Zellic

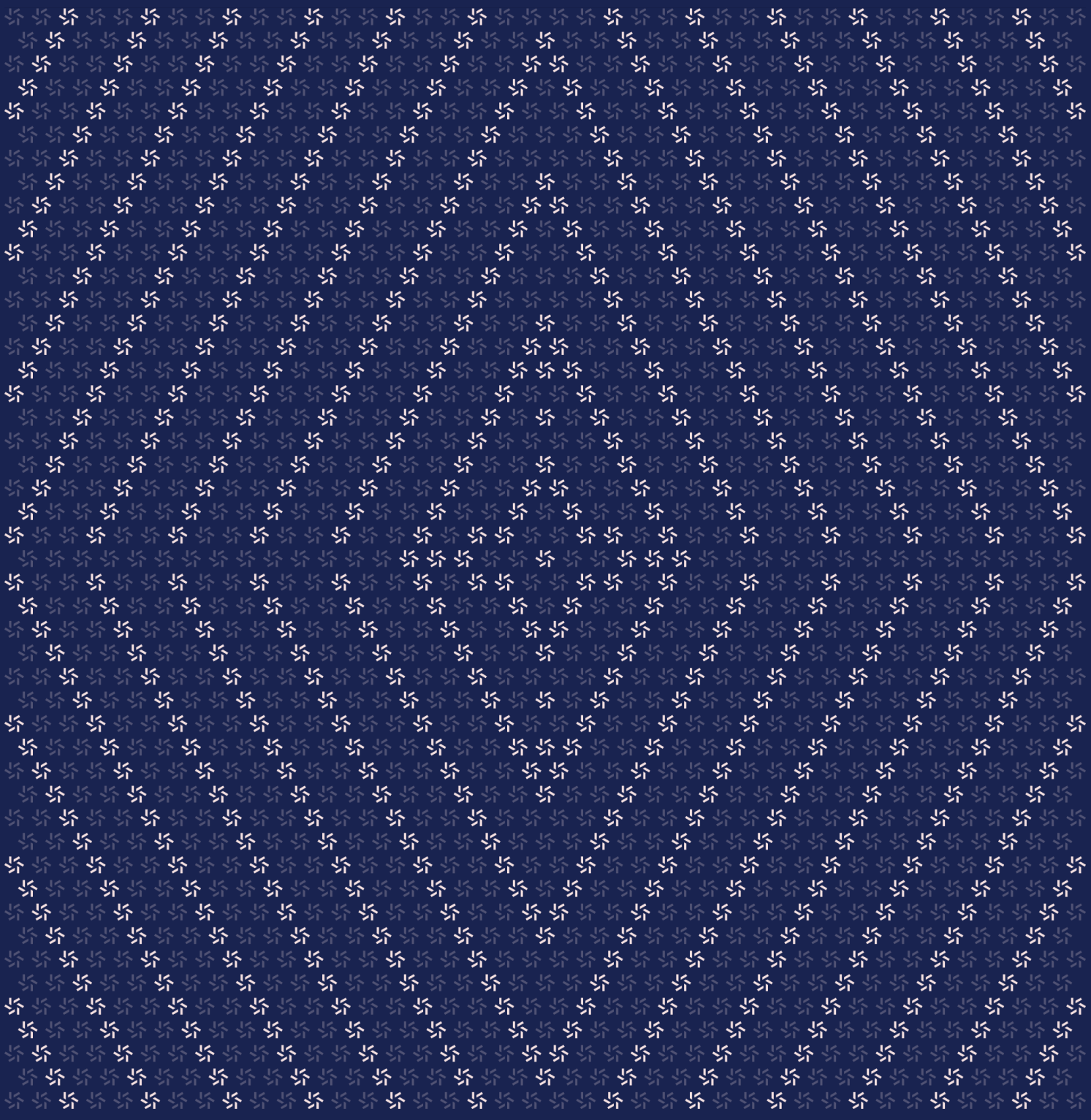
Prepared for
Divine Research

Prepared by
Sunwoo Hwang
Jinheon Lee
Zellic

September 16, 2025

Divine Research Credit Contracts

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Divine Research from September 8th to September 11th, 2025. During this engagement, Zellic reviewed Divine Research Credit Contracts's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any vulnerabilities that could lead to loss of funds?
- Is loan accounting accurate?
- Are there any bugs in the management of loan repayments?
- Does the system run correctly on World Chain?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

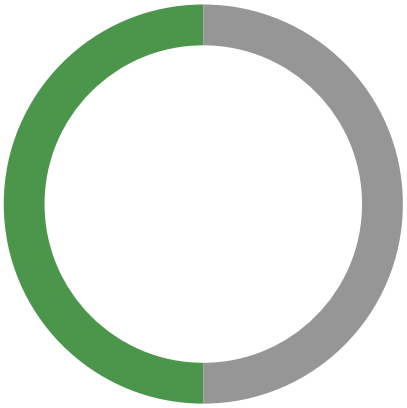
1.4. Results

During our assessment on the scoped Divine Research Credit Contracts contracts, we discovered two findings. No critical issues were found. One finding was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Divine Research in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	0
Indeterminate	1



2. Introduction

2.1. About Divine Research Credit Contracts

Divine Research contributed the following description of the Divine Research Credit contracts:

> Credit is an undercollateralized credit protocol built on World Chain leveraging peer-to-pool lending.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Divine Research Credit Contracts

Type	Solidity
Platform	EVM-compatible
Target	credit-contracts
Repository	https://github.com/DivineResearch/credit-contracts ↗
Version	`6692555b55506c3eab5de98e3ff17e99c4f334ad`
Programs	`LoanManager.sol` `Vault.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of one person-week. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Sunwoo Hwang
Engineer
sunwoo@zellic.io

Jinheon Lee
Engineer
jinheon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 8, 2025	Kick-off call
September 8, 2025	Start of primary review period
September 11, 2025	End of primary review period

3. Detailed Findings

3.1. The protocol depends on the Permit2 signature for repaying loans

Target	LoanManager.sol		
Category	Protocol Risks	**Severity**	Medium
Likelihood	Low	**Impact**	Indeterminate

Description

The `issue` function accepts a Permit2 signature for forcing a repayment. To issue a loan, it requires a signature from the coordinator, which checks the off-chain credit score of the user.

This caveat is documented in the docs ↗.

Here are some other caveats regarding the Permit2 signature:

- The repay tokens are USDC and WLD, which require a user to approve Permit2 to spend them, and the user can set the allowance to zero.
- Users can invalidate a Permit2 signature.
- Users can send their permit token to another address.

Impact

The design assumes that the private key of an Orb-verified account cannot be extracted from the World App. If an attacker were able to obtain multiple verified accounts, each account could borrow funds without repaying.

We understand this finding highlights a known design flaw, but it does carry a serious risk: it could lead to undercollateralization of the protocol. In that case, collected fees (assuming interest is charged to users via the permit2 system) wouldn't be enough to cover defaults.

As the protocol grows and the World ecosystem matures with a larger user base and more apps, the risk of undercollateralization also grows.

Recommendations

There's no easy fix to this issue. To fix the root cause cooperation is needed with the World team to upgrade the airdrop claim mechanism. World team could add the ability for each user to set an airdrop receiver address, this allows the divine team to set a smart contract as the receiver that enforces the loan repayment.

Remediation

This issue is has been acknowledged by Divine Research and they responded as follows:

> The caveats are correct and expected. The economic cost of obtaining World Orb-verified accounts is explicitly factored into the initial credit limit calculations. The cost of acquiring an Orb-verified account exceeds the initial credit limit, making it economically irrational to obtain multiple accounts for the purpose of defaulting. Furthermore, missed forced repayments are factored into the coordinator's calculations for credit limits, interest rates, and when marking down loans.

A note on the ranking of this issue: We originally classified it as High Impact due to the potential risk of fund loss. After further discussion with Divine, we've decided to reclassify it as an Indeterminate Impact. This change is based on two main points.

First, the economic cost of obtaining an Orb-verified account is not straightforward to assess. Divine states that they have factored the economic cost into their initial credit limit calculations. We were unable to determine an exact price for purchasing an account, but it likely exceeds the $5 starting loan threshold that the client intends to use. If an attacker wants to organize a coordinated attack, they would need to invest a significant amount of capital purchasing Orb-verified accounts for a limited upside.

Regardless of the cost to obtain an account, an attacker can still default and steal a single $5 loan before being blacklisted from future borrowing. Blacklisting limits repeat abuse and reduces the attack's upside, but it does **not** eliminate the possibility of a one-time theft. So the attack is economically constrained (limited upside) rather than strictly uneconomical. However, if Orb-verified accounts were obtained below the $5 dollar starting loan threshold, this attack becomes economically viable.

Divine states that the credit-limit controls make a large-scale attack unlikely to drain the protocol before it can respond. However, we cannot independently confirm that without reviewing the off-chain coordinator.

Second, the off-chain coordinator, which plays a key role in the system's health, was not part of our audit. Divine states that they have accounted for this scenario by balancing initial credit limits with normal repayments and their probabilities, and by managing APR based on expected losses from defaults in order to keep the protocol safe. However, we cannot independently verify this, since we have not reviewed that code.

3.2. Wrong linear decay of `totalExpectedRecovery`

Target	Vault.sol		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

`totalExpectedRecovery` is meant to decay linearly to zero over 120 days. However, `applyExpectedDecay` updates an already-decayed amount, so multiple calls produce compounding decay. For example, one call after two days yields `T*(1 - 2/120)`, while two daily calls yield `T*(1 - 1/120)^2`. Newly added recovery is also decayed on the same schedule as already-decayed amounts.

Impact

The value can drift below the intended linear schedule, underestimating expected recovery when calls are frequent.

Recommendations

Implement per-loan linear decay for `totalExpectedRecovery`.

Remediation

This issue has been acknowledged by Divine Research, and fixes were implemented in the following commits:

- `29f6a0d8` ↗
- `678ac11a` ↗

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Missing EOA verification in the contract

The `register` function does not verify whether the borrower address has been verified with an Orb. As a result, any EOA on World Chain can register as a borrower, which may violate the protocol's caveats.

The Divine Research team responded as follows:

> Correct, the borrower's smart account is verified when they attempt to issue a loan - the signature of the coordinator is not created for a borrower unless they are Orb verified.

4.2. Missing slippage protection during repayment token swaps

When a borrower repays with a token that isn't a `Vault` asset, the `LoanManager` swaps the repayment token into the `Vault` asset before crediting the repayment. Because there's no `minAmountOut` or slippage limit, normal market moves can leave a small residual balance even when the borrower expects to fully repay. If pool state changes between quote and execution, the transaction still succeeds but credits less than the nominal repayment amount, leaving loan state unpaid.

4.3. Loan default feedback loop

One concern, related to the Permit2 signature issue described in Finding 3.1. ↗, is that a high number of coordinated defaults could trigger a negative feedback loop: defaults would push borrowing rates higher, potentially making the APR too high to attract new participants — or, in a worst-case scenario, incentivizing some participants to take out loans at high APRs with the expectation of not repaying the loan.

Divine Research confirmed that this falls within their expected operational scenarios. They note that their off-chain coordinator, which was not reviewed as part of this audit, is designed to maintain solvency and mitigate losses through a combination of dynamic interest rates, risk-adjusted credit limits, and automated repayment enforcement. Defaults are economically disincentivized at higher credit levels, and potential losses from initial or higher-risk loans are accounted for in advance.

The Divine Research team responded as follows:

> We acknowledge this scenario. That said, coordinated defaults aren't a profitable attack vector in isolation, as acquiring multiple World Orb-verified accounts costs far more than the initial credit limits, making it economically irrational to scale for drainage. The coordinator oversight gives control over this scenario as it can selectively restrict access when conditions deteriorate.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: LoanManager.sol

Function: `forceRepay(address[] calldata borrowers, ISignatureTransfer.PermitTransferFrom[][] calldata permits, bytes[][] calldata permitSignatures)`

The function processes force repayments for multiple borrowers with overdue loans using submitted permits. It returns per-borrower totals for principal repaid, debt repaid, and penalty paid.

Inputs

- `borrowers`
  - **Control**: Must contain at least one address. Each address must have an overdue loan at execution time.
  - **Impact**: Selects which borrowers are targeted for force repayment.
- `permits`
  - **Control**: The length must equal the length of `borrowers`. For each borrower, every permit must reference a token listed in `forceRepayTokens`.
  - **Impact**: Authorizes token transfers from the borrower to the borrower's deposit surrogate for repayment.
- `permitSignatures`
  - **Control**: The length must equal the length of `permits`. Inner array lengths must align with the corresponding borrower's permits. Each signature must be valid for its permit.
  - **Impact**: Enables `permitTransferFrom` to move funds under each permit.

Branches and code coverage (including function calls)

Intended branches

- For each borrower with an overdue loan and for each valid permit, transfers from the borrower to the borrower's deposit surrogate the lesser of the permitted amount and the borrower's current token balance.

- For each borrower with an overdue loan and for each valid permit, calls `_repay` with `forceRepaid` set to true and adds the returned principal, debt, and penalty to the per-borrower totals.
- For each borrower with an overdue loan and for each valid permit, if the borrower's loan is fully resolved such that the state is no longer `OUTSTANDING`, proceeds to the next borrower.

☑ Test coverage

Negative behavior

- Reverts when `borrowers` is empty.

☑ Negative test

- Reverts when the length of `permits` does not equal the length of `borrowers`.

☑ Negative test

- Reverts when the length of `permitSignatures` does not equal the length of `permits`.

☑ Negative test

- Reverts when a borrower does not have an overdue loan at the time of processing.

☑ Negative test

- Reverts when any permit references a token that is not listed in `forceRepayTokens`.

☑ Negative test

- Reverts when the caller is not the coordinator.

☑ Negative test

- Reverts on reentrancy.

☑ Negative test

Function: `issue(uint256 principal, uint256 debt, uint256 overdueTime, ISignatureTransfer.PermitTransferFrom[] calldata permits, bytes[] calldata permitSignatures, uint256 coordinatorSignatureExpiresTime, bytes calldata coordinatorSignature)`

The function issues a loan to the caller. Permits serve attestation purposes only, while loan terms are validated off chain by the coordinator.

Inputs

- `principal`
  - **Control**: Must be greater than zero and not exceed `MAX_PRINCIPAL`.
  - **Impact**: Amount borrowed, tracked as `remainingPrincipal`.

- `debt`
    - **Control**: Must be greater than or equal to `principal`.
    - **Impact**: Total amount to be repaid, tracked as `remainingDebt`.
- `overdueTime`
    - **Control**: Must be between the minimum duration and the maximum duration from the current timestamp.
    - **Impact**: Determines when the loan state changes to `OVERDUE`.
- `permits`
    - **Control**: The length must equal `permitSignatures.length`. If validation is enabled, the length must be at least the number of `forceRepayTokens` multiplied by `REPAY_TOKEN_PERMITS_COUNT`. Each permit must correspond to a valid force-repay token, use the maximum amount, have an unused nonce, and a sufficient deadline.
    - **Impact**: Used for attestation and validation. Valid permits trigger `PermitSubmitted` events.
- `permitSignatures`
    - **Control**: The length must equal `permits.length`. Each signature must be valid for the corresponding permit.
    - **Impact**: Allows on-chain verification of permits.
- `coordinatorSignatureExpiresTime`
    - **Control**: Must be greater than the current timestamp.
    - **Impact**: Defines the validity period of the coordinator's signature.
- `coordinatorSignature`
    - **Control**: Must correctly recover to the coordinator address for the encoded loan parameters.
    - **Impact**: Authorizes issuance under the specified terms.

Branches and coverage

Intended branches

- A new loan is issued with a unique `loanId`, stored in `loanInfo`, and funded through the vault.
    - ☑ Test coverage
- If validation is enabled, permits are verified and allowances are confirmed.
    - ☑ Test coverage

Negative behavior

- Reverts when the borrower state is not `ELIGIBLE`.

☑ Negative test

- Reverts when `principal` is zero or greater than `MAX_PRINCIPAL`.

☑ Negative test

- Reverts when `debt` is smaller than `principal`.

☑ Negative test

- Reverts when `overdueTime` is not within the allowed duration range.

☑ Negative test

- Reverts when `permitSignatures.length` does not equal `permits.length`.

☑ Negative test

- Reverts when `coordinatorSignatureExpiresTime` is expired.

☑ Negative test

- Reverts when validation is enabled and the number of permits is insufficient.

☑ Negative test

- Reverts when a permit token is invalid.

☑ Negative test

- Reverts when a permit amount is not set to the maximum.

☑ Negative test

- Reverts when a permit nonce has already been used.

☑ Negative test

- Reverts when a permit deadline is too short.

☑ Negative test

- Reverts when a permit signature is invalid.

☑ Negative test

- Reverts when allowances for a force-repay token are missing.

☑ Negative test

- Reverts when the coordinator signature is invalid.

☑ Negative test

- Reverts on reentrancy.

☑ Negative test

Function: `register(ISignatureTransfer.PermitTransferFrom[] calldata permits, bytes[] calldata permitSignatures)`

The function registers the caller as a borrower and deploys a deposit surrogate proxy.

Inputs

- `permits`
  - **Control**: The length must match `permitSignatures`; if `VALIDATE_PERMITS` is enabled, it must also match `forceRepayTokens.length`. Each token of `permits` must equal the corresponding `forceRepayTokens`.
  - **Impact**: Authorizes the contract to transfer at least 1 WEI of each force-repay token.
- `permitSignatures`
  - **Control**: The length must equal the length of `permits`. Each signature must be valid for the corresponding permit.
  - **Impact**: The `receiver` receives the `assets`.

Branches and code coverage (including function calls)

Intended branches

- The caller is successfully registered as a borrower and associated with a newly deployed deposit surrogate proxy.
  - ☑ Test coverage
- If `VALIDATE_PERMITS` is enabled, the contract verifies token balances, sends 1 WEI to the caller, and retrieves it back with the permit to ensure valid authorization.
  - ☑ Test coverage

Negative behavior

- The function reverts when the caller's state is not `UNREGISTERED`.
  - ☑ Negative test
- The function reverts when the lengths of `permitSignatures` and `permits` do not match.
  - ☑ Negative test
- The function reverts when `VALIDATE_PERMITS` is enabled and the lengths of `permitSignatures` and `permits` do not match.
  - ☑ Negative test
- The function reverts when the contract has less than 1 WEI of a `forceRepayToken`.
  - ☑ Negative test
- The function reverts when a permit token does not match the expected `forceRepayToken`.
  - ☑ Negative test
- The function reverts on invalid permit execution (e.g., invalid signature).

☑ Negative test

- The function reverts on reentrancy.

☑ Negative test

Function: `repay(address[] calldata borrowers, address[] calldata tokens)`

The function processes repayments for multiple borrowers with outstanding loans using their deposit surrogates. It returns per-borrower amounts for principal repaid, debt repaid, and penalty paid.

Inputs

- `borrowers`
  - **Control**: Must contain at least one address. Each address must correspond to a borrower whose state is `OUTSTANDING`.
  - **Impact**: Determines which borrowers are targeted for repayment.
- `tokens`
  - **Control**: The length must equal the length of `borrowers`. Each token must be the vault asset or a token with a configured swap-fee tier.
  - **Impact**: Determines the repayment token used for each borrower.

Branches and code coverage (including function calls)

Intended branches

- For each index, `_repay` is executed with the matching borrower and token, with `forceRepaid` set to false. Returned values populate the arrays in order.

☑ Test coverage

Negative behavior

- The function reverts when `borrowers` is empty.

☑ Negative test

- The function reverts when the length of `tokens` does not equal the length of `borrowers`.

☑ Negative test

- The function reverts when a borrower's state is not `OUTSTANDING`.

☑ Negative test

- The function reverts when a token is neither the vault asset nor a token with a

configured swap-fee tier.

☑ Negative test

- The function reverts on reentrancy.

☑ Negative test

- The function reverts when the caller is not the coordinator.

☑ Negative test

Function: `_repay(address borrower, address token, bool forceRepaid)`

This is an internal function that processes repayment for a borrower using their deposit surrogate. Repayments may be partial or cover the full outstanding amount.

Inputs

- `borrower`
  - **Control**: Must have an outstanding loan — otherwise, the call reverts.
  - **Impact**: Identifies which borrower's debt and principal are reduced.
- `token`
  - **Control**: Must either be the vault asset or a token that can be swapped into the asset.
  - **Impact**: Determines which token is withdrawn from the borrower's deposit surrogate for repayment.
- `forceRepaid`
  - **Control**: Boolean flag controlled by the caller.
  - **Impact**: Marks whether the repayment was triggered as part of a forced repayment process.

Branches and code coverage (including function calls)

Intended branches

- Funds are withdrawn from the borrower's deposit surrogate and converted to the vault asset if necessary.

☑ Test coverage

- The amount paid is capped at the outstanding total; any excess is returned to the borrower.

☑ Test coverage

- Loan state updates to `REPAID` reset the borrower's `outstandingLoanId`.

☑ Test coverage

- The repaid amount is transferred to the vault, and `vault.handleRepay` is called.

☑ Test coverage

Negative behavior

- Reverts when the borrower has no outstanding loan.

☑ Negative test

- Reverts when the deposit surrogate has zero balance for the given token.

☑ Negative test

- Reverts when the outstanding total is zero.

☑ Negative test

5.2. Module: Vault.sol

Function: `handleIssue(uint256 loanId, uint256 principal)`

The function handles loan issuance by transferring principal funds to the borrower and updating vault state.

Inputs

- `loanId`
  - **Control**: Must not be zero. Must correspond to a loan in state `ACTIVE`. The lender of the loan must be this contract.
  - **Impact**: Identifies the loan being issued and validated against state and lender.
- `principal`
  - **Control**: Must be the correct principal value determined by the loan terms.
  - **Impact**: Amount of asset transferred to the borrower and added to the outstanding principal.

Branches and code coverage (including function calls)

Intended branches

- Transfers the `principal` amount of the vault's asset to the borrower tied to `loanId`.

☑ Test coverage

- Increases `totalOutstandingPrincipal` by the principal amount.

☑ Test coverage

Negative behavior

- Reverts when `loanId` is zero.
  - ☑ Negative test
- Reverts when the loan state is not `ACTIVE`.
  - ☑ Negative test
- Reverts when the lender of the loan is not this contract.
  - ☑ Negative test
- Reverts on reentrancy.
  - ☑ Negative test
- Reverts when called by a non–loan manager.
  - ☑ Negative test

Function: `handleRepay(uint256 loanId, uint256 principalRepaid)`

The function handles loan repayments, which may be partial or full; applies expected decay; and updates vault aggregates.

Inputs

- `loanId`
  - **Control**: Must not be zero. Must not be in state `UNISSUED`.
  - **Impact**: Identifies the loan whose repayment affects vault-level metrics.
- `principalRepaid`
  - **Control**: When the lender of `loanId` is this contract and the loan is not marked down, the value must not exceed `totalOutstandingPrincipal`.
  - **Impact**: Reduces `totalOutstandingPrincipal` if the loan is not marked down — otherwise reduces `totalExpectedRecovery` by the smaller of this value and the current expectation.

Branches and code coverage (including function calls)

Intended branches

- If the lender of `loanId` is this contract and the loan is not marked down, decreases `totalOutstandingPrincipal` by `principalRepaid`.
  - ☑ Test coverage

- If the lender of `loanId` is this contract and the loan is marked down, decreases `totalExpectedRecovery` by the minimum of `principalRepaid` and `totalExpectedRecovery`.

  ☑ Test coverage

Negative behavior

- Reverts when `loanId` is zero.

  ☑ Negative test
- Reverts when the loan state is `UNISSUED`.

  ☑ Negative test
- Reverts when the lender is this contract, the loan is not marked down, and `principalRepaid` is greater than `totalOutstandingPrincipal`.

  ☑ Negative test
- Reverts when called by a non–loan manager.

  ☑ Negative test
- Reverts on reentrancy.

  ☑ Negative test

Function: `markDownLoans(uint256[] calldata loanIds, uint256[] calldata maxExpectedRecoveries)`

The function marks overdue loans down as losses, moves principal out of outstanding, and books expected recoveries.

Inputs

- `loanIds`
  - **Control**: Must contain at least one element. The length must equal `maxExpectedRecoveries.length`. Each ID must be nonzero, correspond to a loan in state `OVERDUE`, and not already be marked down. The array must not contain duplicates.
  - **Impact**: Selects the loans to mark down.
- `maxExpectedRecoveries`
  - **Control**: The length must equal `loanIds.length`. Each entry is capped at the remaining principal of the corresponding loan.
  - **Impact**: Determines the expected recovery booked for each loan.

Branches and code coverage (including function calls)

Intended branches

- For each loan, sets `markedDown[loanId]` to true.
  - ☑ Test coverage
- For each loan, aggregates totals — subtracts the full `remainingPrincipal` from `totalOutstandingPrincipal` (via subtotal) and adds `expectedRecovery` to `totalExpectedRecovery` (via subtotal).
  - ☑ Test coverage

Negative behavior

- Reverts when `loanIds` is empty.
  - ☑ Negative test
- Reverts when the lengths of `loanIds` and `maxExpectedRecoveries` do not match.
  - ☑ Negative test
- Reverts when any `loanId` is zero.
  - ☑ Negative test
- Reverts when any loan is not in state `OVERDUE`.
  - ☑ Negative test
- Reverts when any loan is already marked down (including duplicates within the input).
  - ☑ Negative test
- Reverts when the subtotal of remaining principals exceeds `totalOutstandingPrincipal`.
  - ☑ Negative test
- Reverts when called by a noncoordinator.
  - ☑ Negative test
- Reverts on reentrancy.
  - ☑ Negative test

6. Assessment Results

During our assessment on the scoped Divine Research Credit Contracts contracts, we discovered two findings. No critical issues were found. One finding was of low impact.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.