INVARIANT TESTING

CAP

GETRECON.XYZ
@NICANOR
@OXKNOT

Recon Invariant Testing

Introduction

Nican0r and 0xKnot performed 4 weeks of Invariant Testing on Cap. They wrote a Stateful Fuzzing Suite that is run with Echidna and can be debugged and run with Foundry as well as Halmos and Medusa.

Repo: https://github.com/cap-labs-dev/cap-contracts Commit Hash: 14d982ead2d67cfaf3e90c9032ccbdf2b9e9acb3

This review uses Code4rena Severity Classification

The Review is done as a best effort service. While a lot of time and attention was dedicated to the security review, it cannot guarantee that no bug is left.

As a general rule, we always recommend doing one additional security review until no bugs are found. This, in conjunction with a Guarded Launch and a Bug Bounty, can help further reduce the likelihood that any specific bug was missed.

About Recon

Recon offers boutique security reviews, invariant testing development, and is pioneering Cloud Fuzzing as a best practice by offering Recon Pro, the most complete tool to run tools such as Echidna, Medusa, Foundry, Kontrol, and Halmos in the cloud with just a few clicks.

About Nican0r

Nican0r has led many invariant testing engagements at Recon for projects such as:

- Liquity, Centrifuge, Badger, and Corn

Nican0r also authored some of the most read articles on the topic, published at: https://getrecon.substack.com/

About 0xKnot

A Software Engineer with 20 years of experience, Knot was formerly a Bot Racer at Code4rena, has built many tools used by Recon both internally and externally, and has been writing invariant testing suites for the last few months along with the team.

Additional Services by Recon

Recon offers:

- Invariant Testing Audits - We'll write your invariant tests, then perform an audit on them.
- Cloud Fuzzing as a Service - The easiest way to run invariant tests in the cloud. Ask about Recon Pro.
- Audits - High quality audits performed by highly qualified reviewers that work with Alex personally.

Table of Contents

- M-02 [Med] Vault redeem will always revert

- **Low**
  - L-01 [Low] Liquidation can worsen an agent's health factor
  - L-02 [Low] Full repay doesnt set isBorrowing to false and emit TotalRepayment
  - L-03 [Low] Permenant DOS of CapToken when using tokens with 18 decimals
  - L-04 [Low] `debtToken` `totalSupply` can be less than the total debt for a given asset

- **QA**
  - Q-01 [QA] incorrect use of `debtToken` for `totalInterest` value
  - Q-02 [QA] Custom revert when adding an already existing asset
  - Q-03 [QA] Missing access for Lender to delegation::distributeRewards function
  - Q-04 [QA] Missing grantAccess for addAsset, removeAsset, rescueERC20 and setWhitelist for Vault (in DeployVault.sol)

I-01 Properties Written

#	Function Name	Property Description
1	property_sum_of_deposits	Sum of deposits is less than or equal to total supply
2	property_sum_of_withdrawals	Sum of deposits + sum of withdrawals is less than or equal to total supply
3	property_vault_solvency_assets	totalSupplies for a given asset is always <= vault balance + totalBorrows + fractionalReserveBalance
4	property_vault_solvency_borrows	totalSupplies for a given asset is always >= totalBorrows
5	property_utilization_index_only_increases	Utilization index only increases
6	property_utilization_ratio	Utilization ratio only increases after a borrow or realizing interest
7	property_vault_balance_does_not_change_redeemAmountsOut	If the vault invests/divests it shouldn't change the redeem amounts out
8	property_agent_cannot_have_less_than_minBorrow_balance_of_debt_token	Agent can never have less than minBorrow balance of debt token
9	property_repaid_debt_equals_zero_debt	If all users have repaid their debt (have 0 DebtToken balance), reserve.debt == 0
10	property_borrowed_asset_value	loaned assets value < delegations value (strictly) or the position is liquidatable
11	property_health_not_changed_with_realizeInterest	health should not change when interest is realized
12	property_total_system_collateralization	System must be overcollateralized after all liquidations
13	property_delegated_value_greater_than_borrowed_value	Delegated value must be greater than borrowed value, if not the agent should be liquidatable
14	property_ltv	LTV is always <= 1e27
15	property_cap_token_backed_1_to_1	cUSD (capToken) must be backed 1:1 by stable underlying assets
16	doomsday_debt_token_solvency	DebtToken balance ≥ total vault debt at all times
17	property_total_borrowed_less_than_total_supply	Total cUSD borrowed < total supply (utilization < 1e27)
18	property_staked_cap_value_non_decreasing	Staked cap token value must increase or stay the same over time
19	property_utilization_ratio_never_greater_than_1e27	Utilization ratio is never greater than 1e27
20	property_maxWithdraw_less_than_loaned_and_reserve	sum of all `maxWithdraw` for users should be <= loaned + reserve
21	capToken_burn	User always receives at least the minimum amount out
22	capToken_burn	User always receives at most the expected amount out
23	capToken_burn	Total cap supply decreases by no more than the amount out
24	capToken_burn	Fees are always nonzero when burning
25	capToken_burn	Fees are always <= the amount out
26	capToken_burn	Burning reduces cUSD supply, must always round down
27	capToken_burn	Burners must not receive more asset value than cUSD burned
28	capToken_burn	User can always burn cap token if they have sufficient balance of cap token
29	capToken_divestAll	ERC4626 must always be divestable
30	capToken_mint	User always receives at least the minimum amount out
31	capToken_mint	User always receives at most the expected amount out
32	capToken_mint	Fees are always nonzero when minting
33	capToken_mint	Fees are always <= the amount out
34	capToken_mint	Minting increases vault assets based on oracle value
35	capToken_mint	User can always mint cap token if they have sufficient balance of depositing asset
36	capToken_mint	Asset cannot be minted when it is paused
37	capToken_redeem	User always receives at least the minimum amount out
38	capToken_redeem	User always receives at most the expected amount out

39	capToken_redeem	Total cap supply decreases by no more than the amount out
40	capToken_redeem	Fees are always <= the amount out
41	capToken_redeem	User can always redeem cap token if they have sufficient balance of cap token
42	doomsday_liquidate	Liquidate should always succeed for liquidatable agent
43	doomsday_liquidate	Liquidating a healthy agent should not generate bad debt
44	doomsday_repay	Repay should always succeed for agent that has debt
45	lender_borrow	Asset cannot be borrowed when it is paused
46	lender_borrow	Borrower should be healthy after borrowing (self-liquidation)
47	lender_borrow	Borrower asset balance should increase after borrowing
48	lender_borrow	Borrower debt should increase after borrowing
49	lender_borrow	Total borrows should increase after borrowing
50	lender_borrow	Borrow should never revert with arithmetic error
51	lender_initiateLiquidation	agent should not be liquidatable with health > 1e27
52	lender_initiateLiquidation	Agent should always be liquidatable if it is unhealthy
53	lender_liquidate	Liquidate should never revert with arithmetic error
54	lender_liquidate	Liquidation should be profitable for the liquidator
55	lender_liquidate	Agent should not be liquidatable with health > 1e27
56	lender_liquidate	Liquidations should always improve the health factor
57	lender_liquidate	Emergency liquidations should always be available when emergency health is below 1e27
58	lender_liquidate	Partial liquidations should not bring health above 1.25
59	lender_liquidate	Agent should have their totalDelegation reduced by the liquidated value
60	lender_liquidate	Agent should have their totalSlashableCollateral reduced by the liquidated value
61	lender_realizeInterest	agent's total debt should not change when interest is realized
62	lender_realizeInterest	vault debt should increase by the same amount that the underlying asset in the vault decreases when interest is realized
63	lender_realizeInterest	vault debt and total borrows should increase by the same amount after a call to `realizeInterest`
64	lender_realizeInterest	health should not change when `realizeInterest` is called
65	lender_realizeInterest	interest can only be realized if there are sufficient vault assets
66	lender_realizeInterest	realizeInterest should only revert with `ZeroRealization()` if paused or `totalUnrealizedInterest == 0`, otherwise should always update the realization value
67	lender_realizeRestakerInterest	vault debt should increase by the same amount that the underlying asset in the vault decreases when restaker interest is realized
68	lender_realizeRestakerInterest	vault debt and total borrows should increase by the same amount after a call to `realizeRestakerInterest`
69	lender_realizeRestakerInterest	health should not change when `realizeRestakerInterest` is called
70	lender_realizeRestakerInterest	restakerinterest can only be realized if there are sufficient vault assets
71	lender_repay	repay should never revert with arithmetic error
72	property_fractional_reserve_vault_has_reserve_amount_of_underlying_asset	fractional reserve vault must always have reserve amount of underyling asset
73	property_liquidation_does_not_increase_bonus	liquidation does not increase bonus
74	property_borrower_cannot_borrow_more_than_ltv	borrower can't borrow more than LTV
75	property_health_should_not_change_when_realizeRestakerInterest_is_called	health should not change when realizeRestakerInterest is called
76	property_no_operation_makes_user_liquidatable	no operation should make a user liquidatable
77	property_dust_on_repay	after all users have repaid their debt, their balance of `debtToken` should be 0

78	property_zero_debt_is_borrowing	if the debt token balance is 0, the agent should not be isBorrowing
79	property_agent_always_has_more_than_min_borrow	agent always has more than minBorrow balance of debtToken
80	property_lender_does_not_accumulate_dust	lender does not accumulate dust
81	property_debt_zero_after_repay	after all users have repaid their debt, the `reserve.debt` should be 0
82	doomsday_repay_all	repaying all debt for all actors transfers same amount of interest as would have been transferred by `realizeInterest`
83	doomsday_manipulate_utilization_rate	borrowing and repaying an amount in the same block shouldn't change the utilization rate
84	property_previewRedeem_greater_than_loaned	`previewRedeem(totalSupply)` >= `loaned`
85	capToken_divestAll	no assets should be left in the vault after divesting all
86	property_available_balance_never_reverts	available balance should never revert
87	property_maxBorrow_never_reverts	maxBorrowable should never revert
88	property_no_agent_borrowing_total_debt_should_be_zero	if no agent is borrowing, the total debt should be 0
89	property_no_agent_borrowing_utilization_rate_should_be_zero	if no agent is borrowing, the utilization rate should be 0
90	doomsday_maxBorrow	maxBorrowable after borrowing max should be 0
91	doomsday_maxBorrow	if no agent is borrowing, the current utilization index should be 0
92	doomsday_compound_vs_linear_accumulation	interest accumulation should be the same whether it's realized or not
93	property_debt_token_total_supply_greater_than_vault_debt	debtToken.totalSupply should never be less than reserve.debt
94	property_healthy_account_stays_healthy_after_liquidation	A healthy account (collateral/debt > 1) should never become unhealthy after a liquidation
95	property_no_bad_debt_creation_on_liquidation	A liquidatable account that doesn't have bad debt should not suddenly have bad debt after liquidation
96	lender_cancelLiquidation	cancelLiquidation should always succeed when health is above 1e27
97	lender_cancelLiquidation	cancelLiquidation should revert when health is below 1e27
98	property_maxLiquidatable_never_reverts	maxLiquidatable should never revert due to arithmetic error
99	property_bonus_never_reverts	bonus should never revert due to arithmetic error
100	property_staked_cap_total_assets_never_reverts	staked cap total assets should never revert due to arithmetic error

I-02 Suggested Next Steps

Executive Summary

Over the course of the 4 week engagement, the Recon team implemented 100 properties which would test the most important logic of the system and are outlined in the outlined in the properties-table file. This allowed us to uncover 2 medium and 4 low severity issues which were either fixed or acknowledge due to their low severity or ability to be worked around.

Recommendation

We recommend spending more time on the invariant tests: defining additional properties and adding optimization tests for the properties that broke.

Room for improvement

- properties checking for agent health should be made after all operations. This led to the discovery of issue 22 but refactoring/extending this property to exclude this case and check for others may prove fruitful in finding similar edge cases. - `StakedCap` and `DebtToken` could use further analysis and properties defined for them

Further considerations

- possible side effects of `debtToken` insolvency in issue 20
- ways to manipulate the utilization rate unfairly
- entire interest realization flow and possibility to game it so that it forces other users to pay more interest, avoid paying same interest as others, potential to force insolvency via interest realization
- edge cases related to the fee rate capping mechanism
- `MathUtils` calculations and how compound interest can affect overall system health
- `repay` flow could be broken down and have more specific properties defined for it

M-01 [Med] agent health changes after `realizeRestakerInterest`

If the `restakerRate` is reduced for an agent before restaker interest is realized it can cause the agent's `totalDebt` to increase and subsequently causes the agent's health to decrease after calling `realizeRestakerInterest`.

See the following reproducer which can be run on the `feat/recon` branch:

```
    function test_property_health_should_not_change_when_realizeRestakerInterest_is_called_6() public {
        switch_asset(0);

        // set initial rate to 0.5%
        oracle_setRestakerRate(0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, 0.05e27);

        capToken_mint_clamped(100711969);

        lender_borrow_clamped(1157920892373161954235709850086879078532699846656405640394575840079131296399 35);

        (,, uint256 totalDebtBefore,,, uint256 healthBefore) = _getAgentParams(_getActor());

        console2.log("rate before %e", oracle.restakerRate(_getActor()));
        oracle_setRestakerRate(0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, 33056249739822063734181);
        console2.log("rate after %e", oracle.restakerRate(_getActor()));

        vm.warp(block.timestamp + 56837);

        lender_realizeRestakerInterest();

        (,, uint256 totalDebtAfter,,, uint256 healthAfter) = _getAgentParams(_getActor());

        console2.log("totalDebtBefore %e", totalDebtBefore);
        console2.log("totalDebtAfter %e", totalDebtAfter);
        console2.log("healthBefore %e", healthBefore);
        console2.log("healthAfter %e", healthAfter);
        console2.log("maxDecreaseHealthDelta %e", maxDecreaseHealthDelta);
        console2.log("optimize_max_health_decrease %e", optimize_max_health_decrease());

        property_health_should_not_change_when_realizeRestakerInterest_is_called();
    }
```

M-02 [Med] Vault redeem will always revert

https://github.com/cap-labs-dev/cap-contracts/blob/8127e7284abd1a82d239458c77388de881ae1d8f/contracts/vault/Vault.sol#L136C5-L146C10

```
function redeem(uint256 _amountIn, uint256[] calldata _minAmountsOut, address _receiver, uint256 _deadline)
        external
        whenNotPaused
        returns (uint256[] memory amountsOut)
    {
        uint256[] memory fees;
        uint256[] memory totalDivestAmounts = new uint256[](amountsOut.length);
        (amountsOut, fees) = getRedeemAmount(_amountIn);
        for (uint256 i; i < amountsOut.length; i++) {
            totalDivestAmounts[i] = amountsOut[i] + fees[i];
        }
```

in line 142 we create an array with length of amountsOut (which is named return value of function so its length is 0) and in 145 when it tries to access totalDivestAmounts[i] it will give out-of-bounds all the time,

line 142 and 143 should swap

L-01 [Low] Liquidation can worsen an agent's health factor

Impact

One of our properties was: "Liquidations should always improve the health factor." However, due to price changes, this assumption can break. Below is a reproduction of the issue:

```
    function test_lender_liquidate_0() public {
        switchActor(1);

        capToken_mint_clamped(10005653326);

        lender_borrow(501317817);

        switchChainlinkOracle(2);

        mockChainlinkPriceFeed_setLatestAnswer(49869528211447337507581);

        lender_liquidate(1);
    }
```

Logs:

```
[FAIL: Liquidation did not improve health factor: 69998647576334689405074741 <= 699986475763401870594867624]
```

Rationale

Whenever an agent health is below 1e18, for each unit of debt, we'll remove 1 unit of collateral

This causes the Agent health to decrease

Additional Impact

Based on the premium chosen liquidators may chose to perform partial liquidations on Agents in order to increase the premium they receive

This should not substantially change the math, however it can result in paying out a higher premium than expected

L-02 [Low] Full repay doesnt set isBorrowing to false and emit TotalRepayment

https://github.com/Recon-Fuzz/cap-contracts/blob/14d982ead2d67cfaf3e90c9032ccbdf2b9e9acb3/contracts/lendingPool/libraries/BorrowLogic.sol#L109-L112

The reason is that we burn the debtToken at the end of this function, so the balance becomes zero after repaid is fully processed.

The check should be `balanceOf == repaid` instead of `balanceOf == 0`, because we burn the repaid amount at the end. So, if the user's balance is equal to repaid, they will have zero debt balance after the burn. or we move the check after the burn.

```
    function test_property_zero_debt_is_borrowing_0() public {
        capToken_mint_clamped(1210366228196525416932125);

        lender_borrow_clamped(381970873);

        lender_repay(381970873);

        property_zero_debt_is_borrowing();
    }
```

L-03 [Low] Permenant DOS of CapToken when using tokens with 18 decimals

Summary

Calling `mint` followed by `redeem` can bring the price of `capToken` down to zero. Once this happens, all subsequent `mint`, `burn`, or `redeem` operations **revert** due to the `getPrice` function reverts.

Reproduction

```
function test_capToken_burn_0() public {
    capToken_mint_clamped(10254117454);
    capToken_redeem_clamped(3158103005);

    switchActor(1);
    capToken_mint_clamped(1e18); // => reverts
}
```

Root Cause

The problem originates in the `price()` function of `CapTokenAdapter.sol`:

```
totalUsdValue += supply * assetPrice / supplyDecimalsPow;
```

When an asset has a very low supply (< 1e10) and high decimals (anything more than 8, like 18), the multiplication of `supply * assetPrice` (where `assetPrice` has 8 decimals) becomes smaller than `supplyDecimalsPow = 1e18`. This causes the result of the division to round down to zero, so the `totalUsdValue` becomes 0.

Later, the final `latestAnswer` becomes:

```
latestAnswer = totalUsdValue * decimalsPow / capTokenSupply; // => 0
```

As a result:

latestAnswer becomes 0

This causes `getPrice()` to revert, due to the following check:

```
            if (price == 0 || _isStale(_asset, lastUpdated)) revert PriceError(_asset);
```

Any function that depends on `getPrice()` — such as `mint`, `burn`, or `redeem` — will start reverting

The system becomes non-recoverable

```
function price(address _asset) external view returns (uint256 latestAnswer, uint256 lastUpdated) {
        uint256 capTokenSupply = IERC20Metadata(_asset).totalSupply();
        if (capTokenSupply == 0) return (1e8, block.timestamp);

        address[] memory assets = IVault(_asset).assets();
        lastUpdated = block.timestamp;

        uint256 totalUsdValue;

        for (uint256 i; i < assets.length; ++i) {
            address asset = assets[i];
            uint256 supply = IVault(_asset).totalSupplies(asset);
            uint256 supplyDecimalsPow = 10 ** IERC20Metadata(asset).decimals();
            (uint256 assetPrice, uint256 assetLastUpdated) = IOracle(msg.sender).getPrice(asset);

            totalUsdValue += supply * assetPrice / supplyDecimalsPow; <= @@@@ audit
            if (assetLastUpdated < lastUpdated) lastUpdated = assetLastUpdated;
        }

        uint256 decimalsPow = 10 ** IERC20Metadata(_asset).decimals();
        latestAnswer = totalUsdValue * decimalsPow / capTokenSupply;
    }
```

L-04 [Low] `debtToken` `totalSupply` can be less than the total debt for a given asset

See the following reproducer test_property_debt_token_balance_gte_total_vault_debt_1:

```
    function test_property_debt_token_balance_gte_total_vault_debt_1() public {
        capToken_mint_clamped(10000718111);


        lender_borrow(100014444, 0x00000000000000000000000000000000DeaDBeef);


        vm.warp(block.timestamp + 6);


        vm.roll(block.number + 1);


        switchActor(1);

lender_borrow_clamped(115792089237316195423570985008687907853269984665640564039457584007913129639935);

       property_debt_token_balance_gte_total_vault_debt();
    }
```

which shows that the sum of debts for all agents (as reported by `ViewLogic::debt`) can be greater than the `totalSupply` of the `debtToken`.

Q-01 [QA] incorrect use of `debtToken` for `totalInterest` value

TODO: determine what the correct calculation would be/if it's actually overestimating.

In `ViewLogic`:

```
    function accruedRestakerInterest(ILender.LenderStorage storage $, address _agent, address _asset)
        public
        view
        returns (uint256 accruedInterest)
    {
        ILender.ReserveData storage reserve = $.reservesData[_asset];
        uint256 totalInterest = IERC20(reserve.debtToken).balanceOf(_agent);
        uint256 rate = IOracle($.oracle).restakerRate(_agent);
        uint256 elapsedTime = block.timestamp - reserve.lastRealizationTime[_agent];

        accruedInterest = totalInterest * rate * elapsedTime / (1e27 * SECONDS_IN_YEAR);
    }
```

the `totalInterest` is taken to be the user's balance of the `debtToken`, but this is actually just their total outstanding debt, not the total interest that they have to pay on their debt.

In `realizeRestakerInterest` the debtToken balance of the user is increased by the amount of realized and unrealized interest:

```
    function realizeRestakerInterest(ILender.LenderStorage storage $, address _agent, address _asset)
        public
        returns (uint256 realizedInterest)
    {
        ILender.ReserveData storage reserve = $.reservesData[_asset];
        uint256 unrealizedInterest;
        (realizedInterest, unrealizedInterest) = maxRestakerRealization($, _agent, _asset);
        ...

        IDebtToken(reserve.debtToken).mint(_agent, realizedInterest + unrealizedInterest);
        IVault(reserve.vault).borrow(_asset, realizedInterest, $.delegation);
        IDelegation($.delegation).distributeRewards(_agent, _asset);
        emit RealizeInterest(_asset, realizedInterest, $.delegation);
    }
```

this seems like it would lead to the calculation of `accruedRestakerInterest` to be greater than it should.

Q-02 [QA] Custom revert when adding an already existing asset

`addAsset` reverts with `AssetNotSupported` instead of the more appropriate `AssetAlreadySupported` when adding an asset that's already contained in the set:

```
    function addAsset(IVault.VaultStorage storage $, address _asset) external {
        if (!$.assets.add(_asset)) revert AssetNotSupported(_asset);
        emit AddAsset(_asset);
    }
```

Q-03 [QA] Missing access for Lender to delegation::distributeRewards function

```
+        accessControl.grantAccess(Delegation.setLastBorrow.selector, infra.delegation, infra.lender);
```

https://github.com/Recon-Fuzz/cap-contracts/blob/71951b1b1c574a9ce85eb6c097959e5ffb21dfd2/contracts/deploy/service/ConfigureAccessControl.sol#L4

Q-04 [QA] Missing grantAccess for addAsset, removeAsset, rescueERC20 and setWhitelist for Vault (in DeployVault.sol)

```
 function _initVaultAccessControl(InfraConfig memory infra, VaultConfig memory vault, UsersConfig memory users)
        internal
    {
...
+       accessControl.grantAccess(Vault.addAsset.selector, vault.capToken, users.vault_config_admin);
+       accessControl.grantAccess(Vault.removeAsset.selector, vault.capToken, users.vault_config_admin);
+       accessControl.grantAccess(Vault.rescueERC20.selector, vault.capToken, users.vault_config_admin);
+       accessControl.grantAccess(Minter.setWhitelist.selector, vault.capToken, users.vault_config_admin);
```

https://github.com/cap-labs-dev/cap-contracts/blob/0f4085d7f1f24f53144cc6df2d6e4d70b61ff97a/contracts/deploy/service/DeployVault.sol#L105