation	Undetermined
8	redeemDueInterestAndRewards does not validate output tokens	Data Validation	Informational
9	Malicious users can execute swaps without sandwiching protection to exfiltrate leveraged funds	Data Validation	High
10	Native tokens may be drained from margin accounts via lack of value validation	Data Validation	High
11	Native tokens may be drained by invoking unexpected compliance functions	Undefined Behavior	Undetermined

12	Lack of zero-value checks in constructor function	Code Quality	**Informational**
13	Unused code/features	Code Quality	**Informational**
14	Missing/inadequate function-level and library documentation	Code Quality	**Informational**

Detailed Findings

1. AccessControlDS uses AccessControl, which has storage collision risks	
Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ARK-1
Target: `contracts/base/auth/AccessControlDS.sol`	

Description

The `AccessControlDS` contract inherits from OpenZeppelin's `AccessControl` contract, creating a risk of storage collisions due to `AccessControlDS`'s use in upgradeable contracts. The definition of `AccessControlDS` and the variable creating the collision risk are shown in figures 1.1 and 1.2.

```
abstract contract AccessControlDS is AccessControl, OwnableReadonlyDS {
    function hasRole(bytes32 _role, address _account) public view virtual override
returns (bool) {
        return (isOwnerRole(_role) && _owner() == _account) || super.hasRole(_role,
_account);
    }
```

Figure 1.1: The facet contract `AccessControlDS` inherits from OpenZeppelin's `AccessControl` contract. (`contracts/base/auth/AccessControlDS.sol#7–10`)

```
abstract contract AccessControl is Context, IAccessControl, ERC165 {
    struct RoleData {
        mapping(address account => bool) hasRole;
        bytes32 adminRole;
    }

    mapping(bytes32 role => RoleData) private _roles;
```

Figure 1.2: The OpenZeppelin `AccessControl` contract is not upgrade-aware and stores role information in the first slot. (`openzeppelin-contracts/contracts/access/AccessControl.sol#49–55`)

Exploit Scenario

A new facet is added to a diamond proxy that already implements a facet using `AccessControlDS`. This new facet stores a variable in slot 0 (or inherits from a contract that stores a variable in slot 0), so these two storage variables collide, causing undefined behavior.

Recommendations

Short term, replace `AccessControl` with `AccessControlUpgradeable`. This contract is upgrade-aware and uses EIP-7201 to store role information at non-colliding storage slots.

Long term, use Slither to detect storage collision risks in upgradeable contracts. The `slither --print variable-order` command can be used manually or in a CI pipeline to detect when upgradeable contracts store variables in contentious slots, as shown in figure 1.3.

```
(slither) → audit-arkis git:(dev) ✗ slither --print variable-order ./contracts/base/auth/AccessControlDS.sol
'forge config --json' running
'/Users/bsamuels/.pyenv/versions/3.8.13/envs/slither/.solc-select/artifacts/solc-0.8.22/solc-0.8.22 --version'
'/Users/bsamuels/.pyenv/versions/3.8.13/envs/slither/.solc-select/artifacts/solc-0.8.22/solc-0.8.22 forge-std/=
_modules/hardhat/ solady/=node_modules/solady/ ./contracts/base/auth/AccessControlDS.sol --combined-json abi,as
ontracts/base/auth' running
INFO:Printers:
AccessControlDS:
+----------------------+-----------------------------------------------+------+--------+
| Name                 |                                          Type | Slot | Offset |
+----------------------+-----------------------------------------------+------+--------+
| AccessControl._roles | mapping(bytes32 => AccessControl.RoleData) |    0 |      0 |
+----------------------+-----------------------------------------------+------+--------+

Address:
+------+------+------+--------+
| Name | Type | Slot | Offset |
+------+------+------+--------+
+------+------+------+--------+

DiamondLibrary:
+------+------+------+--------+
| Name | Type | Slot | Offset |
+------+------+------+--------+
+------+------+------+--------+

INFO:Slither:./contracts/base/auth/AccessControlDS.sol analyzed (10 contracts)
```

Figure 1.3: Slither can be used to quickly identify problematic storage slot usage in upgradeable or diamond facet contracts.

2. EnumerableSet.add/remove return value not checked

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARK-2
Target: `contracts/agreement/Agreement.sol`	

Description

The return values of the `EnumerableSet.remove` and `EnumerableSet.add` functions are not validated in the `Agreement` contract, as shown in figures 2.1 and 2.2.

This means that adding a duplicate collateral token address or removing a collateral token address that does not exist will still emit a `CollateralAdded` or `CollateralRemoved` event, respectively.

```
function removeCollaterals(address[] calldata tokens) external override onlyFactory
{
    Storage storage $ = _storage();
    for (uint256 i = 0; i < tokens.length; i++) {
        $.collaterals.remove(tokens[i]);
        emit CollateralRemoved(tokens[i]);
    }
}
```

Figure 2.1: The return value of `EnumerableSet.remove` is ignored, leading to an erroneous event emission. (`contracts/agreement/Agreement.sol#158–164`)

```
function addCollaterals(address[] calldata tokens) public override onlyFactory {
    Storage storage $ = _storage();
    for (uint256 i = 0; i < tokens.length; i++) {
        $.collaterals.add(tokens[i]);
        emit CollateralAdded(tokens[i]);
    }
}
```

Figure 2.2: The return value of `EnumerableSet.add` is ignored, leading to an erroneous event emission. (`contracts/agreement/Agreement.sol#135–141`)

Exploit Scenario

The Arkis admin intends to remove USDC as a valid collateral token for agreement A. Due to a typo, the transaction sent tries to remove USDC as a valid collateral for agreement B, which already does not have USDC as a valid collateral.

When the transaction is executed, the `Agreement` still emits the `CollateralRemoved` event even though no action took place, reducing the probability that the error will be caught immediately.

Recommendations

Short term, add code to handle the Boolean returned by `add` and `remove`, and consider having the functions revert if a collateral token is added twice or there is an attempt to remove a collateral token that never existed.

Long term, run Slither as a regular part of the development process. This finding was detected using the `unused-return` Slither detector, and regular use of Slither in the development process may have prevented the issue from being introduced.

3. Proxy storage slots may collide due to lack of a namespacing scheme

Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-ARK-3
Target: Various files	

Description

The Arkis contracts use pointers to unstructured storage to avoid storage slot collision; however, the pointer locations are not determined with a sufficient namespacing scheme that avoids collisions. One example of a storage slot determined without sufficient namespacing is shown in figure 3.1.

```
// keccak256("Margin Account Factory State V1")
bytes32 private constant STORAGE_SLOT =
    0x543a5afad456720a20bb954f3229d0931809b27405d3298f8b5e666962637d83;
```

Figure 3.1: The storage slot location name does not have sufficient namespacing to prevent collisions.
(contracts/marginAccount/libraries/AccountFactoryRepository.sol#10–12)

Recommendations

Short term, modify all storage slot strings in the project to use EIP-7201 to implement adequate namespacing and versioning to avoid collisions. Using this EIP will not only avoid collisions, but it will also make it easier for static analysis tools to analyze the Arkis contracts and will provide additional gas optimization when Ethereum implements Verkle tries.

4. Margin accounts can be registered with suspended agreements

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARK-4
Target: `contracts/dispatcher/Dispatcher.sol`	

Description

The registration logic for margin accounts lacks a check to ensure the assigned agreement contract is not suspended.

The agreement contract inherits OpenZeppelin's `PausableUpgradeable` library, which allows the contract to be paused and unpaused. However, only the pause logic is exposed via the `suspend` function, so paused agreement contracts cannot be unpaused.

```
184    function suspend() external override onlyFactory {
185        _pause();
186    }
```

Figure 4.1: contracts/agreement/Agreement.sol#L184–L186

During a margin account registration, an agreement contract address is passed as a parameter to the `registerMarginAccount` function in the `Dispatcher` contract. The `verifyThresholds` function in the `ThresholdVerifier` contract validates the input before proceeding with registration.

```
39    function registerMarginAccount(
40        address agreement,
41        Asset[] calldata _collateral,
42        Asset calldata _leverage
43    ) external override returns (address account) {
44        verifyThresholds(agreement, _collateral, _leverage);
45        IAgreement(agreement).enforceCanBorrow(msg.sender);
46
47        account = factory.borrowAccount();
48
49        for (uint256 i; i < _collateral.length; i++) {
50            _collateral[i].forward(account);
      ...
54    }
```

Figure 4.2: contracts/dispatcher/Dispatcher.sol#L39–L54

However, the current logic does not check whether the agreement contract is suspended. This oversight could result in critical operations on newly registered accounts being blocked if the assigned agreement contract is paused.

```
21    function verifyThresholds(
22        address agreement,
23        Asset[] calldata _collaterals,
24        Asset calldata _leverage
25    ) internal view {
26        if (_leverage.amount == 0) revert AmountMustNotBeZero(_leverage.token);
27
28        if (_collaterals.length == 0) revert ArrayMustNotBeEmpty();
29
30        (IAgreement.Metadata memory metadata, ) = IAgreement(agreement).info();
31        if (metadata.leverage != _leverage.token) revert WrongLeverage(agreement,
_leverage.token);
32
33        for (uint256 x; x < _collaterals.length; x++) {
34            Asset calldata c = _collaterals[x];
35            bool isValidCollateral = false;
36            for (uint y = 0; y < metadata.collaterals.length; y++) {
37                if (c.token == metadata.collaterals[y]) isValidCollateral = true;
38            }
39            if (!isValidCollateral) revert WrongCollateral(agreement, c.token);
40            if (c.amount == 0) revert AmountMustNotBeZero(c.token);
41        }
42    }
```

Figure 4.3: contracts/dispatcher/base/ThresholdsVerifier.sol#L21–L42

Exploit Scenario

Alice registers a new margin account, unaware that the agreement contract used had been suspended. The account is created successfully, but critical operations like trading are blocked due to the contract's paused state.

Recommendations

Short term, include an agreement status check within the validation logic for margin account registration to ensure that the assigned agreement contract is not suspended before proceeding with the registration.

Long term, expand the unit test suite to explore a wide variety of edge case scenarios, validating both happy and unhappy code execution paths within the system.

5. BeaconDS can create broken margin accounts via an undeployed Account implementation

Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ARK-5
Target: `contracts/base/proxy/BeaconDS.sol`	

Description

The `AccountFactory` contract uses the `BeaconDS` contract for setting/retrieving the `Account` implementation; however, `BeaconDS` does not perform contract existence checks when the `Account` implementation is set, as shown in figure 5.1.

```
function setImplementation(address newImplementation) external override onlyOwner {
    if (newImplementation == address(0)) revert ImplementationAddressIsZero();
    if (_implementation() == newImplementation) revert AlreadyUpToDate();

    assembly {
        sstore(IMPLEMENTATION_SLOT, newImplementation)
    }
}
```

Figure 5.1: The `setImplementation` function does not check whether a contract is deployed to `newImplementation`. (`contracts/base/proxy/BeaconDS.sol#13–20`)

Exploit Scenario

There are plans to update the `Account` contract to a `CREATE2`-based implementation whose address can be calculated ahead of time. The transaction to deploy the `Account` contract fails, and the failure is not noticed. When `AccountFactory.setImplementation` is called, the contract implementation is set to an address with no contract deployed to it.

At this point, newly created accounts will be represented by an `ImmutableBeaconProxy` contract that `delegatecall`s into an address with no contract deployed to it. All calls to an address with no contract deployed will succeed and return `true`.

Recommendations

Short term, update `BeaconDS.setImplementation` to check for contract existence using `extcodesize` before accepting a new implementation.

Long term, add a code review checklist item to ensure all proxy contracts check for the presence of the target contract before accepting a new implementation.

6. ThresholdsVerifier is initialized using the compiler contract instead of the compliance contract

Severity: **Informational**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-ARK-6
Target: `contracts/dispatcher/Dispatcher.sol`	

Description

The `ThresholdsVerifier` contract constructor accepts an address for the compliance contract, as shown in figure 6.1. However, the `Dispatcher` contract constructs `ThresholdsVerifier` using the compiler contract address, as shown in figure 6.2.

```
abstract contract ThresholdsVerifier {
    IWhitelistingController internal immutable compliance;
    [...]
    constructor(address _compliance) {
        compliance = IWhitelistingController(_compliance);
    }
```

Figure 6.1: The `ThresholdsVerifier` contract accepts the `WhitelistingController` contract in its constructor.
(`contracts/dispatcher/base/ThresholdsVerifier.sol#10–19`)

```
constructor(
    address _compiler,
    address _factory,
    string memory _chainName
) ThresholdsVerifier(_compiler) {
    compiler = _compiler;
    chainNameHash = keccak256(bytes(_chainName));
    factory = IAccountFactory(_factory);
}
```

Figure 6.2: `Dispatcher` constructs the `ThresholdsVerifier` using the `JitCompiler` contract, which does not implement `IWhitelistingController`.
(`contracts/dispatcher/Dispatcher.sol#29–37`)

Recommendations

Short term, modify the constructor for `Dispatcher` to accept a contract address that implements `IWhitelistingController`. Alternatively, since the `IWhitelistingController` contract is not used in `ThresholdsVerifier`, simply remove the parameter and state variable from `ThresholdsVerifier`.

Long term, ensure that refactored code is properly cleaned up after the fact. Running Slither can help triage code that is no longer used.

7. Risk of leveraged fund drain via hostile agreement contract

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARK-7
Target: `contracts/dispatcher/Dispatcher.sol`	

Description

The `registerMarginAccount` function, intended to be callable by any user on the network, accepts an arbitrary address for the agreement contract, as shown in figure 7.1. This address simply needs to implement the `IAgreement` interface, and no restrictions are in place to prevent the user from supplying the address to an agreement contract whose functionality they directly control.

```
function registerMarginAccount(
    address agreement,
    Asset[] calldata _collateral,
    Asset calldata _leverage
) external override returns (address account) {
    verifyThresholds(agreement, _collateral, _leverage);
    IAgreement(agreement).enforceCanBorrow(msg.sender);

    account = factory.borrowAccount();

    for (uint256 i; i < _collateral.length; i++) {
        _collateral[i].forward(account);
    }

    IAccount(payable(account)).register(msg.sender, agreement, _collateral,
_leverage);
```

Figure 7.1: The `registerMarginAccount` function can be called using an agreement contract that was not created by the `AgreementFactory` contract. (`contracts/dispatcher/Dispatcher.sol#39–53`)

A malicious user may register a margin account using an agreement whose parameters they dynamically control. This would allow the malicious user to modify critical functionality, such as `withdraw`, `claim`, and `repay` behavior.

This finding's severity is set to "undetermined" because it heavily relies on the out-of-scope off-chain components behaving in a specific way. If the components behave in the way specified in the exploit scenario, then the severity of this finding is "high."

Exploit Scenario

A malicious user registers a margin account using an agreement contract they have direct control over. The off-chain code sees the registration event and opens and funds the margin account for the malicious user. Instead of funding the margin account when `Agreement.deposit` is called, the agreement contract transfers the leveraged funds directly to the malicious user.

Recommendations

Short term, add an `Agreement` registry to `AgreementFactory` to allow contracts to determine whether an agreement was deployed by the factory. Then modify `registerMarginAccount` to query the registry and ensure the agreement contract was deployed by the factory.

Long term, consider using a singleton design to replace the `Agreement` and `AgreementFactory` contracts. In a singleton contract, each distinct agreement would be stored as a struct within a map of an `AgreementRegistry` contract. Each agreement would be identified using a UUID or similar identifier. This would likely save gas on contract deployment and simplify the architecture by reducing the number of contracts in the system and the number of proxies that need to be managed.

8. redeemDueInterestAndRewards does not validate output tokens

Severity: **Informational**	Difficulty: **N/A**
Type: Data Validation	Finding ID: TOB-ARK-8
Target: `contracts/compliance/pendle/PendleValidatorMisc.sol`	

Description

The Pendle compliance validator exposes the `redeemDueInterestAndRewards` function, which harvests accrued interest and rewards from the user's liquidity provider (LP) position and yield token (YT) position. As shown in figure 8.1, the compliance function does not validate the tokens that will be received when the function is called.

```
function redeemDueInterestAndRewards(
    address user,
    address[] calldata sys,
    address[] calldata yts,
    address[] calldata markets
) external view returns (Command[] memory cmds_) {
    cmds_ = Command({
        target: getOperator(Support.NotRequired),
        value: 0,
        payload: user == msg.sender
            ? msg.data
            : abi.encodeWithSelector(msg.sig, msg.sender, sys, yts, markets)
    }).asArray();
}
```

Figure 8.1: The `redeemDueInterestAndRewards` validator function does not verify that the tokens redeemed are supported.
(`contracts/compliance/pendle/PendleValidatorMisc.sol#36–49`)

The `redeemDueInterestAndRewards` function may cause ERC-20 tokens configured as gauge rewards to be redeemed, which are not authorized Pendle tokens or ERC-20 principal tokens.

The user will then be unable to sell these tokens since they are not authorized to do so.

Exploit Scenario

A user creates a Pendle position and then later harvests its rewards using `redeemDueInterestAndRewards`. The market their position was in had a gauge configured that emits USDT rewards, but USDT is not configured as an allowed token by Arkis. The user is then unable to swap the USDT for their preferred token.

Recommendations

Short term, add logic to the validator to check the rewards tokens and gauges for each market and ensure they are supported using a function like `getRewardsTokens`. Alternatively, configure the off-chain components to dynamically check the tokens associated with gauge emissions and dynamically update supported tokens based on the current gauge status.

Long term, ensure extra care is taken when calling various protocols' rewards claim functions. These functions often claim unexpected tokens that users desire to sell.

9. Malicious users can execute swaps without sandwiching protection to exfiltrate leveraged funds

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARK-9
Target: `contracts/compliance/pendle/PendleValidatorSwapPT.sol`	

Description

Arkis allows users to execute their own transactions as long as they match a specific set of validation criteria. However, sandwiching protection is not enforced by Pendle's swap validator at any point. This allows malicious users to construct a swap that will execute at a poor price, potentially consuming their safety margin and resulting in a loss for lenders.

The primary method of sandwiching protection is the user-provided "minimum amount out" value, which limits the impact of sandwiching attacks against swaps. Under normal circumstances, users are incentivized to set their swap's minimum amount out to a value very close to their expected result from the swap to minimize sandwiching losses.

Without sandwiching protection using minimum amount out values, swap transactions can be sandwiched and executed at poor prices at the user's loss and the sandwicher's benefit. Depending on how well funded the sandwicher is, transactions without protection can be sandwiched up to 30% of the value of the transaction or more.

In Arkis, users trade with leveraged, undercollateralized funds. As long as the value of the account minus the user's collateral is larger than the lender's initial lent amount, the lender does not have to take a loss. This will be referred to as the "loss threshold."

These funds and their positions will be liquidated if their value drops below a safety threshold, a value that would be reached long before the loss threshold. The intention is that as long as accounts are liquidated soon after dropping below their safety threshold and before reaching the loss threshold, lenders can recoup lending losses from the user's collateral.

The key to this attack is that an account close to its safety threshold may intentionally execute an unprotected swap that causes a large, double-digit loss, causing the account's value to drop below the loss threshold.

Exploit Scenario

An attacker opens a margin account and provides 100 USDC of collateral. They are funded with 1,000 USDC of leverage, with a safety threshold of 950 USDC and a loss threshold of 900 USDC (1,000 – 100).

The attacker creates a swap transaction with a `minimumAmountOut` of zero and builds a set of sandwiching transactions to sandwich their own swap. The output of this sandwiched swap would be tokens valued at 700 USDC. The attacker's sandwich transactions earn roughly 300 USDC from the sandwich, depending on the fee environment. The attacker's 100 USDC is liquidated, but they still earn a profit.

Recommendations

Short term, ensure the safety threshold of each account is larger than the loss threshold by an acceptable margin. The possible loss from a sandwiching attack is cited as 30% in the example above, but this value may be higher or lower depending on the AMM, its liquidity conditions, and its fees. Arkis may calculate the largest possible loss on a maximally sandwiched swap and factor this value into the safety threshold.

Long term, inform the safety threshold based on the largest possible sandwiching loss; this will greatly constrain the amount of leverage users can take out based on their collateral. One alternate option is to use an on-chain pricing oracle and factor its data into compliance calculations, preventing users from setting the minimum amount out to undesirable levels. This option would require a robust, manipulation-resistant pricing oracle and may increase the system's gas requirements.

10. Native tokens may be drained from margin accounts via lack of value validation

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARK-10
Target: `contracts/marginAccount/base/CommandSafeExecutor.sol`	

Description

The `CommandSafeExecutor` library is used to call into the compliance contract and validate user commands. However, it does not perform validation on the native tokens transferred by the user's command. Instead, the `CommandSafeExecutor` library simply uses the value provided by the untrusted user-provided `Command` value and executes it, as shown in figure 10.1. This lack of validation allows users to exfiltrate native tokens through specific protocol operators such as the Uniswap router.

```
function execute(Command calldata _cmd, bytes memory _encodedCmds) private {
    Command[] memory cmdsToExecute = abi.decode(_encodedCmds, (Command[]));

    cmdsToExecute.last().value = _cmd.value;
    cmdsToExecute.safeCallAll();
}
```

Figure 10.1: The `execute()` function uses the value specified by the command provided by the user. (`contracts/marginAccount/base/CommandSafeExecutor.sol#30–35`)

The Uniswap router has a relatively unknown property where any tokens left in the router contract after a transaction ends can be stolen by any user. This does not impact users during normal operations because tokens transferred to the router are consumed by the swap and credited to the user initiating the swap. Many other automated market makers use the same design pattern in their router contracts.

If Arkis adds Uniswap or a similar protocol as an authorized operator, a malicious user may create a swap that intentionally leaves tokens behind in the router, allowing a follow-up transaction to steal the funds.

Note that even if Arkis does not support a vulnerable Uniswap-like protocol, users may still leverage this issue to grief Arkis by destroying the leverage tokens and transferring them to a contract that does not have a rescue mechanism. This attack would cost the user their collateral, so the risk introduced by griefing is directly related to the maximum leverage a user can obtain for a specific amount of collateral.

Exploit Scenario

A malicious user opens an account with native ETH as its leverage token. This account is authorized to interact with the Uniswap router and has USDC as an authorized token.

The user then constructs a multi-call with the three following transactions:

1. Call `marginAccount.execute()` with a command that has the following configuration:
   Target: Uniswap router
   Value: 1 `wei`
   Payload: Call `swapExactETHForTokens`, swapping 1 wei of ETH for USDC.

2. Call `marginAccount.execute()` with a command that has the following configuration:
   Target: Uniswap router
   Value: `balanceOf(marginAccount)`
   Payload: Call `swapExactTokensForTokens`, swapping USDC for WETH.

3. Call `router.swapExactETHForTokens`, swapping `balanceOf(router)` for USDC, sending the results to an attacker-controlled account.

The first transaction is to simply create an amount of ERC-20 tokens in the margin account that can be used in the ERC-20 for ERC-20 swap in the second transaction.

The second transaction sends the entire account's native token balance to the Uniswap router. Since it calls an ERC-20 for ERC-20 swap function, none of the native ETH in the router will be refunded.

The last transaction is called outside the context of Arkis and is used to convert the native ETH in the Uniswap router into USDC that is sent to an attacker-controlled address.

Recommendations

Short term, refactor the system's compliance mechanism so that the command value transferred can be validated by the compliance functions. More precise recommendations in addition to long-term recommendations are documented in TOB-ARK-11.

11. Native tokens may be drained by invoking unexpected compliance functions

Severity: **Undetermined**	Difficulty: **N/A**
Type: Undefined Behavior	Finding ID: TOB-ARK-11
Target: `contracts/marginAccount/base/CommandSafeExecutor.sol`	

Description

The `CommandSafeExecutor` contract performs arbitrary calls into the compliance diamond, potentially allowing an unintentional function from the diamond to be called and trigger unexpected behavior up to and including the drainage of native tokens from the margin account contract.

The compliance diamond implements two sets of contracts, the `WhitelistController` contract and those folding the various validator functions for each supported protocol. The validator functions for each protocol use the same ABI as the function the user intends to call on the targeted protocol.

The intended use case is the following: when the `CommandSafeExecutor` contract receives a transaction the user wants to call, such as `swapExactTokenForPt(address,address,uint256)`, it calls the compliance contract using the same ABI and the same parameters the user has provided. The compliance facet of the diamond implements `swapExactTokenForPt(address,address,uint256)` and performs all of the intended validations, finally returning a `Command` struct that represents the sanitized version of the transaction. The `CommandSafeExecutor` contract then executes the sanitized command and includes a `msg.value` from the original user call.

The primary issue is that `CommandSafeExecutor` cannot determine whether a function being called on the compliance diamond is actually compliance-related. This means a user can call other facets of the compliance diamond, such as the functions implemented by `WhitelistController` or functions implemented by the diamond proxy.

Several functions implemented by `WhitelistController`, such as `getOperators`, return memory structures that could be manipulated into being decoded as a `Command[]`. If an attacker can manipulate the “target” field of the decoded command to an address they control, they can exfiltrate native token values due to how `execute()` uses the value specified in the original command, as shown in figure 11.1. This is a similar mechanism to that described in TOB-ARK-10.

```
function execute(Command calldata _cmd, bytes memory _encodedCmds) private {
    Command[] memory cmdsToExecute = abi.decode(_encodedCmds, (Command[]));

    cmdsToExecute.last().value = _cmd.value;
    cmdsToExecute.safeCallAll();
}
```

Figure 11.1: The `execute()` function uses the value specified by the command provided by the user. (`contracts/marginAccount/base/CommandSafeExecutor.sol#30–35`)

This finding would normally have a severity of "high," but due to time constraints, we were unable to verify the finding's exploitability and have left the severity undefined.

The `abi.decode()` function will revert if a naive attempt is made to convert an encoded `string[] memory` to `Command[] memory`, but this conversion may still be technically possible given the right data structure or array contents.

Exploit Scenario

A malicious user constructs a `Command` to be executed with its target set to the zero address, a payload mapping to the ABI of a function implemented by `WhitelistingController`, and a value equal to all of the native tokens stored in the margin account.

The `CommandSafeExecutor` contract executes the `WhitelistingController` function, which returns an encoded memory payload that will decode into a malicious `Command` payload that targets a malicious user-controlled contract.

The `CommandSafeExecutor` contract then sets the value on the command to be executed to the value in the original command, the value equal to all of the native tokens stored in the margin account.

When the command is executed, the fallback function of the malicious user's contract is called, and the contract successfully receives the account's native tokens.

Recommendations

Short term, modify the compliance-related functions to be gated behind a specific function such as `validatePayload(Command cmd)`. This function would iterate over all supported compliance functions and select the correct validator based on the command's function signature. This would prevent unauthorized functions from being called by `CommandSafeExecutor` and provide a route where the `command.value` can be provided to the validator function.

Long term, avoid patterns that allow contracts to call arbitrary function signatures. If the decision to use the diamond proxy pattern was based on needing the ability to arbitrarily add function validators to the diamond, it is recommended to remove the complexity

introduced by the diamond proxy and move to a more traditional proxy standard to aid the project's complexity management and maintainability.

12. Lack of zero-value checks in constructor function

Severity: **Informational**	Difficulty: **N/A**
Type: Code Quality	Finding ID: TOB-ARK-12
Target: `contracts/agreement/Agreement.sol`	

Description

The `Agreement` constructor function fails to validate incoming arguments, so callers of this function could mistakenly set important state variables to a zero value, misconfiguring the system. The `Agreement` constructor is shown in figure 12.1.

```
constructor(address factory_, address compliance_, address dispatcher_) {
    factory = factory_;
    compliance = compliance_;
    dispatcher = dispatcher_;
}
```

Figure 12.1: The `Agreement` constructor fails to perform zero value checks on the configured contracts. (`contracts/agreement/Agreement.sol#54–58`)

Recommendations

Short term, add zero-value checks to all function arguments to ensure that callers cannot set incorrect values, misconfiguring the system.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's CI pipeline, pre-commit hooks, or build scripts.

13. Unused code/features

Severity: **Informational**	Difficulty: **N/A**
Type: Code Quality	Finding ID: TOB-ARK-13
Target: Various files	

Description

The Arkis smart contracts contain several unused functions and features that should be removed from the project.

1. The nonce logic used in `AccountFactoryRepository`, shown in figure 13.1, is unused and can be safely removed.

```
function getAndIncrementNonce() internal returns (bytes32) {
    return bytes32(_storage().nonce++);
}

function getNonce() internal view returns (bytes32) {
    return bytes32(_storage().nonce);
}
```

Figure 13.1 The nonce logic used in `AccountFactoryRepository` is unused. (`contracts/marginAccount/libraries/AccountFactoryRepository.sol#29–35`)

2. The `AddressRegistry` library is unused and can be safely removed.
3. The `neq()` function in `StateMachine.sol` is unused and can be safely removed.
4. The `deleteTransition()` function in `StateMachine.sol` is unused and can be safely removed.

Recommendations

Short term, remove any unused functions to help reduce code bloat, improve maintainability, and reduce deployment costs.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's CI pipeline, pre-commit hooks, or build scripts.

14. Missing/inadequate function-level and library documentation	
Severity: **Informational**	Difficulty: **N/A**
Type: Code Quality	Finding ID: TOB-ARK-14
Target: Various files	

Description

Throughout the Arkis contracts, some functions and libraries lack explicit documentation, which makes the code harder to read and will reduce maintainability in the future.

For example, `CommandLibrary` and `CommandSafeExecutor` both implement an `execute()` function that does not have NatSpec documentation. The libraries they are contained in do not have any documentation either, despite the incredibly important difference between these two functions. The only way a maintainer can differentiate between these functions is to compare the code between the two functions or infer their differences based on which diamond they are implemented in.

This pattern is very common across the project; many functions have similar names to other components' functions but with drastically different functionality.

Some instances of NatSpec documentation do not thoroughly describe how the code is intended to be used. For example, the compiler diamond's entire purpose is to compile and validate command sequences for liquidation—using the same validations for a user command would introduce a security vulnerability. This limitation is not mentioned in any of the supporting documentation or NatSpec comments.

The only way for a maintainer to know that the compiler diamond is safe only for liquidation logic is to be told by a team member or to exhaustively read the code; maintainers that do not fully understand this fact are more likely to introduce errors in future versions of the code.

Recommendations

Short term, ensure functions, libraries, and contracts across the protocol are adequately documented. NatSpec documentation should be as specific as possible, especially when similar functionality is implemented in a different component.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

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

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

C. Mutation Testing

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not result in test failures indicate gaps in the test coverage. Although mutation testing is a slow process, it allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests.

Slither version 0.10.2 introduced a built-in Solidity mutation testing tool with first-class support for Foundry projects. A mutation testing campaign was run against the target smart contracts using the following command:

```
slither-mutate ./contracts \
  --test-cmd='forge test'
```

Figure C.1: A command that runs a mutation testing campaign against all smart contracts in the `contracts` repository

Note that the overall runtime of this campaign is approximately five hours on a consumer-grade laptop. Several examples of interesting mutation test results are shown in figure C.2 through C.4.

```
INFO:Slither-Mutate:Mutating contract CommandSafeExecutor
INFO:Slither-Mutate:[CR] Line 33: 'cmdsToExecute.last().value = _cmd.value' ==>
'//cmdsToExecute.last().value = _cmd.value' --> UNCAUGHT
```

Figure C.2: Selected results from the mutation testing campaign on `CommandSafeExecutor.sol`

In figure C.2, we can see that the native token value passed to the command to be executed can be mutated without causing any tests to fail. This indicates there is no test verifying that the native token value is passed from the source command sequence to the sequence to be executed.

```
INFO:Slither-Mutate:[CR] Line 83: 'factory.returnAccount(address(account))' ==>
'//factory.returnAccount(address(account))' --> UNCAUGHT
INFO:Slither-Mutate:[CR] Line 84: 'success_ =
liquidationPlan.onComplete.run(compiler, chainNameHash) == 0' ==> '//success =
_liquidationPlan.onComplete.run(compiler, chainNameHash) == 0' --> UNCAUGHT
```

Figure C.3: Selected results from the mutation testing campaign on `Dispatcher.sol`

As shown in figure C.3, commenting out key calls like `returnAccount` or `onComplete.run` do not cause the test suite to fail. This indicates these functions are untested.

```
INFO:Slither-Mutate:Mutating contract AgreementFactory
INFO:Slither-Mutate:[CR] Line 61: 'wc.addTokens(agreement, whitelist.tokens)' ==>
'//wc.addTokens(agreement, whitelist.tokens)' --> UNCAUGHT
INFO:Slither-Mutate:[CR] Line 62: 'wc.addOperators(agreement, whitelist.operators)'
==> '//wc.addOperators(agreement, whitelist.operators)' --> UNCAUGHT
```

Figure C.4: Selected results from the mutation testing campaign on `AgreementFactory.sol`

As shown in figure C.4, commenting out `wc.add(Tokens|Operators)` during agreement creation does not cause a unit test failure. Tests should be added/modified to verify that these values are copied over into the agreement correctly.

```
INFO:Slither-Mutate:Mutating contract ERC20Evaluator
INFO:Slither-Mutate:[RR] Line 24: 'token.enforceTokenSupported()' ==> 'revert()' -->
UNCAUGHT
INFO:Slither-Mutate:[RR] Line 27: 'return
Command({target: _request.recipient, value: _request.amountIn, payload: ""})
.asArray()' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[RR] Line 31: 'return CommandPresets.transfer(token,
_request.recipient, _request.amountIn).asArray()' ==> 'revert()' --> UNCAUGHT
INFO:Slither-Mutate:[MIA] Line 26: 'token.isEth()' ==> 'true' --> UNCAUGHT
INFO:Slither-Mutate:[MIA] Line 26: 'token.isEth()' ==> 'false' --> UNCAUGHT
INFO:Slither-Mutate:[MIA] Line 26: 'token.isEth()' ==> '!(token.isEth())' -->
UNCAUGHT
INFO:Slither-Mutate:[MVIE] Line 23: 'address token = ' ==> 'address token ' -->
UNCAUGHT
```

Figure C.5: Selected results from the mutation testing campaign on `ERC20Evaluator.sol`

In figure C.5, we can see that much of the functionality of `ERC20Evaluator` is untested; however, while measuring unit test coverage, we found that each line is run at least four times. This evidence suggests that line coverage should not be used to evaluate the effectiveness of Arkis's unit test suite.

We believe these testing deficiencies are caused by how unreliable line-of-code or branch coverage is for measuring test coverage in complex projects, and that these measurements may have been originally used to measure Arkis's test maturity. This is especially true in examples like that in figure C.5, where the coverage report indicates that each line is called at least four times, but the mutation testing results indicate that most of the functionality of the code is actually untested.

We recommend that Arkis review the existing tests and add additional tests that would catch the aforementioned types of mutations. Then, use a script similar to that provided in figure C.1 to rerun a mutation testing campaign to ensure that the added tests provide adequate coverage.

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From November 6 to November 7, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Arkis team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 14 issues described in this report, Arkis has resolved 11 issues and has temporarily accepted the risk of the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	AccessControlDS uses AccessControl, which has storage collision risks	Resolved
2	EnumerableSet.add/remove return value not checked	Resolved
3	Proxy storage slots may collide due to lack of a namespacing scheme	Undetermined
4	Margin accounts can be registered with suspended agreements	Resolved
5	BeaconDS can create broken margin accounts via an undeployed Account implementation	Resolved
6	ThresholdsVerifier is initialized using the compiler contract instead of the compliance contract	Resolved
7	Risk of leveraged fund drain via hostile agreement contract	Undetermined
8	redeemDueInterestAndRewards does not validate output tokens	Resolved
9	Malicious users can execute swaps without sandwiching protection to exfiltrate leveraged funds	Resolved
10	Native tokens may be drained from margin accounts via lack of value validation	Resolved

11	Native tokens may be drained by invoking unexpected compliance functions	**Resolved**
12	Lack of zero-value checks in constructor function	**Resolved**
13	Unused code/features	**Resolved**
14	Missing/inadequate function-level and library documentation	**Undetermined**

Detailed Fix Review Results

TOB-ARK-1: AccessControlDS uses AccessControl, which has storage collision risks
Resolved in PR #271. `AccessControlDS` now inherits from `AccessControlUpgradeable`.

TOB-ARK-2: EnumerableSet.add/remove return value not checked
Resolved in PR #271. `Agreement` now checks the Boolean result of set operations.

TOB-ARK-3: Proxy storage slots may collide due to lack of a namespacing scheme
Risk accepted by Arkis. Arkis states that since the issue introduces no vulnerabilities at this time, it will be mitigated in a future version of the protocol.

TOB-ARK-4: Margin accounts can be registered with suspended agreements
Resolved in PR #271. `Agreement.enforceCanBorrow` now uses the `whenNotPaused` modifier. This prevents a paused agreement contract from being processed by the `registerMarginAccount` function when it is called by an honest user. While a user could still theoretically create their own `Agreement` contract that does not respect the pause functionality, this possibility is out of this finding's scope and falls within the impact domain of TOB-ARK-7.

TOB-ARK-5: BeaconDS can create broken margin accounts via an undeployed Account implementation
Resolved in PR #271. The `BeaconDS` contract now verifies whether an implementation contract supports the intended interfaces using high-level calls. These high-level calls will revert if there is no contract deployed at the implementation address, resolving this issue.

TOB-ARK-6: ThresholdsVerifier is initialized using the compiler contract instead of the compliance contract
Resolved in PR #271. The state variable for the compliance address and the constructor parameter have been removed.

TOB-ARK-7: Risk of leveraged fund drain via hostile agreement contract
Risk accepted by Arkis. Arkis states that the attack is prevented in the off-chain application, and given how the off-chain application works, it is impossible for a hostile `Agreement` contract to be funded.

TOB-ARK-8: redeemDueInterestAndRewards does not validate output tokens
Resolved through discussion with Arkis. Arkis states that the functionality described in the finding is the intended operation.

TOB-ARK-9: Malicious users can execute swaps without sandwiching protection to exfiltrate leveraged funds
Resolved in PR #271 and commit hash `d3b7831300719`. A short-term fix has been applied outside of the system's smart contracts. The safety buffer will be configured such that the maximum theoretical sandwich profit cannot exceed the size of the safety buffer. In

addition, a guard has been added to prevent multiple user transactions from being executed in the same block. This guard allows the Arkis liquidation engine time to react to an attacker trying to drain funds using self-constructed sandwiches. Over the long term, it is still recommended that an on-chain fix is implemented to prevent unfavorable swaps.

TOB-ARK-10: Native tokens may be drained from margin accounts via lack of value validation
Resolved in PR #271. The native token value is now passed to the compliance diamond for validation. The existing validation functions refund the native token unless it is explicitly required for the transaction.

TOB-ARK-11: Native tokens may be drained by invoking unexpected compliance functions
Resolved in PR #271 by the TOB-ARK-10 fix. Because the native token amount is now explicitly validated, this finding's potential impact is negligible.

TOB-ARK-12: Lack of zero-value checks in constructor function
Resolved PR #271. Multiple zero-value checks have been added to the `Agreement` constructor.

TOB-ARK-13: Unused code/features
Resolved in PR #271. The unused logic has been removed from the system's smart contracts.

TOB-ARK-14: Missing/inadequate function-level and library documentation
Risk accepted by Arkis. Arkis states that this finding will be addressed in the long term when addressing technical debt in the future.

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.