FEATURE AUDIT - AAVE-V3 `REPAY()`

Valya Zaitseva

February 10, 2026

Table of Contents

I. Feature Scope

1. The feature under the audit: **`repay()`**.

2. **Economic purpose:** it allows users to repay user debt (partially or totally) for themselves or on behalf of users, provided that they were explicitly delegated. The protocol can accept more assets than remaining debt at the call boundary, but must cap the effective repayment to outstanding debt and refund any excess.

3. **State modified:**

- protocol liquidity,
- total debt amount,
- user debt balances and borrowing configuration.

II. Invariants / Design-Level Analysis

Table 1. Grouped Invariants with Failure Surface and Threat Model

State	#	Type	Invariant in plain English	Failure Surface (what can go wrong)	5-lens questions	Threat Model / Attack	Enforced / Assumed	Severity / Impact	Test	Tested
Pre-state	**1**	DEPENDENCY SAFETY	repay() must not rely on oracle values for correctness - e.g. oracle is set and price is not zero.	Oracle values are not correct	External: what happens if oracle returns zero or a stale price?	No threat attack because there is no code path to touch an oracle	ENFORCED: enforced implicitly	-	-	Safe by design
	2	PROTOCOL-LEVEL SAFETY	reserve should be active and asset is not paused	Reserve is not active and/or asset is paused	Economic/State: Can debt be paid if reserve is not active and/or asset is paused?	Attacker can bypass that reserve is not active and/or asset is paused and repay	ENFORCED: ValidationLogic.validateRepay()	Medium / Low	Unit	Yes
	3	STATE VALIDITY	repay() must revert if effective user debt is zero	Repay attempted on user debt == 0	State: Can zero debt be repaid leaving dust or triggering events?	Attacker triggers a no-op, causing greifing, monitoring degradation	ENFORCED: ValidationLogic.validateRepay()	Low / Low	Unit	Yes
Mid-process	**4**	AUTHORIZATION	anybody can repay 'onBehalfOf' debt	Allowed by design	Authorization: Can a caller repay on behalf of another user without delegation approval?	No threat attack because repay only improves debtor's HF	ENFORCED implicitly	-	-	Safe by design
Post-state	**5**	USER ACCOUNTING	effective **debt** reduction <= user debt	Protocol reduces a user's debt by more than they owe	User accounting: Can the protocol forgive debt that never existed?	Attacker repays and protocol reduces debt by more than they owe due to rounding, index mismatch, or stale debt snapshot	ENFORCED: IStableDebtToken/IVariableDebtToken._burn()	High / High	Invariant test, fuzz	No
	6	PROTOCOL SOLVENCY	**burned debt tokens** amount should be <= user debt	**Surplus burn** - debt tokens are burned in amount more that user's debt without accepting liquidity that covers extra burned debt tokens	Economic: Can amount of burned debt tokens be more that user debt?	Attacker **sends more assets t**o the protocol in order to burn more debt tokens than the user's debt, but protocol refunds assets difference and **burns more than debt tokens** amount because of rounding issues. Attacker s**ends less assets tha**n debt, protocol **burns more than user's debt** because of rounding issues.	ENFORCED: IStableDebtToken/IVariableDebtToken.burn()	High / High	Invariant test, fuzz	No
	7	PROTOCOL SOLVENCY	**burned debt tokens** amount should be <= effective repay amount to the protocol	**Surplus burn** - burned debt tokens amount is more than effective repay amount transferred to the protocol	Economic: Can amount of burned debt tokens be more than effective repay amount transferred to the	Attacker sends less assets but debt is burned more than effective repay amount	ENFORCED: executeRepay()	High / High	Invariant test, fuzz	No

					protocol?					
	8	INDEX CONSISTENCY	indices may update, but must remain monotonic and consistent with accrued interest - **reserve indices are monotonic non-decreasing** and independent of individual user repay actions	indices decrease	Economic: Can indices be manipulated in order to reduce APR?	Attacker repays minimal amount that reduces indices -> repays other part of debt with decreased indices	ENFORCED: _updateIndexes()	High / High	Unit	No
	9	DUST SAFETY	protocol must not create unrepayable or irreducible debt dust - **applies to variable debt**, not stable	unrepayable or irreducible debt dust stays in protocol, mutating indeces and internal accounting and fragmentating protocol liquidity?	Economic: Can user repay in a way some dust remains in protocol?	Attacker creates a lot of small debts, repays them in a way that some dust remains in protocol, leading to unrepayable liquidity amount is locked in protocol and will never be repaid and returned to the protocol effective liquidity. Indices will be unfluenced by this unpayable amount.	ENFORCED: _burnScaled() -> amount.rayDiv(index) rounding up ensures no unrepayable scaled debt remains	High / High	Invariant test, fuzz	No
	10	USER ACCOUNTING	user health factor must never decrease as a result of 'repay()'	user health factor decreases as a result of repay.	Accounting: Can user's HF decrease as a result of repay?	No, because debt is reduced but collateral value and liquidation threshold are not touched by 'repay()'	semantic invariant - not runtime require() check	-	Invariant test, fuzz	Safe by design
	11	USER ACCOUNTING	user health factor must not improve unless the user's debt decreases	user health factor improves without debt decrease	Accounting: Can user's HF improve as a result of repay without debt decrease?	No. HF may improve without debt decrease if only collateral increases or liquidation threshold increases, but these both are not results of 'repay()' and are not an attack vector	semantic invariant - not runtime require() check	-	-	Safe by design
	12	INDEX CONSISTENCY	a user's debt exposure is fully eliminated when scaled debt becomes zero **(full repay amount covering total scaled debt)**, without mutating reserve indices	user still **has debt tokens even if debt becomes zero** and this leads to miscalculating of reserve indices	Economic: Can user still have debt tokens when his debt is already zero?	Attacker repays all debt but debt tokens are still on attacker's balance, mutating reserve indices	ENFORCED: _updateIndexes() and _burnScaled()	Medium / High	Invariant test, fuzz	No
	13	USER ACCOUNTING	user state is updated correctly on full repay	user state is updated wrongly on full repay	Accounting: Can user's state be updated incorrecly?	Attacker can manipulate user's state, especially if this attack is combined with anauthorized repay possibility.	ENFORCED: IStableDebtToken.burn() and IVariableDebtToken.burn()	Medium / Medium	Unit	No

III. Cross-Feature Invariant Consistency

`repay()` should reverse all borrow-induced state changes: decrease debt tokens and restore liquidity, should reverse principal effects, not necessarily index history.

IV. Code Inspection

1. ‘repay()’ - Call Map and Structural Scan

```
repay()                                                  -> reduce what the user owes, pay value either by
   |                                                        burning aTokens or transfer underlying into the protocol
   |-> BorrowLogic.executeRepay()
       |-> VIEW: reserve.cache()                                          -> in-memory snapshots of addresses,
       |                                                                    indices and rates of a reserve
       |-> IRREVERSIBLE/STORAGE: reserve.updateState()                    -> updates indices and timestamp
       |-> VIEW: Helpers.getUserCurrentDebt                               -> gets user's stableDebt and
       |                                                                    variableDebt
       |-> ACCOUNTING BOUNDARIES/VIEW: ValidationLogic.validateRepay      -> validates repay
       |-> determine 'paybackAmount'
           |-> if InterestRateMode == STABLE -> paybackAmount = stableDebt
           |-> else -> paybackAmount = variableDebt
       |-> partial vs full repayment                                     -> reconsider 'paybackAmount' depending
       |                                                                    on effective repay amount
       |                                                                    transferred to protocol
       |-> if InterestRateMode == STABLE                                 |-> debt tokens are burned before
                                                                            transfer and update of interest rates
           |-> IRREVERSIBLE PIVOT/EXTERNAL: IStableDebtToken.burn()       |
       |-> else                                                           |
           |-> IRREVERSIBLE PIVOT/EXTERNAL: IVariableDebtToken.burn()     |
       |-> IRREVERSIBLE/STORAGE: reserve.updateInterestRates              -> updates interest rates and
       |                                                                    utilisation, affecting borrow and
       |                                                                    supply rates
       |-> IRREVERSIBLE/STORAGE: userConfig.setBorrowing                  -> user's borrowing flag is zeroed if
       |                                                                    he repaid all debt
       |-> IRREVERSIBLE/STORAGE: IsolationModeLogic.updateIsolatedDebtIfIsolated  -> adjust isolated debt if reserve is
       |                                                                    isolated
       |-> if useATokens == true
           |-> IRREVERSIBLE/STORAGE: IAToken.burn()                       -> no approve/transfer of underlying
           |                                                                token, burn aToken and reduce the
           |                                                                user's claim on reserve liquidity
       |-> else
           |-> IRREVERSIBLE/EXTERNAL: IERC20.safeTransferFrom()           -> transfer underlying into the
           |                                                                protocol
       |-> EMIT: Repay(asset, onBehalfOf, msg.sender, paybackAmount, useATokens)
```

2. ‘burn()’

a. ‘IStableDebtToken.burn()’ - Call Map and Structural Scan

```
IStableDebtToken.burn()
    |->_calculateBalanceIncrease()
        |-> VIEW: currentBalance = computed new principal which includes interest (it is realized, i.e. previous principal
                  + interest) - not stored, it will be used totally or partially to burn repaid amount.
        |-> VIEW: balanceIncrease = just interest accrued till now using a user stableRate
    |-> cache a debtToken totalSupply
    |-> cache a 'userStableRate'
    |
    |-> if (debtToken.totalSupply <= repaid amount)  -> reset storage - overflow/precision protection
        |-> STORAGE: _totalSupply = 0
        |-> STORAGE: _avgStableRate = 0
    |-> else
        |-> STORAGE: calculate _totalSupply = (current debtToken totalSupply - repaid amount)
        |-> calculate 'total stable debt' (rate-weighted total stable debt)
        |-> calculate 'individual repaid stable debt' (rate-weighted repaid portion)
        |-> if 'individual repaid stable debt' >= 'total stable debt' -> reset storage - overflow/precision protection
            |-> STORAGE: _totalSupply = 0
            |-> STORAGE: _avgStableRate = 0
        |-> else
            |-> STORAGE: _avgStableRate = ('total stable debt' - 'individual repaid stable debt') / 'remained debt (i.e.
                        total supply after repaid amount)'

    |-> if user repays all his stableDebt (repaid amount == computed new principal)
        |-> STORAGE: clear 'userStableRate'
        |-> STORAGE: clear user timestamp
    |-> else
        |-> STORAGE: update user timestamp

    |-> STORAGE: update total supply timestamp (always updated)

    |-> if user repays less than cumulatedInterest on his principal (cumulatedInterest is not stored by this point because
        it will be mutated later)
        |-> STORAGE: _mint()     -> realise interest - add unpaid part of interest (cumulatedInterest - repaid amount) to
                    user's stableDebtToken balance (add to his principal, i.e. interest is capitalised)
        |-> EMIT: Transfer(address(0), from, amountToMint)
        |-> EMIT: Mint(...params)
    |-> else
        |-> repaid amount will cover all cumulatedInterest (cumulatedInterest is not written to storage yet) plus
            excessive amount = (repaid amount - cumulatedInterest)
        |-> STORAGE: _burn()   -> burn excessive amount calculated on the previous stem - it is a part of the principal
        |-> EMIT: Transfer(from, address(0), amountToBurn)
        |-> EMIT: Burn(...params)
```

Summarised storage effects for ‘IStableDebtToken.burn()’

Zone	Condition	Storage effect
Only interest covered	repaid <= cumulatedInterest	`_mint()` capitalises unpaid interest (increases principal), `_avgStableRate` updated
Interest + partial principal	cumulatedInterest < repaid < totalDebt	`_burn()` reduces principal, `_avgStableRate` updated
Full repayment	repaid == totalDebt	Clear user rate and timestamp, `_avgStableRate` updated

b. ‘IVariableDebtToken.burn()’ - Call Map and Structural Scan

```
IVariableDebtToken.burn()
    |-> _burnScaled() - actual reduction of user's scaledBalance in userState happens here;
            |-> compute repaid amount scaled down (to make it subtractable from scaledBalance) - based on current variable
                borrow index;
            |-> VIEW: get user's scaledBalance;
            |-> !!! part needed for emitting events only !!!
              |-> compute 'balanceIncrease' - it is the interest accrued since the last index update in the user's state;
            |-> STORAGE: update index in user's state;
            |-> STORAGE: _burn() -> old user's scaled balance is reduced by scaled repaid amount;
            |-> !!! event's emitting part !!!
                |-> if accrued interest 'balanceIncrease' > repaid amount (not scaled down)
                    |-> EMIT: Transfer(address(0), user, amountToMint);
                    |-> EMIT: Mint(user, user, amountToMint, balanceIncrease, index);
                |-> else
                    |-> EMIT: Transfer(user, address(0), amountToBurn);
                    |-> EMIT: Burn(user, target, amountToBurn, balanceIncrease, index);
    |-> returns scaledTotalSupply() - actual current variable debt token total supply after repaid amount subtraction in
        _burn() above
```

Summarised storage effects for ‘IVariableDebtToken.burn()’

Zone	Condition	Storage effect	Events
Only interest covered	repaid < balanceIncrease	`_totalSupply` reduced by scaled repaid amount `userState.balance` reduced by the scaled repaid amount `userState.index` updated to the current one	EMIT: Transfer(address(0), user, amountToMint); EMIT: Mint(user, user, amountToMint, balanceIncrease, index);
Interest + partial scaledBalance	balanceIncrease <= repaid && scaled repaid amount < user’s scaledBalance		EMIT: Transfer(user, address(0), amountToBurn); EMIT: Burn(user, target, amountToBurn, balanceIncrease, index);
Full repayment	repaid == user’s scaledBalance	`_totalSupply` reduced by scaled repaid amount `userState.balance` cleared `userState.index` updated to the current one	

3. Invariants Enforcement / Assumption Map

```
executeRepay()                              ENFORCED: #7 - burned debt tokens amount should be <= transferred repay amount
    |                                                    to the protocol
    |-> reserve.updateState()
    |    |-> _updateIndexes() -             ENFORCED: #8 - indices may update, but must remain monotonic and consistent
    |                                                     with accrued interest - reserve indices are monotonic
    |                                                     non-decreasing and independent of individual user repay
    |                                                     actions;
    |                                       PARTIALLY ENFORCED: #12 - a user's debt exposure is fully eliminated when
    |                                                                 scaled debt becomes zero, without mutating reserve
    |                                                                 indices - ensures indices remain consistent;
    |
    |-> ValidationLogic.validateRepay() -   ENFORCED: #2 - reserve should not be paused or frozen
    |                                       ENFORCED: #3 - repay() must revert if effective user debt is zero
    |
    |-> IStableDebtToken.burn() -           PARTIALLY ENFORCED: #13 - user state is updated correctly on full repay -
```

```
    |   |                                                    clear userStableRate and user timestamp
    |   |-> _burn() -                      ENFORCED: #5 - effective debt reduction <= user debt
    |                                      ENFORCED: #6 - burned debt tokens amount should be <= user debt
    |                                      PARTIALLY ENFORCED: #13 - user state is updated correctly on full repay - user
    |                                                    principal is updated
    |
    |-> IVariableDebtToken.burn()
        |-> _burnScaled() -                ENFORCED: #9 - protocol must not create unrepayable or irreducible debt dust
        |                                  PARTIALLY ENFORCED: #12 - a user's debt exposure is fully eliminated when
        |                                                    scaled debt becomes zero, without mutating reserve
        |                                                    indices - here dust/rounding is handled;
        |                                  PARTIALLY ENFORCED: #13 - user state is updated correctly on full repay -
        |                                                    userState index updated
        |
        |->_burn() -                       ENFORCED: #5 - effective debt reduction <= user debt
                                           ENFORCED: #6 - burned debt tokens amount should be <= user debt
                                           PARTIALLY ENFORCED: #13 - user state is updated correctly on full repay -
                                                    userState scaledBalance updated
```

V. Adversarial PoCs & Property Tests

Low-risk and design-level invariants are justified by call-graph analysis; state-dependent invariants are covered by unit or property tests.

Invariant #1 - `repay()` must not rely on oracle values for correctness

Enforced implicitly by design: `repay()` and all downstream calls do not read oracle prices or price-dependent state.

Invariant #2 - reserve should not be paused or frozen

- Severity / Impact: Medium / Low.
- Unit Test.

ValidationLogic.validateRepay() is asserted with the following reserve configurations setup:

```
reserveConfigurationActivePaused = DataTypes.ReserveConfigurationMap({
            data: (1<<56) | (1<<60)
        });

reserveConfigurationNotActiveNotPaused = DataTypes.ReserveConfigurationMap({
            data: 0
        });

reserveConfigurationActiveNotPaused = DataTypes.ReserveConfigurationMap({
            data: (1<<56)
        });
```

ValidationLogic.validateRepay() reverts for `paused || !active` and succeeds only for `active && !paused`.
Full test implementation: test/valya-test/Invariant_2.t.sol

Invariant #3 - repay() must revert if effective user debt is zero

- Severity / Impact: Low / Low.
- Unit Test.

ValidationLogic.validateRepay() is asserted with `stableDebt` and `variableDebt` = 0 and with the corresponding interestRateModes - STABLE / VARIABLE and tested for both STABLE and VARIABLE rate modes to cover distinct debt token paths.

Full test implementation: test/valya-test/Invariant_3.t.sol

Invariant #4 - anybody can repay `onBehalfOf` debt

Enforced implicitly. No threat attack because `repay()` only improves or doesn’t change the debtor's HF. Safe by design.