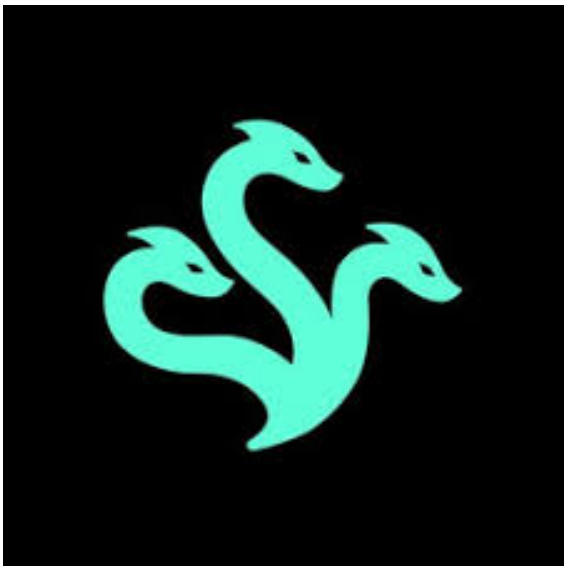
Hybra Finance: Post-Audit Protocol Report



Audited by: rubencrxz

Table of Contents

---

1. Protocol Overview

Hybra Finance is a ve(3,3) decentralized exchange protocol forked from Blackhole, which itself builds upon Solidly's vote-escrow model and Uniswap V3's concentrated liquidity AMM. The protocol is deployed on HyperEVM.

ve(3,3) Model: Core Characteristics

The name ve(3,3) combines two ideas that typically appear together in gauge-based DEXs:

1. ve (vote-escrow)

A Curve Finance-style mechanism: the user locks the base token (e.g. HYBR) for a period *T* and receives a ve position (in many designs this is a veNFT, in others it is not). This position grants voting power and/or boost that decays linearly toward 0 as the unlock date approaches.

Typical mental rule:

- More tokens locked and longer duration → higher initial vePower.
- vePower decreases over time (linear).

2. (3,3)

A cultural reference popularized by OlympusDAO: it represents a coordination incentive — if the majority "cooperates" by locking/participating in the system instead of selling, the aggregate outcome tends to be better. In ve(3,3) it is used as a narrative to align LPs, voters, and protocols; it is not a universal mathematical guarantee, but rather an incentive design.

Typical ve(3,3) Pillars in a DEX:

Pillar	Mechanism
Lock-to-Vote	Lock tokens → receive a ve position / veNFT with linearly decaying voting power
Gauge Voting	ve-holders assign weights to gauges/pools per epoch (often weekly) to determine where emissions go
Proportional Emissions	Each gauge receives emissions ∝ vote weight, then distributed to the gauge's LPs/stakers according to protocol rules
Bribes	Third parties (other protocols, projects) deposit incentives into "bribe markets" to buy votes toward their pools/gauges
Anti-dilution / Rebase (optional)	Many forks include an emission allocation or "rebase" for ve-holders to mitigate inflation dilution; not mandatory in all ve(3,3) designs
Flywheel	More liquidity → more volume/fees → more bribing capacity + more attractive to vote → more emissions to those pools → more liquidity

The system comprises two independently compiled codebases:

- `cl/` -- Concentrated Liquidity AMM (Solidity 0.7.6), a modified Uniswap V3 fork.
- `ve33/` -- Vote-escrowed tokenomics, gauges, emission distribution, and governance (Solidity 0.8.13).

The `ve33` codebase directly integrates `cl` contracts via compiled bytecode export, since the two pragma versions are incompatible for co-compilation.

In-scope contracts (14 files, ~6,082 nSLOC total):

Contract	Lines	Module
`VotingEscrow.sol`	1,355	ve33
`CLPool.sol`	1,079	cl

Contract	Lines	Module
GovernanceHYBR.sol	635	ve33
GaugeManager.sol	555	ve33
GaugeV2.sol	410	ve33
GaugeCL.sol	373	ve33
RewardHYBR.sol	339	ve33
CLFactory.sol	273	cl
VoterV3.sol	263	ve33
MinterUpgradeable.sol	223	ve33
DynamicSwapFeeModule.sol	214	cl
HybrSwapper.sol	160	ve33
HYBR.sol	102	ve33
GaugeFactoryCL.sol	101	ve33

2. Architecture

The high-level system architecture follows the ve(3,3) model:

```
HYBR (base ERC-20 token)
  |
  v
VotingEscrow (lock HYBR --> veNFT with time-decaying voting power)
  |
  +---> VoterV3 (allocate veNFT votes to pool gauges)
  |       |
  |       v
  +---> GaugeManager (create gauges, distribute emissions proportional to votes)
  |       |
  |       +---> GaugeV2 (V2 AMM LP staking, Synthetix-style rewards)
  |       +---> GaugeCL (CL NFT position staking, tick-range-aware rewards)
  |       +---> Bribe contracts (internal = trading fees, external = incentives)
  |
  +---> MinterUpgradeable (weekly emission with 1% decay, rebase, team share)
  |       |
  |       +---> RewardsDistributor (rebase to veNFT holders -- out of scope)
  |       +---> GaugeManager.notifyRewardAmount (gauge emissions)
  |
  +---> RewardHYBR (rHYBR -- non-transferable intermediary reward token)
  |       |
  |       +---> Redeem to HYBR (with penalty, default 20%)
  |       +---> Redeem to veHYBR (1:1, permanent max lock)
  |       +---> Redeem to gHYBR (1:1)
```

```
    |
    +---> GovernanceHYBR (gHYBR -- auto-compounding vault wrapping a single veNFT)
            |
            +---> HybrSwapper (convert non-HYBR bribe rewards to HYBR)
```

---

3. Codebase Composition

Compiler Configuration

Module	Solidity	Optimizer	Runs	EVM Target	IR Pipeline
`cl`	0.7.6	Yes	10	default	No
`ve33`	0.8.13	Yes	200	london	Yes (`via_ir`)

Dependency Resolution

- **cl**: Uses `forge` with git submodule libs (`forge-std`, `openzeppelin-contracts`, `ExcessivelySafeCall`, `solidity-lib`, `base64`).
- **ve33**: Uses `npm` dependencies (`@openzeppelin/contracts`, `@openzeppelin/contracts-upgradeable`, `@cryptoalgebra/*`) alongside `forge-std` git submodule.

Remappings in `ve33/foundry.toml`:

```
@openzeppelin/contracts/ -> node_modules/@openzeppelin/contracts/
@openzeppelin/contracts-upgradeable/ -> node_modules/@openzeppelin/contracts-
upgradeable/
@cryptoalgebra/integral-core/ -> node_modules/@cryptoalgebra/integral-core/
```

---

4. Concentrated Liquidity (CL) Module

4.1 CLPool

`CLPool` implements the `ICLPool` interface, composed of six sub-interfaces: `ICLPoolConstants`, `ICLPoolState`, `ICLPoolDerivedState`, `ICLPoolActions`, `ICLPoolEvents`, and `ICLPoolOwnerActions`.

Pools are deployed as **EIP-1167 minimal proxy clones** by `CLFactory` using `Clones.cloneDeterministic`. The implementation contract is shared and each pool is initialized post-deployment via `initialize()` rather than a constructor.

Core functions follow the Uniswap V3 pattern:

- `swap()` (lines 682-878): Iterative tick-crossing loop. Reads `slot0` into memory, fetches the dynamic fee from the factory on each call. For each step, calls `SwapMath.computeSwapStep()`. Fee distribution uses the three-way `calculateFees()` split (see Section 6). On tick crossings, updates reward growth lazily (only on first crossing), then calls `ticks.cross()` which flips outside trackers for fees, rewards, and

time-weighted accumulators. The swap updates both `liquidity` and `stakedLiquidity` independently.

- **`mint()`** (lines 453-481): Calls `_modifyPosition` with positive `liquidityDelta`. Uses the callback pattern (`ICLMintCallback`) and balance-check verification.
- **`burn()`** (lines 547-589): Calls `_modifyPosition` with negative `liquidityDelta`. Has two overloads: one for `msg.sender` as owner, one for the NFT manager (`onlyNftManager` modifier).

Built-in Gauge Staking (major deviation from Uniswap V3):

The pool natively tracks `stakedLiquidity` and distributes gauge rewards via a Synthetix-style streaming model:

- `stake()` is called by the gauge. It updates `stakedLiquidity` for the current tick range, transfers position ownership virtually from the NFT manager's position to the gauge's position, and updates `stakedLiquidityNet` on boundary ticks.
- `_updateRewardsGrowthGlobal()` accumulates `rewardRate * timeDelta` into `rewardGrowthGlobalX128`, distributed per unit of `stakedLiquidity`. If no staked liquidity exists, rewards accumulate in `rollover`.
- `syncReward()` is called by the gauge to set new reward parameters and clear rollover.

4.2 CLFactory

`CLFactory` acts as the central pool deployer and **fee oracle**. It uses deterministic CREATE2 deployment with `salt = keccak256(abi.encode(token0, token1, tickSpacing))`.

Default fee tiers:

Tick Spacing	Default Fee
1	100 (0.01%)
50	500 (0.05%)
100	500 (0.05%)
200	3,000 (0.30%)
2,000	10,000 (1.00%)

Three fee types are resolved through pluggable fee modules:

- **`getSwapFee(pool)`**: Queries `swapFeeModule` via `excessivelySafeStaticCall` (200k gas limit, 32 bytes return). Falls back to `tickSpacingToFee[pool.tickSpacing()]` if module fails or returns > 10%.
- **`getUnstakedFee(pool)`**: Only active when gauge IS alive. Queries `unstakedFeeModule`, falls back to `defaultUnstakedFee` (100,000 = 10%).
- **`getProtocolFee(pool)`**: Only active when gauge is NOT alive. Queries `protocolFeeModule`, falls back to `defaultProtocolFee` (250,000 = 25%).

Protocol fee and unstaked fee are **mutually exclusive** based on gauge liveness.

4.3 Three-Way Fee Split

`CLPool.calculateFees()` (lines 1015-1038) splits swap fees into three buckets:

1. **All staked** (`liquidity == stakedLiquidity`): All fees go to `stakedFeeAmount` (gauge collects).
2. **All unstaked** (`stakedLiquidity == 0`): `applyUnstakedFees` taxes the full fee.
3. **Mixed**: `splitFees` divides proportionally by staked/unstaked liquidity ratio; unstaked portion is additionally taxed.

`applyUnstakedFees()` (lines 996-1012) operates in two modes:

- **Gauge alive** (`protocolFee == 0`): Percentage of unstaked fees redirected to the gauge/staked bucket.
- **Gauge dead** (`protocolFee > 0`): Percentage of unstaked fees taken as protocol fee.

This creates an economic incentive: unstaked LPs always pay a penalty that goes either to staked LPs or to the protocol.

---

5. ve(3,3) Tokenomics Module

5.1 HYBR Token

Hand-rolled ERC-20 (no OpenZeppelin base). Supply is controlled by a single `minter` address. `initialMint()` is a one-shot 500M HYBR distribution. `burn()` and `burnFrom()` are publicly accessible. The `minter` role is transferred via `setMinter()`, callable only by the current minter.

5.2 VotingEscrow

Hand-rolled ERC-721 implementing Curve-style vote-escrow. Users lock HYBR to receive veNFTs with time-decaying voting power (`bias - slope * elapsed`).

Lock types:

- **Standard locks**: Linear voting power decay from lock creation to expiry (`lock_end`). `MAX_LOCK` determines the maximum lock duration.
- **Permanent locks**: `lockPermanent()` sets a non-decaying lock. Voting power equals the locked amount at all times. Unlockable only by the owner via `unlockPermanent()`.

Position operations:

- `create_lock` / `create_lock_for`: Create new veNFT with specified amount and duration.
- `increase_amount` / `increase_unlock_time`: Modify existing lock.
- `merge`: Combine two veNFTs. Handles permanent-to-permanent merges.
- `multiSplit`: Split one veNFT into 2-10 new NFTs proportional to a weight array.
- `withdraw`: Claim underlying HYBR after lock expiry.

Checkpoint system:

- Per-user point history: `user_point_history[tokenId][epoch]` stores `(bias, slope, ts, blk, permanent)`.
- Global point history: `point_history[epoch]` tracks aggregate supply decay.
- `slope_changes` mapping records scheduled global slope decrements at future week boundaries.

- Binary search lookups for historical balance/supply queries.

Libraries used:

- `VotingBalanceLogic`: Encapsulates `balanceOfNFT`, `balanceOfAtNFT`, `totalSupplyAtT`, `totalSupplyAt` with binary search.
- `VotingDelegationLib`: EIP-712 compliant delegation with checkpoint-based token-ID-level tracking. `MAX_DELEGATES = 1024`.
- `HybraTimeLibrary`: Epoch/week math utilities.

Flash-NFT protection: `ownership_change[_tokenId] == block.number` returns 0 voting power, preventing flash-loan governance attacks.

Auto-poke on deposit: When `deposit_for` or `increase_amount` is called on a voted token, the contract calls `IVoter(voter).poke(_tokenId)` to update vote weights with the new balance.

5.3 VoterV3

Upgradeable (`OwnableUpgradeable`, `ReentrancyGuardUpgradeable`). Handles vote allocation from veNFTs to pool gauges.

- `vote()`: Normalizes weights proportional to veNFT balance, deposits into internal and external bribes. Restricted to one vote per epoch via `onlyNewEpoch` modifier.
- `reset()`: Removes all votes for a tokenId, withdraws from bribes.
- `poke()`: Re-casts existing votes with updated veNFT balance. Callable by owner/approved OR the VotingEscrow contract.
- `lastVoted` is set to `epochStart + 1` (not actual timestamp) to ensure epoch-based vote-once logic.
- Dead gauge votes are skipped during `_vote` but their weight still counts in normalization.

5.4 GaugeManager

Upgradeable. Central orchestrator for gauge creation and emission distribution.

- **Gauge creation**: `_createGauge()` supports type 0 (V2 AMM) and type 1 (CL). Validates token whitelisting via `ITokenHandler`. Creates internal + external bribe contracts via `IBribeFactory`.
- **Emission distribution**: `notifyRewardAmount()` (called by Minter) distributes proportional to vote weights using a global `index` pattern (Synthetix-derived). `_distribute()` sends claimable emissions to gauges.
- **Kill/Revive**: Dead gauges return their claimable to minter; revived gauges resume from the current index.
- CL gauges receive special treatment: `ICLPool.setGaugeAndPositionManager()` is called during creation to link the pool to its gauge.

5.5 GaugeV2 (V2 AMM Gauge)

Non-upgradeable (`ReentrancyGuard`, `Ownable`). Standard Synthetix-style staking: users deposit LP tokens, earn `rewardToken` over `DURATION`.

Reward redemption goes through rHYBR (see Section 7): `getReward()` mints rHYBR via `IRHYBR(rHYBR).depostionEmissionsToken(reward)`, then calls `redeemFor(reward, _redeemType,`

`_user)`.

5.6 GaugeCL (CL Gauge)

Non-upgradeable. Users deposit NFT positions from `NonfungiblePositionManager`.

- **Deposit**: Validates position matches this gauge's pool, collects pending fees to user, transfers NFT, calls `clPool.stake()`.
- **Reward calculation**: Uses `rewardGrowthInside` (per-tick reward accumulation, leveraging `FullMath` and `FixedPoint128` for Q128 precision).
- **notifyRewardAmount**: Handles rollover of unspent rewards. Syncs reward state to the CL pool.
- Same rHYBR redemption pattern as GaugeV2.

5.7 MinterUpgradeable

Controls the emission schedule:

- **Weekly decay**: `calculate_emission() = weekly * 9900 / 10000` (99% of previous week).
- **Tail emission floor**: `circulating_emission() = totalSupply * 25 / 10000` (0.25% of circulating supply).
- **Effective emission**: `max(calculate_emission(), circulating_emission())`.
- **Rebase**: Proportional anti-dilution for veNFT holders based on `lockedShare`, capped at 30%.
- **Distribution**: 5% to team, rebase to RewardsDistributor, remainder to GaugeManager.
- Starting weekly emission: 2.6M HYBR.
- `circulating_supply()` excludes VE-locked tokens AND burned tokens (address `0xdead`).

---

6. Dynamic Fee System

DynamicSwapFeeModule

Pluggable module implementing `IDynamicFeeModule`. The factory calls `getFee(pool)` on each swap to compute the fee dynamically.

Fee formula:

```
totalFee = baseFee + dynamicComponent
totalFee = min(totalFee, feeCap)
totalFee = totalFee - discount   (if tx.origin is discounted)
```

Dynamic component calculation (`_getDynamicFee`):

1. Read current tick and observation cardinality from the pool's `slot0`.
2. Compute TWAP tick over `secondsAgo` (default 600s = 10 minutes) using the pool's oracle.
3. `absTickDelta = |currentTick - twAvgTick|`
4. `dynamicFee = absTickDelta * scalingFactor / SCALING_PRECISION`

Higher price volatility (larger tick deviation from TWAP) produces higher fees.

Per-pool configuration (`DynamicFeeConfig` struct, packed in a single storage slot):

Field	Type	Purpose
`baseFee`	uint24	Base fee in pips
`feeCap`	uint24	Maximum total fee
`scalingFactor`	uint64	Volatility sensitivity multiplier

Global defaults: `defaultScalingFactor`, `defaultFeeCap`, `secondsAgo` (TWAP lookback period).

Special value: `ZERO_FEE_INDICATOR = 420` -- sentinel for explicitly setting 0% fee.

Discount system: The `discounted` mapping allows the `swapFeeManager` to register addresses (keyed by `tx.origin`) that receive a percentage discount on fees. Maximum discount: 50%. Using `tx.origin` means the discount applies based on the EOA that initiated the transaction, not the direct caller.

Safety measures:

- `excessivelySafeStaticCall` pattern from the factory with gas limits prevents griefing.
- Oracle failures handled gracefully via `try/catch` (returns 0 dynamic component).
- Insufficient oracle data (cardinality < `secondsAgo / MIN_SECONDS_AGO`) returns 0 dynamic fee.

---

7. rHYBR Intermediary Reward Layer

`RewardHYBR` (rHYBR) is a **non-transferable** ERC-20 that acts as an intermediary between gauge emissions and users. This is Hybra's key differentiator from standard ve(3,3) implementations.

Flow:

1. Gauges earn HYBR emissions from GaugeManager.
2. When users claim rewards, gauges deposit HYBR into rHYBR contract (`depostionEmissionsToken`), receiving 1:1 rHYBR.
3. Users choose a redemption path:

Redemption Type	Ratio	Destination
`TO_HYBR` (0)	`fixedConversionRate / 10000` (default 80%)	Liquid HYBR to user
`TO_VEHYBR` (1)	1:1	New veNFT with max lock for user
`TO_GHYBR` (2)	1:1	gHYBR vault deposit for user

The 20% penalty from `TO_HYBR` redemption flows to the gHYBR contract via `receivePenaltyReward()` and compounds into its veNFT, creating a flywheel: penalties increase gHYBR backing, increasing its share value.

Transfer restrictions: `transfer()`, `transferFrom()`, and `approve()` all revert. Internal transfers (minting/burning/gauge interactions) are whitelisted via `exempt` and `exemptTo` sets.

---

8. gHYBR Auto-Compounding Vault

`GovernanceHYBR` (gHYBR) is an ERC-4626-like vault that wraps all deposited HYBR into a single permanent-lock veNFT.

Deposit: Users deposit HYBR, receive gHYBR shares (`shares = amount * totalSupply / totalAssets`, 1:1 for first deposit). Deposited HYBR is added to the contract's permanent-lock veNFT.

Withdrawal: Uses `multiSplit` on the veNFT to carve out proportional amounts. Creates 3 new veNFTs: remaining (stays with contract), user amount, and fee amount. Users receive a veNFT (not raw HYBR). A configurable withdrawal fee (default 1%, range 0.1%-10%) creates a veNFT for the `Team` address. Withdrawals are restricted to specific time windows within each epoch.

Auto-compounding flow (operator-driven):

1. `claimRewards()` -- collect rebase + bribe rewards.
2. `executeSwap()` -- convert non-HYBR rewards to HYBR via `HybrSwapper`.
3. `compound()` -- deposit accumulated HYBR back into the veNFT.

Voting: The operator/owner can vote and reset via the Voter contract using the contract's veNFT.

Transfer locks: New deposits are locked for `transferLockPeriod` (configurable 1-240 minutes) enforced via `_beforeTokenTransfer`.

---

9. Storage Layout and Slot Packing

CLPool Slot0 (single 256-bit word)

Field	Type	Bits
`sqrtPriceX96`	uint160	160
`tick`	int24	24
`observationIndex`	uint16	16
`observationCardinality`	uint16	16
`observationCardinalityNext`	uint16	16
`unlocked`	bool	8
Total		**240 bits**

Deviation from Uniswap V3: The `feeProtocol` field is removed. Fees are computed dynamically on each call via the factory.

CLPool Additional Packed Slots

Slot	Fields	Packing
GaugeFees	`token0: uint128` + `token1: uint128`	256 bits
ProtocolFees	`token0: uint128` + `token1: uint128`	256 bits

Slot	Fields	Packing
Slot 16	`stakedLiquidity: uint128` + `lastUpdated: uint32` + `tickSpacing: int24`	184 bits
Slot 17	`liquidity: uint128` + `maxLiquidityPerTick: uint128`	256 bits

Tick.Info Struct (per tick, in mapping)

Slot	Fields	Packing
0	`liquidityGross: uint128` + `liquidityNet: int128`	256 bits
1	`stakedLiquidityNet: int128`	128 bits (128 wasted)
2	`feeGrowthOutside0X128: uint256`	256 bits
3	`feeGrowthOutside1X128: uint256`	256 bits
4	`rewardGrowthOutsideX128: uint256`	256 bits
5	`tickCumulativeOutside: int56` + `secondsPerLiquidityOutsideX128: uint160` + `secondsOutside: uint32` + `initialized: bool`	256 bits

Added vs Uniswap V3: `stakedLiquidityNet` (int128) and `rewardGrowthOutsideX128` (uint256).

Position.Info Struct

Slot	Fields	Packing
0	`liquidity: uint128`	128 bits
1	`feeGrowthInside0LastX128: uint256`	256 bits
2	`feeGrowthInside1LastX128: uint256`	256 bits
3	`tokensOwed0: uint128` + `tokensOwed1: uint128`	256 bits

The `update()` function takes a `bool staked` parameter. When `staked == true`, fee accrual is skipped -- staked positions do not accumulate swap fees individually; their fee share goes to the gauge's `gaugeFees` accumulator.

Oracle.Observation Struct (ring buffer of 65,535)

Field	Type	Bits
`blockTimestamp`	uint32	32
`tickCumulative`	int56	56
`secondsPerLiquidityCumulativeX128`	uint160	160
`initialized`	bool	8

Field	Type	Bits
Total		**256 bits**

DynamicFeeConfig (per pool, single slot)

Field	Type	Bits
`baseFee`	uint24	24
`feeCap`	uint24	24
`scalingFactor`	uint64	64
Total		**112 bits**

10. Libraries and Dependencies

CL Module Libraries (Solidity 0.7.6)

Library	Source	Purpose
`TickMath`	Uniswap V3	`tick <-> sqrtPriceX96` conversions (MIN_TICK: -887272, MAX_TICK: 887272)
`SqrtPriceMath`	Uniswap V3	Next sqrt price from amounts, token deltas between prices
`SwapMath`	Uniswap V3	Single-step swap computation with fee deduction
`FullMath`	Uniswap V3	512-bit multiply-divide (Chinese Remainder Theorem + Newton-Raphson)
`Tick`	Modified	Tick state with added `stakedLiquidityNet`, `rewardGrowthOutsideX128`
`TickBitmap`	Uniswap V3	Bitmap for initialized tick discovery
`Position`	Modified	Position state with `staked` flag skipping fee accrual
`Oracle`	Uniswap V3	TWAP ring buffer (65,535 observations)
`LowGasSafeMath`	Uniswap V3	Gas-optimized safe math for uint256/int256
`SafeCast`	Uniswap V3	Safe casting between uint/int sizes
`LiquidityMath`	Uniswap V3	Safe liquidity add/subtract
`FixedPoint128` / `FixedPoint96`	Uniswap V3	Q128 and Q96 constants
`TransferHelper`	Uniswap V3	Safe ERC20 transfers
`UnsafeMath`	Uniswap V3	Unchecked division for gas savings
`BitMath`	Uniswap V3	Most/least significant bit operations

Library	Source	Purpose
`ExcessivelySafeCall`	Nomad	Gas-limited safe external calls for fee module queries
`Clones`	OpenZeppelin	EIP-1167 minimal proxy deployment
`EnumerableSet`	Custom	Set data structure for enumeration
`ProtocolTimeLibrary`	Custom	Epoch boundary and time window calculations

ve33 Module Libraries (Solidity 0.8.13)

Library	Source	Purpose
`VotingBalanceLogic`	Custom	`balanceOfNFT`, `totalSupplyAtT` with binary search through point history
`VotingDelegationLib`	Custom	Checkpoint-based delegation (MAX_DELEGATES = 1024)
`VoterFactoryLib`	Custom	Pair/gauge factory array management
`HybraTimeLibrary`	Custom	Epoch/week time math
`Math`	Custom	General math utilities
`FullMath`	Uniswap V3 (copy)	Used by GaugeCL for reward precision
`FixedPoint128`	Uniswap V3 (copy)	Q128 constant for CL reward math
`SafeCast`	Copy	Integer casting safety
`EnumerableSet`	OpenZeppelin	Used by RewardHYBR for whitelist sets

External Dependencies

Package	Version Context	Used By
`@openzeppelin/contracts`	0.8.x	ve33 (ERC20, Ownable, ReentrancyGuard, Pausable)
`@openzeppelin/contracts-upgradeable`	0.8.x	ve33 (OwnableUpgradeable, ReentrancyGuardUpgradeable)
`openzeppelin-contracts` (git submodule)	0.7.6 compat	cl (Clones)
`ExcessivelySafeCall` (git submodule)	-	cl (CLFactory)
`forge-std`	-	Both (testing)

11. Deployment and Cross-Compilation Strategy

The two modules use incompatible Solidity pragma versions:

- **cl**: `pragma solidity =0.7.6` -- locked to exactly 0.7.6 (Uniswap V3 requirement).
- **ve33**: `pragma solidity 0.8.13` -- uses 0.8.x with `via_ir` compilation pipeline.

Bytecode bridge: A Foundry script (`cl/script/ExportDeployments.s.sol`) compiles all CL contracts and exports their bytecode to a JSON file. The `ve33` test suite imports this JSON to deploy CL contracts at test time, with storage slots directly mutated to replicate constructor initialization.

Pool deployment: CL pools are deployed as EIP-1167 minimal proxy clones (`Clones.cloneDeterministic`) from a single `poolImplementation` contract. Deterministic addresses are derived from `keccak256(abi.encode(token0, token1, tickSpacing))`.

---

12. Access Control Model

Core Multisigs

Role	Description
`hybraMultisig`	Main 4/6 multisig; can add/remove roles
`hybraTeamMultisig`	Team 2/2 multisig
`emergencyCouncil`	Emergency functions control

Role-Based Access (via PermissionsRegistry)

Role	Scope
`GOVERNANCE`	High-level governance decisions, kill/revive gauges
`VOTER_ADMIN`	Voter contract settings
`GAUGE_ADMIN`	Gauge creation, factory management
`BRIBE_ADMIN`	Bribe contract management

Per-Contract Privileged Roles

Contract	Role	Capabilities
HYBR	`minter`	Token minting, minter transfer
VotingEscrow	`team`	Set voter/artProxy, toggle split, partner NFTs
VotingEscrow	`voter`	Mark tokens voted/abstained, attach/detach
CLFactory	`owner`	Set fee modules, collect protocol fees
CLFactory	`swapFeeManager`	Manage swap fee module and defaults
CLFactory	`unstakedFeeManager`	Manage unstaked fee module
CLFactory	`protocolFeeManager`	Manage protocol fee module

Contract	Role	Capabilities
MinterUpgradeable	`team`	Emission parameters, team rate, two-step transfer
GovernanceHYBR	`owner`	Administrative settings
GovernanceHYBR	`operator`	Claim, compound, swap, vote operations
RewardHYBR	`owner`	Rate setting, whitelist, pause, emergency

13. HyperEVM Considerations

The codebase targets HyperEVM as its deployment chain, as evidenced by `foundry.toml` RPC endpoints (`HYPEREVM_RPC_URL`) and etherscan verification configuration.

No HyperEVM-specific opcode or precompile usage exists in any of the in-scope contracts. The system is standard EVM-compatible. The `ve33` module specifies `evm_version = "london"` in its foundry configuration.

Time configuration: `HybraTimeLibrary` currently uses testnet parameters:

- `WEEK = 1800` seconds (30 minutes) -- mainnet would be `604800` (7 days)
- `MAX_LOCK_DURATION = 2 years` -- mainnet would be 4 years

These testnet values are active in the audited codebase with mainnet parameters commented out.

14. Key Deviations from Upstream

vs. Uniswap V3

Aspect	Uniswap V3	Hybra CL
Fee storage	`feeProtocol` in Slot0	Removed; fees computed dynamically per-swap via factory
Fee model	Static per-pool	Dynamic (volatility-sensitive via TWAP oracle)
Deployment	Constructor with immutables	EIP-1167 clones with `initialize()`
Staking	External (no native support)	Built-in gauge staking with `stakedLiquidity` tracking
Reward distribution	None	Synthetix-style streaming via `rewardGrowthGlobalX128`
Fee split	Protocol fee only	Three-way: unstaked LPs / gauge (staked LPs) / protocol
Tick state	`liquidityGross`, `liquidityNet`	Added `stakedLiquidityNet`, `rewardGrowthOutsideX128`
Position state	Standard fee accrual	`staked` flag skips fee accrual for staked positions

Aspect	Uniswap V3	Hybra CL
Fee module safety	N/A	`excessivelySafeStaticCall` with gas limits
NFT manager integration	Separate contract only	Pool has `burn`/`collect` overloads with `onlyNftManager`
Reward rollover	N/A	Unearned rewards (zero staked liquidity) accumulate in `rollover`

vs. Standard ve(3,3) / Solidly

Aspect	Standard ve(3,3)	Hybra
Reward token	Direct base token to users	rHYBR intermediary with redemption options
Redemption penalty	N/A	20% penalty for liquid HYBR conversion
Auto-compounding	N/A	gHYBR vault with single veNFT wrapper
VotingEscrow	Standard Solidly	Added permanent locks, multiSplit, partner NFT restrictions
Voter	Monolithic	Split into VoterV3 (voting) + GaugeManager (gauge creation/distribution)
CL integration	Varies	Deep integration: pool-native staking, three-way fee split

vs. Blackhole (direct fork parent)

The protocol is a fork of Blackhole with modifications including:

- rHYBR intermediary reward layer (not present in Blackhole)
- gHYBR auto-compounding vault
- HybrSwapper for bribe-to-HYBR conversion
- Modified access control via PermissionsRegistry

15. My Findings

During this contest I was able to find some vulnerabilities that were Valid Findings. However, due to V12 AI questionable incursion in the middle of the contest (it was the first time that V12 was used by Code4rena), they were invalidated few days before the ending of it.

The following findings were identified during the review. They are documented here for completeness and future hardening.

[H-1] Reentrancy in `slash` — No Guard or State Updates Before External Calls

Severity: High

Description

The public `slash` function performs multiple external calls (`JUNIOR_POOL.slash`, `DOLA.approve`, and `IMarket(market).repay`) without any reentrancy protection or internal state updates between them. Execution order is: (1) validations and computation of `slashed`, (2) `JUNIOR_POOL.slash(debt - collateralValue)`, (3) `DOLA.approve(market, slashed)`, (4) `IMarket(market).repay(borrower, slashed)`, (5) `emit Slash` and return. No storage is written (e.g. marking the borrower as slashed) before these calls. A malicious market or token contract could reenter `slash` during one of these calls (e.g. inside `repay` or via a callback from the pool/token) and, before the first invocation completes, trigger additional `slash` executions. This can drain or corrupt the junior pool by repeatedly invoking `slash` for the same or other borrowers before the original call finalizes.

Recommendation

Apply a reentrancy guard (e.g. inherit from OpenZeppelin's `ReentrancyGuard` and use a `nonReentrant` modifier) on `slash`, and/or introduce an internal state change (e.g. marking the borrower as slashed in storage) before performing any external calls so that reentrant calls fail validation or see updated state.

Impacted code (conceptual):

```
function slash(address market, address borrower) public returns(uint) {
    require(allowedMarkets[market], "Market not allowed");
    require(DBR.markets(market), "Market not active FiRM market");
    require(activationTime[market] <= block.timestamp && activationTime[market] >
0, "Market protection not activated");
    uint collateralValue = IMarket(market).getCollateralValue(borrower);
    uint debt = IMarket(market).debts(borrower);
    require(debt > collateralValue, "No bad debt");
    require(debt >= minDebt, "Debt too low");
    require(collateralValue <= maxCollateralValue, "Collateral value too high");
    uint slashed = JUNIOR_POOL.slash(debt - collateralValue);  // external
    DOLA.approve(market, slashed);                              // external
    IMarket(market).repay(borrower, slashed);                   // external —
reentrancy vector
    emit Slash(market, borrower, slashed);
    return slashed;
}
```

[H-2] Incorrect `IERC20` Interface in WithdrawEscrow — Wrong `transfer` Signature

Severity: High

Description

The `IERC20` interface used in `WithdrawEscrow.sol` defines `transfer` with a non-standard signature: `transfer(address from, uint amount)`. The standard ERC-20 specification uses `transfer(address to, uint256 amount)`. The first parameter is the recipient (`to`), not the sender (`from`). This mismatch can cause compilation errors when the contract is used with standard ERC-20 tokens, or worse, lead to incorrect use at

call sites (e.g. passing `from` where `to` is expected), resulting in failed transfers, wrong recipients, or integration failures with other protocols that assume the standard ABI.

Recommendation

Align the interface with EIP-20: `function transfer(address to, uint256 amount) external returns (bool);`. Ensure all call sites use the correct argument order (recipient, amount) and that the token contract actually implements the standard `transfer`.

Impacted code (conceptual):

```
// Incorrect
function transfer(address from, uint amount) external returns(bool);

// Correct (ERC-20)
function transfer(address to, uint256 amount) external returns (bool);
```

[M-1] Division by Zero in `LinearInterpolationDelayModel.getWithdrawDelay`

Severity: Medium

Description

In `LinearInterpolationDelayModel`, `getWithdrawDelay` computes `maxDelayThreshold = totalSupply * maxDelayThresholdBps / 10_000`. When `maxDelayThresholdBps` is 0 or `totalSupply` is 0, `maxDelayThreshold` becomes 0. The function then divides by `maxDelayThreshold` in the expression `(minDelay * (maxDelayThreshold - totalWithdrawing) + maxDelay * totalWithdrawing) / maxDelayThreshold`, causing a division-by-zero revert. That can make the WithdrawEscrow contract unusable whenever the delay model is queried under these conditions (e.g. initial zero supply or misconfiguration), blocking withdrawals or delay queries.

Recommendation

Check that `maxDelayThreshold` is not zero before dividing. If it is zero, return a safe default (e.g. `maxDelay`) or a dedicated sentinel value, and document the behavior. Optionally, enforce `maxDelayThresholdBps > 0` and/or handle `totalSupply == 0` at initialization.

Impacted code (conceptual):

```
uint maxDelayThreshold = totalSupply * maxDelayThresholdBps / 10_000;
if(totalWithdrawing >= maxDelayThreshold) return maxDelay;
return (minDelay * (maxDelayThreshold - totalWithdrawing) + maxDelay *
totalWithdrawing) / maxDelayThreshold;  // reverts if maxDelayThreshold == 0
```

[M-2] Fee Charged on Full Withdrawal Amount When Exit Window Has Started — Double Charging

Severity: Medium

Description

In `queueWithdrawal`, when a user already has an existing withdrawal and the exit window has started (`block.timestamp > exitWindowStart`), the fee is calculated on the entire `totalWithdrawAmount` instead of only on the new amount being queued. The logic is: `fee = totalWithdrawAmount > amount && block.timestamp > exitWindowStart ? totalWithdrawAmount * withdrawFeeBps / 10000 : amount * withdrawFeeBps / 10000`. So after the exit window has started, any additional queue operation pays fee on the full accumulated amount, including amounts that were already fee-charged in prior queue operations. This effectively double-charges (or over-charges) users for the same principal.

Recommendation

Change the fee basis to only the new amount being added in this call, or introduce explicit tracking of the amount that has already been fee-charged and apply the fee only to the increment. Ensure the invariant is clear: fee is applied once per unit of withdrawal amount, at a well-defined point (e.g. at queue time for that increment only).

Impacted code (conceptual):

```
fee = totalWithdrawAmount > amount && block.timestamp > exitWindowStart ?
    totalWithdrawAmount * withdrawFeeBps / 10000 :
    amount * withdrawFeeBps / 10000;
totalWithdrawAmount -= fee;
```
