WANE PROTOCOL · TECHNICAL WHITEPAPER

Wane

An On-Chain Antibody Registry and Policy Firewall for AI Agents

Detection already exists as a vendor commodity. The part that does not is an open, staked, on-chain place where the party that was just attacked writes the signal back.

Antibody registry · staked write-back commons · policy firewall · EIP-7702 delegate · Uniswap v4 hook · Solana PDA binding

VERSION
1.0

DATE
June 2026

STATUS
Live on Base mainnet

Wane: An On-Chain Antibody Registry and Policy Firewall for AI Agents

Detection already exists as a vendor commodity. The part that does not is an open, staked, on-chain place where the party that was just attacked writes the signal back.

TL;DR

Autonomous agents sign at machine speed with no human reading the destination address. The defensive tooling that exists today is detection: vendors simulate transactions and ship risk feeds to wallets. Detection is a commodity. The part that does not exist in commodity form is a **write-back commons**: an open, staked, on-chain registry where the party holding the freshest signal, the one that was just drained, publishes that signal directly into shared state that the next agent reads before it signs.

Wane is that commons plus the enforcement that makes a written signal bind. An **antibody** is a staked, typed, on-chain claim that a specific subject is hostile. Publishing locks `mintStake = 100e18` WANE; disputing locks `challengeStake = 200e18` WANE; a flagged target is rejected before value moves. The same registry doubles as an on-chain **policy firewall** (spend caps, allowlists, kill switch, time-to-live, ten stable reason codes) and as the trust root a plain EOA borrows through an **EIP-7702 delegate** that can only block, never move funds. A **Uniswap v4 hook** makes the screen automatic at the pool boundary, screening swappers by address and by codehash.

It is live on Base mainnet (chainId `8453`), registry at `0x027F371fB139A57EcD2A2E175d30157eEA1C56de`, seeded with `652` genesis antibodies, with the same model running on Solana devnet. The honest boundary is stated throughout: Wane reduces the cost of the second exposure onward, never the first.

Abstract

Wane is an on-chain antibody registry and policy firewall for AI agents, live on Base mainnet (chainId `8453`) and running on Solana devnet under one SDK, `wane-sdk`. The registry holds **antibodies**: staked, typed, on-chain claims that a specific subject is hostile, keyed by `(ThreatKind kind, bytes32 subject)` over a four-value taxonomy (`Address=0`, `CallPattern=1`, `Bytecode=2`, `Semantic=3`) with identical numeric values on both chains. Reads are a free `view` (`check(kind, subject)`, with helpers `checkAddress` and `checkBytecode`, `checkFee` default `0`); writes are economically weighted, with `mintAntibody` locking `mintStake = 100e18` WANE and `challenge` locking `challengeStake = 200e18` WANE. A claim becomes **enforceable** when it is minted, not revoked, and either matured (`maturity = 21600` blocks) or corroborated to `enforceCorrobs = 2`, and a consumer may demand more. Disputes settle through `resolve(uint64 id, bool falsePositive)`, slashing a proven-false publisher and paying the challenger from `reserved`.

The same contract is a policy firewall (`WanePolicy.sol`): per-agent spend caps (`perTxCap`, `dailyCap`), selector and token allowlists, a guardian-operated kill switch (`paused`, `globalPaused`) and global denylist (`globalDenied`), and a policy time-to-live (`expiresAt`), every gate evaluated in-contract and surfaced through ten stable reason codes (`R_OK=0` through `R_TOKEN=9`). A plain EOA borrows this trust root through an EIP-7702 delegate (`WaneDelegate.sol`) whose `execute()` screens an action before running it and whose only power is to block: the inner call runs with `msg.sender == address(this)`, so Wane holds no funds and cannot redirect value. A Uniswap v4 hook (`WaneHook.sol`) screens every swapper by address and by codehash at the pool boundary, using the presented sender rather than `tx.origin`. The Solana enforcement path binds the screened subject to the paid subject through program-derived address seed equality. The paper states its trust surfaces (governor-arbitrated `resolve`, guardian powers) and its limits (first-exposure gap, on-chain-expressibility of subjects, 7702 direct-key-spend bypass) in dedicated honest-caveat boxes rather than footnotes.

WANE is a fixed supply of `1,000,000,000` tokens, already minted on Base mainnet, and is the stake and reward currency of the registry.

Contents

1. The problem: defense that does not accumulate

Every agent that gets drained teaches the attacker something, while teaching the other agents nothing.

An autonomous agent holding a key is a wallet that signs without a human in the loop. The premise of the entire category is that the agent decides, the agent signs, and value moves at machine speed. That premise is also the attack surface. When an agent approves a malicious spender, routes value to a drainer, or calls into hostile bytecode, there is no pause, no second glance, no human reading the destination address one character at a time. The transaction clears at the same speed every other transaction clears.

The defensive tooling that exists today is real and it works on the axis it was built for. The established vendors simulate a transaction before it is signed, score it, and ship the verdict to a wallet or an agent runtime as a feed. This is detection, and detection is a commodity: several vendors do it competently, the techniques are well understood, and the output is a stream of risk scores consumed at signing time. For a large class of attacks, a good simulation feed is sufficient. We are not claiming otherwise.

The gap is not in detection quality. The gap is in who is allowed to write, and when.

Consider the timeline of a single drainer address across its operating life. It is deployed. It drains its first victim. It drains its second, third, and fortieth. At some point a vendor's pipeline observes enough on-chain behavior to classify it, and from that moment forward subscribers to that vendor's feed are protected. Trace where the freshest possible signal sits in that timeline. It sits with the first victim, at the moment of the drain, before any vendor has classified anything. The party with the earliest, highest-confidence knowledge that an address is hostile is the party that just lost funds to it. That party has the signal. That party has no write path.

This is the structural defect. Defensive knowledge in the agent economy does not accumulate at the point where it is generated. It is generated at the edge, by victims, and it can only be published from the center, by vendors, after a classification lag. The edge holds fresh, specific, ground-truth signal and has nowhere to put it. The center holds the write path and operates one classification cycle behind the edge. Every drain regenerates knowledge that was already known to an earlier victim, because the earlier victim could not write it down anywhere the next agent would read.

A vendor feed inverts trust and it inverts timing, and both inversions cut against the party with the most at stake.

The trust inversion is that the consumer of the feed must trust the vendor's pipeline, the vendor's incentives, and the vendor's coverage decisions, none of which are inspectable on-chain and none of which the consumer can corroborate or contest. A score arrives. It is correct or it is not, and the consumer has no standing to push back, no stake to put behind a disagreement, and no way to add a signal the vendor's pipeline missed. The relationship is read-only by construction. The party with the most at stake in a given verdict, the agent about to sign, is a passive recipient.

The timing inversion is that the write happens at the center after a lag, while the danger is known at the edge immediately. The agent that is about to encounter a drainer for the second time in that drainer's life cannot benefit from the first encounter unless the first victim's knowledge has already completed a round trip through a vendor's classification pipeline and back out as a feed update. The faster the attacker rotates addresses and deploys fresh bytecode, the more often the relevant signal is younger than the feed.

What does not exist in commodity form is an open, staked, on-chain write-back commons: a place where the attacked party, holding the freshest signal in the system, can publish that signal directly to a registry that the next agent reads before it signs. The shape is a record written by a victim and read by a peer, with the write itself carrying economic weight so that a reader can weigh it, rather than a feed pushed from a vendor to a subscriber. The write path belongs to whoever has the signal, and the signal is weighed on-chain rather than trusted on faith.

Wane is that write-back commons, plus the enforcement layer that makes a written signal actionable at the moment an agent signs. It is live on Base mainnet (chainId `8453`), with the same registry model running on Solana devnet. The registry that holds the signal (`0x027F371fB139A57EcD2A2E175d30157eEA1C56de`) is the same contract that doubles as an on-chain policy firewall, and the same trust root that a plain EOA can route through an EIP-7702 delegate. The remainder of this document specifies how a write becomes enforceable, what economic weight a write carries, and exactly where the protection holds and where it does not.

HONEST CAVEAT. A write-back commons reduces the cost of the second exposure onward. It does nothing for the first. The first victim of a brand-new drainer, a freshly deployed address with no prior on-chain history, is unprotected at the moment of the drain, because no antibody for that subject exists yet. Wane does not claim to catch novel one-shot attacks. It claims that the second agent to encounter a given threat should not have to relearn what the first agent already paid to discover. Throughout this paper, when we say protection, we mean protection from the second exposure forward.

2. The analogy, made precise

Borrow three properties from immunity, discard the rest, and pin the word "antibody" to something that has a contract address.

The name is a claim, and a claim that borrows a biological metaphor should say exactly which parts it borrows and which parts it throws away. Loose biological analogies in this space tend to smuggle in properties the system does not actually have. We will do the opposite: name the three immune properties we reproduce, name the two we explicitly do not, and then define the word `antibody` as a concrete on-chain object with fields and a status, not as a figure of speech.

2.1 What we borrow

Three properties of a biological immune response map onto the design, and only these three.

Durable. An adaptive immune response leaves a memory that outlives the infection that produced it. The body does not re-derive the response to a previously seen pathogen from scratch on every exposure. In Wane, the analog is a persistent on-chain record. When a threat is published, it stays published. The record carries an `id`, a `publisher`, a `stake`, a `mintedBlock`, a corroboration count, and a `status`, and it remains readable by any agent at any later block at zero read cost (`checkFee` defaults to `0`). The knowledge that a given address is hostile becomes a durable fact in shared state rather than an event that expires from a vendor's cache.

Shared. Immune memory in a population can be transferred: one organism's adaptive response can protect another, which is the entire principle of passive immunization. In Wane, the registry is a single shared substrate that every enrolled agent reads from. A signal published by one victim is, from the next block forward, readable by every other agent on the chain. The first party to encounter a threat and write it down immunizes the entire population of readers, not just itself. This is the property a vendor feed cannot reproduce on open terms, because in a feed the shared layer is the vendor's private pipeline and membership is a subscription, not a commons.

Front-loaded. A primed immune system pays its detection cost before the pathogen does damage, not after. Recognition happens at the point of contact and gates the response. Wane is front-loaded in exactly this sense: the registry read happens before the action executes, not after it settles. An agent screens a target through `check(ThreatKind kind, bytes32 subject)`, or through a policy evaluation

via `wouldAllow(target, value, data)`, and a flagged target is rejected before value moves. Through the EIP-7702 delegate, an action routed through `execute()` is screened against the registry and the wallet policy first, and a flagged target reverts with `Blocked(target, reason)` before the inner call runs. The check is a precondition, not a postmortem.

2.2 What we discard

A metaphor is only honest if it names what it leaves behind. Two central properties of biological immunity are deliberately absent from Wane, and pretending otherwise would misdescribe the system.

No self/non-self discrimination. A biological immune system distinguishes the organism's own cells from foreign material and, when that discrimination fails, attacks the host. Wane has no such notion. It does not model identity, ownership, or belonging, and it makes no attempt to infer whether a target is "part of" an agent or foreign to it. A subject is flagged or it is not, by typed claim, and the only "self" in the system is the narrow on-chain guard that a legitimate delegate action must satisfy `msg.sender == address(this)`. There is no immune surveillance of the agent's own behavior and no concept of autoimmunity. The agent's owner sets policy; the registry supplies typed claims; neither one decides what the agent "is".

No somatic mutation. Biological antibodies are generated by a randomized, mutate-and-select process that explores antigen space and refines binding affinity over rounds. Wane generates nothing. It does not synthesize candidate threats, does not explore an attack space, and does not evolve its records toward better recognition. Every antibody in the registry was published by a specific party who staked WANE to assert it. The registry is a ledger of human-published and agent-published claims, refined by corroboration and challenge rather than by mutation and selection. There is no generative loop, and any description of Wane as "evolving its own defenses" would be false.

2.3 What an antibody actually is

With the borrowing and the discarding stated, the word can be defined precisely.

An antibody is a staked, typed, on-chain claim that a specific subject is hostile.

Unpack each adjective against the contract, because each one is load-bearing.

Staked. Publishing an antibody is not free assertion. The publisher locks `mintStake` (default `100e18` WANE) into the registry via `mintAntibody(ThreatKind kind, bytes32 subject, bytes32 evidence)`. The stake is the publisher's skin in the claim. It can be slashed if the claim is resolved as a false positive, and it is reclaimable via `reclaimStake(uint64 id)` once the claim matures (`maturity` defaults to `21600` blocks, roughly 72 hours at 2 second blocks). A claim with money behind it is a claim a reader can weigh, which is precisely the property a vendor score lacks.

Typed. An antibody carries a `ThreatKind` rather than an undifferentiated "this is bad" flag. The kind is drawn from a four-value enumeration with identical numeric values on both chains: `Address` (`0`), `CallPattern` (`1`), `Bytecode` (`2`), and `Semantic` (`3`). The type tells a reader what kind of subject the claim binds to and which class of threat it asserts, and it lets a policy enforce some kinds and ignore others through a bitmask (`K_ADDRESS`, `K_CALL`, `K_BYTECODE`, `K_SEMANTIC`). A bytecode antibody catches a drainer that redeploys to a fresh address but reuses its code, because `checkBytecode(bytes32 codehash)` binds to the code, not the location.

On-chain. The claim lives in registry state at a known address (`0x027F371fB139A57EcD2A2E175d30157eEA1C56de` on Base mainnet), readable by anyone, enforced by the same contract that holds it, with no off-chain pipeline between the write and the read. The subject is a `bytes32`, which constrains what can be expressed: a claim can only bind to something reducible to an on-chain value (an address, a codehash, a typed pattern), and a threat that does not resolve to such a subject cannot be written.

A claim, not a proof. This is the adjective the system is most careful about. An antibody asserts hostility; it does not prove it. Confidence is probabilistic and is raised, never settled, by two mechanisms: `corroborate(uint64 id)`, by which other parties add weight, and `challenge(uint64 id)`, by which a party stakes `challengeStake` (default `200e18` WANE, twice the mint stake, which makes frivolous challenges expensive) against the claim and forces governor-arbitrated `resolve(uint64 id, bool falsePositive)`. A reader chooses how much confidence to demand: `check()` returns `active=true` only when an antibody is

minted, not revoked, and either matured or carrying at least `enforceCorrobs` corroborations (default `2`), and a consumer can set its own corroboration threshold above that floor.

The registry ships with `652` genesis antibodies seeded from known drainer addresses, so the commons is not empty at launch. Those genesis records carry the `Genesis` status; subsequently published records move through `Minted`, then `Active` (`1`, live and enforced on `check`), and can enter `Challenged` (`2`, staked against and under dispute) or `Revoked` (`3`, proven false, minter slashed). The full status lifecycle, the economics that price honest and dishonest writes, and the enforcement surfaces that turn a written claim into a reverted transaction are specified in the sections that follow.

HONEST CAVEAT. The analogy is a design aid, not a guarantee of behavior. An antibody is a staked claim, and a staked claim can be wrong: a false positive that a reader enforces will block a legitimate target until it is challenged and resolved, and the resolution path runs through a governor today (`resolve()` is `onlyGovernor`), which is an admitted trust surface that future work must decentralize. The three borrowed properties, durable, shared, and front-loaded, describe the data structure and the timing of the read. They do not promise that any individual claim is true. Truth in this system is corroborated and contested, never assumed.

3. The threat taxonomy

A drainer does not present itself as a single fact. It presents as an address you should not pay, a call shape you should not sign, a contract body that behaves the way three earlier contract bodies behaved, or an intent that reads as legitimate until you price its outcome. A registry that recognized only the first of these would be a blocklist, and a blocklist is something attackers route around by lunchtime. Wane sorts hostile signal into four kinds, and the same four numeric values mean the same four things on Base and on Solana, so a consumer that learns one chain's vocabulary already speaks the other's.

The kind is part of the antibody's identity, not a label bolted on after the fact. An antibody is keyed by `(ThreatKind kind, bytes32 subject)`, and `check()` takes the kind as its first argument. A consumer asks a precise question ("is this codehash flagged?") and gets a precise answer, instead of dumping every signal into one undifferentiated bucket where an address match and a bytecode match are indistinguishable.

Value	`ThreatKind`	Subject is	What it catches
`0`	`Address`	a 32-byte-encoded recipient or contract address	the known-bad counterparty: a drainer wallet, a sweep contract, a fake-airdrop claimer
`1`	`CallPattern`	a hash committing to a call shape (selector plus argument structure)	the hostile interaction shape: an `approve` to an unexpected spender, a `setApprovalForAll`, a permit signed to a sweeper
`2`	`Bytecode`	a contract codehash (`bytes32`)	the contract body itself, independent of the address it is deployed at
`3`	`Semantic`	a hash committing to a higher-level intent	the outcome that reads wrong even when address, selector, and code all look clean

The policy layer addresses these kinds by bitmask, so a wallet can choose which classes it wants enforced. The flags are `K_ADDRESS = 1<<0`, `K_CALL = 1<<1`, `K_BYTECODE = 1<<2`, `K_SEMANTIC = 1<<3`, and `K_ALL` is all four set. A conservative agent runs `K_ALL`. A latency-sensitive one that already screens addresses elsewhere can enforce, say, only `K_BYTECODE | K_SEMANTIC` and let its existing address feed cover the rest. A consumer enforces the kinds it wants and ignores the rest.

3.1 Why Bytecode is the load-bearing kind

The naive way to ban a drainer is to ban its address. It is also the way that ages worst. Deploying a fresh copy of a malicious contract costs the attacker a deployment fee and nothing else: the new contract lives at a new address, the address antibody that flagged the original now points at a dead instance, and the blocklist is one redeployment behind. This is the structural reason address-only feeds decay. The signal is attached to the most disposable property the attacker controls.

The contract body is not disposable in the same way. When an attacker redeploys identical malicious logic, the new instance carries the same `EXTCODEHASH`, because the codehash is a commitment to the runtime bytecode, not to the deployment address. One `Bytecode` antibody, minted once against the codehash of the original drainer, recognizes every byte-identical redeployment automatically, at every address, forever, with no second mint and no human noticing the new deployment. The attacker can change addresses for free. Changing the codehash means writing genuinely different code, which raises the floor of every copycat campaign from "redeploy" to "re-engineer."

This is what `checkBytecode(bytes32 codehash)` exists for, and it is why the Uniswap v4 hook screens swappers by codehash in addition to address. A swapper that arrives at a flagged-codehash contract is rejected on its first interaction with the pool, even though that specific contract instance has never been seen before, because the body has. `checkAddress(address)` catches the named instance; `checkBytecode(bytes32 codehash)` catches the family. The two together are what turn a registry of past incidents into coverage of future ones.

HONEST CAVEAT. Bytecode immunity is exact-match on the codehash, not behavioral. An attacker who pads the constructor, reorders independent operations, or compiles the same logic under a different optimizer setting produces a different `EXTCODEHASH` and a different subject, which the original antibody does not cover until someone mints against the new body. Bytecode raises the cost of cheap copy-paste redeployment from zero to "modify and recompile." It does not detect logically-equivalent rewrites, and it makes no claim to. `CallPattern` and `Semantic` exist precisely because address and bytecode are both literal: they catch the hostile shape and the hostile outcome when the attacker has changed enough bytes to escape the first two kinds but not enough to change what the interaction actually does.

4. The mechanism

The registry is small enough to state in three verbs. An agent reads it before acting, a victim writes back to it after being attacked, and a third party arbitrates when a write is disputed. Everything else in `WaneRegistry.sol`, the staking, the maturity clock, the slashing, the reward accounting, exists to make those three verbs economically honest. The thesis of the protocol lives here: detection feeds already exist and are commoditized, but an open, staked, on-chain write-back commons, where the freshest signal (the party that was just attacked) can publish back into shared memory, is the part a vendor feed cannot structurally reproduce.

4.1 Read: `check()` and the enforceable rule

Reading is free by default (`checkFee` is `0`) and is the hot path: every screened action calls it.

```
check(ThreatKind kind, bytes32 subject) view returns (bool active, uint64 id)
```

with the convenience wrappers `checkAddress(address)` and `checkBytecode(bytes32 codehash)` . The return is deliberately two-valued: `active` is the enforcement answer, and `id` is the antibody's handle so a consumer can inspect, corroborate, or challenge the exact record that flagged it.

`active` is not simply "an antibody exists." A claim that anyone can mint for `100e18` WANE cannot be trusted the instant it is minted, or the registry becomes a denial-of-service tool where an attacker mints a false flag against a legitimate counterparty and freezes honest agents. So `check()` applies the **enforceable rule**:

`check()` returns `active = true` if and only if the antibody is minted, is not `Revoked` , AND it is either matured (at least `maturity = 21600` blocks old, about 72 hours at 2-second blocks) OR has reached `enforceCorrobs = 2` corroborations.

The two clauses are two independent paths to enforceability, and they encode two different kinds of confidence. Maturity is confidence through unchallenged time: a claim that has stood for 21600 blocks without being successfully disputed has survived the window in which anyone with `200e18` WANE could have challenged it. Corroboration is confidence through independent agreement: once two other publishers stake their own claim that the same `(kind, subject)` is hostile, the flag enforces immediately, before maturity, because fresh attacks are exactly when the protection needs to be fast. A drainer launched an hour ago has no 72-hour history, but if two attacked parties corroborate it, agents are protected within the hour.

The consumer keeps the final say. The contract enforces `enforceCorrobs = 2` as the floor, but a consumer reading `check()` can demand a higher corroboration count before it personally acts on a flag (the antibody record exposes `corroborations` , a `uint32`). A treasury agent moving large value can require more agreement than a low-stakes bot. The registry sets the minimum bar for "enforceable"; each consumer sets its own.

4.2 Write: mint, corroborate, challenge, resolve

Verb	Function	Stake / effect
Publish	`mintAntibody(kind, subject, evidence)`	locks `mintStake = 100e18` WANE; creates the antibody with an `evidence` commitment
Agree	`corroborate(uint64 id)`	independent publisher backs the claim; raises `corroborations` toward `enforceCorrobs`
Dispute	`challenge(uint64 id)` `nonReentrant`

| locks `challengeStake = 200e18` WANE; moves the antibody to `Challenged` | | Arbitrate | `resolve(uint64 id, bool falsePositive)` `onlyGovernor nonReentrant` | settles the dispute, slashes the losing side | | Recover | `reclaimStake(uint64 id)` / `claimRewards()` | minter pulls stake back after maturity; anyone pulls `earned` rewards |

Minting is a staked claim, not a free assertion. To publish that `(kind, subject)` is hostile, a publisher locks `100e18` WANE alongside a `bytes32 evidence` commitment. The stake is the publisher's skin in the game: it is reclaimable after `maturity` if the claim stands (`reclaimStake`), and it is slashable if the claim is proven false. The antibody record that results carries `id` , `publisher` , `stake` (a `uint96`), `mintedBlock` (a `uint64`), `corroborations` (a `uint32`), and `status` .

4.3 The challenge asymmetry

Anyone may dispute a live antibody with `challenge(id)` , which locks `challengeStake = 200e18` WANE and moves the record to `Challenged` . The challenge stake is deliberately twice the mint stake. This asymmetry is the registry's defense against the cheaper of the two abuses.

There are two ways to attack a write-back commons. You can mint false flags to censor honest counterparties, and you can challenge true flags to unstick a drainer you control. Minting already costs `100e18` and exposes the minter to slashing on `resolve` . Challenging is the move an attacker makes to clear a correct antibody that is blocking their address, so it must be the more expensive action, or sweeping the registry of true flags would be cheap. At `200e18` , a frivolous challenge risks twice what a mint risks, and a challenger who loses forfeits the entire bond.

4.4 Resolution and slashing

`resolve(uint64 id, bool falsePositive)` is `onlyGovernor` and `nonReentrant` , and it is the one arbitrated step in the lifecycle.

If `falsePositive` is true, the antibody is `Revoked` , the publisher is slashed, and the challenger receives their bond back plus the slashed stake, with the payout drawn from `reserved` . If the flag is upheld instead, the challenger's bond is forfeit and the antibody returns to enforcement.

The payouts are conservation-respecting: the slashed stake does not appear from nowhere, it moves from the loser's locked balance to the winner inside the registry's accounting. On the read side, when `checkFee` is nonzero, `publisherBps = 8000` directs 80% of any check fees to the publisher of the antibody that did the work, which is what makes maintaining a useful flag a revenue position rather than pure cost.

4.5 Status

| Value | `Status` | Meaning | Effect on `check()` | |-------|----------|---------|---------------------| | (genesis) | `Genesis` | one of the 652 seeded antibodies from known drainer addresses | enforced like any live antibody | | (minted) | `Minted` | freshly minted, before it is enforceable | not yet `active` unless matured or corroborated | | `1` | `Active` | live and enforceable | returns `active = true` | | `2` | `Challenged` | staked against, under dispute | under dispute pending `resolve` | | `3` | `Revoked` | proven false, minter slashed | never returns `active = true` |

`Revoked` is terminal and is the only status that survives a successful challenge. A revoked antibody can never re-enforce, and its minter has been slashed, which is the cost side of the false-flag math below.

4.6 Token accounting: `reserved` and `surplus`

All WANE used by the registry lives inside the registry contract, and the accounting cleanly separates committed value from free value. `reserved` is the sum of every locked mint stake, every challenge bond, and every `earned` (claimable) reward balance. `earned[addr]` is what a given address can pull via `claimRewards()` . `surplus` is `balance - reserved` : the genuinely unencumbered WANE the registry holds. The invariant the contract maintains is that every payout on `resolve` comes out of `reserved` , moving value from a loser's locked position to a winner's claimable balance, so no settlement can pay out value the contract has not already accounted as committed. Slashing is a transfer inside `reserved` , never a mint.

4.7 The economic argument

Theorem (informal, an argument and not a proof). Under the default parameters, the expected cost to a rational actor of minting a false flag exceeds its expected benefit, so the equilibrium behavior of a self-interested publisher is to mint only claims it expects to survive challenge.

Argument. Consider an actor who mints a knowingly false antibody, for instance to censor a legitimate counterparty. The mint locks `100e18` WANE. The false flag is, by construction, wrong, so any party harmed

by it (or any watcher earning by policing the registry) can `challenge` it for `200e18`. On `resolve` with `falsePositive = true`, the antibody is `Revoked` and the minter's `100e18` is slashed and handed, together with the challenger's returned bond, to the challenger. The minter's downside is the full `100e18` stake. The minter's upside is whatever transient censorship it bought during the window before challenge, and that window is itself bounded: a false flag has no real attack behind it to corroborate, so it cannot reach `enforceCorrobs = 2` from honest publishers and must instead wait out `maturity` to enforce, which is `21600` blocks of exposure to a `200e18` challenge it will lose. The honest case is the mirror image: a true flag attracts corroboration, enforces early, survives challenge, returns its stake on `reclaimStake`, and earns `publisherBps = 8000` of any check fees. The parameters are arranged so that being right is a recoverable-stake, fee-earning position and being wrong is a slashed-stake position, with the challenge bond set high enough (`2x` the mint) that clearing a true flag is the most expensive move on the board.

HONEST CAVEAT. This is an economic argument, not a cryptographic guarantee, and it has named gaps. It assumes a rational, capital-constrained adversary; an actor willing to burn `100e18` WANE purely to grief a specific counterparty for one maturity window can do so until challenged, and Wane only guarantees that doing so is loss-making, not impossible. It assumes the challenge path is live, which means it assumes watchers are actually policing the registry and that `resolve` is arbitrated honestly. And `resolve` is `onlyGovernor` today, so the dispute outcome rests on the governor's integrity. That trust surface is stated plainly in the governance and limitations sections; decentralizing arbitration is future work, not a solved problem.

4.8 The antibody lifecycle

Stage	Trigger	State	Enforced by `check()`?
Mint	`mintAntibody`, lock `100e18`	`Minted`	No, not yet enforceable
Early enforce	`corroborate` reaches `enforceCorrobs = 2`	enforceable	Yes, before maturity
Mature enforce	`maturity = 21600` blocks elapse unchallenged	`Active`	Yes
Challenge	`challenge`, lock `200e18`	`Challenged`	Under dispute
Resolve (false)	`resolve(id, true)` by governor	`Revoked`, minter slashed	No, terminal
Resolve (upheld)	`resolve(id, false)` by governor	back to `Active`, challenger bond forfeit	Yes
Reclaim	`reclaimStake` after maturity, claim is standing	stake returned	Yes, still live

The path a legitimate antibody walks is mint, corroborate or mature, enforce, and after maturity reclaim its stake while continuing to enforce. The path a false one walks is mint, fail to corroborate, get challenged during its maturity exposure, and resolve to `Revoked` with the stake slashed. The two paths diverge at exactly the point where the claim's truth gets tested, which is the property a write-back commons needs to stay trustworthy as it grows.

5. Non-bypassability of the binding

A protection layer is only as good as the cheapest way around it. The interesting question is never whether Wane blocks a flagged target when asked nicely. The interesting question is what happens when the screening account is the wrong one, omitted entirely, or swapped for a clean lookalike at the moment a transaction is signed. If the binding between a transaction's real destination and the antibody that covers it can be loosened, the firewall becomes decoration.

This section sets out why the binding holds. The Solana enforcement path is treated first because its address derivation makes the argument concrete and inspectable, then the EVM equivalent is shown to reach the same property through `check(kind, subject)`. The claim is labeled an argument, not a proof, and the caveat at the end states exactly what it does not cover.

5.1 What the binding has to guarantee

The enforcing program signs an outflow only after it has read the registry for the actual destination of that outflow. Three distinct cheats have to fail for that guarantee to mean anything:

1. **Substitution.** The caller supplies a registry account for some clean address, hoping the program reads coverage for the wrong subject and waves the transfer through.
2. **Omission.** The caller supplies no antibody account at all, hoping the program skips the read and defaults to allow.
3. **Misclassification.** The caller supplies the genuine account but under a threat kind the program does not consult for this action.

A binding that survives all three forces the screened subject to be the same subject the transaction actually pays. On Solana that identity is enforced by program-derived address (PDA) seed derivation rather than by a runtime equality check the caller could route around.

5.2 Argument: the Solana PDA binding (argument, not proof)

On the Solana side, every antibody lives at a deterministic address. The client and the program both compute it the same way:

`findProgramAddressSync(["antibody", kind, subject], REGISTRY_PROGRAM)`

The seeds are the literal tag `"antibody"`, the `ThreatKind` discriminant, and the `subject` (the 32 byte destination or codehash under scrutiny). The address is a pure function of those seeds under `REGISTRY_PROGRAM`. There is no nonce a caller can vary, no alternate preimage that lands on the same PDA without already being the same `(kind, subject)` pair.

The enforcing program does not accept a passed-in antibody account on faith. It re-derives the expected address from the real destination of the transaction being signed, then requires that exact account at that exact position. The argument proceeds case by case against the three cheats:

Against substitution. Suppose the caller wants to move value to a flagged address `D` but presents the antibody account for a clean address `C`. The program re-derives the expected PDA from `D`, the field it is actually about to pay, yielding `addr(D) = findProgramAddressSync(["antibody", kind, D], REGISTRY_PROGRAM)`. The account the caller supplied sits at `addr(C)`, and `addr(C) != addr(D)` because the seed `subject` differs. The supplied account is not the account the program demands, and the instruction fails. The caller cannot grind a clean address whose antibody PDA collides with `D`'s, because that would require two distinct `subject` seeds mapping to one PDA, which is the collision the derivation rules out.

Against omission. Suppose the caller supplies no antibody account, betting the program will read nothing and allow by default. The program still re-derives `addr(D)` and requires precisely that account to be present in the instruction. A missing account is not the required account, so the seeds do not match and the instruction fails closed. There is no allow-by-default branch reachable by leaving the slot empty, because the required account is named by derivation, not by the caller's generosity.

Against misclassification. Suppose the caller presents the genuine account for `D` but under a `ThreatKind` the program is not screening for this outflow. The `kind` is itself a seed. An account derived under `Address=0` lives at a different PDA than one derived under `Bytecode=2`, so presenting the wrong-kind account presents the wrong account, and the same seed-mismatch failure applies. The program consults the kinds the action requires, and only the account at the PDA for that exact `(kind, subject)` satisfies the constraint.

The same derivation discipline governs the rest of the enforcement state. The protocol configuration lives at the `["config"]` PDA, each owner's policy at `["policy", owner]`, and each owner's session wallet at `["vault", owner]`. Because these are derived, not passed by reference, a caller cannot point the program at a permissive config it authored or a policy belonging to a different, laxer owner. The session wallet screens every outflow on-chain: the `["vault", owner]` PDA is the thing that signs, and it signs only after the `["antibody", kind, subject]` read for the real destination has been satisfied. A clean substitute or an omission fails for the same reason in every case, the seeds would not match.

Argument, not proof. This is a soundness argument about the derivation, not a machine-checked proof of the deployed program. It assumes the program in fact re-derives from the true destination of the outflow and requires the derived account, which is the design and the intended invariant. It does not certify the absence of an unrelated bug elsewhere in the instruction handler, and it does not extend to value the `["vault", owner]` PDA never authorizes. The property it establishes is narrow and exact: an attacker cannot make the screened subject differ from the paid subject by swapping, dropping, or misclassifying the registry account, because the account is named by the destination itself.

5.3 The EVM equivalent

Base mainnet reaches the same property by a different route. There is no PDA derivation on the EVM, so the binding is carried by the call itself. The consumer (the policy contract, the 7702 delegate, or the v4 hook) calls into the registry at `0x027F371fB139A57EcD2A2E175d30157eEA1C56de`:

`check(ThreatKind kind, bytes32 subject) view returns (bool active, uint64 id)`

with the helpers `checkAddress(address)` and `checkBytecode(bytes32 codehash)`. The `subject` passed to `check` is computed from the transaction the consumer is about to perform: the actual target address for an `Address` screen, the actual runtime codehash for a `Bytecode` screen. The registry returns coverage for exactly that `subject` and no other. The binding holds because the screening consumer derives `subject` from the real call it is gating, in the same execution, before it runs.

This is why the EVM enforcement surfaces, the policy firewall, the delegate, and the hook, all read the registry with a `subject` taken from the live action rather than from an attacker-supplied hint. The recipient a swap actually pays is the `subject` the hook screens. The target an `execute()` call actually invokes is the `subject` the delegate screens. The mismatch the Solana derivation forecloses by seed equality, the EVM forecloses by deriving `subject` in-line from the transaction under enforcement, so a clean substitute never reaches `check` in place of the real destination.

5.4 The shared property, stated once

Across both chains the non-bypassability rests on a single discipline: the subject that gets screened is computed from the action being authorized, not accepted from the party who wants the action authorized. On Solana the computation is PDA seed derivation that makes the wrong subject land on the wrong account. On the EVM the computation is the consumer building `subject` from the live call before it dispatches. Neither path has an allow-by-default branch a caller can reach by handing over a convenient lookalike.

HONEST CAVEAT. Non-bypassability of the binding is not non-bypassability of the protocol. This section shows only that the screened subject equals the paid subject. It says nothing about whether an antibody for that subject exists yet (Section 6 is about coverage at the frontier), nor about the EIP-7702 direct-key-spend bypass, where a key signs a raw transaction that never routes through `execute()` and so is never screened at all. A correctly bound check over an empty registry still allows a brand-new drainer. The binding guarantees you are asking about the right address. It does not guarantee anyone has answered yet.

6. The write-back commons

Detection of malicious on-chain activity already exists as a product. The established vendors simulate transactions and ship threat feeds to wallets, and they do it well. So the honest question for Wane is not whether transaction screening is valuable, it plainly is, but whether anything is left to build that a vendor feed does not already cover. The answer is structural rather than qualitative, and it lives in the direction signal can flow.

6.1 The asymmetry of a feed

A vendor feed is a one-way channel. The vendor observes, classifies, and publishes; the subscriber consumes. A subscriber who gets drained by a brand-new address holds the freshest possible signal about that address, the first confirmed hit, and has no path to put it back into the feed except by filing a report the vendor may or may not action on the vendor's schedule. The economics of the relationship guarantee the asymmetry: the subscriber pays to read and produces nothing the system can use, because the system has no write surface pointed at the subscriber.

This is not a complaint about any particular vendor's diligence. It is a property of the topology. A closed feed concentrates the right to publish in the operator. The people at the actual frontier of new attacks, the freshly drained, are exactly the people the topology silences.

6.2 What a write-back commons changes

Wane keeps the read surface free and adds a write surface anyone can reach. A participant reads the registry with `check(kind, subject)` for free, the same way they would consume any feed. The difference is what happens after a hit lands. A participant who confirms a new malicious address does not file a ticket. They call `mintAntibody(kind, subject, evidence)`, stake `100e18` WANE, and the claim becomes a registry record that every other consumer's next `check` can enforce against.

The signal flows back from the frontier into the shared state. The freshly drained, the participants holding the newest and most expensive information, are precisely the ones the write surface is built for. Coverage at the frontier grows with the number of participants who can publish, not with the headcount of a single operator's research team. This is the network effect a closed feed cannot reproduce by topology alone: every reader is a potential writer, and the marginal antibody can come from anyone who just saw the attack.

Argument (network effect), not proof. The claim is that an open write surface dominates a closed feed at the attack frontier, where the freshest signal is held by the victim rather than the operator. It assumes

participants who get hit are willing to stake `100e18` WANE to publish, and that corroboration and challenge filter the resulting claims well enough to keep the commons usable (Section 4 covers that filtering). It does not claim the commons is more accurate than a vendor on day-old threats the vendor has already curated. It claims the commons is structurally faster at the one place that matters most, the second exposure to a thing the operator has not seen yet, because the first victim can write back.

6.3 Trust the operator versus trust the stake

The two models also differ in what a consumer is asked to trust. A feed asks the consumer to trust the operator: trust that the operator classified correctly, trust that the operator has no incentive to under-report a paying customer's address, trust that the operator's review queue moved fast enough. The trust is in a reputation and a process the consumer cannot inspect.

Wane asks the consumer to trust a stake. Publishing a false antibody costs a `100e18` WANE bond exposed to challenge. Anyone may post `challengeStake` of `200e18` WANE, twice the mint, against a claim they believe is wrong. If a `resolve(id, falsePositive=true)` finds the claim false, the publisher is slashed and the challenger is paid the bond plus the slashed stake from `reserved`. The cost of a frivolous challenge is symmetric: a challenge of a true antibody forfeits the `200e18` bond. The result is a market where putting bad data into the commons has a price, taking bad data out has a reward, and both are denominated in the same staked token rather than in an operator's goodwill.

The threat model under each is different in kind. To corrupt a feed you compromise or pressure one operator. To corrupt the commons you have to out-stake the people willing to challenge you, repeatedly, while every false mint you post bleeds slashed `WANE` to whoever calls it. The honest version of this claim is bounded, and the next subsection states the bound.

6.4 Why a closed feed cannot copy this

A vendor could in principle open a public submission form and pay bounties for new addresses. That narrows the gap without closing it, for two structural reasons.

First, the right to publish stays gated. A bounty form routes every submission through the operator's acceptance decision, so the frontier signal still waits on the operator's queue before it can be enforced. The commons enforces the moment a mint clears its corroboration threshold, with no operator in the path between the victim's write and the next consumer's `check`.

Second, the unit of trust does not transfer. A vendor can publish data, but a vendor cannot let its data be challenged by an outside bond that slashes the vendor when the vendor is wrong, without becoming a staking registry, at which point it has rebuilt the commons. The thing that makes a Wane antibody enforceable to a stranger is not the publisher's reputation, it is the `stake` field on the record and the open right to challenge it. A closed feed can copy the data and cannot copy the bond, because the bond is the part the operator would have to put at risk.

HONEST CAVEAT. The write-back commons reduces the cost of the second exposure onward, never the first. The first victim of a genuinely novel drainer reads an empty registry and is unprotected, exactly as they would be against any system, vendor feed included. The commons does not make claims into proofs: an antibody is a staked assertion, and confidence is probabilistic, raised by corroboration and challenge but never guaranteed. The dispute backstop, `resolve()`, is governor-arbitrated today, which is an admitted trust surface, and decentralizing it is future work. The argument of this section is narrow and load-bearing: at the frontier, where the freshest signal sits with the attacked party, an open staked write surface routes that signal back into enforceable state faster than a closed feed can, and that is the one advantage a feed's topology structurally denies it.

7. The policy firewall

A staked registry tells you a target is bad, and the firewall is what refuses to let your agent touch it.

The registry described so far answers one question: is this subject a known threat? That answer is necessary, and it is not sufficient. An agent runtime that calls `check()`, reads `active=true`, and then proceeds anyway has consulted an oracle and ignored it. Detection that is advisory is detection that gets skipped under load, mis-wired in integration, or simply disabled by a careless operator. The signal has to be load-bearing at the moment value moves, or it does not bind.

`WanePolicy.sol` makes the same registry load-bearing. It is an on-chain policy firewall for agent wallets and for plain EOAs, and it sits in the call path so that a flagged or out-of-scope action reverts before any value leaves the account. The antibody lookup is one of several gates an action must pass. The others are spending caps, allowlists, a kill switch, and a time-to-live on the policy authority itself. Every gate is evaluated in the contract, not in the SDK, not in the agent prompt, not in a relayer. An operator who wants to be reckless has to change on-chain state to do it, and that state is owned by the account holder.

7.1 Enrollment and the dry-run

An account opts in with `enroll(agent, ...)`, which writes a `Policy` record for that agent address into `policies[agent]`. From that point the policy is consulted on enforced paths. Enrollment is the account holder declaring intent: this wallet should be screened, under these caps, against these threat kinds, for this window of time.

Before an action is attempted, a caller can ask the contract what would happen. `wouldAllow(target, value, data)` returns `(bool allowed, uint8 reason)`. It runs the full gate sequence as a view call and returns the verdict plus a reason code, without moving anything. An agent runtime uses `wouldAllow` to pre-screen a planned action and surface a human-readable cause when the answer is no. A monitoring process uses it to audit what a policy will and will not permit, given current registry state. The dry-run and the enforced path share the same logic, so the answer `wouldAllow` returns is the answer the enforced path will produce for the same inputs against the same state.

HONEST CAVEAT. `wouldAllow` is a view over current state. Registry state and policy state can change between the dry-run and the action: a corroboration can cross `minCorrobs`, an antibody can be revoked, the daily window can roll, a guardian can pause. The dry-run tells you the verdict now, not a guarantee about the next block. The enforced path is the only thing that binds, and it re-evaluates at execution time.

7.2 The Policy struct

A policy is a compact on-chain record. Each field is a separately enforced rule, and the combination is the agent's operating envelope.

Field	Type	Meaning
`owner`	`address`	The account that controls this policy. Only the owner changes its terms.
`enabled`	`bool`	Owner intent. When false, the policy is off and the firewall does not enforce for this agent.
`paused`	`bool`	Fast kill switch. Settable by owner or guardian. When true, every action for this agent is refused regardless of the other fields.
`selectorScoped`	`bool`	When true, the function selector allowlist is enforced. When false, selector scoping is not applied.
`tokenScoped`	`bool`	When true, the token allowlist is enforced. When false, token scoping is not applied.
`blockKinds`	`uint8`	Bitmask of the threat kinds this policy enforces (`K_ADDRESS`, `K_CALL`, `K_BYTECODE`, `K_SEMANTIC`, or `K_ALL`).
`minCorrobs`	`uint32`	Sensitivity. The corroboration count an antibody must reach before this policy treats it as enforceable.
`expiresAt`	`uint40`	Policy TTL as a unix timestamp. `0` means never. Otherwise the policy authority dies at this time.
`perTxCap`	`uint128`	Maximum value per single action. `0` means no per-tx cap.
`dailyCap`	`uint128`	Maximum value over a rolling day. `0` means no daily cap.
`spentToday`	`uint128`	Running total of value spent in the current day window.
`dayStart`	`uint64`	Start of the current rolling-day accounting window.

Three of these fields deserve their mechanics spelled out, because they are where the firewall does its quietest work.

`blockKinds` and `minCorrobs` decide which antibodies bind. The threat taxonomy has four kinds, `Address=0`, `CallPattern=1`, `Bytecode=2`, `Semantic=3`, and a policy declares which of them it cares about with the `blockKinds` bitmask (`K_ADDRESS=1<<0`, `K_CALL=1<<1`, `K_BYTECODE=1<<2`, `K_SEMANTIC=1<<3`, `K_ALL` covering all four). A conservative agent that interacts only with a known set of contracts might enforce `K_ADDRESS` alone. An agent that executes arbitrary calldata enforces `K_ALL`. Within the kinds it enforces, `minCorrobs` sets how much agreement an antibody needs before this policy honors it. This is the consumer-side corroboration threshold the registry was designed to expose: the contract's `check()` already gates on its own `enforceCorrobs`, and a policy layers its own stricter `minCorrobs` on top, so a cautious operator can demand more independent agreement than the network minimum before an antibody stops its agent.

`perTxCap`, `dailyCap`, `spentToday`, and `dayStart` bound the blast radius. Antibody screening stops known-bad targets. Caps bound the damage from anything screening did not catch, including the first victim of a novel drainer, which by construction no antibody yet covers. `perTxCap` refuses any single action

over its limit. `dailyCap` refuses any action that would push `spentToday` past the rolling-day limit; the contract advances `dayStart` and resets `spentToday` when the window rolls. A policy with a tight daily cap converts a total-drain into a bounded loss, and it does so without depending on anyone having published an antibody first.

`expiresAt` is a time-to-live on authority. A policy with a nonzero `expiresAt` stops being authoritative after that timestamp. This is least-privilege in the time dimension: grant an agent a spending envelope for a defined window, and let the grant lapse on its own rather than relying on someone to remember to revoke it. An expired policy yields reason code `R_EXPIRED`, distinct from a disabled or paused one, so an operator can tell a lapsed grant apart from an intentional shutoff.

7.3 The guardian and the global surface

Per-agent policy is owner-controlled. Some failures are faster than an owner can react to, and some bad recipients are bad for everyone at once. The firewall carries a separate watcher role and a global layer for exactly those cases.

The **guardian** is a watcher role that can kill agents and set protocol-wide controls. A guardian can flip an individual agent's `paused` flag, and it can set `globalPaused`, a single boolean that pauses every enrolled agent at once. `globalPaused` is the protocol-wide kill switch: when a live exploit is hitting many agents through a shared dependency, one guardian action stops all of them while the situation is assessed, rather than waiting for each owner to pause individually.

The guardian also curates `globalDenied`, a mapping of recipients that no agent may pay regardless of that agent's own allowlist. When an address is added to `globalDenied`, every enrolled agent refuses to send to it, and the action yields reason code `R_GLOBAL_DENY`. This is distinct from per-agent blocklisting: a global denial is a curated, protocol-wide statement that a recipient is hostile to all, used for confirmed drainer infrastructure rather than one agent's local preference.

HONEST CAVEAT. The guardian is a trust surface. A guardian can pause agents and deny recipients globally, which is exactly the power needed to stop a live exploit and exactly the power an abusive or compromised guardian would misuse. The guardian can only block, never move funds, so the worst a hostile guardian does is denial of service, not theft. That bound is real and it is not zero. Constraining and ultimately decentralizing the guardian role is acknowledged future work, not a solved problem.

7.4 The five mappings

The policy state that the gate sequence reads lives in five mappings. Four are scoped per agent; one is global.

Mapping	Scope	Purpose
`policies[agent]`	per agent	The `Policy` record for an enrolled agent.
`allowlist[agent][target]`	per agent	Targets explicitly permitted for this agent.
`blocklist[agent][target]`	per agent	Targets explicitly forbidden for this agent, by local owner choice.
`allowedSelector[agent][bytes4]`	per agent	Function selectors permitted when `selectorScoped` is true.
`allowedToken[agent][token]`	per agent	Tokens permitted when `tokenScoped` is true.
`globalDenied[recipient]`	global	Recipients forbidden for all agents, curated by the guardian.

`allowlist` and `blocklist` are the agent owner's local statements about targets. `allowedSelector` and `allowedToken` are scoping rules that bind only when their respective `selectorScoped` and `tokenScoped` flags are set, which lets an operator pin an agent to a known set of functions and assets without forcing every policy to maintain those lists. `globalDenied` is the guardian's protocol-wide statement, sitting above all per-agent state.

7.5 The ten reason codes

Every verdict the firewall returns carries a reason code. `R_OK=0` means the action passed every gate. The other nine name the specific gate that refused it. The codes are stable, so an agent runtime and an auditor read the same vocabulary, and a refusal is never a bare boolean: it always says which rule fired.

Code	Value	Fires when
`R_OK`	0	The action passed every gate. Allowed.
`R_BLOCKLIST`	1	The target is on this agent's local `blocklist`.
`R_ANTIBODY`	2	The registry has an enforceable antibody covering the target, under this policy's `blockKinds` and `minCorrobs`.
`R_PERTX`	3	The action's value exceeds `perTxCap`.
`R_DAILY`	4	The action would push `spentToday` past `dailyCap`.
`R_PAUSED`	5	The agent's `paused` flag or the global kill switch is set.
`R_GLOBAL_DENY`	6	The recipient is on the guardian-curated `globalDenied` list.
`R_EXPIRED`	7	The policy's `expiresAt` TTL has elapsed

and its authority has lapsed. | | `R_SELECTOR` | 8 | `selectorScoped` is on and the call's function selector is not in `allowedSelector`. | | `R_TOKEN` | 9 | `tokenScoped` is on and the token is not in `allowedToken`. |

The ordering matters less than the property that every non-zero code maps to exactly one enforced rule. When `wouldAllow` returns `(false, 4)`, the operator knows the daily cap stopped the action, not an antibody and not a pause, and can respond to the actual constraint. When the enforced path reverts, it reverts for one of these nine named reasons.

7.6 Least-privilege, enforced before value moves

Put the pieces together and the firewall is a least-privilege envelope evaluated in the contract. An action survives only if it clears every gate at once: not on the local `blocklist` (`R_BLOCKLIST`), not on the global `globalDenied` (`R_GLOBAL_DENY`), not covered by an enforceable antibody under this policy's `blockKinds` and `minCorrobs` (`R_ANTIBODY`), within `perTxCap` (`R_PERTX`), within the remaining `dailyCap` after `spentToday` (`R_DAILY`), not paused locally or globally (`R_PAUSED`), inside the policy's `expiresAt` window (`R_EXPIRED`), using an allowed selector if `selectorScoped` (`R_SELECTOR`), and an allowed token if `tokenScoped` (`R_TOKEN`). Anything that fails any single gate reverts with the matching reason code, and it reverts before the transfer or call runs.

Argument (least-privilege binds because it is in-contract, not because the agent is well-behaved). A policy enforced in application code is advisory: the agent, the SDK, or the operator can route around it. A policy enforced in `WanePolicy.sol` is consulted on the call path itself, so changing the envelope means changing on-chain state that only the policy `owner`, and for pause and global controls the `guardian`, can change. The argument is that an action which would violate a gate cannot execute without first executing a state change to that gate, and the right to make that state change is held by the account holder, not the agent. This is an argument, not a proof: it depends on the call actually routing through the firewall, which on EVM is what the EIP-7702 delegate path and the Uniswap v4 hook are for, and which a raw key spend bypasses. The next section addresses precisely that routing.

The firewall does not replace the registry. It consumes the registry, and it adds the dimensions a pure threat lookup cannot cover: how much an agent may spend, on which functions, in which assets, for how long, and who can hit the kill switch when something is moving faster than an owner can watch. Detection tells the agent a target is bad. The firewall is what makes that knowledge, and the spending bounds around it, refuse the action before the money is gone.

8. The EIP-7702 delegate

The guards described so far protect contracts: an agent wallet that routes its actions through Wane, a Uniswap pool that mounts a hook, a Solana vault that screens its own outflows. The largest population of vulnerable signers is something else. It is a plain externally owned account, one private key, no contract code, no policy struct, holding funds and signing transactions the way every wallet has signed them since the genesis block. EIP-7702 is the first mechanism that lets such an account borrow contract behavior for the duration of a transaction without surrendering the key or migrating to a smart account. `WaneDelegate.sol` is the code an EOA borrows.

8.1 One account update

Under EIP-7702 a wallet signs a single account update that points its code at a delegate address. After that update the account still has its own balance, its own nonce, and its own authority, and it now also has executable code: the delegate's code, running in the account's own context. For Wane the target of that update is `WaneDelegate.sol`. The wallet signs it once.

From that point on, an action the wallet wants to take is submitted through the delegate's `execute()` entrypoint rather than as a bare value transfer or a bare contract call. `execute()` is the screening boundary. Before the inner call runs, the delegate resolves the action's target and consults two sources in order: the antibody registry at `0x027F371fB139A57EcD2A2E175d30157eEA1C56de`, and the wallet's own policy in `WanePolicy.sol`. The registry answers whether the target (or its codehash) carries an `Active` antibody at or above the policy's `minCorrobs` sensitivity. The policy answers whether the action clears the wallet's own caps, allowlists, kill switch, and TTL, returning one of the reason codes `R_OK` through `R_TOKEN` defined in Section 7.

If both clear, the inner call proceeds with the wallet's full authority. If either refuses, the action does not run. The delegate reverts with `Blocked(target, reason)`, where `reason` is the same `uint8` reason code surfaced

by `wouldAllow`, so the caller learns not only that the action was stopped but precisely which rule stopped it: a curated global denylist hit (`R_GLOBAL_DENY=6`), a matured antibody on the recipient (`R_ANTIBODY=2`), an over-cap spend (`R_PERTX=3` or `R_DAILY=4`), an expired policy (`R_EXPIRED=7`), and so on.

8.2 The trust model: the delegate can only block

A delegate runs in the account's own context, which is the same property that makes EIP-7702 useful and the same property that makes a malicious delegate catastrophic. A delegate the wallet should not trust could move every asset the wallet holds, because for the duration of the transaction it is the wallet. The design of `WaneDelegate.sol` is therefore constrained by a single requirement: the delegate must be able to refuse an action and must have no power to do anything else.

That requirement is met structurally, not by audit promise. The delegate never holds funds. It exposes no path that moves the wallet's assets to a destination the wallet did not itself name in the action it submitted. When `execute()` clears an action and runs the inner call, the call is dispatched from the account itself, so `msg.sender == address(this)`: the recipient sees the wallet as the originator, exactly as it would for an unguarded transaction, and the value, target, and calldata are the ones the wallet supplied. Wane sits before the call as a gate that can return false. It does not sit inside the call as a party that can rewrite it. There is no Wane-controlled destination, no fee skim on the inner transfer, no substitution of the recipient. Reading the registry to make the gate decision is free; the `check` path charges nothing by default (`checkFee = 0`), so screening adds no rent.

The same `msg.sender == address(this)` property is also the access guard. A legitimate action under 7702 enters `execute()` with the account as caller, because the account is running its own borrowed code. Any other caller is, by definition, an outsider reaching into the wallet's delegated code to try to spend the wallet's funds. The delegate rejects that caller with `NotSelf`. The full error surface of `WaneDelegate.sol` is small and total: `Blocked(address, uint8)` when the registry or policy refuses, `BatchLengthMismatch` when a batched call is malformed, and `NotSelf` when the caller is not the account itself. There is no administrative function, no upgrade hook in the call path, and no privileged third party who can make the delegate act.

Property	Mechanism	What it forecloses
Cannot hold funds	No balance-custody path in `WaneDelegate.sol`	A compromised delegate draining a pooled balance
Cannot redirect value	Inner call runs with `msg.sender == address(this)`, target/value/data from the wallet	Recipient substitution, fee skim
Cannot be driven by outsiders	`NotSelf` guard requires caller `== address(this)`	A third party invoking `execute()` to spend the wallet
Cannot rent-seek on reads	`checkFee = 0`, registry reads free	Screening becoming a per-action tax
Can only refuse	`execute()` returns the call or reverts `Blocked(target, reason)`	Any action other than block

The asymmetry is the point. A 7702 delegate is one of the most powerful objects a wallet can grant code to, and Wane spends that power on exactly one verb. The wallet keeps full custody throughout.

8.3 Sponsored execution

EIP-7702 supports a flow in which the account update and the resulting action are paid for by a third party rather than by the account itself, which matters for agents and freshly funded wallets that hold the asset they want to protect but little or no native gas. Wane operates a sponsored relayer at `0x733B4cc401286a6B5A0249Eac7a456b098ED0Dab` for this path. The relayer pays for inclusion. It does not change the trust model of Section 8.2: the inner call still runs with `msg.sender == address(this)`, the relayer is not the account and cannot itself drive `execute()` past the `NotSelf` guard, and the screening decision is still made against the registry and the wallet's own policy. The relayer's role is gas, not authority.

HONEST CAVEAT. A 7702 delegate cannot intercept a transaction the key signs directly. The protection in this section holds for actions submitted through `execute()`, which is the path the Wane SDK and an agent runtime call. A raw transfer or contract call that the private key signs and broadcasts on its own, with no routing through the delegate, bypasses the delegate entirely. This is not a Wane-specific weakness; it is true of every 7702 guard, because the delegate is code the account runs when asked, not an interceptor sitting in front of the signing key. Wane's guarantee is precise and bounded: actions that go through `execute()` are screened, and the value of that guarantee is exactly as large as the share of the wallet's activity that goes through `execute()`. For an autonomous agent whose runtime issues every action through the SDK, that share is the whole surface. For a human who occasionally signs a raw transaction in a wallet UI, the directly

signed transaction is unprotected. We state this plainly rather than imply that one signed delegation makes a key unspendable. It does not.

9. Automatic immunity: the Uniswap v4 hook

The delegate of Section 8 protects the signer, but it leaves the counterparty untouched. A pool that lists a token does not control who swaps against it, and a victim who has already published an antibody on a drainer's address gains nothing if that drainer can still trade freely in a venue that never reads the registry. Where Section 8 made a wallet consult Wane before it acts, this section makes a market consult Wane before it lets a swapper in.

9.1 Screening at the pool boundary

A Uniswap v4 hook is a contract a pool attaches at creation that runs at defined points in the swap lifecycle, including before a swap executes. `WaneHook.sol` is such a hook. A pool deployed with the Wane hook mounted screens every swap against the antibody registry at the moment the swap is attempted, with no action required from liquidity providers, no allowlist for traders to join, and no per-trade opt-in. Immunity becomes a property of the venue rather than a discipline each participant has to maintain.

Before a swap settles, the hook resolves the party initiating it and queries the registry along two of the threat kinds from Section 3. It screens by address, asking `checkAddress` whether the swapper carries an `Active` antibody under `ThreatKind.Address`. It also screens by code, asking `checkBytecode` whether the swapper's contract codehash carries an `Active` antibody under `ThreatKind.Bytecode`. A swap whose initiator is flagged on either axis is rejected before it executes; a swap from a clean party proceeds normally. The codehash axis is what makes the hook resistant to address rotation: a drainer that deploys a fresh address from the same bytecode presents a new `Address` subject but the same `Bytecode` subject, and an antibody minted against the codehash catches every clone of that contract at once, which is the property argued in Section 3 for `ThreatKind.Bytecode` as a class-level rather than instance-level signal.

The same enforceability rule from the registry applies here without modification. `check` returns `active = true` only when the antibody is minted, not `Revoked`, and either matured past `maturity = 21600` blocks or already carrying `corroborations >= enforceCorrobs`. A bare unmatured claim with no corroboration does not stop a swap. The hook inherits the registry's staking and corroboration discipline rather than inventing a softer or stricter rule of its own, so a pool that mounts the hook is protected at exactly the confidence level the commons has already established for a given subject.

9.2 Hardened against the origin trick

The natural but wrong way to write a swap screen is to check `tx.origin`, the externally owned account that started the transaction. That choice silently breaks two cases that matter. A 4337 account-abstraction wallet routes its operations through a bundler, so `tx.origin` is the bundler, not the user; screening `tx.origin` would test the wrong party and wave the real swapper through. A contract that initiates a swap on a user's behalf likewise has a `tx.origin` that is some upstream EOA rather than the contract actually trading. `WaneHook.sol` does not read `tx.origin`. It screens the sender presented to the hook, the party that is actually performing the swap in the pool's frame, which keeps the screen correct under account abstraction and under contract-initiated swaps alike.

Screen target	Behavior under 4337 / contract caller	Result
`tx.origin`	Resolves to bundler or upstream EOA, not the swapper	Wrong party screened, real swapper unscreened
sender (Wane's choice)	Resolves to the party actually swapping	Correct party screened on both address and codehash

HONEST CAVEAT. The hook protects pools that mount it. It does nothing for a pool that does not, and it cannot reach a venue outside Uniswap v4. A flagged swapper turned away by a hooked pool can still trade in any unhooked or non v4 venue, exactly as it could before. The hook also enforces the registry's freshness boundary and no more: a subject with no `Active` antibody, including the first victim of a brand new drainer whose signal has not yet been published or corroborated, passes the screen, because there is nothing in the commons to screen against yet. The hook front-loads the cost of every exposure after the first; it does not abolish the first.

10. Security analysis

A security model is only as honest as its weakest disclosed boundary. Most agent-safety systems advertise the cases they catch and stay quiet about the cases they cannot. Wane takes the opposite posture: the contract enforces a fixed set of guarantees that hold regardless of who is operating the protocol, and everything outside that set is named as procedural or disclosed. The dividing line is whether a guarantee survives a hostile governor. A guarantee that survives a hostile governor is enforced. A guarantee that depends on the governor or guardian behaving is procedural. A property that cannot be guaranteed at all is disclosed.

10.1 Enforced versus procedural versus disclosed

The table below separates what the contract guarantees by code from what depends on a human role behaving correctly, and from what Wane openly admits it cannot promise. An enforced property holds even if the governor turns adversarial, because the check happens inside `check`, the firewall, or the delegate `execute` path before value moves. A procedural property is real but contingent on a trust surface. A disclosed property is a limit stated so that no integrator builds on an assumption Wane does not back.

Property	Category	Mechanism
A flagged, enforceable antibody reverts the action	Enforced	`check(kind, subject)` returns `active=true` once the record is minted, not revoked, and matured or `corroborations >= enforceCorrobs` the consumer, hook, delegate, or Solana PDA re-derivation acts on that result before the call runs
Spend caps and allowlists bind every screened action	Enforced	`WanePolicy` evaluates `perTxCap`, `dailyCap`, `allowedSelector`, `allowedToken`, `allowlist`, and `blocklist` in `wouldAllow` and returns a reason code before execution
The kill switch stops an agent instantly	Enforced	`paused` (owner or guardian) and `globalPaused` (guardian) are read in-contract; a paused agent returns `R_PAUSED=5` with no further evaluation
A false claim costs the publisher more than it gains	Enforced	minting locks `mintStake = 100e18` an honest challenger locks `challengeStake = 200e18` a `resolve(id, true)` Revokes the record and slashes the publisher stake to the challenger
Reads never cost the protected party	Enforced	`check`, `checkAddress`, `checkBytecode` are `view` with `checkFee` default `0` protection does not depend on the victim paying at the moment of attack
The 7702 delegate can only block, never redirect	Enforced	inside `execute`, `msg.sender == address(this)` a non-self caller hits `NotSelf` the delegate holds no funds and has no path to move wallet assets the wallet did not request
Genesis seeding is set by the governor	Procedural	`seedGenesis` is governor-controlled; the 652 genesis antibodies are a curated set, trusted because the governor curated them
Economic parameters can change	Procedural	`mintStake`, `challengeStake`, `maturity`, `enforceCorrobs`, `publisherBps`, `checkFee`, the enforce window, and pause are governor-changeable at launch
Disputes are arbitrated by the governor	Procedural	`resolve(id, bool falsePositive)` is `onlyGovernor` and `nonReentrant` the final call on a challenge is a human governor decision today
Fast pause and curated denylist are guardian powers	Procedural	the guardian can pause agents, set `globalPaused`, and populate `globalDenied` this is a fast response surface, not a trustless one
Coverage of a new threat depends on a report	Procedural	an address, call pattern, bytecode, or semantic subject is only enforceable after someone mints it; the registry cannot enforce what no one has written back
An antibody is a staked claim, not a proof	Disclosed	confidence is probabilistic; `corroborations` and challenges move it but never make it certain; consumers choose their own `minCorrobs` threshold to set their own bar

The reason this split matters: an integrator reading the enforced rows can reason about a worst case where the governor is fully compromised and still know that a flagged drainer reverts, a daily cap holds, and the delegate cannot move funds. The procedural rows are the surface to watch and the roadmap targets for decentralization in Section 11. The disclosed row is the one Wane refuses to oversell.

10.2 The cost of a false claim

The economic core of the registry is that lying is more expensive than telling the truth, and that the expense is front-loaded. A publisher who mints a false positive against an honest address locks `mintStake = 100e18` WANE at the moment of minting, before any reward accrues. A challenger who disputes that claim locks `challengeStake = 200e18` WANE, deliberately twice the mint, so that frivolous challenges against good antibodies are themselves expensive. When the governor calls `resolve(id, true)`, the antibody moves to `Revoked=3`, the publisher stake is slashed, and the challenger receives the bond plus the slashed stake as payout from `reserved`. The attacker who wanted to grief a legitimate contract by flagging it pays `100e18` to attempt it and loses that stake when challenged and resolved against. The honest minter who is wrongly

challenged keeps their stake when `resolve(id, false)` upholds the antibody and the challenger bond is forfeit.

This is a security property, not just an economic one. The threat being defended against is registry poisoning: an adversary minting false antibodies to make legitimate addresses unusable for every agent that screens them. The defense is that each false mint is a `100e18` bet the adversary loses on resolution, and each frivolous challenge is a `200e18` bet that loses if the antibody is upheld. Spam at scale is bounded by the staking budget of the spammer, and every unit of that budget is at risk.

HONEST CAVEAT. The slash-on-resolve guarantee is enforced by code, but the determination of `falsePositive` is made by the governor through `resolve`, which is `onlyGovernor`. The economic deterrent is real and in-contract; the arbitration of who was right is procedural today. An adversary who controls the governor could resolve disputes dishonestly. That trust surface is named in full in Section 11 and is the primary decentralization target on the roadmap.

10.3 Threats and the mechanism that meets each

The table below enumerates concrete attack classes and the exact mechanism that answers each. The Class column marks whether the answer is Enforced in-contract (E) or Procedural and bounded rather than prevented (P). The pattern across the P rows is consistent: Wane does not claim to prevent these, it claims to bound the loss or to have a recovery path, and it says so.

| Attack | Mechanism | Class | | --- | --- | --- | | Agent about to call a known drainer | `check(Address, subject)` returns `active=true` for the enforceable antibody; the delegate `execute`, the v4 hook, or the Solana vault rejects the call before value moves; on Base the delegate reverts `Blocked(target, R_ANTIBODY)` | E | | Compromised agent attempts an oversized transfer | `WanePolicy` enforces `perTxCap` and `dailyCap` an over-cap action returns `R_PERTX=3` or `R_DAILY=4` and is blocked even if the target is unflagged | E | | Brand-new drainer no one has reported yet | not stopped at first exposure; the registry holds no antibody for an unminted subject; loss is bounded by `perTxCap` and `dailyCap` and the agent can be paused, and the SECOND exposure onward is covered once anyone writes it back | P | | Adversary flags a legitimate address to deny it | `challenge(id)` opens a dispute by locking `challengeStake = 200e18` `resolve(id, true)` Revokes the false antibody and slashes the publisher to the challenger; the false flag costs the attacker their `100e18` mint stake | E | | Outsider tries to drain an EOA delegated under 7702 | inside `execute`, any caller other than the wallet itself fails the `msg.sender == address(this)` guard and reverts `NotSelf` the delegate never holds or forwards funds | E | | Key holder signs a raw transfer directly, bypassing `execute` | not covered; a delegate cannot intercept a raw transaction the key signs directly, the same limit as every EIP-7702 guard; protection holds only for actions routed through `execute` | P | | Governor acts against the protocol | bounded by the two-step governor transfer and the published intent to decentralize `resolve` and parameter control; not prevented in-contract today and named as a trust surface | P | | Hook screened by spoofed origin | the v4 hook screens `sender` and uses no `tx.origin`, so an account-abstraction or 4337 path cannot launder a flagged swapper through a relayer origin | E |

The E rows compose into a defensible claim: a known threat reverts, an over-cap spend is capped, a wrongly delegated wallet cannot be drained by an outsider, a false flag is punished, and a spoofed swap origin does not work. The P rows are the honest perimeter. The first victim of a novel drainer is unprotected, a direct key spend that never touches `execute` is outside the delegate path, and the governor is a trust surface until decentralized. None of these are hidden in a footnote; they are first-class rows in the threat table because an integrator needs them to size real risk.

10.4 Why reads being free is a security property

It is tempting to treat `checkFee = 0` as a fee decision. It is a security decision. The moment an agent is about to interact with a hostile target is the worst possible moment to require it to hold a balance, route a payment, or complete a second transaction in order to learn that the target is hostile. Wane makes the protective read a `view` call with default fee `0` so that screening is available to a wallet that is empty, paused, or mid-attack. The party with the freshest signal, the one about to be drained, can read the registry without precondition, and the party who published the antibody is still compensated through `publisherBps = 8000` on any check fees an integrator chooses to charge downstream. The protective path and the reward path are separated so that the protective path is never gated on the victim's ability to pay.

11. Trust model and roadmap

Every protocol that claims to protect users has a trust model, and most of them bury it. Wane's trust model is two human roles and a set of named limits. This section states both in full, then gives the order in which the trust surfaces are intended to shrink.

11.1 The two roles

The protocol has a **governor** and a **guardian**, and they are different in kind. The governor is the slow, high-authority role: it seeds genesis via `seedGenesis`, sets the economic parameters (`mintStake`, `challengeStake`, `maturity`, `enforceCorrobs`, `publisherBps`, `checkFee`, the enforce window, and pause), and arbitrates disputes through `resolve(id, bool falsePositive)`, which is `onlyGovernor` and `nonReentrant`. Governor transfer is two-step, so a single mistaken transaction cannot hand the role to an unrecoverable address. The guardian is the fast, low-authority role: it can pause agents, set `globalPaused`, and curate `globalDenied`, the global recipient denylist that applies to all agents. The guardian can stop things quickly but cannot resolve disputes, change parameters, or seed genesis.

The reason for two roles is failure containment. A fast pause power needs to be reachable on short notice, which argues for a smaller operational surface and therefore a weaker key. A parameter and dispute power needs to be deliberate, which argues for a stronger, slower key. Collapsing them into one role would force a single key to be both fast and high-authority, which is the worst combination for an attacker to capture.

HONEST CAVEAT. The governor and the guardian are trust surfaces, and today they are not decentralized. A compromised or malicious governor can resolve disputes dishonestly through `resolve`, change `mintStake` or `challengeStake` to distort the economic deterrent, or seed antibodies that should not exist. A compromised guardian can pause honest agents and add honest recipients to `globalDenied`. The two-step governor transfer and the role split bound some of this, but neither removes the trust. Wane states this plainly rather than describing the system as trustless. Decentralizing these roles is the central item on the roadmap below, and until it ships, an integrator should treat the governor and guardian as parties it is choosing to trust.

HONEST CAVEAT. Wane reduces the cost of the SECOND exposure to a threat onward, never the first. The first victim of a brand-new drainer is unprotected, because no antibody exists for a subject no one has minted. The registry's value is the write-back commons: once the freshest signal, the attacked party, mints the antibody, every subsequent agent that screens the subject is covered. This means Wane is structurally a second-loss-and-onward defense, not a clairvoyant one. An integrator who needs first-shot protection against never-before-seen attacks should pair Wane with predictive simulation, and should not read the registry as a promise of first-victim safety.

11.2 Other disclosed limits

Three further limits are stated for completeness, each already surfaced in earlier sections and collected here so the trust model is in one place.

First, subjects must be expressible on-chain. The threat taxonomy covers `Address=0`, `CallPattern=1`, `Bytecode=2`, and `Semantic=3`, and an antibody can only enforce against a subject that resolves to one of these. Off-chain social engineering is caught only when it terminates in an on-chain subject the agent actually screens. A phishing instruction that convinces a human to sign is outside the registry until it becomes a concrete on-chain destination.

Second, an antibody is a staked claim, not a proof. `corroborations` and `challenge` move confidence in the right direction, and `enforceCorrobs = 2` lets a consumer require corroboration before enforcement, but no count of corroborations turns a probabilistic claim into a certainty. Consumers set `minCorrobs` to choose their own bar, which means the protocol exposes the confidence dial rather than pretending the dial does not exist.

Third, the 7702 delegate protects only what is routed through `execute`. A key that signs a raw transfer directly bypasses the delegate, the same limit that applies to every EIP-7702 guard. The sponsored relayer at `0x733B4cc401286a6B5A0249Eac7a456b098ED0Dab` exists for the sponsored-7702 flow, but it does not change the fact that a direct key spend is outside the screened path.

11.3 Roadmap

The roadmap is ordered by trust surface, smallest blast radius first, because the point of the roadmap is to retire the procedural rows of the Section 10 table.

Decentralize `resolve`. Today dispute resolution is `onlyGovernor`. The first target is to move the `falsePositive` determination from a single governor to a mechanism that does not require trusting one party: staked juror voting, an escalation game, or a challenge market that settles on corroboration and counter-stake rather than on a human call. This retires the largest enforced-versus-procedural gap, because `resolve` is the one governor power that can directly Revoke an honest antibody or uphold a dishonest one.

Decentralize parameters. The governor's ability to change `mintStake`, `challengeStake`, `maturity`, `enforceCorrobs`, `publisherBps`, and `checkFee` is appropriate for launch tuning and inappropriate as a permanent power. The intent is to move parameter changes behind on-chain governance with timelocks, so that a parameter shift is observable and contestable before it takes effect rather than instantaneous at governor discretion.

Constrain the guardian. The guardian's pause and `globalDenied` powers are valuable as a fast response and dangerous as an unaccountable one. The direction is to bound the guardian with expiry on pauses, transparency on `globalDenied` additions, and a path for an agent owner to contest a denial, so that the fast role cannot quietly become a permanent one.

Mainnet on Solana. The same registry model is live on Solana devnet, with the antibody account at `findProgramAddressSync(["antibody", kind, subject], REGISTRY_PROGRAM)`, the `config` PDA at `["config"]`, the policy PDA at `["policy", owner]`, and the vault PDA at `["vault", owner]`. The enforcing program re-derives the antibody address from the actual destination of the transaction being signed and requires exactly that account, so a clean substitute or an omission fails because the seeds would not match. Promoting this from devnet to mainnet is on the roadmap and extends the write-back commons across both chains under one SDK, `wane-sdk`.

The ordering is deliberate. `resolve` first, because it is the power that can corrupt the registry's core claim. Parameters second, because they shape the economic deterrent. The guardian third, because its blast radius is operational rather than economic. Solana mainnet alongside, because it widens the commons without changing the trust model. Each step turns a procedural row from the Section 10 table into an enforced one, and the protocol is finished decentralizing only when the enforced column is the whole table.

12. The wager

Every defense system is a bet about where the next signal comes from and who is allowed to act on it. A vendor feed bets that the operator's pipeline, running one classification cycle behind the edge, is close enough to the frontier to matter. Wane bets the other direction.

THE WAGER. The freshest signal in the entire system is held by the party that was just attacked, at the moment of the attack, before any vendor has classified anything. A defense that gives that party a write path, with economic weight on the write so the next reader can price it, will cover the second exposure faster than any topology that routes the same signal through a central operator first. Wane stakes its design on this single proposition: that an open, staked, on-chain write-back commons beats a closed feed at the one place the closed feed is structurally slowest, the second time a threat appears that the operator has not yet seen. Everything else in the protocol, the typed taxonomy, the enforceable rule, the policy firewall, the delegate, the hook, the slash-on-resolve economics, exists to make that write trustworthy enough to read. If the wager is right, coverage at the frontier grows with the number of parties who can write, not with the headcount of one operator's research team, and the network effect compounds in a direction a vendor feed cannot follow without becoming a staking registry itself. If the wager is wrong, Wane is an expensive way to reproduce a feed. We have stated the boundary that decides it: the commons never protects the first victim, only every victim after. The bet is that the second victim onward is where most of the loss actually lives.

The wager is falsifiable, and it is the right kind of falsifiable. If victims will not stake `100e18` WANE to write back, the commons stays thin and the bet loses on participation. If corroboration and challenge cannot keep the commons clean at scale, the bet loses on data quality. Both failure modes are observable on-chain, which is the standard a security protocol should be held to.

13. Conclusion

The agent economy is being built on a defensive model that silences the people closest to every new attack. The first victim of a drainer holds the freshest, highest-confidence signal in the system, and the prevailing topology gives that victim nowhere to put it except a vendor's queue. Knowledge that was already paid for,

in real losses, is discarded and then regenerated by the next victim, because there is no shared place the first one could write it down where the second one would read it before signing.

Wane is a single structural correction to that model: the right to publish a threat belongs to whoever holds the signal, the publication carries a stake so a stranger can weigh it, and the same registry that holds the claim enforces it before value moves, through a policy firewall, an EIP-7702 delegate, a Uniswap v4 hook, and a Solana vault. It is live on Base mainnet, seeded with `652` genesis antibodies, with every limit stated in the open: the first exposure is never covered, subjects must reduce to on-chain values, claims are staked assertions rather than proofs, and the governor and guardian are trust surfaces a roadmap is built to retire.

The proposition the whole protocol reduces to is this: defensive knowledge in an adversarial system should accumulate at the point where it is generated, and the party that paid to learn a threat should be the party allowed to write it down.

Appendix A: Parameters

All values are the real defaults in `WaneRegistry.sol` and `WanePolicy.sol`. The threat-kind numeric values and policy bitmask flags are identical on Base and Solana.

A.1 Economic and registry parameters

Parameter	Value	Meaning
`mintStake`	`100e18` WANE	Locked to publish an antibody via `mintAntibody` reclaimable after `maturity` if the claim stands, slashable if proven false
`challengeStake`	`200e18` WANE	Locked to dispute an antibody via `challenge` twice the mint, so frivolous challenges are expensive; forfeit if the antibody is upheld
`maturity`	`21600` blocks	About 72 hours at 2-second blocks; after it elapses unchallenged, the minter may `reclaimStake` and the claim is enforceable by time
`enforceCorrobs`	`2`	Corroborations that make a claim enforceable immediately, before maturity
`publisherBps`	`8000`	80% of any check fees routed to the antibody's publisher
`checkFee`	`0` (default)	Read cost; `check`, `checkAddress`, `checkBytecode` are free `view` calls by default
Genesis antibodies	`652`	Seeded from known drainer addresses via `seedGenesis` at launch

A.2 Threat taxonomy (`ThreatKind`, identical on both chains)

Value	`ThreatKind`	Policy flag
`0`	`Address`	`K_ADDRESS = 1<<0`
`1`	`CallPattern`	`K_CALL = 1<<1`
`2`	`Bytecode`	`K_BYTECODE = 1<<2`
`3`	`Semantic`	`K_SEMANTIC = 1<<3`
(all four)		`K_ALL`

A.3 Antibody status

Value	`Status`	Effect on `check()`
(genesis)	`Genesis`	Enforced like any live antibody
(minted)	`Minted`	Not enforceable until matured or corroborated
`1`	`Active`	Returns `active = true`
`2`	`Challenged`	Under dispute pending `resolve`
`3`	`Revoked`	Never returns `active = true` terminal, minter slashed

A.4 Policy firewall reason codes (`WanePolicy.sol`)

Code	Value	Fires when
`R_OK`	`0`	Action passed every gate
`R_BLOCKLIST`	`1`	Target on the agent's local `blocklist`
`R_ANTIBODY`	`2`	Enforceable antibody covers the target under this policy's `blockKinds` and `minCorrobs`
`R_PERTX`	`3`	Value exceeds `perTxCap`
`R_DAILY`	`4`	Action would push `spentToday` past `dailyCap`
`R_PAUSED`	`5`	Agent `paused` flag or `globalPaused` is set
`R_GLOBAL_DENY`	`6`	Recipient on guardian-curated `globalDenied`
`R_EXPIRED`	`7`	Policy `expiresAt` TTL elapsed
`R_SELECTOR`	`8`	`selectorScoped` on and selector not in `allowedSelector`
`R_TOKEN`	`9`	`tokenScoped` on and token not in `allowedToken`

A.5 Solana PDA seeds

State	Derivation
Antibody	`findProgramAddressSync(["antibody", kind, subject], REGISTRY_PROGRAM)`
Config	`["config"]`
Policy	`["policy", owner]`
Vault (session wallet)	`["vault", owner]`

Appendix B: Live deployment (Base mainnet)

Chain: Base mainnet, chainId `8453`. The same registry model runs on Solana devnet; Solana mainnet is on the roadmap.

B.1 Contract and token addresses

Component	Address
Registry	`0x027F371fB139A57EcD2A2E175d30157eEA1C56de`
WANE token	`0x1465E33f687C557BF275D6d692eC1316126d8e9e`
Delegate	

(EIP-7702) | `0x6350D5850143277F7657549FB505569917641927` | | Sponsored relayer | `0x733B4cc401286a6B5A0249Eac7a456b098ED0Dab` |

B.2 Token

| Property | Value | | --- | --- | | Total supply | `1,000,000,000` (1B) WANE, fixed | | Mint status | Already minted on Base mainnet | | Role | Stake and reward currency for the registry |

B.3 Contracts and resources

| Resource | Location | | --- | --- | | Contracts and SDK | github.com/WaneProtocol | | SDK package | `wane-sdk` (one package, two chains: `Wane.base({agent})` viem-backed, `Wane.solana()` web3.js-backed) | | Site | wane.network | | X | @wanedotnetwork |