Arcade.xyz V3

Security Assessment

July 31, 2023

Prepared for:
Cary Galant
Arcade.xyz

Prepared by: **Alexander Remie, Guillermo Larregay, and Robert Schneider**

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

Arcade engaged Trail of Bits to review the security of the Arcade.xyz V3 protocol. The protocol is an NFT lending platform that allows users to initiate, roll over, and repay loans, with unique NFTs and other assets serving as collateral to back loans. It implements a network of contracts to handle loan management, fee assessment, and asset vaulting, which effectively allows users to bundle assets into a single transferable NFT that can be used as collateral.

A team of three consultants conducted the review from May 15 to June 9, 2023, for a total of eight engineer-weeks of effort. Our testing efforts focused on the security, reliability, and functionality of the protocol. With full access to the source code and documentation, we performed static and dynamic testing of the protocol, using automated and manual processes.

Observations and Impact

Arcade's NFT lending protocol uses adequate access controls and is protected against potential attack vectors such as reentrancy and front-running. Nonetheless, we identified consistent issues across various components, particularly those related to input validation, return value checks, and potential loss of access to funds (TOB-ARCADE-8, TOB-ARCADE-12), indicating a need for enhanced data validity checks and safeguarding mechanisms. Concerns regarding the approval mechanism (TOB-ARCADE-3) and event emissions (TOB-ARCADE-1, TOB-ARCADE-4) indicate a need for more robust logging and ERC-20 token management. Furthermore, we identified issues related to incorrect implementations and operations—like the incorrect encoding of the `itemPredicates` parameter, which obfuscates the verifier address (TOB-ARCADE-14), and misaligned incentives for adhering to the expected order of operations (TOB-ARCADE-16); these issues suggest that further attention is needed to refine certain implementations and prevent invalid or exploitable operations. Although critical security measures are implemented in the protocol, improvements are necessary in the areas of data validation, logging, timing, and configuration.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the Arcade team 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.

- **Ensure that all inputs and expected return values are validated (TOB-ARCADE-5 and TOB-ARCADE-8).** We found issues related to input and return value validation in several contracts, including `LoanCore`, `OriginationController`, `RepaymentController`, `VaultFactory`, and `AssetVault`. These issues suggest that the validity of input data and responses from the contract functions might not be adequately enforced or verified across the protocol.

- **Develop an incident response plan.** Such a plan will help the Arcade team to prepare for failure scenarios and will outline the appropriate responses to them. Refer to appendix G for guidance on creating an incident response plan.

- **Update the user-facing documentation to reflect V3 of the protocol.** The current documentation is related to V2 of the protocol. Up-to-date user-facing documentation will limit the risk of user confusion due to changes between V3 and V2.

- **Improve the inline comments.** Some parts of the implementation (such as the rollover mechanism and the `PunksVerifier` contract) are difficult to understand due to insufficient comments to explain the implementation.

- **Improve the unit testing suite.** Although test coverage is at 100%, the unit testing suite still has gaps, which has caused issues like TOB-ARCADE-5 and TOB-ARCADE-9 to persist unnoticed.

Breakdown of Findings

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	4
Low	5
Informational	6
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Configuration	1
Data Validation	6
Error Reporting	1
Testing	1
Timing	2
Undefined Behavior	4

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Robert Schneider, Consultant
robert.schneider@trailofbits.com

Project Timeline

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

Date	Event
May 11, 2023	Pre-project kickoff call
May 22, 2023	Status update meeting #1
May 30, 2023	Status update meeting #2
June 5, 2023	Status update meeting #3
June 12, 2023	Delivery of report draft and report readout meeting
July 31, 2023	Delivery of final report with fix review

Project Goals

The engagement was scoped to provide a security assessment of the Arcade.xyz V3 NFT lending protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do the NFTs used in the protocol comply with standard interfaces like ERC-721, ERC-1155, and ERC-20? Do the NFTs deviate from these standards in a way that could introduce unexpected behaviors or compatibility issues? Are the “safe” versions of the functions of these standards used?
- Is the protocol's claim process (which is triggered when a borrower fails to repay a loan) fair, transparent, and secure, preventing unfair liquidation of NFTs?
- Does the protocol implement measures to prevent front-running, especially for sensitive operations like loan repayment and liquidation?
- Are the asset vaults implemented securely, protecting borrowers’ funds and their access to them at all times? Additionally, does the use of asset vaults in any way prevent lenders from being able to claim collateral when their loans are not repaid?
- Does the system handle loan initialization and repayments properly, including those involving multiple assets and various types of NFTs?
- Does the protocol ensure that the `tokenURI` for each NFT, which points to its metadata, is immutable after minting, preventing a malicious user from changing it?
- Are the rules and parameters surrounding collateral requirements in the lending protocol clearly defined and secure, providing adequate safeguards in case of attempted fraud?
- Does the protocol correctly and securely implement EIP-712 for structured data hashing and signing? Are there robust measures in place to prevent potential security issues related to the misuse of domain separators or type hashes?
- Are role-based access controls securely and effectively implemented within the system? Are there safeguards preventing unauthorized access, and are the roles appropriately assigned to prevent potential exploitation or misuse?

Project Targets

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

Arcade.xyz

Repository	https://github.com/arcadexyz/arcade-protocol
Version	`4f510e0e2287901abb21265f72aa4465166ab09d`
Type	Solidity
Platform	Ethereum

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:

Core

- **`LoanCore`:** The `LoanCore` contract is the core contract of the protocol. Both of the controller contracts call functions in this contract. This contract contains functions to start loans, repay loans, claim the collateral backing loans that have not been repaid, roll over loans, and withdraw fees. We reviewed the implementation for vulnerabilities that could allow attackers to steal funds or cause a denial of service, and we looked for ways that users could lose access to their funds (TOB-ARCADE-8). Additionally, we reviewed the access controls, event emission, input validation, and pausing mechanism of the contract. We also closely reviewed the correctness of the rollover mechanism and state machine and looked for opportunities to perform reentrancy and front-running attacks. Lastly, we reviewed the contract to ensure that it correctly applies fees, uses promissory notes correctly, and safely transfers ERC-20, ERC-721, and ERC-1155 tokens.

- **`OriginationController`:** This is one of the two main controller contracts through which users interact with the protocol. This contract performs validations on calls made to `LoanCore`, which performs the actual operations. This contract contains functions to start loans and to roll over loans. We reviewed the implementation to look for flaws related to input validation, access controls, missing events, loops that exhaust gas, signature verification vulnerabilities, EIP-712 adherence, ERC-2612 (permit) adherence, and front-running and reentrancy vulnerabilities. Additionally, we looked for ways to create loans with invalid terms, create loans with invalid counterparties, roll over loans without fully repaying the previous loans, or subvert the rollover mechanism in other ways.

- **`RepaymentController`:** This is the second of the two main controller contracts through which users interact with the protocol. This contract performs validations on calls made to `LoanCore`, which performs the actual operations. This contract contains functions to repay loans and to claim the collateral backing loans that ended but were not repaid. We reviewed the access controls, input validation (TOB-ARCADE-8), and fee calculations. We looked for front-running and reentrancy vulnerabilities. Additionally, we looked for ways to repay a loan without actually fully repaying it, flaws in the withdrawal of lender funds using the `redeemNote` function, transaction ordering flaws between repayment and liquidation (TOB-ARCADE-13), flaws that could prevent borrowers from fully realizing the value of their loans without risking default (TOB-ARCADE-13), and ways for attackers to illegitimately claim borrowers’ collateral.

- **`FeeController`:** This contract contains the configured fees and functions to update the fee values, adhering to the maximum fee amount per fee category. We reviewed the implementation for issues related to access controls, input validation, and missing configurations of maximum fees (TOB-ARCADE-9). Additionally, we reviewed the effect of changing configured fees while loans are active (TOB-ARCADE-10).

- **`PromissoryNote`:** This contract implements an ERC-721 token that is used to track the lender and borrower parties involved in a loan. This NFT is minted when a loan is started and burned when the loan ends (through repayment or the claiming of collateral). We reviewed the access controls, input validation, and return value validation (TOB-ARCADE-5). Additionally, we reviewed the implementation of overridden OpenZeppelin functions to ensure that the contract still adheres to the ERC-721 standard.

Vaults

- **`VaultFactory`:** This contract can be used by potential borrowers to deploy an `AssetVault` contract through a minimal proxy pattern. The `VaultFactory` contract is both a factory and an ERC-721 token. For each deployed `AssetVault` contract, an accompanying ERC-721 token is minted; the owner of this token is the owner of the associated `AssetVault` contract. We reviewed the implementation to look for flaws related to access controls, reentrancy, event emission, input validation (TOB-ARCADE-8), and return value validation (TOB-ARCADE-5). Additionally, we reviewed the claim fees function for any flaws that might prevent fees from being withdrawable (TOB-ARCADE-7), and we checked for the use of `SafeERC20` functions wherever possible (TOB-ARCADE-12).

- **`AssetVault`:** This contract can be deployed by users through the `VaultFactory` contract and is used to bundle multiple assets (ERC-20, ERC-721, ERC-1155, ETH, CryptoPunks) inside one NFT (the `VaultFactory` NFT—refer to the bullet point above). This bundled NFT can then be used as collateral to back loans. We manually reviewed the contract to look for flaws related to access controls, reentrancy, front-running, input validation, initialization, ownership tracking, and inconsistent error messages. We also looked for ways that attackers could steal funds from an asset vault and that the owner of an asset vault could withdraw its assets while it is being used as collateral in an active loan. Additionally, we investigated whether the functionality allowing users to nest asset vaults could cause any problems, such as the loss of access to the funds within an asset vault (TOB-ARCADE-11). Lastly, we reviewed the use of safe transfer functions to withdraw assets, the approval mechanism (TOB-ARCADE-3), the access controls and implementation of the various “call” functions, and the mechanism for disabling and enabling withdrawals.

- **`CallWhitelist`, `CallWhitelistApprovals`, `CallWhitelistDelegation`, and `CallWhitelistAllExtensions`:** These four contracts together (through inheritance) make up the “whitelist” contract that is attached to deployed `AssetVault` contracts and is used to allow/disallow specific functions to be called from the `AssetVault` contract. This is mostly used to be able to participate in airdrops while an NFT is locked up in an asset vault. The owner of the “whitelist” contract is allowed to whitelist functions; in practice, the owner will be an Arcade-owned account or a governance system. We manually reviewed the contract for flaws related to inheritance, input validation, access controls, and event emission (TOB-ARCADE-4). Additionally, we reviewed the use of the blacklist and whitelist to determine whether whitelisted calls that are also present in the blacklist would still be deemed whitelisted.

Verifiers

- **`ItemsVerifier`**: This contract verifies that asset vaults have an asset or a list of assets (ERC-20, ERC-721, ERC-1155). We manually reviewed the contract to check that the described specification adheres to the implementation and to look for any edge cases that are not handled correctly. Additionally, we reviewed the wildcard process to determine whether it could be somehow used by an attacker.

- **`CollectionWideOfferVerifier`:** This contract checks that the NFT collateral in the loan terms is equal to the NFT specified in the predicate. Any token within an NFT collection would satisfy this verifier—the contract does not check for a specific `tokenId` within an NFT collection. We manually reviewed the contract to check that the described specification adheres to the implementation and that all required validation is present.

- **`UnvaultedItemsVerifier`:** This contract checks that the collateral token in the loan terms and its `tokenId` match those specified in the predicate. We manually reviewed the contract to check that the described specification adheres to the implementation and that all required validation is present.

- **`PunksVerifier`:** This contract checks whether a CryptoPunk (any CryptoPunk or a specific CryptoPunk) is present in the collateral specified in the loan terms. We manually reviewed the contract to check that the described specification adheres to the implementation and that the validation is correct. We also reviewed the CryptoPunks contract to ensure that the integration is implemented correctly.

- **`ArtBlocksVerifier`:** This contract checks whether a token within a list of supported ArtBlocks collections (any ArtBlocks token or a specific token) is present in the collateral specified in the loan terms. We manually reviewed the contract to check that the described specification adheres to the implementation and that the validation is correct. We also reviewed the supported ArtBlocks contract to ensure that the integration is implemented correctly.

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:

- **Decentralized governance:** The Arcade protocol uses a decentralized governance system consisting of multiple smart contracts. These contracts were not part of the repository under review and were considered out of scope. We recommend performing a separate audit to review the governance-related smart contracts.

- **External contracts for unit tests:** The provided repository contains various helper contracts that are used only to write unit tests that interact with these external contracts. For example, these external contracts include an implementation of the EIP-5639 standard and a copy of the CryptoPunks smart contract. These contracts were all considered out of scope and may warrant a separate review.

- **OpenZeppelin libraries:** The Arcade protocol uses the OpenZeppelin-provided smart contracts (version 4.3.2). These contracts were considered out of scope for this audit.

- **Front end:** The Arcade system also includes a front end to interact with the smart contracts. The front end was considered out of scope for this audit and may warrant a separate review.

Automated Testing

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

Test Harness Configuration

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

Tool	Description	Policy
Universalmutator	A deterministic mutation generator that detects gaps in test coverage	Appendix F

Test Results

The results of this focused testing are detailed below.

Universalmutator: The following table displays the proportion of mutants for which all unit tests passed. A small number of valid mutants indicates that test coverage is thorough and that any newly introduced bugs are likely to be caught by the test suite. A large number of valid mutants indicates gaps in the test coverage where errors may go unnoticed. We used the results in the following table to guide our manual review, giving extra attention to code for which test coverage appears to be incomplete.

It is important to note that some of the valid mutants can be false positives, such as mutants that remove the public visibility modifier from a public function or variable. These results must be manually checked before drawing conclusions. Refer to appendix F for more information.

Target	Valid Mutants
`contracts/FeeController.sol`	5.88%
`contracts/LoanCore.sol`	8.68%
`contracts/OriginationController.sol`	5.65%
`contracts/PromissoryNote.sol`	13.65%

contracts/RepaymentController.sol	5.97%
contracts/libraries/FeeLookups.sol	23.21%
contracts/libraries/InterestCalculator.sol	13.33%
contracts/libraries/LoanLibrary.sol	12.20%
contracts/errors/Lending.sol	3.07%
contracts/errors/Vault.sol	2.35%
contracts/vault/AssetVault.sol	8.10%
contracts/vault/CallBlacklist.sol	5.70%
contracts/vault/CallWhitelist.sol	6.25%
contracts/vault/CallWhitelistAllExtensions.sol	8.33%
contracts/vault/CallWhitelistApprovals.sol	8.86%
contracts/vault/CallWhitelistDelegation.sol	9.09%
contracts/vault/OwnableERC721.sol	4.69%
contracts/vault/VaultFactory.sol	11.35%
contracts/verifiers/ArtBlocksVerifier.sol	0.0%
contracts/verifiers/CollectionWideOfferVerifier.sol	0.0%
contracts/verifiers/ItemsVerifier.sol	0.0%
contracts/verifiers/PunksVerifier.sol	0.0%
contracts/verifiers/UnvaultedItemsVerifier.sol	0.0%

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 system contains almost no arithmetic apart from the arithmetic used for calculating fees. The arithmetic that is present features automatic underflow and overflow protection through the use of solc 0.8.x. There are no unchecked blocks.	**Satisfactory**
Auditing	All state-changing functions emit events. Custom errors are used in the codebase to signal specific reasons for reverts. We identified one potentially confusing event emission (TOB-ARCADE-4) and found that an error throws different error messages (TOB-ARCADE-1). We recommend that the Arcade team implement an incident response plan (appendix G) and use an off-chain monitoring system to provide early warnings when problems arise in the protocol.	**Satisfactory**
Authentication / Access Controls	The protocol's contracts all implement appropriate access controls. Privileges are dropped when they are not needed anymore after deployment. Externally owned accounts do not have access to user funds. We did identify one issue related to access controls on asset vaults that could cause users to lose access to funds (TOB-ARCADE-11). We recommend that the Arcade team add a page in the documentation that describes all of the privileged roles per contract and the actions that each role is allowed to perform.	**Satisfactory**
Complexity Management	Overall, the contracts and functions are easy to read, perform a single task or group of tasks, and are well documented through the use of NatSpec comments. We detected no code duplication or redundancy in the	**Moderate**

	codebase. However, one exception to this is the loan rollover mechanism, which is divided into five functions across two contracts. The logic of these functions is hard to follow due to the high number of conditions that are checked in each of the functions. Additionally, all of these functions could use more inline comments to explain what they do and why. We recommend that the Arcade team redesign the functions to make them less complex and add additional inline comments in those newly designed versions.	
Decentralization	The Arcade governance system is in charge of whitelisting allowed currencies, whitelisting certain functions that can be called on asset vaults, and pausing the system. However, when the system is paused, no liquidations can happen, but users can still repay their loans, thereby ensuring that a pause of the system does not enable unfair liquidations. Also, the governance system cannot access any user assets at any time. The governance system used by the Arcade protocol was out of scope for this audit; therefore, further investigation is required.	**Further Investigation Required**
Documentation	The overall user-facing documentation on the Arcade website is thorough and provides numerous real-world use cases to walk users through how the protocol and the UI work. The implementation makes heavy use of NatSpec comments and inline comments. However, the rollover mechanism and the `PunksVerifier` contract could use more inline comments. Additionally, we found some misleading comments and mistakes in comments (as documented in appendix C, which lists our code quality recommendations). Finally, we recommend updating the user-facing documentation to reflect V3 of the protocol instead of V2.	**Moderate**
Transaction Ordering Risks	The protocol, as currently evaluated, does not appear to pose immediate threats to user assets from unforeseen transaction ordering or maximum extractable value (MEV). However, certain design choices obscure	**Moderate**

	incentives related to the anticipated sequence of operations within the system (TOB-ARCADE-13, TOB-ARCADE-16). To address these issues, we recommend implementing a robust testing strategy that accentuates potential risks associated with transaction ordering and timing within the system.	
Low-Level Manipulation	The Arcade protocol does not use assembly, and the use of low-level calls is limited. We did identify one potential issue related to the use of the low-level `transfer()` function instead of `call()`, but it is not an immediate problem due to the current fee recipient contract (TOB-ARCADE-7).	**Satisfactory**
Testing and Verification	The test coverage is 100%, although some of the issues found (such as TOB-ARCADE-5 and TOB-ARCADE-9) could have been prevented if the test scenarios covered edge cases. The Arcade protocol does not use more advanced testing tools such as fuzzing at the moment, which is an area the protocol could improve on in the future. Additionally, consider using mutation testing to check the robustness of the unit tests (appendix F).	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Different zero-address errors thrown by single and batch NFT withdrawal functions	Error Reporting	**Informational**
2	Solidity compiler optimizations can be problematic	Undefined Behavior	**Undetermined**
3	callApprove does not follow approval best practices	Undefined Behavior	**Informational**
4	Risk of confusing events due to missing checks in whitelist contracts	Data Validation	**Low**
5	Missing checks of _exists() return value	Data Validation	**Informational**
6	Incorrect deployers in integration tests	Testing	**Informational**
7	Risk of out-of-gas revert due to use of transfer() in claimFees	Undefined Behavior	**Informational**
8	Risk of lost funds due to lack of zero-address check in functions	Data Validation	**Medium**
9	The maximum value for FL_09 is not set by FeeController	Data Validation	**Low**
10	Fees can be changed while a loan is active	Timing	**Low**
11	Asset vault nesting can lead to loss of assets	Access Controls	**Low**
12	Risk of locked assets due to use of _mint instead of _safeMint	Undefined Behavior	**Medium**

13	Borrowers cannot realize full loan value without risking default	Timing	**Medium**
14	itemPredicates encoded incorrectly according to EIP-712	Configuration	**Low**
15	The fee values can distort the incentives for the borrowers and lenders	Data Validation	**Informational**
16	Malicious borrowers can use forceRepay to grief lenders	Data Validation	**Medium**

Detailed Findings

1. Different zero-address errors thrown by single and batch NFT withdrawal functions	
Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-ARCADE-1
Target: `contracts/vault/AssetVault.sol`	

Description

The `withdrawBatch` function throws an error that is different from the single NFT withdrawal functions (`withdrawERC721`, `withdrawERC1155`). This could confuse users and other applications that interact with the Arcade contracts.

The `withdrawBatch` function throws a custom error (`AV_ZeroAddress`) if the `to` parameter is set to the zero address. The single NFT withdrawal functions `withdrawERC721` and `withdrawERC1155` do not explicitly check the `to` parameter. All three of these functions internally call the `_withdrawERC721` and `_withdrawERC1155` functions, which also do not explicitly check the `to` parameter. The lack of such a check is not a problem: according to the ERC-721 and ERC-1155 standards, a transfer must revert if `to` is the zero address, so the single NFT withdrawal functions will revert on this condition. However, they will revert with the error message that is defined inside the actual NFT contract instead of the Arcade `AV_ZeroAddress` error, which is thrown when `withdrawBatch` reverts.

```
    function withdrawBatch(
        address[] calldata tokens,
        uint256[] calldata tokenIds,
        TokenType[] calldata tokenTypes,
        address to
    ) external override onlyOwner onlyWithdrawEnabled {
        uint256 tokensLength = tokens.length;
        if (tokensLength > MAX_WITHDRAW_ITEMS) revert AV_TooManyItems(tokensLength);
        if (tokensLength != tokenIds.length) revert AV_LengthMismatch("tokenId");
        if (tokensLength != tokenTypes.length) revert AV_LengthMismatch("tokenType");
        if (to == address(0)) revert AV_ZeroAddress();

        for (uint256 i = 0; i < tokensLength; i++) {
            if (tokens[i] == address(0)) revert AV_ZeroAddress();
```

Figure 1.1: A snippet of the `withdrawBatch` function in *`arcade-protocol/contracts/vault/AssetVault.sol`*

Additionally, the CryptoPunks NFT contract does not follow the ERC-721 and ERC-1155 standards and contains no check that prevents funds from being transferred to the zero address (and the function is called `transferPunk` instead of the standard `transfer`). An explicit check to ensure that `to` is not the zero address inside the `withdrawPunk` function is therefore recommended.

```
function transferPunk(address to, uint punkIndex) {
    if (!allPunksAssigned) throw;
    if (punkIndexToAddress[punkIndex] != msg.sender) throw;
    if (punkIndex >= 10000) throw;
    if (punksOfferedForSale[punkIndex].isForSale) {
        punkNoLongerForSale(punkIndex);
    }
    punkIndexToAddress[punkIndex] = to;
    balanceOf[msg.sender]--;
    balanceOf[to]++;
    Transfer(msg.sender, to, 1);
    PunkTransfer(msg.sender, to, punkIndex);
    // Check for the case where there is a bid from the new owner and refund
it.
    // Any other bid can stay in place.
    Bid bid = punkBids[punkIndex];
    if (bid.bidder == to) {
        // Kill bid and refund value
        pendingWithdrawals[to] += bid.value;
        punkBids[punkIndex] = Bid(false, punkIndex, 0x0, 0);
    }
}
```

Figure 1.2: The `transferPunk` function in `CryptoPunksMarket` contract (Etherscan)

Lastly, there is no string argument to the `AV_ZeroAddress` error to indicate which variable equaled the zero address and caused the revert, unlike the `AV_LengthMismatch` error. For example, in the batch function (figure 1.1), the `AV_ZeroAddress` could be thrown in line 203 or 206.

Exploit Scenario

Bob, a developer of a front-end blockchain application that interacts with the Arcade contracts, develops a page that interacts with an `AssetVault` contract. In his implementation, he catches specific errors that are thrown so that he can show an informative message to the user. Because the batch and withdrawal functions throw different errors when `to` is the zero address, he needs to write two versions of error handlers instead of just one.

Recommendations

Short term, add the zero address check with the custom error to the `_withdrawERC721` and `_withdrawERC1155` functions. This will cause the same custom error to be thrown for all of the single and batch NFT withdrawal functions. Also, add an explicit zero-address check inside the `withdrawPunk` function. Lastly, add a string argument to the `AV_ZeroAddress` custom error that is used to indicate the name of the variable that triggered the error (similar to the one in `AV_LengthMismatch`).

Long term, ensure consistency in the errors thrown throughout the implementation. This will allow users and developers to understand errors that are thrown and will allow the Arcade team to test fewer errors.

2. Solidity compiler optimizations can be problematic

Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ARCADE-2
Target: `hardhat.config.ts`	

Description

Arcade has enabled optional compiler optimizations in Solidity. According to a November 2018 audit of the Solidity compiler, the optional optimizations may not be safe.

```
    optimizer: {
        enabled: optimizerEnabled,
        runs: 200,
    },
```

Figure 2.1: The solc optimizer settings in `arcade-protocol/hardhat.config.ts`

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. Another bug due to the incorrect caching of Keccak-256 was reported.

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

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 Arcade 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.

3. callApprove does not follow approval best practices

Severity: **Informational**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-ARCADE-3
Target: `contracts/vault/AssetVault.sol`	

Description

The `AssetVault.callApprove` function has undocumented behaviors and lacks the increase/decrease approval functions, which might impede third-party integrations.

A well-known race condition exists in the ERC-20 approval mechanism. The race condition is enabled if a user or smart contract calls `approve` a second time on a spender that has already been allowed. If the spender sees the transaction containing the call before it has been mined, they can call `transferFrom` to transfer the previous value and then still receive authorization to transfer the new value. To mitigate this, `AssetVault` uses the `SafeERC20.safeApprove` function, which will revert if the allowance is updated from nonzero to nonzero. However, this behavior is not documented, and it might break the protocol's integration with third-party contracts or off-chain components.

```
function callApprove(
   address token,
   address spender,
   uint256 amount
) external override onlyAllowedCallers onlyWithdrawDisabled nonReentrant {
   if (!CallWhitelistApprovals(whitelist).isApproved(token, spender)) {
       revert AV_NonWhitelistedApproval(token, spender);
   }

   // Do approval
   IERC20(token).safeApprove(spender, amount);

   emit Approve(msg.sender, token, spender, amount);
}
```

Figure 3.1: The `callApprove` function in `arcade-protocol/contracts/vault/AssetVault.sol`

```
/**
* @dev Deprecated. This function has issues similar to the ones found in
* {IERC20-approve}, and its usage is discouraged.
*
* Whenever possible, use {safeIncreaseAllowance} and
* {safeDecreaseAllowance} instead.
```

```
    */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
spender, value));
    }
```

Figure 3.2: The `safeApprove` function in
`openzeppelin-contracts/contracts/token/ERC20/utils/SafeERC20.sol`

An alternative way to mitigate the ERC-20 race condition is to use the `increaseAllowance` and `decreaseAllowance` functions to safely update allowances. These functions are widely used by the ecosystem and allow users to update approvals with less ambiguity.

```
    function safeIncreaseAllowance(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        uint256 newAllowance = token.allowance(address(this), spender) + value;
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
spender, newAllowance));
    }

    function safeDecreaseAllowance(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        unchecked {
            uint256 oldAllowance = token.allowance(address(this), spender);
            require(oldAllowance >= value, "SafeERC20: decreased allowance below
zero");
            uint256 newAllowance = oldAllowance - value;
            _callOptionalReturn(token,
abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
        }
    }
```

Figure 3.3: The `safeIncreaseAllowance` and `safeDecreaseAllowance` functions in
`openzeppelin-contracts/contracts/token/ERC20/utils/SafeERC20.sol`

Exploit Scenario

Alice, the owner of an asset vault, sets up an approval of 1,000 for her external contract by calling `callApprove`. She later decides to update the approval amount to 1,500 and again calls `callApprove`. This second call reverts, which she did not expect.

Recommendations

Short term, take one of the following actions:

- Update the documentation to make it clear to users and other integrating smart contract developers that two transactions are needed to update allowances.

- Add two new functions in the `AssetVault` contract: `callIncreaseAllowance` and `callDecreaseAllowance`, which internally call `SafeERC20.safeIncreaseAllowance` and `SafeERC20.safeDecreaseAllowance`, respectively.

Long term, when using external libraries/contracts, always ensure that they are being used correctly and that edge cases are explained in the documentation.

4. Risk of confusing events due to missing checks in whitelist contracts

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARCADE-4
Target: `contracts/vault/CallWhitelist.sol`, `contracts/vault/CallWhitelistDelegation.sol`	

Description

The `CallWhitelist` contract's `add` and `remove` functions do not check whether the given call has been registered in the whitelist. As a result, `add` could be used to register calls that have already been registered, and `remove` could be used to remove calls that have never been registered; these types of calls would still emit events. For example, invoking `remove` with a call that is not in the whitelist would emit a `CallRemoved` event even though no call was removed. Such an event could confuse off-chain monitoring systems, or at least make it more difficult to retrace what happened by looking at the emitted event.

```
function add(address callee, bytes4 selector) external override onlyOwner {
   whitelist[callee][selector] = true;
   emit CallAdded(msg.sender, callee, selector);
}
```

Figure 4.1: The add function in `arcade-protocol/contracts/vault/CallWhitelist.sol`

```
function remove(address callee, bytes4 selector) external override onlyOwner {
   whitelist[callee][selector] = false;
   emit CallRemoved(msg.sender, callee, selector);
}
```

Figure 4.2: The `remove` *function in* `arcade-protocol/contracts/vault/CallWhitelist.sol`

A similar problem exists in the `CallWhitelistDelegation.setRegistry` function. This function can be called to set the registry address to the current registry address. In that case, the emitted `RegistryChanged` event would be confusing because nothing would have actually changed.

```
function setRegistry(address _registry) external onlyOwner {
   registry = IDelegationRegistry(_registry);

   emit RegistryChanged(msg.sender, _registry);
}
```

Figure 4.3: The `setRegistry` function in
`arcade-protocol/contracts/vault/CallWhitelistDelegation.sol`

Arcade has explained that the owner of the whitelist contracts in Arcade V3 will be a (set of) governance contract(s), so it is unlikely that this issue will happen. However, it is possible, and it could be prevented by more validation.

Exploit Scenario

No calls have yet been added to the whitelist in `CallWhitelist`. Through the governance system, a proposal to remove a call with the address `0x1` and the selector `0x12345678` is approved. The proposal is executed, and `CallWhitelist.remove` is called. The transaction succeeds, and a `CallRemoved` event is emitted, even though the “removed” call was never in the whitelist in the first place.

Recommendations

Short term, add validation to the `add`, `remove`, and `setRegistry` functions. For the `add` function, it should ensure that the given call is not already in the whitelist. For the `remove` function, it should ensure that the call is currently in the whitelist. For the `setRegistry` function, it should ensure that the new registry address is not the current registry address. Adding this validation will prevent confusing events from being emitted and ease the tracing of events in the whitelist over time.

Long term, when dealing with function arguments, always ensure that all inputs are validated as tightly as possible and that the subsequent emitted events are meaningful. Additionally, consider setting up an off-chain monitoring system that will track important system events. Such a system will provide an overview of the events that occur in the contracts and will be useful when incidents occur.

5. Missing checks of _exists() return value

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARCADE-5
Target: `contracts/PromissoryNote.sol`, `contracts/vault/VaultFactory.sol`	

Description

The ERC-721 `_exists()` function returns a Boolean value that indicates whether a token with the specified `tokenId` exists. In two instances in Arcade's codebase, the function is called but its return value is not checked, bypassing the intended result of the existence check.

In particular, in the `PromissoryNote.tokenURI()` and `VaultFactory.tokenURI()` functions, `_exists()` is called before the URI for the `tokenId` is returned, but its return value is not checked. If the given NFT does not exist, the URI returned by the `tokenURI()` function will be incorrect, but this error will not be detected due to the missing return value check on `_exists()`.

```
165    function tokenURI(uint256 tokenId) public view override(INFTWithDescriptor,
ERC721) returns (string memory) {
166        _exists(tokenId);
167
168        return descriptor.tokenURI(address(this), tokenId);
169    }
```

Figure 5.1: The `tokenURI` function in `arcade-protocol/contracts/PromissoryNote.sol`

```
48    function tokenURI(address, uint256 tokenId) external view override returns
(string memory) {
49        return bytes(baseURI).length > 0 ? string(abi.encodePacked(baseURI,
tokenId.toString())) : "";
50    }
```

Figure 5.2: The `tokenURI` function in `arcade-protocol/contracts/nft/BaseURIDescriptor.sol`

Exploit Scenario

Bob, a developer of a front-end blockchain application that interacts with the Arcade contracts, develops a page that lists users' promissory notes and vaults with their respective URIs. He accidentally passes a nonexistent `tokenId` to `tokenURI()`, causing his application to show an incorrect or incomplete URI.

Recommendations

Short term, add a check for the `_exists()` function's return value to both of the `tokenURI()` functions to prevent them from returning an incomplete URI for nonexistent tokens.

Long term, add new test cases to verify the expected return values of `tokenURI()` in all contracts that use it, with valid and invalid tokens as arguments.

6. Incorrect deployers in integration tests

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-ARCADE-6
Target: `test/Integration.ts`	

Description

The fixture deployment function in the provided integration tests uses different signers for deploying the Arcade contracts before performing the tests.

All Arcade contracts are meant to be deployed by the protocol team, except for vaults, which are deployed by users using the `VaultFactory` contract. However, in the fixture deployment function, some contracts are deployed from the `borrower` account instead of the `admin` account.

Some examples are shown in figure 6.1; however, there are other instances in which contracts are not deployed from the `admin` account.

```
    const signers: SignerWithAddress[] = await ethers.getSigners();
    const [borrower, lender, admin] = signers;

    const whitelist = <CallWhitelist>await deploy("CallWhitelist", signers[0], []);
    const vaultTemplate = <AssetVault>await deploy("AssetVault", signers[0], []);
    const feeController = <FeeController>await deploy("FeeController", admin, []);
    const descriptor = <BaseURIDescriptor>await deploy("BaseURIDescriptor", signers[0], [BASE_URI])
    const vaultFactory = <VaultFactory>await deploy("VaultFactory", signers[0], [vaultTemplate.address, whitelist.address, feeController.address, descriptor.address]);
```

Figure 6.1: A snippet of the tests in `arcade-protocol/test/Integration.ts`

Exploit Scenario

Alice, a developer on the Arcade team, adds a new permissioned feature to the protocol. She adds the relevant integration tests for her feature, and all tests pass. However, because the deployer for the test contracts was not the `admin` account, those tests should have failed, and the contracts are deployed to the network with a bug.

Recommendations

Short term, correct all of the instances of incorrect deployers for the contracts in the integration tests file.

Long term, add additional test cases to ensure that the account permissions in all deployed contracts are correct.

7. Risk of out-of-gas revert due to use of transfer() in claimFees

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ARCADE-7
Target: `contracts/vault/VaultFactory.sol`	

Description

The `VaultFactory.claimFees` function uses the low-level `transfer()` operation to move the collected ETH fees to another arbitrary address. The `transfer()` operation sends only 2,300 units of gas with this operation. As a result, if the recipient is a contract with logic inside the `receive()` function, which would use extra gas, the operation will probably (depending on the gas cost) fail due to an out-of-gas revert.

```
function claimFees(address to) external onlyRole(FEE_CLAIMER_ROLE) {
    uint256 balance = address(this).balance;
    payable(to).transfer(balance);

    emit ClaimFees(to, balance);
}
```

Figure 7.1: The `claimFees` function in `arcade-protocol/contracts/vault/VaultFactory.sol`

The Arcade team has explained that the recipient will be a treasury contract with no logic inside the `receive()` function, meaning the current use of `transfer()` will not pose any problems. However, if at some point the recipient does contain logic inside the `receive()` function, then `claimFees` will likely revert and the contract will not be able to claim the funds. Note, however, that the fees could be claimed by another address (i.e., the fees will not be stuck).

The `withdrawETH` function in the `AssetVault` contract uses `Address.sendValue` instead of `transfer()`.

```
function withdrawETH(address to) external override onlyOwner
onlyWithdrawEnabled nonReentrant {
    // perform transfer
    uint256 balance = address(this).balance;
    payable(to).sendValue(balance);
    emit WithdrawETH(msg.sender, to, balance);
}
```

Figure 7.2: The `withdrawETH` function in [`arcade-protocol/contracts/vault/AssetVault.sol`]

`Address.sendValue` internally uses the `call()` operation, passing along all of the remaining gas, so this function could be a good candidate to replace use of `transfer()` in `claimFees`. However, doing so could introduce other risks like reentrancy attacks. Note that neither the `withdrawETH` function nor the `claimFees` function is currently at risk of reentrancy attacks.

Exploit Scenario

Alice, a developer on the Arcade team, deploys a new treasury contract that contains an updated `receive()` function that also writes the received ETH amount into a storage array in the treasury contract. Bob, whose account has the `FEE_CLAIMER_ROLE` role in the `VaultFactory` contract, calls `claimFees` with the newly deployed treasury contract as the recipient. The transaction fails because the write to storage exceeds the passed along 2,300 units of gas.

Recommendations

Short term, consider replacing the `claimFees` function's use of `transfer()` with `Address.sendValue`; weigh the risk of possibly introducing vulnerabilities like reentrancy attacks against the benefit of being able to one day add logic in the fee recipient's `receive()` function. If the decision is to have `claimFees` continue to use `transfer()`, update the NatSpec comments for the function so that readers will be aware of the 2,300 gas limit on the fee recipient.

Long term, when deciding between using the low-level `transfer()` and `call()` operations, consider how malicious smart contracts may be able to exploit the lack of limits on the gas available in the recipient function. Additionally, consider the likelihood that the recipient will be a smart wallet or multisig (or other smart contract) with logic inside the `receive()` function, as the 2,300 gas from `transfer()` might not be sufficient for those recipients.

8. Risk of lost funds due to lack of zero-address check in functions

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARCADE-8
Target: `contracts/vault/VaultFactory.sol`, `contracts/RepaymentController.sol`, `contracts/LoanCore.sol`	

Description

The `VaultFactory.claimFees` (figure 8.1), `RepaymentController.redeemNote` (figure 8.2), `LoanCore.withdraw`, and `LoanCore.withdrawProtocolFees` functions are all missing a check to ensure that the `to` argument does not equal the zero address. As a result, these functions could transfer funds to the zero address.

```
function claimFees(address to) external onlyRole(FEE_CLAIMER_ROLE) {
    uint256 balance = address(this).balance;
    payable(to).transfer(balance);

    emit ClaimFees(to, balance);
}
```

Figure 8.1: The `claimFees` function in `arcade-protocol/contracts/vault/VaultFactory.sol`

```
function redeemNote(uint256 loanId, address to) external override {
    LoanLibrary.LoanData memory data = loanCore.getLoan(loanId);
    (, uint256 amountOwed) = loanCore.getNoteReceipt(loanId);

    if (data.state != LoanLibrary.LoanState.Repaid) revert
RC_InvalidState(data.state);
    address lender = lenderNote.ownerOf(loanId);
    if (lender != msg.sender) revert RC_OnlyLender(lender, msg.sender);

    uint256 redeemFee = (amountOwed * feeController.get(FL_09)) /
BASIS_POINTS_DENOMINATOR;

    loanCore.redeemNote(loanId, redeemFee, to);
}
```

Figure 8.2: The `redeemNote` function in `arcade-protocol/contracts/RepaymentController.sol`

Exploit Scenario

A script that is used to periodically withdraw the protocol fees (calling `LoanCore.withdrawProtocolFees`) is updated. Due to a mistake, the `to` argument is left uninitialized. The script is executed, and the `to` argument defaults to the zero address, causing `withdrawProtocolFees` to transfer the protocol fees to the zero address.

Recommendations

Short term, add a check to verify that `to` does not equal the zero address to the following functions:

- `VaultFactory.claimFees`
- `RepaymentController.redeemNote`
- `LoanCore.withdraw`
- `LoanCore.withdrawProtocolFees`

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.

9. The maximum value for FL_09 is not set by FeeController

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARCADE-9
Target: `contracts/FeeController.sol`	

Description

The `FeeController` constructor initializes all of the maximum values for the fees defined in the `FeeLookups` contract except for `FL_09` (`LENDER_REDEEM_FEE`). Because the maximum value is not set, it is possible to set any amount, with no upper bound, for that particular fee.

The lender's redeem fee is used in `RepaymentController`'s `redeemNote` function to calculate the fee paid by the lender to the protocol in order to receive their funds back. If the protocol team accidentally sets the fee to 100%, all of the users' funds to be redeemed would instead be used to pay the protocol.

```
constructor() {
    /// @dev Vault mint fee - gross
    maxFees[FL_01] = 1 ether;

    /// @dev Origination fees - bps
    maxFees[FL_02] = 10_00;
    maxFees[FL_03] = 10_00;

    /// @dev Rollover fees - bps
    maxFees[FL_04] = 20_00;
    maxFees[FL_05] = 20_00;

    /// @dev Loan closure fees - bps
    maxFees[FL_06] = 10_00;
    maxFees[FL_07] = 50_00;
    maxFees[FL_08] = 10_00;
}
```

Figure 9.1: The constructor in `arcade-protocol/contracts/FeeController.sol`

```
function redeemNote(uint256 loanId, address to) external override {
    LoanLibrary.LoanData memory data = loanCore.getLoan(loanId);
    (, uint256 amountOwed) = loanCore.getNoteReceipt(loanId);

    if (data.state != LoanLibrary.LoanState.Repaid) revert
RC_InvalidState(data.state);
    address lender = lenderNote.ownerOf(loanId);
```

```
        if (lender != msg.sender) revert RC_OnlyLender(lender, msg.sender);

        uint256 redeemFee = (amountOwed * feeController.get(FL_09)) /
BASIS_POINTS_DENOMINATOR;

        loanCore.redeemNote(loanId, redeemFee, to);
    }
```

Figure 9.2: The `redeemNote` function in `arcade-protocol/contracts/RepaymentController.sol`

Exploit Scenario

Charlie, a member of the Arcade protocol team, has access to the privileged account that can change the protocol fees. He wants to set `LENDERS_REDEEM_FEE` to 5%, but he accidentally types a 0 and sets it to 50%. Users can now lose half of their funds to the new protocol fee, causing distress and lack of trust in the team.

Recommendations

Short term, set a maximum boundary for the `FL_09` fee in `FeeController`'s constructor.

Long term, improve the test suite to ensure that all fee-changing functions test for out-of-bounds values for all fees, not just `FL_02`.

10. Fees can be changed while a loan is active

Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-ARCADE-10
Target: `contracts/FeeController.sol`	

Description

All fees in the protocol are calculated using the current fees, as informed by the `FeeController` contract. However, fees can be changed by the team at any time, so the effective rollover and closure fees that the users will pay can change once their loans are already initialized; therefore, these fees are impossible to know in advance.

For example, in the code shown in figure 10.1, the `LENDER_INTEREST_FEE` and `LENDER_PRINCIPAL_FEE` values are read when a loan is about to be repaid, but these values can be different from the values the user agreed to when the loan was initialized. The same can happen in `OriginationController` and other functions in `RepaymentController`.

```
    function _prepareRepay(uint256 loanId) internal view returns (uint256 amountFromBorrower, uint256 amountToLender) {
        LoanLibrary.LoanData memory data = loanCore.getLoan(loanId);
        if (data.state == LoanLibrary.LoanState.DUMMY_DO_NOT_USE) revert RC_CannotDereference(loanId);
        if (data.state != LoanLibrary.LoanState.Active) revert RC_InvalidState(data.state);

        LoanLibrary.LoanTerms memory terms = data.terms;

        uint256 interest = getInterestAmount(terms.principal, terms.proratedInterestRate);

        uint256 interestFee = (interest * feeController.get(FL_07)) / BASIS_POINTS_DENOMINATOR;
        uint256 principalFee = (terms.principal * feeController.get(FL_08)) / BASIS_POINTS_DENOMINATOR;

        amountFromBorrower = terms.principal + interest;
        amountToLender = amountFromBorrower - interestFee - principalFee;
    }
```

Figure 10.1: The `_prepareRepay` function in `arcade-protocol/contracts/RepaymentController.sol`

Exploit Scenario

Lucy, the lender, and Bob, the borrower, agree on the current loan conditions and fees at a certain point in time. Some weeks later, when the time comes to repay the loan, they learn that the protocol team decided to change the fees while their loan was active. Lucy's earnings are now different from what she expected.

Recommendations

Short term, consider storing (for example, in the `LoanTerms` structure) the fee values that both counterparties agree on when a loan is initialized, and use those local values for the full lifetime of the loan.

Long term, document all of the conditions that are agreed on by the counterparties and that should be constant during the lifetime of the loan, and make sure they are preserved. Add a specific integration or fuzzing test for these conditions.

11. Asset vault nesting can lead to loss of assets

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ARCADE-11
Target: `contracts/vault/VaultFactory.sol`, `contracts/vault/AssetVault.sol`	

Description

Allowing asset vaults to be nested (e.g., vault A is owned by vault B, and vault B is owned by vault X, etc.) could result in a situation in which multiple asset vaults own each other. This would result in a deadlock preventing assets in the affected asset vaults from ever being withdrawn again.

Asset vaults are designed to hold different types of assets, including ERC-721 tokens. The ownership of an asset vault is tracked by an accompanying ERC-721 token that is minted (figure 11.1) when the asset vault is deployed through the `VaultFactory` contract.

```
    function initializeBundle(address to) external payable override returns (uint256) {
        uint256 mintFee = feeController.get(FL_01);

        if (msg.value < mintFee) revert VF_InsufficientMintFee(msg.value, mintFee);

        address vault = _create();

        _mint(to, uint256(uint160(vault)));

        if (msg.value > mintFee) payable(msg.sender).transfer(msg.value - mintFee);

        emit VaultCreated(vault, to);
        return uint256(uint160(vault));
    }
```

Figure 11.1: The `initializeBundle` function in `arcade-protocol/contracts/vault/VaultFactory.sol`

To add an ERC-721 asset to an asset vault, it needs to be transferred to the asset vault's address. Because the ownership of an asset vault is tracked by an ERC-721 token, it is possible to transfer the ownership of an asset vault to another asset vault by simply transferring the ERC-721 token representing vault ownership. To withdraw ERC-721 tokens from an asset vault, the owner (the holder of the asset vault's ERC-721 token) needs to enable withdrawals (using the `enableWithdraw` function) and then call the `withdrawERC721` (or `withdrawBatch`) function.

```
    function enableWithdraw() external override onlyOwner onlyWithdrawDisabled {
        withdrawEnabled = true;
        emit WithdrawEnabled(msg.sender);
    }
```

Figure 11.2: The `enableWithdraw` function in `arcade-protocol/contracts/vault/AssetVault.sol`

```
    function withdrawERC721(
        address token,
        uint256 tokenId,
        address to
    ) external override onlyOwner onlyWithdrawEnabled {
        _withdrawERC721(token, tokenId, to);
    }
```

Figure 11.3: The `withdrawERC721` function in `arcade-protocol/contracts/vault/AssetVault.sol`

Only the owner of an asset vault can enable and perform withdrawals. Therefore, if two (or more) vaults own each other, it would be impossible for a user to enable or perform withdrawals on the affected vaults, permanently locking all assets (ERC-721, ERC-1155, ERC-20, ETH) within them.

The severity of the issue depends on the UI, which was out of scope for this review. If the UI does not prevent vaults from owning each other, the severity of this issue is higher. In terms of likelihood, this issue would require a user to make a mistake (although a mistake that is far more likely than the transfer of tokens to a random address) and would require the UI to fail to detect and prevent or warn the user from making such a mistake. We therefore rated the difficulty of this issue as high.

Exploit Scenario

Alice decides to borrow USDC by putting up some of her NFTs as collateral:

1. Alice uses the UI to create an asset vault (vault A) and transfers five of her CryptoPunks to the asset vault.

2. The UI shows that Alice has another existing vault (vault X), which contains two Bored Apes. She wants to use these two vaults together to borrow a higher amount of USDC. She clicks on vault A and selects the “Add Asset” option.

3. The UI shows a list of assets that Alice owns, including the ERC-721 token that represents ownership of vault X. Alice clicks on “Add”, the transaction succeeds, and the vault X NFT is transferred to vault A. Vault X is now owned by vault A.

4. Alice decides to add another Bored Ape NFT that she owns to vault X. She opens the vault X page and clicks on “Add Assets”, and the list of assets that she can add shows the ERC-721 token that represents ownership of vault A.

5. Alice is confused and wonders if adding vault X to vault A worked (step 3). She decides to add vault A to vault X instead. The transaction succeeds, and now vault A owns vault X and vice versa. Alice is now unable to withdraw any of the assets from either vault.

Recommendations

Short term, take one of the following actions:

- Disallow the nesting of asset vaults. That is, prevent users from being able to transfer ownership of an asset vault to another asset vault. This would prevent the issue altogether.

- If allowing asset vaults to be nested is a desired feature, update the UI to prevent two or more asset vaults from owning each other (if it does not already do so). Also, update the documentation so that other integrating smart contract protocols are aware of the issue.

Long term, when dealing with the nesting of assets, consider edge cases and write extensive tests that ensure these edge cases are handled correctly and that users do not lose access to their assets. Other than unit tests, we recommend writing invariants and testing them using property-based testing with Echidna.

12. Risk of locked assets due to use of _mint instead of _safeMint

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ARCADE-12
Target: `contracts/vault/VaultFactory.sol`, `contracts/PromissoryNote.sol`	

Description

The asset vault and promissory note ERC-721 tokens are minted via the `_mint` function rather than the `_safeMint` function. The `_safeMint` function includes a necessary safety check that validates a recipient contract's ability to receive and handle ERC-721 tokens. Without this safeguard, tokens can inadvertently be sent to an incompatible contract, causing them, and any assets they hold, to become irretrievable.

```
    function initializeBundle(address to) external payable override returns (uint256) {
        uint256 mintFee = feeController.get(FL_01);

        if (msg.value < mintFee) revert VF_InsufficientMintFee(msg.value, mintFee);

        address vault = _create();

        _mint(to, uint256(uint160(vault)));

        if (msg.value > mintFee) payable(msg.sender).transfer(msg.value - mintFee);

        emit VaultCreated(vault, to);
        return uint256(uint160(vault));
    }
```

Figure 12.1: The `initializeBundle` function in `arcade-protocol/contracts/vault/VaultFactory.sol`

```
    function mint(address to, uint256 loanId) external override returns (uint256) {
        if (!hasRole(MINT_BURN_ROLE, msg.sender)) revert PN_MintingRole(msg.sender);
        _mint(to, loanId);

        return loanId;
    }
```

Figure 12.2: The `mint` function in `arcade-protocol/contracts/PromissoryNote.sol`

The _safeMint function's built-in safety check ensures that the recipient contract has the necessary ERC721Receiver implementation, verifying the contract's ability to receive and manage ERC-721 tokens.

```
function _safeMint(
   address to,
   uint256 tokenId,
   bytes memory _data
) internal virtual {
   _mint(to, tokenId);
   require(
       _checkOnERC721Received(address(0), to, tokenId, _data),
       "ERC721: transfer to non ERC721Receiver implementer"
   );
}
```

Figure 12.3: The _safeMint function in *openzeppelin-contracts/contracts/token/ERC721/ERC721.sol*

The _checkOnERC721Received method invokes the onERC721Received method on the receiving contract, expecting a return value containing the bytes4 selector of the onERC721Received method. A successful pass of this check implies that the contract is indeed capable of receiving and processing ERC-721 tokens.

The _safeMint function does allow for reentrancy through the calling of _checkOnERC721Received on the receiver of the token. However, based on the order of operations in the affected functions in Arcade (figures 12.1 and 12.2), this poses no risk.

Exploit Scenario

Alice initializes a new asset vault by invoking the initializeBundle function of the VaultFactory contract, passing in her smart contract wallet address as the to argument. She transfers her valuable CryptoPunks NFT, intended to be used for collateral, to the newly created asset vault. However, she later discovers that her smart contract wallet lacks support for ERC-721 tokens. As a result, both her asset vault token and the CryptoPunks NFT become irretrievable, stuck within her smart wallet contract due to the absence of a mechanism to handle ERC-721 tokens.

Recommendations

Short term, use the _safeMint function instead of _mint in the PromissoryNote and VaultFactory contracts. The _safeMint function includes vital checks that ensure the recipient is equipped to handle ERC-721 tokens, thus mitigating the risk that NFTs could become frozen.

Long term, enhance the unit testing suite. These tests should encompass more negative paths and potential edge cases, which will help uncover any hidden vulnerabilities or bugs like this one. Additionally, it is critical to test user-provided inputs extensively, covering a broad spectrum of potential scenarios. This rigorous testing will contribute to building a more secure, robust, and reliable system.

13. Borrowers cannot realize full loan value without risking default

Severity: **Medium**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-ARCADE-13
Target: `contracts/LoanCore.sol`	

Description

To fully capitalize on their loans, borrowers need to retain their loaned assets and the owed interest for the entire term of their loans. However, if a borrower waits until the loan's maturity date to repay it, they become immediately vulnerable to liquidation of their collateral by the lender.

As soon as the `block.timestamp` value exceeds the `dueDate` value, a lender can invoke the `claim` function to liquidate the borrower's collateral.

```
// First check if the call is being made after the due date.
uint256 dueDate = data.startDate + data.terms.durationSecs;
if (dueDate >= block.timestamp) revert LC_NotExpired(dueDate);
```

Figure 13.1: A snippet of the `claim` function in `arcade-protocol/contracts/LoanCore.sol`

Owing to the inherent nature of the blockchain, achieving precise synchronization between the `block.timestamp` and the `dueDate` is practically impossible. Moreover, repaying a loan before the `dueDate` would result in a loss of some of the loan's inherent value because the protocol's interest assessment design does not refund any part of the interest for early repayment.

In a scenario in which `block.timestamp` is greater than `dueDate`, a lender can preempt a borrower's loan repayment attempt, invoke the `claim` function, and liquidate the borrower's collateral. Frequently, collateral will be worth more than the loaned assets, giving lenders an incentive to do this.

Given the protocol's interest assessment design, the Arcade team should implement a grace period following the maturity date where no additional interest is expected to be assessed beyond the period agreed to in the loan terms. This buffer would give the borrower an opportunity to fully capitalize on the term of their loan without the risk of defaulting and losing their collateral.

Exploit Scenario

Alice, a borrower, takes out a loan from Eve using Arcade's NFT lending protocol. Alice deposits her rare CryptoPunk as collateral, which is more valuable than the assets loaned to her, so that her position is over-collateralized. Alice plans to hold on to the lent assets for the entire duration of the loan period in order to maximize her benefit-to-cost ratio. Eve, the lender, is monitoring the blockchain for the moment when the `block.timestamp` is greater than or equal to the `dueDate` so that she can call the `claim` function and liquidate Alice's CryptoPunk. As soon as the loan term is up, Alice submits a transaction to the `repay` function, and Eve front-runs that transaction with her own call to the `claim` function. As a result, Eve is able to liquidate Alice's CryptoPunk collateral.

Recommendations

Short term, introduce a grace period after the loan's maturity date during which the lender cannot invoke the `claim` function. This buffer would give the borrower sufficient time to repay the loan without the risk of immediate collateral liquidation.

Long term, revise the protocol's interest assessment design to allow a portion of the interest to be refunded in cases of early repayment. This change could reduce the incentive for borrowers to delay repayment until the last possible moment. Additionally, provide better education for borrowers on how the lending protocol works, particularly around critical dates and actions, and improve communication channels for borrowers to raise concerns or seek clarification.

14. itemPredicates encoded incorrectly according to EIP-712

Severity: **Low**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-ARCADE-14
Target: `contracts/OriginationController.sol`	

Description

The `itemPredicates` parameter is not encoded correctly, so the signer cannot see the verifier address when signing. The verifier address receives each batch of listed assets to check them for correctness and existence, which is vital to ensuring the security and integrity of the lending transaction.

According to EIP-712, structured data should be hashed in conjunction with its `typeHash`. The following is the `hashStruct` function as defined in EIP-712:

$$\mathtt{hashStruct(s : \mathbb{S}) = keccak256(typeHash \parallel encodeData(s))}$$
$$\mathtt{where\ typeHash = keccak256(encodeType(typeOf(s)))}$$

In the protocol, the `recoverItemsSignature` function hashes an array of `Predicate[]` structs that are passed in as the `itemPredicates` argument. The function encodes and hashes the array without adding the `Predicate typeHash` to each member of the array. The hashed output of that operation is then included in the _ITEMS_TYPEHASH variable as a `bytes32` type, referred to as `itemsHash`.

```
(bytes32 sighash, address externalSigner) = recoverItemsSignature(
    loanTerms,
    sig,
    nonce,
    neededSide,
    keccak256(abi.encode(itemPredicates))
);
```

Figure 14.1: A snippet of the `initializeLoanWithItems` function in `arcade-protocol/contracts/OriginationController.sol`

```
bytes32 private constant _ITEMS_TYPEHASH =
    keccak256(
        // solhint-disable max-line-length
        "LoanTermsWithItems(uint32 durationSecs,uint32 deadline,uint160
proratedInterestRate,uint256 principal,address collateralAddress,bytes32
itemsHash,address payableCurrency,bytes32 affiliateCode,uint160 nonce,uint8 side)"
    );
```

Figure 14.2: The `_ITEMS_TYPEHASH` variable in `arcade-protocol/contracts/OriginationController.sol`

However, this method of encoding an array of structs is not consistent with the EIP-712 guidelines, which stipulates the following:

> *"The array values are encoded as the keccak256 hash of the concatenated encodeData of their contents (i.e., the encoding of SomeType[5] is identical to that of a struct containing five members of type SomeType).*
>
> *The struct values are encoded recursively as hashStruct(value). This is undefined for cyclical data."*

Therefore, the protocol should iterate over the `itemPredicates` array, encoding each `Predicate` instance separately with its respective `typeHash`.

Exploit Scenario

Alice creates a loan offering that takes CryptoPunks as collateral. She submits the loan terms to the Arcade protocol. Bob, a CryptoPunk holder, navigates the Arcade UI to accept Alice's loan terms. An EIP-712 signature request appears in MetaMask for Bob to sign. Bob cannot validate whether the message he is signing uses the CryptoPunk verifier contract because that information is not included in the hash.

Recommendations

Short term, adjust the encoding of `itemPredicates` to comply with EIP-712 standards. Have the code iterate through the `itemPredicates` array and encode each `Predicate` instance separately with its associated `typeHash`. Additionally, refactor the `_ITEMS_TYPEHASH` variable so that the `Predicate typeHash` definition is appended to it and replace the `bytes32 itemsHash` parameter with `Predicate[] items`. This revision will allow the signer to see the verifier address of the message they are signing, ensuring the validity of each batch of items, in addition to complying with the EIP-712 standard.

Long term, strictly adhere to established Ethereum protocols such as EIP-712. These standards exist to ensure interoperability, security, and predictable behavior in the Ethereum ecosystem. Violating these norms can lead to unforeseen security vulnerabilities.

15. The fee values can distort the incentives for the borrowers and lenders	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ARCADE-15
Target: `contracts/FeeController.sol`	

Description

Arcade V3 contains nine fee settings. Six of these fees are to be paid by the lender, two are to be paid by the borrower, and the remaining fee is to be paid by the borrower if they decide to mint a new vault for their collateral. Depending on the values of these settings, the incentives can change for both loan counterparties.

For example, to create a new loan, both the borrower and lender have to pay origination fees, and eventually, the loan must be rolled over, repaid, or defaulted. In the first case, both the new lender and borrower pay rollover fees; note that the original lender pays no fees at all for closing the loan. In the second case, the lender pays interest fees and principal fees on closing the loan. Finally, if the loan is defaulted, the lender pays a default fee to liquidate the collateral.

The various fees paid based on the outcome of the loan can result in an interesting incentive game for investors in the protocol, depending on the actual values of the fee settings. If the lender rollover fee is cheaper than the origination fee, investors may be incentivized to roll over existing loans instead of creating new ones, benefiting the original lenders by saving them the closing fees, and harming the borrowers by indirectly raising the interest rates to compensate. Similarly, if the lender rollover fees are higher than the closing fees, lenders will be less incentivized to rollover loans.

In summary, having such fine control over possible fee settings introduces hard-to-predict incentives scenarios that can scare users away or cause users who do not account for fees to inadvertently lose profits.

Recommendations

Short term, clearly inform borrowers and lenders of all of the existing fees and their current values at the moment a loan is opened, as well as the various possible outcomes, including the expected net profits if the loan is repaid, rolled over, defaulted, or redeemed.

Long term, add interactive ways for users to calculate their expected profits, such as a loan simulator.

16. Malicious borrowers can use forceRepay to grief lenders	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ARCADE-16
Target: `contracts/RepaymentController.sol`	

Description

A malicious borrower can grief a lender by calling the `forceRepay` function instead of the `repay` function; doing so would allow the borrower to pay less in gas fees and require the lender to perform a separate transaction to retrieve their funds (using the `redeemNote` function) and to pay a redeem fee.

At any time after the loan is set and before the lender claims the collateral if the loan is past its due date, the borrower has to pay their full debt back in order to recover their assets. For doing so, there are two functions in `RepaymentController`: `repay` and `forceRepay`. The difference between them is that the latter transfers the tokens to the `LoanCore` contract instead of directly to the lender. It is meant to allow the borrower to pay their obligations when the lender cannot receive tokens for any reason.

For the lender to get their tokens back in this scenario, they must call the `redeemNote` function in `RepaymentController`, which in turn calls `LoanCore.redeemNote`, which transfers the tokens to an address set by the lender in the call.

Because the borrower is free to decide which function to call to repay their debt, they can arbitrarily decide to do so via `forceRepay`, obligating the lender to send a transaction (with its associated gas fees) to recover their tokens. Additionally, depending on the configuration of the protocol, it is possible that the lender has to pay an additional fee (`LENDER_REDEEM_FEE`) to get back their own tokens, cutting their profits with no chance to opt out.

```
    function redeemNote(uint256 loanId, address to) external override {
        LoanLibrary.LoanData memory data = loanCore.getLoan(loanId);
        (, uint256 amountOwed) = loanCore.getNoteReceipt(loanId);

        if (data.state != LoanLibrary.LoanState.Repaid) revert
RC_InvalidState(data.state);
        address lender = lenderNote.ownerOf(loanId);
        if (lender != msg.sender) revert RC_OnlyLender(lender, msg.sender);

        uint256 redeemFee = (amountOwed * feeController.get(FL_09)) /
BASIS_POINTS_DENOMINATOR;
```

```

        loanCore.redeemNote(loanId, redeemFee, to);
    }
```

Figure 16.1: The `redeemNote` function in `arcade-protocol/contracts/RepaymentController.sol`

Note that, from the perspective of the borrower, it is actually cheaper to call `forceRepay` than `repay` because of the gas saved by not transferring the tokens to the lender and not burning one of the promissory notes.

Exploit Scenario

Bob has to pay back his loan, and he decides to do so via `forceRepay` to save gas in the transaction. Lucy, the lender, wants her tokens back. She is now forced to call `redeemNote` to get them. In this transaction, she lost the gas fees that the borrower would have paid to send the tokens directly to her, and she has to pay an additional fee (`LENDER_REDEEMER_FEE`), causing her to receive less value from the loan than she originally expected.

Recommendations

Short term, remove the incentive (the lower gas cost) for the borrower to call `forceRepay` instead of `repay`. Consider taking one of the following actions:

- Force the lender to always pull their funds using the `redeemNote` function. This can be achieved by removing the `repay` function and requiring the borrower to call `forceRepay`.

- Remove the `forceRepay` function and modify the `repay` function so that it transfers the funds to the lender in a `try/catch` statement and creates a redeem note (which the lender can exchange for their funds using the `redeemNote` function) only if that transfer fails.

Long term, when designing a smart contract protocol, always consider the incentives for each party to perform actions in the protocol, and avoid making an actor pay for the mistakes or maliciousness of others. By thoroughly documenting the incentives structure, flaws can be spotted and mitigated before the protocol goes live.

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
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
Transaction Ordering Risks	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.

Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code 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.

- **Make fee names more explicit and add getters for each fee.** The names used for the various fee values (e.g., `FL_0x`) do not clearly describe the fees. Readers of the code will have to navigate between the current contract and the `FeeLookups` contract to determine the type of each fee. Additionally, adding custom getter functions to the `FeeController` contract (e.g., `getLenderFee()`) can simplify the system by allowing `FeeLookups` to be removed from the inheritance chain of the `OriginationController`, `RepaymentController`, and `VaultFactory` contracts.

- **Ensure that verifiers follow the provided documentation.** According to the `_runPredicatesCheck()` NatSpec documentation, the function reverts if a verifier returns `false`. However, some of the implemented verifiers can return `true` or `false` or instead revert. Even though this discrepancy has no direct impact on the system, an external entity interacting with Arcade may be confused by getting an `IV_xxx` error when they expected an `OC_PredicateFailed` error.

- **Ensure that structure names are unique throughout the system.** There are two structures named `SignatureItem`: one in `ArtBlocksVerifier` and the other in `ItemsVerifier`. Even though they are defined in different namespaces, it can be confusing to identify them because they have different members. Moreover, in the provided test suite, they are referred to as `ArtBlocksItem` and `SignatureItem`, respectively, making it more confusing for readers.

- **Use automatically generated getters for public variables.** `OriginationController` defines the mappings `allowedVerifiers`, `allowedCurrencies`, and `allowedCollateral` as public. The Solidity compiler automatically adds getters for these variables, but manual getters were added by the team (`isAllowedVerifier`, `isAllowedCurrency`, and `isAllowedCollateral`) with no additional functionality from the default getters.

- **Ensure that comments in the code reflect the code's intended behavior.** Here are two examples of off-by-one comments in `OriginationController`: `OriginationController.sol#L669` and `OriginationController.sol#L673`.

- **Ensure that contract names match their filenames.** The `verifiers/ItemsVerifier.sol` file contains the `ArcadeItemsVerifier` contract.

- **Remove variables that are never used or are used only once.** For example, `id` in `ArcadeItemsVerifier.verifyPredicates()` is used only once in the function body.
- **Avoid redefining constants.** `FL_01` in `FeeLookups` was redefined in `VaultFactory`.
- **Remove unneeded or unreachable code.** For example, the `else` condition in the `ArcadeItemsVerifier.verifyPredicates` function can never be reached because `abi.decode` will revert for incorrect `CollateralType` enum values.

D. Risks with Approving NFTs for Use as Collateral

The Arcade protocol aims to whitelist NFT contracts to be used as collateral to back loans. These NFTs could introduce problems that could allow attackers to steal funds or otherwise impede the correct functioning of the system. We recommend that the Arcade team exercise caution in approving NFTs for use as collateral to ensure that the system keeps working correctly and that no user loses access to funds.

Follow these guidelines when considering which NFTs to approve:

- **Tokens should never be upgradeable.** Upgradeable ERC-721 tokens can introduce substantial risks when used as collateral in an NFT lending protocol. Therefore, smart contract developers should either prevent upgradeable tokens from being used as collateral or implement robust safeguards against the associated risks. Some of these risks include the following:
    - **Unpredictable token logic changes:** Because the contract owner or a designated admin can alter the logic of an upgradeable token, the token's behavior could change unpredictably during the loan period. This could affect the value of the collateral, render it worthless, or prevent its return.
    - **Centralization and minimized trust:** Upgradeable contracts introduce an element of centralization, as the power to upgrade the contract typically lies with a specific address or addresses. This could be a risk in a decentralized environment, where the ethos is to minimize trust in individual parties. The contract's owner could, maliciously or unintentionally, make an upgrade that jeopardizes the token's role as collateral.
    - **Complexity and potential bugs:** Upgradeable contracts are more complex than their non-upgradeable counterparts. This added complexity increases the risk of bugs and vulnerabilities, which could be exploited to the detriment of Arcade's lending protocol and its users.
- **Tokens should not have a self-destruct capability.** The self-destruct function allows a contract to be destroyed by its owner, which essentially removes the contract's bytecode from the Ethereum blockchain, making it nonfunctional. Here are some of the risks of using self-destructible tokens as collateral:
    - **Total loss of collateral:** If an ERC-721 token used as collateral has a self-destruct function and that function is invoked during the loan's lifecycle, the token will be rendered worthless. The borrower could default on their loan and the lender would not be able to claim the collateral, leading to a complete loss.

    - **Damage to the integrity of Arcade's lending protocol:** Such tokens can undermine the integrity of the lending protocol. Lenders will be unwilling to participate if they believe that the collateral could self-destruct, making it harder for the protocol to attract and retain users.
    - **Lack of recourse:** In traditional finance, there are legal protections to prevent the destruction of assets used as collateral. In contrast, in the blockchain world, there is no way to recover a contract once it has self-destructed, making it a significant risk for lenders.
- **Tokens should not be pausable.** A "pause" function, when present in a smart contract, allows certain privileged accounts such as contract owners and administrators to stop specific activities such as token transfers for a period of time. Although this functionality can be useful for halting activities in case of a detected vulnerability or bug, it can pose significant risks to an NFT lending protocol, such as the following:
    - **Possible prevention of repayment and collateral retrieval:** If a token used as collateral is paused, that token cannot be transferred. This means that a borrower could not repay their loan and retrieve their collateral, and similarly, a lender could not claim the collateral if the loan defaults.
    - **Market manipulation:** In a worst-case scenario, a malicious token owner could strategically pause and unpause a token, disrupting the market and possibly manipulating the token's value.
- **Tokens should not be burnable by some authorized third-party.** Tokens that can be burned by a third-party or token admin should not be permitted as collateral in an NFT lending protocol. "Burning" is a process by which tokens are permanently removed from circulation, thereby reducing the total supply of tokens. Although this feature can be useful in certain contexts, it can introduce the following risks when used in an NFT lending protocol:
    - **Total loss of collateral:** If the token used as collateral can be burned by an admin or third party, it could be burned during the duration of the loan, which would leave the lender unprotected in the case of a default.
    - **Loan-to-value manipulation:** Those with the ability to burn tokens could engage in manipulative behaviors that disrupt the loan-to-value ratio, such as artificially influencing a token's scarcity and thereby its market value, thus leading to over-collateralization or under-collateralization.
- **Tokens should not hold or have access to other assets.** Tokens can be structured to hold or interact with other assets on the blockchain. An ERC-721 token can be a

"wrapper" for specific ERC-20 tokens, generate yields from a DeFi protocol, or represent in-game characters with their own assets. Using them as collateral in lending protocol comes with the following risks:

  - **Value fluctuation:** If the token holds or has access to other assets, its value can change during the loan period if the value of the underlying assets changes. If a collateral changes value, it may no longer cover the value of the loan, creating significant risk for the lender.

  - **Asset removal or addition:** If the token allows assets to be added to or removed from it, the value of the collateral could be altered during the life cycle of the loan. A borrower or a third party could remove assets from the collateral, decreasing the collateral's value; the lender and the protocol would have no means to prevent this.

  - **Valuation complexity:** Valuing tokens that hold other assets is more complex than valuing simpler tokens. Some assets are interest bearing, some undergo rebasing, and most are traded on public markets. The complexities involved in accurately determining the value of such tokens introduces additional risks and complexities for the lender, the borrower, and the protocol in general.

- **Gaming tokens with alterable intrinsic value should be avoided.** Tokens are often used to represent unique digital assets in gaming environments, such as characters, equipment, and virtual real estate. Using them as collateral in a lending protocol carries some risks, such as the following:

  - **Developer control:** Game developers often maintain a degree of control over in-game assets, which may include the ability to create, modify, or destroy assets. If a game developer decides to flood the market with copies of a previously rare asset, or alter its capabilities within the game, the value of the collateral could be significantly affected.

  - **In-game rules and actions:** In-game actions by other players or changes to in-game rules can influence the value of the token. For instance, if the game involves player competition, other players' actions could diminish the value of the collateral token.

E. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

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

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

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

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.

- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.

- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

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

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.

- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

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

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

Risks of ERC20 Extensions

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

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

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

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

ERC721 Tokens

ERC721 Conformity Checks

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

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

Common Risks of the ERC721 Standard

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

- ❏ **The onERC721`Received` callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals

F. Mutation Testing

The goal of mutation testing is to gain insight into a codebase's test coverage. Mutation tests go line-by-line through the target file, mutate the given line in some way, run tests, and flag changes that do not trigger test failures. Depending on the complexity of the logic in any given line and the tool used to mutate, mutation tests could test upwards of 50 mutants per line of source code. Mutation testing is a slow process, but by highlighting areas of the code with incomplete test coverage, it allows auditors to focus their manual review on the parts of the code that are most likely to contain latent bugs.

In this section, we provide information on available mutation testing tools that could be used in the Arcade V3 codebase, and we describe the mutation testing campaign that we conducted during this audit.

The following are available mutation testing tools:

- Universalmutator: This tool generates deterministic mutants from regular expressions; it supports many source code languages, including Solidity and Vyper. Refer to the 2018 ICSE paper on the tool and this guest blog post about the tool on the Trail of Bits blog for more information.

- Necessist: This tool was developed in-house by Trail of Bits. It operates on tests rather than source code, although it has a similar end goal. Necessist could provide a nice complement to source-focused mutation testing. Due to the time-boxed nature of this review, we deprioritized the use of Necessist to conduct an additional mutation testing campaign.

- Vertigo: This tool was developed by security researchers at Consensys Diligence. Integration with Foundry is planned, but the current progress on that work is unclear. Known scalability issues are present in the tool.

- Gambit: This tool generates stochastic mutants by modifying the Solidity AST. It is optimized for integration with the Certora prover.

We used universalmutator to conduct a mutation testing campaign during this engagement because the mutants it generates are deterministic and because it is a relatively mature tool with few known issues. This tool can be installed with the following command:

```
pip install universalmutator
```

Figure F.1: The command used to install universalmutator

Once installed, a mutation campaign can be run against all Solidity source files using the following bash script:

```
1   find contracts \
2     -name '*.sol' \
3     -not -path '*/interfaces/*' \
4     -not -path '*/test/*' \
5     -not -path '*/external/*' \
6     -print0 | while IFS= read -r -d '' file
7   do
8     name="$(basename "$file" .sol)"
9     dir="mutants/$name"
10    mkdir -p "$dir"
11    echo "Mutating $file"
12    mutate "$file" \
13      --cmd "timeout 200s npx hardhat test" \
14      --mutantDir "$dir" \
15      > "mutants/$name.log"
16  done
```

Figure F.2: A bash script that runs a mutation testing campaign against each Solidity file in the `contracts` directory

Consider the following notes about the above bash script:

- The overall runtime of the above script against all non-excluded Solidity files in the target `contracts` repository is approximately one week on a modern M2 Mac. This execution time is directly related to the `npx hardhat test` tool runtime and the number of contracts to be mutated.

- The `--cmd` argument on line 13 specifies the command to run for each mutant. This command is prefixed by `timeout 200s` (`timeout` is a tool included in the `coreutils` package on macOS) because a healthy run of the test suite was measured to take approximately 150 seconds. A timeout longer than the average test suite runtime is used only to cut off test runs that are badly stalled.

The results of each target's mutation tests are saved in a file, per line 15 of the script in figure F.2. An illustrative example of such output is shown in figure F.3.

```
*** UNIVERSALMUTATOR ***
MUTATING WITH RULES: audit-arcade/custom-solidity.rules
FAILED TO FIND RULE audit-arcade/custom-solidity.rules AS BUILT-IN...
SKIPPED 458 MUTANTS ONLY CHANGING STRING LITERALS
2761 MUTANTS GENERATED BY RULES
...
PROCESSING MUTANT: 121:         if (_feeController == address(0)) revert
OC_ZeroAddress();  ==>          if (_feeController != address(0)) revert
OC_ZeroAddress();...INVALID
PROCESSING MUTANT: 121:         if (_feeController == address(0)) revert
OC_ZeroAddress();  ==>          if (_feeController <= address(0)) revert
OC_ZeroAddress();...VALID [written to
mutants/OriginationController/OriginationController.mutant.25.sol]
```

```
PROCESSING MUTANT: 121:         if (_feeController == address(0)) revert
OC_ZeroAddress();  ==>          if (_feeController >= address(0)) revert
OC_ZeroAddress();...INVALID
...
PROCESSING MUTANT: 375:     ) public override returns (uint256 newLoanId) {  ==>
) public override returns (uint256  {...INVALID
PROCESSING MUTANT: 375:     ) public override returns (uint256 newLoanId) {  ==>
) public override returns (uint256 newLoanId) ...INVALID
PROCESSING MUTANT: 375:         _validateLoanTerms(loanTerms);  ==>
/*_validateLoanTerms(loanTerms);*/...VALID [written to
mutants/OriginationController/OriginationController.mutant.46.sol]
PROCESSING MUTANT: 375:         _validateLoanTerms(loanTerms);  ==>
selfdestruct(msg.sender);...INVALID
PROCESSING MUTANT: 375:         _validateLoanTerms(loanTerms);  ==>
revert();...INVALID
...
156 VALID MUTANTS
2605 INVALID MUTANTS
0 REDUNDANT MUTANTS
Valid Percentage: 5.650126765664615%
```

Figure F.3: Abbreviated output from the mutation testing campaign on `OriginationController.sol`

The output of universalmutator starts with the number of mutants generated and ends with a summary of how many of these mutants are valid. A small percentage of valid mutants indicates thorough test coverage.

The first highlighted snippet in the middle of the output is focused on mutations made to line 121 of the `OriginationController` source code. This particular line shows a mutation with a false positive: this means that while the mutant compiles and passes all tests, the mutation does not imply failure in test coverage because the address cannot be negative. Other types of common false positives include removing the public visibility modifier from variable or function declarations.

However, the second highlighted snippet shows that commenting out line 375 of the `OriginationController` source code makes the test run succeed. Because this change can have consequences in the results of the function, this mutant is expected to be invalid given thorough test coverage. In that particular case, it means that none of the implemented tests for the `OriginationController` contract tries to roll over a loan with invalid loan terms. For auditors, this is a cue to take an extra close look at the implementation of this method and at its use throughout the rest of the codebase.

We recommend running mutation tests on the code every time major changes are made to the code or to the test suite, and we recommend filtering the results to ensure that the test coverage is correct.

G. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which the Arcade protocol will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

H. 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.

On July 24 to July 25, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Arcade 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 16 issues described in this report, Arcade has resolved 12 issues and has not resolved the remaining four issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Different zero-address errors thrown by single and batch NFT withdrawal functions	Informational	Resolved
2	Solidity compiler optimizations can be problematic	Undetermined	Unresolved
3	callApprove does not follow approval best practices	Informational	Resolved
4	Risk of confusing events due to missing checks in whitelist contracts	Low	Resolved
5	Missing checks of _exists() return value	Informational	Resolved
6	Incorrect deployers in integration tests	Informational	Resolved
7	Risk of out-of-gas revert due to use of transfer() in claimFees	Informational	Resolved
8	Risk of lost funds due to lack of zero-address check in functions	Medium	Resolved

9	The maximum value for FL_09 is not set by FeeController	Low	Resolved
10	Fees can be changed while a loan is active	Low	Resolved
11	Asset vault nesting can lead to loss of assets	Low	Unresolved
12	Risk of locked assets due to use of _mint instead of _safeMint	Medium	Resolved
13	Borrowers cannot realize full loan value without risking default	Medium	Resolved
14	itemPredicates encoded incorrectly according to EIP-712	Low	Resolved
15	The fee values can distort the incentives for the borrowers and lenders	Informational	Unresolved
16	Malicious borrowers can use forceRepay to grief lenders	Medium	Unresolved

Detailed Fix Review Results

TOB-ARCADE-1: Different zero-address errors thrown by single and batch NFT withdrawal functions

Resolved in commits `00e8130` and `8542c46`. The `AV_ZeroAddress` error was updated to include a `string addressType` argument that is used to indicate which address parameter violated a zero-address check requirement. Additional zero-address checks were added to functions in the `AssetVault` contract, unit tests were implemented to ensure that these checks work as expected, and the NatSpec comments were updated to reflect this update.

TOB-ARCADE-2: Solidity compiler optimizations can be problematic

Unresolved. Arcade provided the following statement about this issue:

> *The Arcade team understands that Solidity compiler optimizations may potentially be problematic. However to remove our compiler optimization we would need to downsize four of our smart contracts in extensive ways, breaking them up into smaller contracts and considerably increasing our code footprint, introducing more complexity and possibly new risks.*

We elected not to do this at this time, given that the existence of a vulnerability remains undetermined. Before the next major protocol release, the Arcade team will revisit this issue.

TOB-ARCADE-3: callApprove does not follow approval best practices
Resolved in commits a3ce932 and 77c1db0. The AssetVault contract now includes two new functions, callIncreaseAllowance and callDecreaseAllowance, which enable safe interactions with Arcade for third-party integrations. The callApprove function was temporarily removed in commit a3ce932 but was subsequently added back in commit 77c1db0. To aid third-party integrators and test the expected functionality of the newly added functions, documentation and unit tests have been added appropriately.

TOB-ARCADE-4: Risk of confusing events due to missing checks in whitelist contracts
Resolved in commit 2434705. Two new error types, CW_AlreadyWhitelisted and CW_NotWhitelisted, have been added and implemented for whitelisting functions. If the address has already been added or the address targeted for removal is not found, the whitelisting functions will now revert. NatSpec comments have been added to describe each error, and unit tests have been included to test the add and remove functions for the expected revert.

TOB-ARCADE-5: Missing checks of _exists() return value
Resolved in commit e04502b. New reasons for DoesNotExist reverts were added to the Lending and Vault contracts. The VaultFactory and PromissoryNote contracts now use these revert reasons in the tokenURI functions when the requested tokenId is nonexistent.

Additionally, new tests were implemented for VaultFactory and PromissoryNote that check for the expected revert reason when a nonexistent tokenId is used.

TOB-ARCADE-6: Incorrect deployers in integration tests
Resolved in commit dddc905. The incorrect deployers in integration test cases were changed from signers[0] to admin. A new test was implemented to check that the correct permissions are set when the protocol is deployed.

TOB-ARCADE-7: Risk of out-of-gas revert due to use of transfer() in claimFees
Resolved in commit a948cfc. Comments were added to inform users and developers about the issue, but no further changes were made to the contract.

TOB-ARCADE-8: Risk of lost funds due to lack of zero-address check in functions
Resolved in commit a6dbd53. Existing error types related to ZeroAddress were modified to include a string parameter indicating the address that failed the zero-address check (also refer to the fix status for issue TOB-ARCADE-1). The uses of the old error type with no arguments were replaced with the new version.

New zero-address checks for the token and destination addresses were implemented in the `LoanCore.withdraw` and `LoanCore.withdrawProtocolFees` functions. Checks for the destination address were added to the `RepaymentController.redeemNote` and `VaultFactory.claimFees` functions.

Existing tests were modified to account for the new string parameter in error types, and new tests were implemented for zero-address parameters in the fixed functions.

TOB-ARCADE-9: The maximum value for FL_09 is not set by FeeController
Resolved in commit `a51dfd8`. The value of `maxFees[FL_09]` is now set properly in the constructor of the `FeeController` contract. The unit test suite has also been expanded to include tests that ensure that all maximum fee values are properly set.

TOB-ARCADE-10: Fees can be changed while a loan is active
Resolved in commit `f7b87a7`. The fix implemented for this issue consists of several parts.

Fee names were changed. Previously, the `FeeLookups` constants ranged from `FL_01` to `FL_09`, where `FL_01` was the vault minting fee and `FL_02` to `FL_09` were the different fees that the borrowers and lenders should pay for a loan. The Arcade team renamed `FL_02` through `FL_09` to `FL_01` through `FL_08` and removed the former `FL_01` from `FeeLookups` and replaced it with `vaultMintFee` in `FeeController`.

The functions for setting and getting the fees were renamed to `setLendingFee` and `getLendingFee`, respectively. The new `getFeesOrigination` and `getFeesRollover` functions were added to simplify the process of retrieving the fees for the loan origination and rollover processes.

A new `FeeSnapshot` structure was created to take a snapshot of the fee values at the moment a loan is originated to ensure that future changes to fees do not affect existing loans. However, only the values for the default, interest, and principal fees (fees `FL_05` through `FL_07`) are stored, and the redeem fee (`FL_08`) is read from `feeController` at the moment of redeeming. Even though this feels counterintuitive, it is consistent with the statement given by Arcade for the fix status of finding TOB-ARCADE-16; having the redeem fee read from `feeController` allows protocol admins to change the fee to prevent griefing.

Existing tests were modified to comply with the new changes. New tests were added in `RepaymentController` and `feeController`.

TOB-ARCADE-11: Asset vault nesting can lead to loss of assets
Unresolved. Arcade provided the following statement about this issue:

> *The Arcade team understands the likelihood of this problem occurring is quite low because our UI does not allow for vault keys (vault tracking ERC721 tokens) to be*

deposited inside another vault contract. A power user would need to execute this type of vault key transfer via Etherscan.

We elect to address this issue by:

- *Thoroughly documenting this risk and providing clear and comprehensive warnings against this specific action in the documentation*
- *Maintaining our user interface to ensure that it does not present the option of using vault keys as collateral for loans*

TOB-ARCADE-12: Risk of locked assets due to use of _mint instead of _safeMint
Resolved in commit `20b7c66`. The `mint` function in the `PromissoryNote` contract has been updated to use `_safeMint` instead of `_mint`. Similarly, the `initializeBundle` function in the `VaultFactory` contract now uses `_safeMint` instead of `_mint`. These changes ensure that an ERC-721 token is not sent to a contract address that is not configured to receive it. The NatSpec comments have also been updated to reflect these changes. Additionally, the unit testing suite and mock contracts have been updated to adequately test the new expected behavior of the two altered functions.

TOB-ARCADE-13: Borrowers cannot realize full loan value without risking default
Resolved in commits `6586c37` and `f1eb8ae`. A new grace period after the original loan's due date was introduced with the first commit. The grace period added in this commit was a configurable setting that can be between one hour and seven days. An admin-only function was added to set the new value, which emits events on changes or errors. A new set of unit tests was added for the `setGracePeriod` functionality, and existing tests were modified to take the grace period into account.

However, in the second commit, the variable grace period was replaced with a constant 10-minute period. The variable period tests were removed, and the remaining tests were modified to account for the change.

TOB-ARCADE-14: itemPredicates encoded incorrectly according to EIP-712
Resolved in commit `c95e21c`. Item predicates are now encoded in compliance with the EIP-712 standard. Rather than a hash representing an array of item predicates, an array of the actual `Predicate` structs to be signed is now presented to the signer. A `_PREDICATE_TYPEHASH` has been created, and the `_ITEMS_TYPEHASH` has been updated to correctly account for the array of `Predicate` structs that is now represented in the signature. The unit tests dealing with signatures that include items have been updated to reflect the changes to the signature scheme.

TOB-ARCADE-15: The fee values can distort the incentives for the borrowers and lenders
Unresolved. Arcade provided the following statement about this issue:

Our V3 documentation will comprehensively outline all fees within the lending protocol, pointing the users to their values at loan initiation. The Arcade team's objective is to help users grasp potential profits and anticipated fees linked with loan repayment, rollover, default, or redemption scenarios.

TOB-ARCADE-16: Malicious borrowers can use forceRepay to grief lenders
Unresolved. Arcade provided the following statement about this issue:

The Arcade.xyz team is aware that for honest lenders, having forceRepay called incurs additional gas cost and possible fees. Nevertheless, the team has elected to keep the implementation as-is: we feel that having two separate functions is a more explicit, less "surprising" design compared to having a single function whose effects may change based on external state.

In general, we believe the vector allowing borrower griefing is best mitigated through proper incentive management and counterparty relationship management: borrowers who have griefed lenders in the past are likely to receive lending offers with higher premiums, as lenders try to mitigate their risk. In a larger sense, if griefing becomes a protocol-wide issue, redeem fees can be set to 0.