Daemon

A Mined ERC-20 with a Self-Hook

May 2026

Abstract

DMN is an ERC-20 token released entirely through proof of work, with no team allocation, no insider presale, no admin keys, and no upgrade path. The token contract is also its own Uniswap V4 swap hook and its own miner. One bytecode, one address. Once deployed, the rules cannot change.

The supply is 21 million tokens, identical to Bitcoin in scale. Five percent funds an open genesis sale, five percent seeds a Uniswap V4 DMN/ETH pool whose liquidity is locked forever by the same hook that collects a 1% swap fee. The remaining ninety percent is mined: anyone with a browser can solve a keccak256 puzzle bound to their wallet address and call `mine()` to receive the current era's reward.

This document describes why the design exists, how each piece works, and what limits remain.

Why this exists

Most ERC-20 launches today are pre-mints. A team deploys a contract, mints the supply to a multisig, and decides how to distribute it later. The "fair launch" variants typically replace the multisig with a vesting contract, which is just a slower version of the same problem.

A different model has existed since 0xBitcoin in 2018: the contract refuses to mint to anyone. New tokens emerge only when someone solves a hash puzzle on chain. The supply schedule is mechanical, the distribution is permissionless, and the team has no special power because there is no team-controlled state to abuse.

DMN takes that model and adds two pieces that 0xBitcoin and its imitators could not have. The first is address-bound proofs of work: a mined nonce only works for the wallet that found it, so solutions cannot be copied from the mempool. The second is a self-hook, made possible by Uniswap V4 hooks shipping to mainnet in January 2025. The token contract is also the swap hook for its own liquidity pool, which lets the pool reject every liquidity-modification call and effectively lock the LP without any external service.

Address-binding kills front-running at the cryptographic level. The self-hook removes the need to trust a third-party lock contract or a multisig timeout. Together they remove the two most common forms of soft-rug that survive even in supposedly fair launches.

How the contract is organized

The contract moves through four phases in its lifetime.

Phase 0, Empty. The contract is deployed and holds nothing. Constructor parameters set the addresses of the Uniswap V4 PoolManager, PositionManager, and Permit2 for the target chain. The deployer becomes the `controller`, the only address with permission to claim accumulated swap fees later. No other state is mutable from outside.

Phase 1, Genesis. Anyone can call `mintGenesis(units)` with `units * 0.01 ETH`. Each unit yields 1,000 DMN, capped at five units per transaction. The transaction cap exists so a single mempool slot cannot sweep the entire allocation in one go. Excess ETH is refunded in the same call. Three days after deploy, if the pool has not been seeded, `refundGenesis` opens and lets any holder of genesis DMN redeem it back for ETH at the original price (see Refund escape hatch).

Phase 2, Seeding. Once the genesis cap of 1,050,000 DMN is sold out, anyone can call `seedPool()`. The function mints the remaining 19,950,000 DMN to the contract itself, creates the V4 DMN/ETH pool with the contract as its hook, deposits all genesis ETH plus 1,050,000 DMN as liquidity, and sets the initial mining difficulty. The LP position NFT is minted to the controller. A fallback `partialSeed()` exists for the case where genesis stalls below the cap; it can be called by the controller no earlier than thirty minutes after deploy.

Phase 3, Mining. The remaining 18,900,000 DMN is released through proof of work. Each successful `mine(nonce)` call pays the current era's reward, starting at 100 DMN and halving every 100,000 mints. Mining ends when the cap is reached.

Phase transitions are gated by storage flags. No admin function exists to skip a phase, rewind state, or change parameters.

The mining protocol

A mined nonce is valid for a wallet at a given epoch when

```
keccak256(abi.encode(challenge, nonce)) < currentDifficulty
```

where

```
challenge = keccak256(abi.encode(chainId, contractAddress, miner, epoch))
epoch     = blockNumber / 100
```

Three properties matter here.

The challenge changes every hundred blocks, roughly twenty minutes on mainnet. A miner who fails to find a solution within one epoch cannot carry the work forward. The seed shifts and the search

starts over. This caps the cost of difficulty mispricing and gives the network natural retargeting windows.

The `miner` field is `msg.sender`. Bob's nonce search is for a different challenge than Alice's. Even if Bob copies Alice's pending transaction from the mempool and rebroadcasts it under his own address, his version computes a different hash and almost certainly fails the difficulty check. Front-running mined solutions is impossible at the cryptographic level, not at the transaction-ordering level.

Replay protection lives in a `usedProofs` mapping keyed on `keccak256(miner, nonce, epoch)`. The same triple cannot be claimed twice. Storage grows by one slot per successful mint.

Difficulty retargets every 2,016 mints, the same cadence as Bitcoin. The new target equals the old target times the ratio of blocks actually elapsed over the expected number of blocks. The expected count is 2,016 mints times 5 blocks per mint, that is, 10,080 blocks. The retarget is clamped to a factor of four in either direction per period.

A hard cap of ten mints per block prevents one party with surplus hashpower from sweeping every block. The eleventh `mine()` call in the same block reverts.

Starting difficulty after seeding is `type(uint256).max >> 32`, which means roughly one in four billion hashes qualifies. A modern laptop finds a solution in seconds, so the first generation of miners does not need specialised hardware.

The self-hook

A Uniswap V4 hook is a contract whose address contains specific bits in its lower fourteen bits. Those bits encode which lifecycle functions the hook actually implements. The same contract that implements `transfer` and `balanceOf` for the ERC-20 also implements `beforeSwap`, `afterSwap`, and `beforeInitialize` for the pool. To make this work, the contract is deployed via CREATE2 with a salt chosen so the resulting address has the correct permission bits. Salt mining happens offline before deployment, typically takes a few seconds, and is fully reproducible.

After seeding, the pool exists at coordinates: currency0 is native ETH, currency1 is DMN, tickSpacing is 200, hooks is the DMN contract itself. The hook intercepts swaps and routes 1% of the ETH side of every swap to the contract's own balance.

The hook also denies every liquidity modification. `beforeAddLiquidity` and `beforeRemoveLiquidity` revert unconditionally. The pool starts with the seed liquidity and stays at that exact shape forever. Nobody, including the deployer who holds the LP NFT, can move the liquidity out or add more on top of it.

Accumulated ETH fees sit on the contract until the controller calls `claimFees()`, which transfers the contract's entire ETH balance to the controller. This is the only privileged function in the contract. It cannot affect the token supply, the pool liquidity, the mining schedule, or any other parame-

ter. It only moves the contract's own ETH balance to the controller, where the controller had no influence over how that ETH got there in the first place.

The launch model that results does not depend on a third-party lock service, a multisig timeout, or any external commitment. The lock is the bytecode.

Tokenomics

Allocation	Amount	Share
Genesis sale	1,050,000 DMN	5%
Locked LP	1,050,000 DMN	5%
Mining	18,900,000 DMN	90%
Total	**21,000,000 DMN**	**100%**

Genesis is priced at 0.01 ETH per 1,000 DMN, fixed. A fully subscribed genesis raises 10.5 ETH, every unit of which goes directly into the V4 pool as the ETH side of seed liquidity. Nothing reaches the deployer at genesis.

The mining schedule halves every 100,000 successful mints. Era zero pays 100 DMN per mint, era one pays 50, era two pays 25, and so on. In practice the schedule terminates at era four because the cumulative reward up to that point reaches the 18.9M mining cap.

Target throughput is one mint per minute on average. With Ethereum's twelve-second blocks, that translates to a difficulty retarget every 33.6 hours of real time, assuming the network mines at exactly the target rate. The retarget mechanism corrects for deviations within a factor of four per period.

Seeding strategy

Seeding the pool is the only timed decision the controller has to make. Two functions enable it:

```
seedPool()    — permissionless, requires genesisMinted == GENESIS_CAP
partialSeed() — controller only, requires block.timestamp >= deployedAt + 30
minutes
```

Both functions call the same internal logic. The only difference is whether the genesis has reached the full 1,050,000 DMN cap or not. `seedPool` is the happy path where the community fills the cap on its own; `partialSeed` is the controller's escape hatch when genesis stalls.

The interesting property of `partialSeed` is what it does with the unsold genesis allocation. The body uses `genesisMinted`, not `GENESIS_CAP`:

```
function _seedBody() internal {
    uint256 eth    = genesisEthRaised;
    uint256 pickLP = genesisMinted;
    _mint(address(this), pickLP + MINING_SUPPLY);
    // create the pool with (eth, pickLP) as liquidity
}
```

The unsold genesis DMN is never minted. Not burned, not held by the controller, not stuck in storage. It simply does not exist. If genesis stops at 300,000 DMN sold, the contract's total supply becomes 300,000 plus 18,900,000 = 19,200,000 DMN, and the pool receives 300,000 DMN paired against whatever ETH was raised at that point.

This is clean from a contract-design perspective. No surplus tokens to manage, no governance over unsold inventory, no airdrops to plan. The 30-minute delay on `partialSeed` is the only anti-grief constraint: it prevents the controller from instantly seeding an empty pool, which would yield zero liquidity and break the AMM. The `genesisMinted > 0` check is the same idea: at least one external buyer must have participated.

The downside of early seeding is economic, not technical. The mining supply is fixed at 18,900,000 DMN regardless of how much genesis sold. The LP side scales linearly with genesis sales. Seeding at 30% means the pool is roughly one third the size it would be at full sell-out, and miners produce new supply at the same rate either way. A thin pool combined with active mining means each mined block pushes the price down harder than it would against a deep pool.

Three strategies are reasonable.

Strict sell-out. Wait for genesis to hit the full cap before seeding. Anyone can call `seedPool` at that point, so the call costs the controller nothing. Maximum pool depth, maximum credibility, but the timeline is uncertain. If the sale stalls below the cap, trading and mining never open.

Threshold seed. The controller commits publicly to seeding when genesis reaches a chosen threshold, for example 50% or 70%. The threshold acts as a coordination point: buyers know how much further the sale has to go before trading opens, and they can pile in if they want the gap closed. The controller publishes a deadline as a fallback, for instance "if 70% is not reached by day seven, `partialSeed` is called at the current level".

Aggressive early seed. Seed within hours of deploy regardless of how much sold. Trading and mining open immediately, but the pool is tiny and slippage is severe. Useful only when the project's appeal depends on the miner being live at launch and the controller is willing to accept a fragile initial market.

The contract enforces none of these. The controller's discretion within the time gate is total. The recommendation is the threshold strategy with a published target and deadline, because it gives the community a number to rally around and removes the open-ended uncertainty of waiting for a strict sell-out that may never come.

Refund escape hatch

The genesis sale collects ETH on the contract and trusts that one of the seed functions will eventually be called. If neither succeeds, the ETH would otherwise be locked forever. A buyer with no recourse is exposed to two distinct failure modes: a controller who disappears, and a controller who tries to seed but cannot because of a V4 edge case the fork tests missed.

`refundGenesis` exists to cover both:

```
function refundGenesis(uint256 tokenAmount) external nonReentrant {
    if (genesisComplete)                                  revert GenesisAlrea
dyComplete();
    if (block.timestamp < deployedAt + REFUND_GRACE)      revert RefundGraceN
otPassed();
    if (tokenAmount == 0 || tokenAmount % GENESIS_UNIT != 0) revert MustBeUni
tMultiple();

    uint256 units   = tokenAmount / GENESIS_UNIT;
    uint256 ethBack = units * GENESIS_PRICE;

    _burn(msg.sender, tokenAmount);
    genesisMinted    -= tokenAmount;
    genesisEthRaised -= ethBack;

    (bool ok,) = msg.sender.call{value: ethBack}("");
    if (!ok) revert EthTransferFailed();

    emit GenesisRefund(msg.sender, ethBack, tokenAmount);
}
```

`REFUND_GRACE` is fixed at 3 days. Any holder of genesis-minted DMN can call `refundGenesis(amount)` once the grace has passed, provided the pool has not yet been seeded. The function burns the caller's DMN and returns ETH at the original 0.01 ETH per 1,000 DMN rate. Multiple partial refunds are permitted.

Three properties keep the function safe.

The function is gated on `!genesisComplete`. After `seedPool` or `partialSeed` runs, the ETH is no longer on the contract; it is liquidity in the V4 pool, locked by the hook, unreachable. Refunds become impossible from that moment forward, by design. Holders who want out post-seed sell into the pool at the AMM price.

The unit-multiple check forces `tokenAmount` to be a multiple of 1,000 DMN. This avoids rounding issues and matches the genesis purchase granularity exactly. Half-unit refunds would create dust counters that drift apart from `genesisEthRaised`.

The `_burn` happens before the ETH transfer. Reentrancy through the recipient cannot replay a refund the caller has not first burned. Combined with the `nonReentrant` modifier, this leaves no

path for double-spending the position.

The three-day window is short enough that a committed controller can seed before refunds open. A buyer who panics on day one cannot exit through this function; they have to live with the position until the pool opens or the grace expires. The window is long enough that a controller in good faith has time to debug a failing `partialSeed` and retry without being interrupted by refund traffic competing for the same ETH balance.

After three days, if seeding has not succeeded, the system effectively becomes a no-op refund market. Genesis buyers withdraw, the contract's ETH balance trends to zero, and the deployment is over. No tokens were minted to anyone who has not since burned them back.

Verification

The contract source is published under the MIT license alongside its build configuration. After deployment to any chain, the source can be verified on Etherscan, Sourcify, or any compatible explorer. The compiled bytecode, the constructor arguments, and the CREATE2 salt are reproducible from the source.

The mining algorithm is implemented in Rust, compiled to WebAssembly, and runs entirely in the user's browser. The Rust source is in the same repository as the contract. The compiled WASM bundle is published as a static asset alongside the frontend, served directly from disk with no server-side computation.

A user who wants to confirm the contract does what this document claims can do three things independently before mining.

First, check that the contract address has the correct V4 hook permission bits. Take the deployed address, mask with `0x3FFF`, and verify the result equals `0x20CC`. Any address that does not pass this check is not the real contract.

Second, read the constants directly from chain. Call `TOTAL_SUPPLY`, `MINING_SUPPLY`, `GENESIS_CAP`, `BASE_REWARD`, `ERA_MINTS`, `EPOCH_BLOCKS`, `ADJUSTMENT_INTERVAL`, and `MAX_MINTS_PER_BLOCK`. Each must match the values stated here.

Third, try to remove liquidity from the DMN/ETH pool through any V4 router or position manager. The call reverts with `InvalidAction()`. The revert is unconditional and originates from the hook itself, not from any access-controlled flag.

Agent alignment

The DMN contract maps cleanly onto the ERC-8004 agent registry model. Three properties of a "trustless agent" are spelled out in the EIP: a stable identity, observable reputation, and verifiable behavior. DMN satisfies all three without any wrapper layer.

The identity is the contract's own address. It is fixed at deploy time, derived deterministically from the CREATE2 init-code hash, and cannot be migrated. There is no upgradable proxy in front, no admin key to rotate, no governance to change ownership. The address is the agent.

The reputation is the contract's on-chain state. `totalMints`, `totalMiningMinted`, accumulated swap fees, the V4 LP NFT held by the controller, the genesis ETH raised: every metric that matters is queryable directly from chain by any indexer. There is no reputation oracle to trust because every claim DMN makes about itself can be checked against its own storage.

The behavior is verified by the source. The bytecode is the source, the source is the spec, and both are immutable. The hook reverts on liquidity removal. The mint function refuses to pay more than `MINING_SUPPLY`. The `claimFees` function only moves ETH the contract has accumulated, never tokens it has minted. Each of these guarantees is unconditional and originates from the hook itself.

When the canonical ERC-8004 Identity Registry is deployed on mainnet (`0x8004A169FB4a3325136EB29fA0ceB6D2e539a432`), the controller registers DMN with a single call to `register(agentURI)`. The `agentURI` points to a hosted JSON file (`/agent.json`) that lists the contract's capabilities, endpoints, and validation hooks. Indexers like 8004scan pick the new agent up automatically from the `Transfer` event of the registry NFT.

Miner Agent NFTs

A separate ERC-721 collection, `MinerAgent`, gives each DMN participant an on-chain identity in the ERC-8004 sense without modifying the core contract. The mapping is one NFT per address, claimable once, soulbound after mint. Anyone whose DMN balance is non-zero may call `claim()` to mint their own agent NFT. Genesis buyers, miners, and aftermarket buyers all qualify; the only requirement is that the wallet currently holds DMN.

The metadata is generated entirely on-chain. `tokenURI` returns a base64-encoded JSON whose `image` field embeds a base64-encoded SVG rendered from live state. The SVG shows the agent ID, the truncated wallet address, the current DMN balance, and a tier label that scales with holdings: Initiate, Bronze, Silver, or Gold. There are no hosted images, no IPFS pins, and no central server in the loop. Every render is reproducible from the contract.

Transfers between EOAs revert with `Soulbound()`. Burning and minting are both allowed at the protocol level so that future upgrades to the soulbound semantics are possible without a redeploy, but no burn function is exposed to users today. The NFT is meant to remain attached to the wallet that earned it.

The collection is intentionally lightweight. It does not give holders new rights over DMN. It does not change the supply distribution. It does not introduce new tradeable surface area. Its job is to make ownership of DMN legible to agent-aware tooling and to give participants a single, queryable identity that ERC-8004 indexers can resolve.

Limits and caveats

The contract has no upgrade path. If a critical bug is discovered post-launch, there is no fix. The mitigation is the same as Bitcoin's: keep the surface small, audit before deploy, deploy code that has been forked from a contract running in production elsewhere.

Mining on Ethereum mainnet costs real gas. At twenty gwei a `mine()` transaction costs roughly five dollars worth of ETH. The reward in dollar terms must exceed this for mining to remain economically rational. As the price of DMN in the AMM falls below the mining cost, hashrate drops, the network produces fewer than one mint per minute, and difficulty retargets downward. Equilibrium price is whatever clears that condition.

Address-binding makes solutions unstealable from the mempool but does nothing against a miner who controls multiple wallets. A miner with surplus hashrate can mine to any address they own. This is the same property Bitcoin has and is not considered a problem; the cost of computing power is what discourages over-extraction.

The 1% swap fee paid to the controller represents an ongoing extraction from swap volume. It is not a token tax. The swap itself executes at the pool's posted price, and the fee is taken from the ETH side via the hook. Buyers and sellers pay equally. The fee can never be raised, lowered, or rerouted because the contract is immutable.

DMN is not a promise. There is no team behind it in the conventional sense. There is no roadmap to deliver against. The contract does what its bytecode does, and nothing more.