Sigil Smart Contract Review

6 contracts in `contracts/src/`. Solidity 0.8.26 (except PoolReward at 0.8.24). Targets **Base mainnet** (Uniswap V4 + EAS).

All tests pass: **60/60** ✅ | Build: **clean** ✅

Architecture Overview

```
graph TB
    User["User / Agent"] -->|"launch()"| Factory
    Factory["SigilFactory"] -->|"deploys"| Token["SigilToken (ERC-20)"]
    Factory -->|"creates pool"| PM["Uniswap V4 PoolManager"]
    Factory -->|"registerPool()"| Hook["SigilHook"]
    PM -->|"afterSwap()"| Hook
    Hook -->|"USDC fees"| Vault["SigilFeeVault"]
    Hook -->|"Native token fees"| Escrow["SigilEscrow"]
    Vault -->|"80% to dev"| Dev["Developer Wallet"]
    Vault -->|"20% to protocol"| Treasury["Protocol Treasury"]
    Escrow -->|"unlock via governance"| Dev
    PoolReward["PoolReward"] -->|"EAS attestation claim"| Dev

    style Factory fill:#6C5CE7,color:#fff
    style Hook fill:#6C5CE7,color:#fff
    style Vault fill:#00B894,color:#fff
    style Escrow fill:#E17055,color:#fff
```

---

1. SigilToken

> *SigilToken.sol — 58 lines*

What It Does

Minimal ERC-20 token. Every project launched on Sigil gets one.

Key Facts

- **Fixed supply**: 100 billion tokens (set by Factory's `DEFAULT_SUPPLY`)
- **No mint/burn**: all supply minted at construction to the `_recipient` (the Factory)
- **Standard ERC-20**: `transfer`, `transferFrom`, `approve`, `balanceOf`, `allowance`
- **No ownership or admin functions** — immutable after deploy
- **Max-allowance optimization**: `type(uint256).max` skips deduction

How Sigil Uses It

Factory deploys one per `launch()` call. All 100B tokens go to Factory, which places them as single-sided liquidity in the V4 pool. No tokens go directly to the developer.

---

2. SigilFactory

> *SigilFactory.sol — 298 lines*

What It Does

One-click token launch: deploys the token, creates a Uniswap V4 pool, places liquidity, and registers it with the Hook.

Key Function: `launch(name, symbol, projectId, dev)`

1. Deploys a new `SigilToken` (100B supply → minted to Factory)
2. Sorts token vs USDC addresses for V4 pool key
3. Computes the `PoolKey` (token/USDC pair, 1% LP fee, 200 tick spacing)
4. Registers the pool with `SigilHook.registerPool()`
5. Initializes the pool on PoolManager with a starting price
6. Places **single-sided liquidity** across 7 tick bands (bonding curve effect)
7. Stores launch info and emits `TokenLaunched` event

Constants

Constant	Value	Purpose
`DEFAULT_SUPPLY`	100B tokens	Fixed for all launches
`DEFAULT_FEE`	10000 (1%)	LP fee tier (but set to 0 — hook takes 1% separately)
`DEFAULT_TICK_SPACING`	200	V4 pool tick spacing

Liquidity Placement

- Supply is split across **7 bands** of tick ranges
- Creates depth at various price levels (bonding curve)
- Token is placed single-sided (no USDC required to launch)
- **LP is permanently locked** — `beforeRemoveLiquidity` reverts in the Hook

Access Control

- `owner` can call `setOwner()` — no other admin functions
- Anyone can call `launch()`

---

3. SigilHook

> *SigilHook.sol — 351 lines*

What It Does

Uniswap V4 hook that intercepts every swap to collect 1% fees. This is the core revenue engine.

Fee Flow (afterSwap)

```
Every swap → afterSwap fires → calculates 1% of output

If output is USDC:
  → 80% to dev via FeeVault
  → 20% to protocol via FeeVault
```

```
If output is native token:
  → 100% to tokenEscrow contract
```

Hook Permissions

Hook	Enabled	Why
`beforeInitialize`	✅	Validates pool is registered
`afterSwap`	✅	Collects 1% fee on every swap
`beforeRemoveLiquidity`	✅	**Permanently blocks LP removal**
All others	❌	Not needed

Key Functions

Function	Access	Purpose
`registerPool(key, dev, tokenIsCurrency0)`	Factory only	Registers a new Sigil pool
`setPoolDev(poolId, newDev)`	Owner only	Updates dev address (assigns escrowed fees)
`setTokenEscrow(newEscrow)`	Owner only	Updates escrow contract address
`setFactory(newFactory)`	Owner only	Updates factory address
`setOwner(newOwner)`	Owner only	Transfers ownership

State Tracking

- `isRegisteredPool[poolId]` — only registered pools get fee collection
- `sigilTokenIsToken0[poolId]` — which side of the pair is the Sigil token
- `poolDev[poolId]` — maps pool to its developer (for fee routing)

LP Lock Mechanism

`beforeRemoveLiquidity` always reverts with `"SIGIL: LP_LOCKED"`. Once liquidity is placed by Factory, it can **never** be removed. This is a core trust property.

---

4. SigilFeeVault

> *SigilFeeVault.sol — 386 lines*

What It Does

Accounting contract for USDC fee splits. Accumulates dev (80%) and protocol (20%) fees from swaps. Handles the case where a dev hasn't verified yet (escrow mode).

Fee States

```
graph LR
    A["Hook deposits fees"] --> B{Dev known?}
    B -->|Yes| C["devFees[dev][token] += 80%"]
    B -->|No| D["unclaimedFees[poolId][token] += 80%"]
    D -->|"assignDev()"| C
    D -->|"30 days expire"| E["sweepExpiredFees() → protocol"]
    C -->|"claimDevFees()"| F["Dev wallet"]
```

Key Functions

Function	Access	Purpose
`depositFees(poolId, dev, token, devAmt, protocolAmt)`	Hook only	Records incoming fee split
`claimDevFees(token)`	Any dev	Withdraw accumulated fees for a token
`claimAllDevFees()`	Any dev	Withdraw all accumulated fees across all tokens
`assignDev(poolId, dev)`	Owner	Assign escrowed fees to a verified dev
`sweepExpiredFees(poolId)`	Anyone	Move 30-day-old unclaimed fees to protocol
`claimProtocolFees(token)`	Owner	Withdraw protocol's 20% accumulation

Escrow Logic (Unclaimed Devs)

- If `dev == address(0)`, fees are escrowed under the `poolId`
- `assignDev()` transfers all escrowed fees to the dev's balance
- After **30 days** without assignment, anyone can call `sweepExpiredFees()` to move them to protocol

Fee Tracking

- `devFees[address][token]` — claimable balance per dev per token
- `protocolFees[token]` — claimable balance for protocol per token
- `totalDevFeesEarned[address][token]` — lifetime earnings (for dashboards)
- `devFeeTokens[address]` — list of tokens a dev has earned (for `claimAll`)

---

5. SigilEscrow

> *SigilEscrow.sol — 371 lines*

What It Does

DAO governance contract for token unlocks. Native tokens from swaps flow here and are locked. The developer proposes milestones, the community votes, and tokens are released upon verified completion.

Proposal Lifecycle

```
Created → Voting (5 days) → Approved/Rejected
  If Approved → Dev builds → submitProof() → CompletionVoting (3 days)
    If Confirmed → Completed (tokens released to dev)
    If Rejected → Disputed (protocol can override)
```

Key Constants

Constant	Value	Purpose
`PROPOSAL_THRESHOLD_BPS`	5 (0.05%)	Min token balance to propose (non-dev)
`QUORUM_BPS`	400 (4%)	Min participation for valid vote
`VOTING_DURATION`	5 days	Initial vote window
`COMPLETION_VOTING_DURATION`	3 days	Completion vote window

Who Can Propose

- **Developer wallet** — always (no threshold)
- **Token holders** — if they hold ≥ 0.05% of total supply (50M of 100B)

Key Functions

Function	Access	Purpose
`createProposal(title, desc, amount, targetDate)`	Dev or 0.05% holder	Create milestone unlock proposal
`vote(proposalId, support)`	Any token holder	Vote yes/no (weight = token balance at snapshot)
`voteWithComment(proposalId, support, comment)`	Any token holder	Vote + emit comment on-chain
`finalizeVote(proposalId)`	Anyone	Close voting after deadline
`submitProof(proposalId, proofUri)`	Dev only	Submit completion evidence (URL)
`voteCompletion(proposalId, confirmed)`	Any token holder	Vote on whether milestone was completed
`voteCompletionWithComment(...)`	Any token holder	Vote completion + comment
`finalizeCompletion(proposalId)`	Anyone	Close completion vote
`protocolOverride(proposalId)`	Protocol only	Override disputed completion (release tokens)

Vote Mechanics

- **Snapshot-based**: voter weight = `token.balanceOf(voter)` at proposal creation block
- **Double-vote prevention**: `hasVoted[proposalId][voter]` mapping
- **Quorum**: 4% of total supply must participate for vote to be valid

- **Majority wins**: yes > no = approved, no ≥ yes = rejected

Protocol Override

- Only on **Disputed** proposals (completion vote failed)
- Only callable by `protocol` address
- Releases tokens to dev — safety valve for unfair community rejection

Token Release

On `Completed` or `Overridden` status, the proposal's `tokenAmount` is transferred from the escrow contract to `devWallet`.

---

6. PoolReward

> *PoolReward.sol — 195 lines*

What It Does

EAS (Ethereum Attestation Service) based reward claiming. Allows verified project owners to claim token rewards by proving ownership via on-chain attestations.

Flow

```
1. createPool(projectId, token, amount)  →  Tokens deposited
2. Developer verifies identity off-chain (GitHub, domain, tweet)
3. Backend creates EAS attestation: wallet → project
4. Dev calls claimReward(attestationUID)  →  Tokens released
```

Key Functions

Function	Access	Purpose
`createPool(projectId, token, amount)`	Anyone	Deposit tokens for a project
`topUpPool(projectId, amount)`	Anyone	Add more tokens to existing pool
`claimReward(attestationUid)`	Attested owner	Claim pool using EAS proof
`emergencyWithdraw(projectId, to)`	Owner only	Rescue unclaimed pool funds
`getPool(projectId)`	View	Check pool status

EAS Verification (on-chain)

The claim verifies all of these on-chain:

1. Attestation created by `trustedAttester` (Sigil backend)
2. Attestation recipient == `msg.sender`
3. Not revoked
4. Correct schema UID
5. `isOwner == true` in the attestation data

Schema

```
string platform, string projectId, address wallet, uint64 verifiedAt, bool isOwner
```

Contract Relationships

From	To	Relationship
Factory → Token	deploys	Creates new ERC-20 per launch
Factory → Hook	calls `registerPool()`	Registers pool for fee collection
Factory → PoolManager	calls `initialize()` + `modifyLiquidity()`	Creates V4 pool + places liquidity
Hook → FeeVault	calls `depositFees()`	Routes USDC fees (80/20 split)
Hook → Escrow	transfers native tokens	Routes native token fees
FeeVault → Dev	`claimDevFees()`	Dev withdraws USDC earnings
Escrow → Dev	governance unlock	Tokens released after approved milestone
PoolReward → Dev	`claimReward()`	EAS-verified reward distribution

Deployment Addresses (Configured)

Contract	Base Mainnet	Base Sepolia
PoolManager	`0x498581fF718922c3f8e6A244956aF099B2652b2b`	`0x05E73354cFDd6745C338b50BcFDfA3Aa6fA034`
USDC	`0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913`	`0x036CbD53842c5426634e7929541eC2318f3dCF`
EAS	`0x4200000000000000000000000000000000000021`	`0x4200000000000000000000000000000000000000`

> *[!IMPORTANT] The Hook address must encode V4 permission flags in its last 14 bits. Deployment uses `HookMiner` to find a valid salt via CREATE2.*

Security Notes for Reviewer

1. **LP is permanently locked** — `beforeRemoveLiquidity` reverts. This is intentional and a core trust guarantee.
2. **Protocol override** is limited to **completion disputes only** — cannot override initial votes.
3. **Fee expiry** — unclaimed dev fees expire to protocol after 30 days if no dev is assigned.
4. **No reentrancy guards** — contracts use checks-effects-interactions pattern. Reviewer should verify this is sufficient.
5. **Snapshot voting** — uses `balanceOf` at proposal creation block, not current balance. Prevents flash-loan vote manipulation.
6. **All token transfers use low-level `.call()`** — success is checked, but return data is not decoded. Standard for known ERC-20s.