Mephis: A Recursive Reinforcement Hive for Coordinating Agent Swarms in Weaponized Exploitation of Smart Contracts

Mephis Research

.

August 20, 2026

Abstract

We present **Mephis**, a closed-loop audit system that pairs a council of specialized language-model agents with a deterministic runtime - RPC simulation, atomic impact measurement, an authorization-policy engine, and persistent cross-audit memory - for reviewing smart contracts the operator is explicitly authorized to test. Rather than scanning for generic "issues," Mephis forms and tests falsifiable hypotheses about *permissionless fund-moving paths*: a proposing agent conjectures a path by which an unprivileged caller could relocate value, and every conjecture is confirmed or refuted against on-chain simulation before it is reported. Two controls make the system defensive by construction. First, it is **authorization-gated**: a signed **target-authorization allowlist** of `{chainId, address, mandate-ref, expiry}` entries is checked before any analysis begins, and a contract absent from it is refused. Second, its only on-chain capability is a **rescue-and-return lane** that relocates at-risk funds to a pre-registered, multi-party recovery multisig fixed at authorization time, under a mandatory responsible-disclosure workflow - never an extraction to an operator-chosen address. We contribute: (i) a six-agent council coordinated over a shared blackboard; (ii) value-of-information probe scheduling over a simulation oracle; (iii) funded simulation via state overrides to model flash-capital adversaries safely inside a sandbox; (iv) reusable cross-audit memory; and (v) an integrated authorization-and-rescue safety architecture. Scope: Ethereum mainnet, verified contracts, defensive use only.

1 Introduction

Authorized security review of modern decentralized-finance (DeFi) protocols is labor-intensive and unevenly covered. A production protocol composes upgradeable proxies, external price oracles, callback-bearing token standards, and cross-contract accounting, and the reviewer must reason about the emergent state machine rather than any single function. Among the defects that survive into deployment, the highest-severity class is narrow and well-defined: a *permissionless fund-moving path* - a sequence of calls available to an unprivileged caller that ends with protocol-controlled value leaving its intended custody. Mephis is built specifically to find, and to help remediate, that class on contracts the operator is authorized to examine.

The system is a closed loop of specialized language-model agents plus a deterministic runtime (RPC simulation, impact measurement, a live authorization policy, and memory). It does not enumerate generic weaknesses; it hunts permissionless paths that could move funds, and then confirms or refutes each one against on-chain simulation - and, for authorized rescue only, against live transactions. Grounding is the central methodological commitment: a language model is used to *propose* hypotheses, never to *adjudicate*

them. Every claim reduces to a measured, atomic balance delta produced by executing the candidate path against forked mainnet state, which defeats the hallucination failure mode that makes unaided LLM audit output unreliable.

The runtime is a local **Ollama** endpoint at `http://localhost:11434`, default model `minimax-m3:cloud`, driven in JSON mode. Proposing agents run at temperature `0.2`; an ensemble-diversity configuration raises this to `0.6` when deliberately widening the hypothesis space. Each call carries a `240s` timeout and performs exactly one strict-JSON retry on a parse failure before the turn is abandoned. A neutral **test-principal** address - an unprivileged, sandbox-only identity that models the adversary - is injected into every agent's system prompt so that all reasoning is framed from the standpoint of what an outside caller can and cannot do.

Mephis is intended solely for targets the operator owns or has deployed, or for which the operator holds an explicit written bug-bounty scope or engagement letter. Authorization is not a usage guideline layered over a general-purpose tool; it is a precondition enforced by the runtime, described in Section 2. The contributions of this paper are:

- **A six-agent council with a shared blackboard.** Specialized agents - cartography, hypothesis proposal, adversarial critique, probe planning, simulation adjudication, and disclosure - collaborate through a common structured workspace rather than a fixed pipeline.
- **Value-of-information probe scheduling.** Simulation is the scarce resource; Mephis schedules the next probe against the simulation oracle by expected information gain, prioritizing experiments most likely to confirm or refute an open hypothesis.
- **Funded simulation via state overrides.** Balance and storage overrides let the sandbox model a flash-capital adversary - a test principal transiently commanding large capital - without any real funds, capturing capital-dependent paths safely.
- **Cross-audit memory.** Confirmed and refuted findings, protocol fingerprints, and probe outcomes persist across engagements, so later audits inherit earlier structural knowledge.
- **An authorization-and-rescue safety architecture.** A signed target allowlist, an operator-identity lock, and a rescue-only live lane bound to a fixed recovery multisig together constrain every capability to authorized targets and to fund return.

2 Threat Model, Scope, and Authorization

This section defines the actor Mephis reasons about, the controls that bind its capabilities to authorized targets, and the behaviors it deliberately places out of scope. The controls described here are prior to, and independent of, the audit methodology in later sections: they determine *whether* a target may be examined at all, not *how* it is examined.

2.1 Adversary model

The adversary Mephis models is an **unprivileged caller**, represented throughout by a neutral **test principal** - a sandbox-only address holding no protocol role, no ownership, no governance rights, and no allowances beyond those any external account could obtain permissionlessly. Modeling from this standpoint is what makes findings meaningful: a path is interesting precisely when the test principal, starting from

public state, can drive protocol value out of custody. *In scope* for hypothesis generation are permissionless fund-moving paths, including authorization-bypass paths in which the caller first defeats an access-control check and then moves value - but such paths are pursued only on authorized targets and only to demonstrate the defect within simulation or to effect an authorized rescue. The adversary is assumed able to command transient capital (modeled by state overrides, Section on funded simulation) and to order and combine calls arbitrarily within a transaction, but is *not* assumed to hold any legitimate privilege.

2.2 Authorization architecture

The primary safety control is the **target-authorization allowlist**: a signed manifest enumerating exactly the contracts the operator may examine. Each entry binds a target to a mandate and an expiry, and the manifest additionally fixes the recovery destination for any rescue action. Each allowlist entry has the shape `{chainId, address, mandate-ref, expiry}`, where `mandate-ref` references the ownership record, bug-bounty scope, or engagement letter that authorizes the target and `expiry` bounds the authorization in time. Alongside the target entries the manifest carries a single fixed **recovery-multisig safe-harbor address** - the multi-party destination to which at-risk funds may be relocated.

The allowlist is verified *before cartography* - before any bytecode is fetched for analysis, any hypothesis is proposed, or any simulation is run. A target whose `{chainId, address}` pair is absent from a valid, unexpired, signature-checked manifest entry is refused outright; there is no analysis path that bypasses this check. Authorization is conjunctive: **both** operator authorization (Section 2.3) **and** target authorization must hold simultaneously. A correctly authenticated operator may not examine a target absent from the allowlist, and a listed target may not be examined by an unauthenticated operator. Neither condition alone is sufficient.

Table 1: In-scope versus out-of-scope behaviors under the Mephis threat model.

Dimension	**In scope**	**Out of scope**
Target	Contracts on the signed allowlist (owned/deployed, or under written bounty scope or engagement letter), unexpired.	Any contract not on the allowlist, expired entries, or wrong `chainId`.
Actor modeled	Unprivileged caller (neutral test principal) starting from public state.	Legitimate owner, DAO, or role-holder exercising already-held privilege.
Defect class	Permissionless fund-moving paths, including authorization-bypass that then extracts value.	Denial of service, griefing, gas-exhaustion, and other non-fund-moving nuisances.
On-chain action	Rescue-and-return of at-risk funds to the fixed recovery multisig, under mandatory disclosure.	Any transfer to an operator-chosen address; any extraction for profit.
Capital	Simulated flash-capital via sandbox state overrides.	Deployment of real attacker capital against a live target.

Note. The target-authorization allowlist is the primary safety control: it is checked before any analysis begins, and a contract not on it is refused. The recovery-multisig safe-harbor address is fixed in the signed manifest at authorization time and is never selected at runtime, so no rescue transaction can route funds to a destination the authorizing parties did not pre-register.

2.3 Operator identity lock

Independently of target authorization, Mephis binds execution to an authorized operator. Invoking `Run()`, and every live action thereafter, requires that `SHA-256(lowercase operator key material)` equal a configured `AUTHORIZED_OPERATOR_HASH`. Only the hash is configured; the allowed identity is *not* stored in plaintext anywhere in the user interface, so the interface reveals no credential even to a party with full read access to it. A mismatched wallet or key fails the unauthorized gate and halts execution before analysis. Because the identity check is re-evaluated on entry to any live path - not merely once at startup - a session cannot be escalated to a live rescue action by an identity other than the one that satisfied the gate.

2.4 Out of scope

Several behaviors are deliberately excluded, both to keep the system defensive and to keep its findings actionable:

- **Legitimate privileged use.** Actions available only to an owner, administrator, DAO, or other role-holder exercising already-held privilege are not modeled as attacks. Mephis reasons from the unprivileged test principal; a "finding" that requires legitimate privilege is not a permissionless fund-moving path and is refuted.
- **Denial of service and griefing.** Paths that degrade availability, waste gas, or inconvenience users without moving protocol funds are outside the defect class Mephis targets.
- **Unauthorized targets.** Any contract absent from the signed allowlist is refused before analysis, regardless of how interesting it may appear; there is no reconnaissance mode that operates outside the manifest.
- **Value extraction to any non-recovery destination.** The live lane can relocate at-risk funds only to the pre-registered recovery multisig. It has no capability to route value to an operator-chosen or runtime-supplied address, and it is never described or used as a profit or exploitation tool.

3 Protocol Cartography

Protocol cartography is the phase in which Mephis constructs a bounded map of the contracts it is permitted to reason about. It runs only after the **target-authorization allowlist** of Section 2 has admitted the primary target; cartography inherits that decision and never widens it silently. A target is admitted only when it is a *verified* contract exposing a published ABI (retrieved through the Etherscan v2 verified-source interface). An unverified address, or an externally owned account (EOA) with no code, is refused at this gate: without source and ABI there is nothing an auditor can reason over, and the system declines to guess.

Around the primary target, Mephis performs **related-contract ingestion** - the discovery of *satellites*, the collaborating contracts (tokens, vaults, oracles, routers, factories) without which a protocol cannot be understood. The satellite set is deliberately capped at **14** addresses so that the cartographic graph remains legible to both the human operator and the reasoning agents, and so that ingestion cannot fan out unboundedly across an entire chain.

3.1 Satellite discovery sources

Candidate satellites are gathered from several independent channels, each of which is a normal artifact of reading a protocol rather than a probing behavior:

- **Analyst-supplied addresses** typed alongside the primary target at engagement setup.
- **Hex addresses mined from source text** - literal `0x...` constants embedded in the verified Solidity of already-loaded contracts.
- **Zero-argument address getters** invoked on loaded ABIs. Mephis recognizes the conventional accessor names that expose a protocol's wiring, including `asset`, `token`, `underlying`, `want`, `stakingToken`, `rewardToken(s)`, `loanToken`, `collateralToken`, `factory`, `router`, `pair`, `pool`, `oracle`, `priceOracle`, `vault`, `strategy`, `lender`, `lendingPool`, `aToken`, `debtToken`, `gauge`, `minter`, `weth/WETH`, and `usdc/USDC`.
- **Model-requested ingestion** - explicit `ingest:[0x...]` lists proposed by the analyst, strategist, or hunter agents during reasoning.
- **Addresses returned in probe results** during sandboxed simulation.

Discovery is filtered before any fetch. Mephis discards the zero address, the sentinel values `0x1`, `0xfff...fff`, and `0xeee...eee` (the conventional native-asset placeholder), addresses that are already loaded, and addresses whose source fetch fails. The filters keep the graph free of degenerate or duplicate nodes.

3.2 Satellite typing and authorization boundary

Each satellite that is itself verified receives its own flattened source and ABI, and is subjected to the same cartography as the primary. A satellite that is *unverified* is still usable where it presents a standard token interface: Mephis binds a built-in ERC-20 ABI covering `approve`, `transfer`, `transferFrom`, `balanceOf`, `allowance`, and `decimals`, which is sufficient to model value flows through the token without pretending to full knowledge of its implementation.

> **Note. Authorization applies transitively, but never expands the mandate.** A satellite is admitted to the graph as analytic context regardless of ownership, because understanding a vulnerable contract requires reading its neighbors. However, the **rescue mandate** does not follow the graph. A satellite that lies *outside* the authorization boundary established in Section 2 - one for which the operator holds neither ownership nor an explicit written bug-bounty scope - is treated as **read-only context only**. Its source informs analysis; it is never a target of a live rescue transaction. The set of contracts Mephis may act upon is the intersection of the discovered graph with the signed authorization manifest, and that intersection is computed before the live lane is ever reachable.

3.3 The protocol graph

The assembled cartography is rendered as a **protocol graph** and presented to the reasoning agents as two clearly delimited regions: a **PRIMARY** section carrying the mandated target's function signatures and

source windows, and a **RELATED** section carrying each admitted satellite with its signatures and source windows. The partition is not cosmetic - it mirrors the authorization boundary, so that at every step an agent can distinguish the contracts it may propose rescuing from the contracts it may only read.

4 Source Windowing (Coverage Rotation)

Verified contracts arrive from Etherscan as a multi-file JSON bundle. Mephis first **flattens** this bundle into a single Solidity blob so that cross-file references resolve within one coordinate space, then **chunks** the blob into overlapping windows of approximately **12,000 characters** with a **1,000-character overlap**. The overlap ensures that a function straddling a window boundary is never severed - it appears whole in at least one window.

Alongside the windows, Mephis builds a **source map**: an index listing every window together with the names of the functions it contains. The source map is what lets an agent navigate by intent ("show me the withdrawal path") rather than by byte offset.

4.1 Agent-directed focus

Agents steer their own reading through a `sourceFocus` request with the shape `{ contract: 'primary' | 0x..., chunks: [], functions: [], next: bool }`. An agent may name specific chunk indices, name functions (which the runtime resolves to their containing windows through the source map), or set `next: true` to advance to the next unread material. The runtime loads the requested windows and **marks them seen**.

4.2 Coverage as a completeness discipline

The core guarantee of this phase is that an audit does not conclude on an unread contract. If the model neglects to advance, the runtime **auto-advances** unread primary windows on its behalf, so that coverage progresses even when the agent's attention drifts. The stop condition is coupled to coverage: Mephis *prefers not to finish* while any primary window remains unseen, and will only accept termination when the model asserts `done: true` *after* every primary window has been either read or explicitly dismissed. A **coverage status** summary - how many primary windows are seen, how many remain - is injected into the context every round, so the decision to stop is always made against a current, visible accounting rather than an assumption.

Algorithm 1: Primary-source coverage rotation

```
input:  windows W = [w0 ... wN]   // flattened, chunked primary source
        sourceMap M            // window -> { functions }
state:  seen  W  (initially )

each round:
    // 1. honor the agent's explicit focus request, if any
    if agent emits sourceFocus f:
        targets <-
        for c in f.chunks:      targets <- targets  { W[c] }
        for fn in f.functions:  targets <- targets  resolve(M, fn)
```

```
        if f.next:              targets <- targets  { firstUnseen(W, seen) }
        load(targets);  seen <- seen  targets

    // 2. guard against a forgetful model: never stall coverage
    else if (W \ seen) != :
        w <- firstUnseen(W, seen)
        load({ w });  seen <- seen  { w }     // auto-advance

    // 3. report current coverage to the agent every round
    inject status: |seen| / |W| primary windows seen,
                   unseen = W \ seen

    // 4. termination is gated on coverage
    if agent asserts done = true:
        if (W \ seen) = :            accept  -> stop
        else if unseen windows are explicitly dismissed:
                                      accept  -> stop
        else:                         refuse  -> continue

output: analysis over fully covered (or explicitly dismissed) primary source
```

The asymmetry is intentional. Satellite windows may be sampled as context demands, but *primary* windows - the source of the contract actually under mandate - are held to a rotation that must complete. Coverage rotation converts "the auditor read what caught their eye" into "the auditor accounted for every line of the target, and said so."

5 Value-Sink Coverage

Reading every line of source guarantees exposure but not attention: the lines that matter most to a fund-rescue audit are those where value can leave a contract. Mephis therefore maintains a distinct completeness discipline over **value sinks** - the public functions through which assets move, are minted, or are authorized to move.

5.1 Identifying value sinks

A public function is classified as a value sink when its name matches any of the following recognized value-movement verbs:

Table 2: Value-sink name signatures

Category	**Matched function names**
Asset egress	`withdraw`, `redeem`, `claim`, `harvest`, `collect`, `skim`, `exit`, `unstake`
Debt & leverage	`borrow`, `liquidate`, `flash`
Supply change	`mint`, `burn`, `deposit`
Token authority	`transfer`, `transferFrom`, `approve`, `permit`
Execution & routing	`swap`, `execute`, `settle`, `bridge`, `finalize`, `performUpkeep`

5.2 The open/touched ledger

Every discovered sink is entered into a ledger and displayed with a status tag: `[open]` for a sink that has not yet been reasoned about, and `[touched]` once an agent has analyzed it. The ledger is a running audit checklist - at any moment the operator can see which value-movement paths have received attention and which have not.

This ledger drives an **audit-completeness guarantee** that operates in addition to, and independently of, the source-coverage rotation of Section 4. A model's assertion of `done: true` is **refused while any sink remains `[open]`**. The audit cannot be declared finished with an unexamined path through which funds can exit the contract.

The guarantee admits one disciplined escape. A sink may be closed without full finding analysis when the agent's recorded *thoughts* contain an explicit **dismiss reason** - a stated justification for why that path is not a concern (for example, an egress function guarded by an access control the agent has verified) - or the summary attestation `value sinks reviewed`. Dismissal is thus always *on the record*: the completion path requires either that every value sink was analyzed, or that a named rationale exists for setting it aside. There is no silent exit.

> **Note.** Together, Sections 4 and 5 form a two-axis completeness contract. Coverage rotation ensures no *line* of the mandated target goes unread; value-sink coverage ensures no *egress path* goes unconsidered. An audit that terminates has, by construction, either accounted for every window and every value sink, or left a written record of what it consciously dismissed - the evidentiary basis a responsible-disclosure report and any subsequent rescue action are expected to rest upon.

6 The Council: Six Specialized Agents

Mephis's analytic core is a heterogeneous multi-agent ensemble - the **Council** - that reasons over a single authorized target rather than a monolithic prompt. Each agent has a narrow remit, a fixed activation schedule, and a typed output contract, and all of them read from a shared **blackboard**: `council = { analyst, strategist, composer, memos[] }`. A helper, `formatCouncilIntel()`, concatenates the current analyst model, strategist thesis, composer fusion, the knowledge graph, and the last eight cross-agent memos into a single intelligence briefing; every hunter, strategist, composer, and divergent turn receives this briefing as context. Because the entire Council is scoped to a contract already cleared by the target-authorization allowlist (Section 1), the roles below describe an *audit* of an authorized system: they map **unprivileged execution paths**, produce **findings**, and, where warranted, assemble a sandbox **demonstration**. No agent selects an extraction address; where a proof-of-impact requires an on-chain step, it is expressed as a rescue plan whose destination is the pre-registered recovery multisig fixed in the signed manifest.

The vocabulary is deliberately defensive. Where an unscoped tool would speak of an *attacker* crafting an *exploit*, the Council speaks of a **test principal** - a neutral, unprivileged caller modeled only inside the simulation harness (Section 11) - traversing an unprivileged path to a **finding**. A hypothesis that reaches `status: confirmed` is a finding an auditor can reproduce and hand to the protocol team through the responsible-disclosure workflow, not a payload.

6.1 Roles, Schedules, and Outputs

The six agents divide the audit into distinct cognitive modes. The **Analyst** builds the ground-truth model once at bootstrap. The **Strategist** sets impact priorities and chains multi-stage theses. The **Hunter** is the main loop that turns theses into concrete probes against the sandbox. The **Composer** fuses partial results into stronger multi-stage findings using the knowledge graph. The **Divergent** agent is an adversarial second opinion that assumes the Council missed a subtle path. The **Critic** is a skeptic that refutes false positives before any finding is reported.

The activation schedule is what keeps the ensemble convergent rather than merely verbose. The Analyst and the bootstrap Strategist run before the Hunter's first turn, so round 1 begins against a complete model rather than a blank contract. The Strategist and Composer then re-enter on a fixed 3-round cadence and, crucially, *on any confirm*, so a newly reproduced finding immediately triggers re-planning and fusion instead of waiting for the next scheduled slot. The Divergent and Critic agents bracket the settled result: divergence widens the search after the first settlement, and the Critic narrows it, refuting before disclosure. Severity, scope, and the confirm/refute discipline follow the detection taxonomy of Section 4 - the Council reports classes of finding an auditor can verify, never weaponized recipes.

7 Intelligence Sharing and Working Memory

The Council behaves as a single investigator only because its agents share memory in two distinct registers: a **short-term working memory** that is common to the hive, and a **long-lived conversational memory** that belongs to the Hunter alone. Keeping these separate is what lets specialized agents cross-pollinate without collapsing into one undifferentiated context window.

7.1 The Bootstrap Handoff

An audit opens with a fixed relay: `analyst` → `strategist` → `hunter round 1`. The Analyst produces the protocol model; the Strategist converts it into an impact thesis and opening hypotheses; the Hunter receives both - folded into the Council intel briefing - together with the protocol bundle for the authorized target. Only then does probing begin, and every probe is dispatched against the sandbox, never against live state.

7.2 Cross-Agent Memos: The Hive's Short-Term Memory

On every Hunter turn, the `intelOut` field is decomposed into memo lines addressed to specific peers - `HUNTER` → `strategist`, `HUNTER` → `composer`, `HUNTER` → `analyst` - each truncated to keep the channel terse. These lines are appended to `council.memos`, which is maintained as a **rolling last-8 list**. That window is the hive's short-term working memory: it is exactly the set of memos that `formatCouncilIntel()` replays into the next strategist, composer, hunter, and divergent turn. Bounding it at eight entries keeps the shared context focused on recent developments and prevents stale intelligence from dominating re-planning.

7.3 Scheduled Re-Planning and Probe Injection

Two conditions trigger a refresh of shared intelligence: *every three successful probe passes*, or *any confirmed hypothesis*. On a refresh, the Strategist re-plans against the updated blackboard, the Composer fuses partial results into multi-stage findings, and newly produced hypotheses are merged into the working set by `id`. Any

Table 3: The six Council agents - remit, activation schedule, and typed output - Every output line is written to the shared blackboard; hypotheses across agents are merged by stable `id`.

Agent	When it runs	Output
Analyst	Bootstrap, once.	A full protocol model of the authorized target: purpose, actors, externals and their interactions, state variables with adversarial relevance, value flows, invariants, trust assumptions, user journeys, keeper loops, composition edges, and blind spots. Also emits an `ingest` list (may pull in missing peer contracts, limit 6), a `sourceFocus`, and a `summary`.
Strategist	Bootstrap, then every 3 probe rounds, and on any confirm.	A maximum-impact thesis, prioritized surfaces, and multi-stage chained plans (setup → manipulate → demonstrate, or relocate-in-simulation, across contracts). Emits a variable matrix (amounts, actors, tokens, ordering, externals), `hypothesesToOpen`, and a `memoToHunter`. Assumes the test principal can flash-borrow inside the sandbox only (Section 11); draws on prior-audit memory; incorporates hunter feedback.
Hunter	Main loop, up to 120 rounds.	Turns Council intel into hypotheses and probes; must not ignore a strategist or composer chain without probing or refuting it, and may invent new paths. Schema: `thoughts`, `hypotheses[]`, `probes[]`, `ingest`, `sourceFocus`, `intelOut`, `done`. Each hypothesis carries `technique`, `severity`, `status`, `variables{}`, `explanation`, `mechanism`, `outcome`, `evidence`, `test`, and `plan.steps[]` with phases and a `rescue{}` block. Emits `intelOut {forStrategist, forComposer, forAnalyst}` into the memos.
Composer	Every 3 probe rounds and on confirm; also every 2 divergent rounds.	Fuses the analyst model, strategist chains, and hunter hypotheses and results into stronger multi-stage findings, using the knowledge graph - a hypothesis that *pushes* a token (+) can feed one that *pulls* the same token (−). May return upgraded hypotheses (merged by `id`) and preferred probes, and may set `fund:true` on a fused probe to request sandbox capitalization.
Divergent	After first settlement, up to 8 rounds.	A rival mind that assumes the Council missed a subtle multi-stage path; uses the same schema as the Hunter and is merged by `id`.

probes the Composer prefers are injected into the Hunter's next user message, marked as *'prefer these,'* so fusion output steers the main loop directly rather than through a side channel. During divergence, the same refresh fires on even rounds, keeping the rival agent synchronized with the rest of the Council.

7.4 The Hunter's Conversational Memory

Separate from the shared memos, the Hunter maintains its own growing `messages[]` array - system prompt, successive user briefings, its own assistant turns, and the probe-result user messages returned from the sandbox. This gives the Hunter continuous **intra-agent memory** of the entire audit: it can recall a probe it ran forty rounds earlier and the exact delta that came back, whereas the other agents see only the compressed last-8 memo window. The two registers are complementary - the memos carry cross-agent intelligence in bounded form, and the Hunter's transcript preserves the full investigative trail behind a single line of that intelligence.

Algorithm 2: Information flow across a Mephis audit

```
BOOTSTRAP
  analyst.model      := ANALYST(target, peers)        // once
  strategist.thesis  := STRATEGIST(analyst.model)     // once
  intel              := formatCouncilIntel()
                        // = analyst + strategist + composer
                        //   + knowledgeGraph + memos[last 8]

MAIN LOOP  (hunter, up to 120 rounds)
  for round r = 1 .. 120:
      hunter.messages += user(intel, probe-results)
      out := HUNTER(hunter.messages)         // intra-agent memory grows
      dispatch out.probes  ->  SANDBOX  ->  deltas
      memos += truncate(out.intelOut.forStrategist)
      memos += truncate(out.intelOut.forComposer)
      memos += truncate(out.intelOut.forAnalyst)
      memos := memos[-8:]                    // rolling short-term memory

      if (r mod 3 == 0 successful passes) OR (any hyp confirmed):
          strategist.thesis := STRATEGIST(intel + hunter feedback)
          composer.fused    := COMPOSER(analyst, strategist, hyps, graph)
          hyps := mergeById(hyps, composer.upgraded)
          nextUserMsg += preferProbes(composer.probes)   // 'prefer these'

      intel := formatCouncilIntel()          // replay updated memory
      if out.done: break

AFTER FIRST SETTLEMENT
  for d = 1 .. 8:                            // divergent agent
      hyps := mergeById(hyps, DIVERGENT(intel, VoI, reflexion))
      if d even: refresh strategist + composer   // same cadence

ON ANY CONFIRM
  hyps := CRITIC(hyps, probeDeltas, rescueSimHashes)  // refute FPs, once
```

The schematic makes the loop's memory discipline explicit: the Hunter's `messages[]` array only ever grows, preserving the complete trail, while `memos` is truncated to eight after every turn and `intel` is recomputed

from it. Re-planning, fusion, divergence, and criticism all read the same replayed briefing, so each specialized agent contributes to - and inherits from - a shared, bounded picture of the authorized target, with the Hunter alone retaining the full-resolution history behind it.

8 Persistent Cross-Audit Memory

Mephis accumulates auditing experience across engagements through a bounded, browser-local memory store. State is held under the `localStorage` key `mph_hunt_memory_v1`, capped at **500 lessons**. The store is deliberately modest in size: it is a distillation of prior audit outcomes, not a corpus of raw transaction traces, and it exists to sharpen the analyst's and strategist's priors on subsequent authorized targets.

8.1 Lesson distillation

When an audit completes, every hypothesis that reached a terminal state - **confirmed** (the detection signature was verified in sandboxed simulation) or **refuted** (the signature failed to hold) - is distilled into a compact **lesson** record. Each lesson captures the *detection technique*, a short *title*, the assessed *severity*, the *outcome*, a free-text *note* bounded at 240 characters, a bag of **protocol-shape tokens** (observed function names together with analyst-labelled externals, state touchpoints, and purpose descriptors), the target *address* and *chainId*, and a timestamp `ts`. Records are deduplicated by the composite key `technique|title|address`, so a repeated finding on the same authorized contract updates rather than multiplies its lesson.

8.2 Relevance scoring

At the start of an authorized audit, `retrieveRelevantLessons(8)` ranks stored lessons against the current target's protocol shape and returns at most the eight highest-scoring above a relevance threshold. Scoring combines a cosine-like similarity over token bags with additive priors for locality, recency, and prior confirmation:

Algorithm 3: Lesson relevance score for a candidate lesson ℓ against target τ

```
score(, ) =
      shape(, )                          // token-bag overlap, in [0,1]
    + 0.60 . [ address() = address() ]    // same authorized contract
    + 0.05 . [ chainId() = chainId() ]    // same chain
    + recency()                            // decaying weight on ts
    + 0.25 . [ outcome() = confirmed ]     // reward verified findings

where  shape(, ) =  ( Sigma_t  w_(t) . w_(t) )

                     w_ . w_               // cosine over shape-token bags

and    [ P ] = 1 if predicate P holds, else 0.

retain     score(, ) > 0.12
```

The `0.60` same-address term dominates: a prior lesson on the very same authorized contract is almost always surfaced, letting the system re-check whether a previously refuted concern has been reintroduced or a confirmed one properly remediated. The lower `0.05` same-chain term is a weak tie-breaker rather than a driver of relevance.

8.3 Briefing injection

`formatMemoryBriefing()` renders the retained lessons into a short natural-language briefing that is injected into both the analyst and strategist prompts. The briefing steers deliberation in two directions: it *biases toward* detection techniques that **confirmed** on structurally similar look-alike contracts, and it *cautions against* techniques that have accumulated repeated **refutations** (false positives) on similar shapes, reducing wasted probe budget. The memory encodes only surviving state - distilled outcomes that persist across sessions - never live audit intermediates.

> **Note. Privacy.** The lesson store is per-browser `localStorage`. It is not synchronized to a server, not shared between operators, and not transmitted off the analyst's machine. Cross-audit learning is therefore local to a single auditor's workstation, and target addresses recorded in lessons are those of authorized engagements only.

9 Intra-Audit Knowledge Graph

Where cross-audit memory (Section 8) persists distilled outcomes across engagements, the **knowledge graph** is the working memory of a single audit. It is reset at the start of each authorized engagement so that no state leaks between targets, and it is rebuilt incrementally as probing proceeds.

9.1 Structure

The graph carries four node and edge kinds:

- **Contracts** - the authorized target and any in-scope contracts it composes with.
- **Tokens** - the assets observed to move within the sandbox.
- **Composition edges** - inter-contract call and dependency relationships, populated by the analyst as it maps the protocol surface.
- **Hypothesis nodes** - candidate detection signatures, each annotated with the *observed token deltas* recorded when the hypothesis is exercised in simulation.

The graph is updated after every probe batch, so token deltas attached to hypotheses reflect actual sandbox observations rather than analyst conjecture.

9.2 Shared council intel

`formatHuntGraph()` renders the current graph as part of the **shared council intel** visible to the composer and hunter roles. The rendering exposes: the contracts in play, the composition edges between them, each hypothesis annotated with the **test-principal token sign** - whether, in simulation, the unprivileged **test principal** ends a probe with a token balance that increased (+) or decreased (−) - and an explicit *fuse instruction* directing the council to consider whether separate findings can be composed.

9.3 Sign edges and composition

The token-sign annotation is what makes the graph actionable for the composer. Each hypothesis is a directed effect on the test principal's modelled balance:

- A **value-pushing** finding carries a + sign: exercising it moves value *toward* the test principal (for example, a signature where an accounting update credits a caller that supplied nothing of matching value).
- A **value-pulling** finding carries a − sign: exercising it moves value *out of* a contract, but on its own leaves the test principal net-neutral or short (for example, a signature that drains a pool into a settlement path the test principal does not initially control).

Because the graph makes both signs and the composition edges explicit, the composer can search for a + hypothesis whose *output token* matches the *input token* of a − hypothesis on a composition-linked contract, and chain them: the value-pushing finding establishes a credited position, and the value-pulling finding realizes it across the composed contract. The fuse instruction asks the council to test precisely such chains in the sandbox and report the *net* test-principal delta of the combined sequence, which is frequently larger in magnitude - and higher in assessed severity - than either finding scored in isolation.

Algorithm 4: Sign-edge fuse search over the intra-audit graph (sandbox only)

```
for each hyp h with sign(h) = +           // value-pushing
  for each hyp h with sign(h) =          // value-pulling
    if token_out(h) = token_in(h)
       and composition_linked(contract(h), contract(h)):
         propose chain h -> h
         simulate chain against the test principal in sandbox
         record net (test-principal, token)
         if |net | > max(| h|, | h|):
             raise combined finding; escalate severity
```

Note. This composition analysis is a *severity-assessment* device: it establishes, inside the sandbox, whether two individually minor findings amount to a materially at-risk position that an auditor must report. It identifies which authorized-target funds are at risk so they can be routed to the pre-registered recovery multisig under Section 3's rescue-and-return lane; it is never a value-extraction procedure and produces no attacker-directed transfer.

10 Self-Consistency Ensemble (round 1)

Hypothesis generation is inherently sample-dependent: a single decoding of the hunter prompt may, by chance, omit a valid detection signature that the target's surface actually warrants. Mephis treats the opening round of hunting as a **recall** problem - the cost of missing a real vulnerability class is far higher than the cost of carrying an extra hypothesis that later refutes cheaply in the sandbox - and applies a self-consistency ensemble to widen coverage before probing begins.

10.1 Procedure

After the first hunter response is parsed as JSON, Mephis draws **two additional samples at temperature 0.6** against the *same opening prompt*. These are independent re-draws, not continuations: each sample sees the identical context and produces its own hypothesis set, so their disagreements reflect genuine sampling variance rather than accumulated conversational drift.

Algorithm 5: Round-1 self-consistency merge

```
H <- hypotheses(primary_sample)          // first hunter JSON
added <- 0
for k in 1..2:                           // two extra samples, T = 0.6
    s <- sample(opening_prompt)          // fresh draw, not a continuation
    if parse(s) fails: continue          // failed samples skipped
    for hyp in hypotheses(s):
        if title(hyp) is distinct from every title in H:
            if id(hyp) missing: id(hyp) <- "e" + next_index   // e1, e2, ...
            H <- H  { hyp }
            added <- added + 1
        probes(H) <- probes(H)  probes(hyp)   // concatenate probe lists
memo <- "self-consistency: added " + added + " hypotheses"
```

Merging is by *distinct title*: a sampled hypothesis whose title is not already present is admitted as a new hypothesis, receiving a synthetic identifier of the form `eN` if it lacks one. Probe suggestions from all samples are concatenated onto the working set, so even where a sample restates a known hypothesis it may still contribute a fresh probe. Samples that fail to parse are skipped without aborting the round, and a short memo records how many hypotheses the ensemble added.

10.2 Rationale

The construction follows the general principle of *self-consistency sampling* - that drawing several independent stochastic completions and aggregating them recovers signal that any single greedy or low-temperature decode may drop. Mephis adapts the idea from answer-voting to **set-union**: rather than selecting a single majority answer, it takes the union of distinct hypotheses, because the objective in round 1 is to maximize *recall* of candidate detection signatures. The subsequent simulation and verification stages supply the precision, refuting spurious additions at bounded probe cost, so a small number of extra false candidates is an acceptable price for reducing missed vulnerability classes on an authorized target.

11 Funded Simulation (modeling flash / whale capital, safely)

A defensive audit must answer a sharper question than "can an unprivileged caller move funds today?" It must answer "can a *well-capitalized* adversary move funds?" Because flash loans and whale balances are freely available on public networks, an audit that only exercises the operator's own modest balance systematically under-reports risk. **Mephis** closes this gap entirely inside simulation: it grants the neutral **test principal** synthetic capital through RPC state overrides, models the well-capitalized adversary, and measures the outcome. This capital exists only as an in-memory override for a read-only simulated call. It is never signed, never broadcast, and never touches a live transaction.

11.1 State overrides and balance-slot detection

The `eth_simulateV1` method accepts a `blockStateCalls` array in which each entry may carry a `stateOverrides` map. Mephis uses this map to set the test principal's balances before the probe steps execute. Native ETH is trivial (the override sets the account `balance` field directly), but ERC20 balances live in a contract storage slot whose layout is not standardized: a Solidity `mapping(address => uint256)` and a Vyper `HashMap` hash the holder key differently, and the base slot index varies by contract. To write a synthetic balance, Mephis must first locate the slot that backs `balanceOf(holder)`.

Detection is empirical and cached per token. For candidate base slots `0..15`, Mephis computes both the Solidity and the Vyper storage key for the holder, writes a distinctive **sentinel** value into each candidate in parallel, and then reads `balanceOf(holder)` under each override. Whichever override causes `balanceOf` to return the sentinel identifies the true balance slot and mapping layout. If no candidate reproduces the sentinel, funding is skipped for that token and the simulation proceeds with the token unfunded. The discovered `{slot, layout}` pair is cached so the probe of subsequent turns pays the detection cost once.

Algorithm 6: Balance-slot detection (per token, cached)

```
input:  token, holder, balanceOf(.) reader
output: {slot, layout} that backs balanceOf(holder), or NONE

sentinel <- a distinctive large constant unlikely to occur naturally
candidates <- []
for base in 0 .. 15:
    key_sol <- keccak256( pad32(holder)  pad32(base) )      # Solidity mapping
    key_vy  <- keccak256( pad32(base)    pad32(holder) )    # Vyper HashMap
    candidates += { (token, key_sol, layout:'solidity', base),
                    (token, key_vy,  layout:'vyper',    base) }

# probe all candidates in parallel; each is an independent simulated read
for c in candidates in parallel:
    override <- { c.token : { stateDiff : { c.key : sentinel } } }
    observed <- simulate_read( balanceOf, holder, stateOverrides = override )
    if observed == sentinel:
        return { slot: c.base, layout: c.layout }     # first exact match wins

return NONE          # funding skipped for this token; sim proceeds unfunded
```

11.2 Building funding overrides

Once slots are known, `buildFundingOverrides` assembles the override map for a probe. The test principal's ETH balance is set to `1e24` wei (approximately `1e6` ETH), and each tracked token balance is set to `1e30` base units, unless a probe specifies a model-supplied amount. A probe requests funding declaratively: `fund:true` applies the default flash-sized balances, while `fund:{eth, tokens:[{token, amount}]}` requests specific amounts. These magnitudes are chosen to exceed any realistic pool depth so that capital availability is never the limiting factor when Mephis is testing whether a value-moving path exists.

11.3 Auto-funding retry

Many probes are dispatched unfunded first, because the least-privileged case is the most important to characterize. When an unfunded write reverts with an error whose shape indicates a balance or allowance shortfall, Mephis re-runs the probe exactly once with flash-sized balances. The classifier matches revert reason strings containing tokens such as *insufficient*, *exceeds*, *ERC20*, or *underflow*, as well as common error selectors including `0xe450d38c` and `0xfb8f41b2`. The re-run result is tagged `funded:'auto'` (from the retry heuristic) or `funded:'requested'` (when the probe explicitly asked to be funded), so downstream analysis records exactly which capital assumption produced the observation.

A path that yields no gain unfunded but yields a positive delta once funded is *not* discounted as an artifact of the override. It is treated as a genuine finding, because it faithfully models a flash-borrowing adversary who can summon that capital atomically on a live network. The funding assumption is realistic, not a cheat; recording it merely documents the adversary class the finding assumes.

> **Note.** Scope boundary. Funding is a property of the *simulation lane only*. The rescue-and-return lane of Section 18 never applies these overrides: rescue transactions execute against real balances and real allowances, because their sole purpose is to move at-risk funds that actually exist into the pre-registered recovery multisig. If the configured RPC does not honor `stateOverrides`, or a token's balance slot cannot be located, the simulation simply proceeds unfunded and annotates the reduced-capital assumption rather than fabricating a result.

12 Impact Oracle (Atomic Impact Measurement)

Detection reasoning is driven in part by a language model, and a language model will, unprompted, assert that a path "drains" or "steals" funds. Such assertions are hypotheses, not evidence. Mephis disciplines every value-movement claim with the **impact oracle**: an atomic pre/post measurement of the test principal's token balances taken inside a single simulation, so that the measured delta is the arbiter of whether value actually moved.

12.1 Atomic wrapping

For any multi-step or state-changing probe, Mephis wraps the probe's own steps between two measurement passes and submits the whole sequence as one `eth_simulateV1` request. First it reads

`balanceOf(principal)` for up to eight **tracked tokens**; then it runs the probe steps; then it reads `balanceOf(principal)` again. Because all of this executes in one simulated block, state changes from the probe steps carry forward into the post-measurement, and the difference is computed atomically:

Δt = balanceOfpost(principal, t) − balanceOfpre(principal, t), for each tracked token t.

The tracked set is derived automatically: any contract loaded into the engine whose ABI exposes a `balanceOf` function is a candidate, capped at eight tokens per measurement to bound the call count. The oracle defines `deltaPositive` as *any* tracked token showing $\Delta t > 0$ accruing to the principal. Native ETH is deliberately *not* measured inside simulation and this omission is recorded in the evidence; live snapshots (Section 18) do measure ETH together with tokens, so the live lane carries no such blind spot.

12.2 Why atomicity is the ground truth

Measuring balances across two *separate* calls would be worthless: state does not persist between independent simulated calls, so the post-read would observe the pre-probe world and every delta would be zero. Wrapping the sequence in one simulation is what makes the measurement meaningful. It reproduces exactly the property a real adversary depends on, namely that the whole sequence executes atomically within one transaction or bundle, and it yields a signed, quantitative answer to "did the principal end richer?" that owes nothing to model narration.

12.3 The demotion rule

The oracle's decisive role is negative. A fund-moving hypothesis that the model has marked `confirmed` is subject to `postFilterConfirms`: if the probe matching that hypothesis was *sim-observable* - that is, its value effect should have appeared as a positive ERC20 delta - and the oracle measured *no* positive delta, the hypothesis is demoted from `confirmed` to `investigating`, and the evidence is annotated with the null measurement. In effect, a claim of value extraction that the ground-truth measurement fails to reproduce is not allowed to stand as confirmed. This single rule is the primary guard against hallucinated confirmations: the model may propose, but only an atomic, measured, positive delta lets a value-movement finding reach confirmed status.

13 Value-of-Information Probe Scheduling

Simulation is the scarce resource. Mephis caps execution at `PROBE_CAP = 10` probes per turn, and the set of candidate probes generated across active hypotheses routinely exceeds that cap. Scheduling is therefore an explicit value-of-information problem: spend the fixed simulation budget where the expected information gain is highest, rather than servicing candidates round-robin. The scheduler `rankProbes` assigns each candidate a scalar score, runs the ten highest, and persists the keys it has already spent budget on so that repeats are deprioritized on later turns.

13.1 Scoring

Each probe's score is the product of six multiplicative factors - severity, plausibility, novelty, chain depth, a composition bonus, and a funding bonus:

score = severity × plausibility × novelty × depth × compose × fund

where depth = 1 + 0.12 × (steps − 1), capped at a maximum contribution of +4 (i.e. the depth factor is clamped to ≤ 5). A multiplicative form is deliberate: a probe must be simultaneously severe, plausible, and informative to rank highly, and any single near-zero factor - for example a refuted hypothesis - suppresses the whole score.

Table 4: Scoring factors used by `rankProbes` (identical in the divergent pass).

Factor	Condition	Value	Intent
severity	critical	4	Weight by the stake of the hypothesis under test.
	high	3	
	medium	2	
	low	1	
	default (unrated)	1.5	
plausibility	investigating	1.4	Favor open questions; a confirmed or refuted claim yields little new information.
	confirmed	0.5	
	refuted	0.1	
novelty	unseen key	1.25	Reward exploration; penalize re-probing an already-spent key.
	already probed	0.35	
chain depth	per extra step beyond the first	×(1 + 0.12·(steps−1)), cap +4	Longer composed chains reach deeper states worth more per successful probe.
compose / fuse bonus	probe fuses multiple hypotheses	1.2	Reward probes that test an interaction rather than a single call.
fund bonus	`fund:true`	1.1	Slightly favor the well-capitalized-adversary case when otherwise tied.

13.2 Budget spending and persistence

The ten highest-scoring probes run; the rest wait. Seen keys persist across turns, so the novelty factor drops a re-encountered probe from 1.25 to 0.35 and it will only re-run if its other factors have strengthened - for example, a hypothesis newly escalated to critical severity. The same ranking governs the divergent pass, so both the convergent and exploratory phases spend the budget by expected information gain rather than by arrival order.

14 Reflexion

A round in which every probe reverted and nothing was confirmed is not a dead end; the revert signatures themselves carry information about why the hypotheses failed. Mephis runs a self-critique step, `reflectOnRound`, that reads those signatures and turns them into concrete guidance for the next round rather than repeating the same attempts.

14.1 Clustering revert signatures

`reflectOnRound` triggers when, after a round of probes, at least one probe reverted and no hypothesis reached confirmed status. It normalizes each revert message by stripping embedded hex (addresses, amounts, and selectors that would otherwise fragment identical failures into distinct strings), clusters the normalized messages, and takes the three largest clusters as the dominant failure modes of the round.

14.2 The memo

From those clusters Mephis composes a short memo proposing next-round adjustments: change magnitudes or step ordering, try a different entrypoint, or set `fund:true` to test the well-capitalized case. The memo is state-aware in one important respect: it records whether a *funded* retry has already run for the failing path. If flash-sized balances were already supplied and the path still reverts with a shortfall- or authorization-shaped error, the memo notes that the obstacle is *likely a genuine guard* - a real access-control check or invariant - rather than a mere capital limitation. Recognizing genuine guards is as valuable as finding a bypass: it steers budget away from a defended path and toward untested surface, and it is exactly the conclusion a careful auditor should record.

14.3 Propagation

The memo is written into the **shared council intel** so that the observation is visible to every analysis agent, and it is pasted into the hunter agent's next user message so the guidance is present in-context when the following round's hypotheses are formed. The loop is thus closed: revert signatures from one round become adjusted hypotheses - or a documented "defended" conclusion - in the next.

15 Simulation Engine

The simulation engine is the deterministic substrate beneath every capability described above. It dispatches each probe to the cheapest RPC method that can answer it, carries state across the steps of a composed probe, resolves call targets against the loaded ABIs, and fails over across a fixed set of public endpoints.

15.1 Method selection

A single `view` call that requires no state overrides is dispatched as a plain `eth_call`, the least expensive path. Anything else - a state-changing step, a multi-step sequence, or any probe carrying overrides - is dispatched as `eth_simulateV1` with the request shape:

Algorithm 7: `eth_simulateV1` request envelope

```
eth_simulateV1 {
    blockStateCalls: [
        { calls: [ step, step, ... , step ],
          stateOverrides?: { ... }        # present only when funding is requested
        }
    ],
```

```
    validation:      false,            # skip full block validation
    traceTransfers:  false             # deltas come from the impact oracle, not traces
}
at block tag 'latest'
```

Both `validation` and `traceTransfers` are disabled: validation is unnecessary for a hypothetical bundle, and transfer tracing is redundant because value movement is measured by the atomic `balanceOf` deltas of the impact oracle (Section 12). All simulation is pinned to the `latest` block tag.

15.2 Stateful sequences

Within one `blockStateCalls` entry the steps execute in order and state carries between them, which is what makes composed probes and the pre/probe/post measurement wrapping possible. The chain is strict: the *first* revert stops the sequence, and every later step is marked `skipped` rather than executed. This gives the reflexion step (Section 14) a clean signal - the specific step and reason at which a chain failed - instead of a cascade of dependent failures.

15.3 Target and placeholder resolution

Each step names a function; the engine resolves that name against the primary ABI first, then any satellite ABIs of loaded contracts, then the fallback ERC20 ABI. The contract a step actually calls is selected by its `steps[].to` field, so a single sequence can interleave calls across several contracts. Before dispatch, any occurrence of the **test-principal placeholder** in a probe field is rewritten to the concrete principal address, so that funding overrides, call arguments, and balance measurements all refer to one consistent unprivileged caller.

15.4 Transport and failover

The engine rotates across three public RPC endpoints on failure: `ethereum.publicnode.com`, `eth.drpc.org`, and `1rpc.io/eth`. An optional Tor HTTP gateway may be configured for request egress; a SOCKS proxy address may be stored but is not usable for dialing from a browser context, so only the HTTP gateway path is exercised there. Failover is transparent to the higher layers: a probe that cannot be served by one endpoint is retried against the next before any error is surfaced to the scheduler.

16 Vulnerability-Class Detection Taxonomy

This section presents Mephis's vulnerability-class catalog strictly as an **auditor's detection taxonomy**. It is *not* an exploitation playbook: for each class the catalog records only (i) the **observable tells** an auditor can read from source, bytecode, or state; (ii) **what the auditor verifies** to distinguish an incidental pattern from a genuine defect; (iii) the **confirm/refute condition** evaluated inside the sandbox against a modeled unprivileged **test principal**; and (iv) a defensive **fingerpost** to the historical incident class at the level of the SWC registry and public post-mortem literature. No entry contains a step-by-step weaponized recipe, novel exploit code, or a drop-in payload.

These families are supplied to hunter agents as **optional appendix packs** injected as context. Agents are explicitly told they are *not bound* to the taxonomy: it seeds hypotheses and standardizes reporting vocabulary, but it does not constrain what an agent may investigate, and it is *not exhaustive*. A finding that matches no listed class is reported on its own terms; a listed class that does not apply is dismissed with a recorded reason. Every confirm/refute condition is a sandbox invariant check, never an instruction to move value on-chain - the only live lane in Mephis is the rescue-and-return lane defined elsewhere in this paper.

16.1 Composition

Table 5: Value paths that span two or more contracts.

Class	Tells	Auditor verifies	Confirm/refute condition
`cross_protocol`	A value path where funds or accounting cross a trust boundary between two or more contracts (external call, adapter, wrapper, or integrated protocol) without a single owner of the invariant.	That each contract's local invariant is preserved under the *composed* call, and that assumptions one contract makes about another (return semantics, decimals, pausing, reentrancy guarantees) actually hold at the boundary.	Confirm if, in simulation, a sequence legal for each contract individually drives the composed system into a state where a conservation invariant (Σ credited $\leq \Sigma$ backing) is violated; refute if the boundary assumptions are enforced or the composed invariant holds.

16.2 Token-Send Semantics

16.3 Price / State Manipulation

16.4 Withdrawal / Exit Extraction

16.5 Vault NAV

16.6 Oracle

16.7 Cryptographic Verification

16.8 Accounting Invariants

16.9 Reentrancy

16.10 Authorization Bypass Affecting Funds

Note. Each confirmed class is reported as an audit finding with its confirm condition, the sandbox trace that established it, and its SWC/incident-class fingerpost, then routed to the responsible-disclosure

Table 6: Discrepancies between assumed and actual ERC-20 transfer behavior.

Class	Tells	Auditor verifies	Confirm/refute condition
`fot_rebasing`	Accounting credits a *requested* amount rather than the *delta actually received*; no balance-before/balance-after measurement around `transferFrom`; support advertised for arbitrary tokens.	Whether a fee-on-transfer or rebasing token would credit gross while the contract receives net, and whether internal share/debt math reads cached rather than live balances.	Confirm if a modeled FoT/rebasing token yields credited > received (or share value drifting from backing) in the sandbox; refute if credit equals measured delta.
`ignored_erc20_ret`	Bare `transfer`/`transferFrom` calls whose boolean return is unchecked; no `SafeERC20`; state updated before or without confirming success.	That a token returning `false` (or returning nothing) cannot leave internal state advanced as though the transfer succeeded.	Confirm if a token modeled to return `false`/void leaves balances or credits updated without the value having moved; refute if the return is enforced or the call reverts.
`approval_race`	Reliance on `approve` without allowance-reset discipline; permit-based flows without nonce/deadline scrutiny; UX that changes a nonzero allowance directly.	Whether an allowance change or a permit can be ordered adversarially relative to a spend, and whether nonces and deadlines bound the window.	Confirm if two orderings of an allowance/permit and a spend yield a net authorized amount exceeding the intended one in simulation; refute if nonce/deadline/reset invariants preclude it.
`raw_transfer_accounting`	Net asset value or share price derived from a live balance that any address can raise by a raw `transfer` into the contract; a `transferInto`-style deposit path distinct from the accounted one.	That share price and accounting reference *tracked* deposits, not ambient token balance, so an unsolicited transfer cannot move NAV.	Confirm if a direct transfer by the test principal shifts NAV or share price without a corresponding accounted deposit; refute if accounting ignores untracked balance.
`permit_token_crypto`	Custom `permit` or in-token `ecrecover` use; missing domain separator, chainId, or malleability guards; signature checks that admit `address(0)` recovery.	Correctness of the EIP-712 domain, nonce handling, deadline, and rejection of degenerate signatures on the token authorization path.	Confirm if a malformed or replayed signature validates or an `ecrecover` to `address(0)` is accepted as authorization in the sandbox; refute if all degenerate cases revert.

Table 7: Same-transaction manipulation of a value read the contract trusts.

Class	**Tells**	**Auditor verifies**	**Confirm/refute condition**
`flash_inflate_redeem`	An atomic path where a value the contract prices against can be minted or borrowed, moved, and unwound within one transaction; pricing read from a mutable same-block source.	Whether the priced quantity is protected against atomic inflation (time-weighting, snapshotting, or independent oracle) across mint → move → redeem.	Confirm if an atomic round-trip by the test principal ends with system backing reduced (value extracted from the contract's own reserves) in simulation; refute if pricing resists the atomic move.
`thin_pool_push`	Pricing or collateral valuation sourced from an AMM whose reserves are small relative to the position; spot reserve-ratio reads.	Depth of the referenced pool relative to positions it prices, and whether a single trade can move the reference materially within tolerance.	Confirm if a bounded trade shifts the referenced price enough to cross a solvency or liquidation threshold against the modeled position; refute if depth or bounds prevent it.
`share_price_sandwich`	Vault entry and exit priced from within-transaction state that a preceding action in the same transaction can move; no per-block or snapshot guard.	That deposit and withdraw prices cannot both be influenced by an action sandwiched in the same transaction.	Confirm if a same-transaction enter/exit around a state-moving action yields net gain drawn from other depositors' backing in the sandbox; refute if pricing is snapshotted.

Table 8: Exits that remove more backing than the exiter is entitled to.

Class	Tells	Auditor verifies	Confirm/refute condition
`insolvent_redeem`	Redemption honored at face while aggregate backing may be below aggregate claims; no solvency gate or pro-rata haircut; first-come-first-served exit queue.	Whether the contract can pay early exiters in full out of backing owed to others when total backing < total claims.	Confirm if, under modeled underbacking, an early exit by the test principal is honored at full value leaving remaining holders short; refute if exits are gated or pro-rated to solvency.
`rounding_extract`	Division before multiplication; truncation that consistently favors the caller; loops that repeat a rounding step; dust that accrues to the actor.	Direction and accumulation of rounding across repeated operations, and whether per-operation dust can compound into material value.	Confirm if a repeated rounding-favorable loop nets the test principal value beyond entitlement above a de-minimis threshold in simulation; refute if rounding is neutral or rounds against the caller.
`reward_index_skim`	Reward or interest index updated on a schedule that a deposit/withdraw can straddle; claimable computed from a stale index; join-just-before / leave-just-after timing.	That reward accrual is proportional to time-at-risk and cannot be captured by momentary position around an index update.	Confirm if a position taken immediately before and released immediately after an index update claims rewards disproportionate to time held; refute if accrual is time-weighted.
`escape_hatch_race`	Withdrawal queue, cooldown, or epoch boundary that a haircut, pause, or repricing is applied at; a path to exit ahead of the adjustment.	Whether a holder can order an exit before a solvency haircut or repricing that was meant to bind all holders equally.	Confirm if the test principal exits at pre-haircut value by racing the queue/epoch boundary in the sandbox; refute if the adjustment binds in-flight exits.

Table 9: Share-price integrity for ERC-4626-style vaults.

Class	Tells	Auditor verifies	Confirm/refute condition
donation_nav	Share price computed from live asset balance; no internal accounting variable insulating NAV from unsolicited transfers.	That a direct token transfer into the vault cannot change price-per-share for existing or incoming depositors.	Confirm if a donation by the test principal moves price-per-share (advantaging one party at another's expense) in simulation; refute if NAV tracks accounted assets only.
first_depositor	Empty-vault share math with no minimum initial mint, no dead-shares burn, and no virtual-offset; rounding that lets the first depositor set an extreme share price.	Whether the first deposit can be structured so a later depositor's shares round to a value far below assets contributed.	Confirm if, from an empty vault, a modeled first deposit plus donation causes a subsequent depositor to lose material value to rounding; refute if minimum-mint or virtual-offset mitigations hold.

Table 10: Trust placed in externally settable price or settlement data.

Class	Tells	Auditor verifies	Confirm/refute condition
oracle_sanity	A price, settlement, or update entry callable without permission or without bounds/staleness/deviation checks; single-source pricing; missing sequencer or heartbeat guards.	Whether the price/settle path enforces caller authorization, freshness, deviation caps, and sane bounds before the value is trusted downstream.	Confirm if an unprivileged or unbounded update sets a value that downstream logic acts on to move backing in the sandbox; refute if sanity/authorization checks reject it.

Table 11: Signature and proof checks on value-bearing paths.

Class	Tells	Auditor verifies	Confirm/refute condition
`trivial_crypto`	A verification path that accepts an empty proof, a zero signature, or a default-initialized commitment; short-circuit branches where an absent proof is treated as valid.	That the verifier rejects degenerate inputs and that no code path treats "no proof supplied" as "proof passed".	Confirm if a zero/empty proof or signature validates on a value-bearing path in simulation; refute if all degenerate inputs are rejected.
`unbound_claim`	A membership/inclusion proof that authorizes a claim but is not cryptographically bound to the *amount* or *recipient*; payout parameters read outside the signed/committed data.	That the proven statement fixes every payout-relevant parameter, so a valid proof cannot be reused with a different amount or recipient.	Confirm if a valid proof can be paired with an amount/recipient it does not commit to and still pays out; refute if the proof binds the full payout tuple.

Table 12: Internal bookkeeping that must reconcile with real balances.

Class	Tells	Auditor verifies	Confirm/refute condition
`invariant_accounting`	Tracked total supply, debt, or reward totals maintained separately from real balances; interactions with FoT tokens; reward math with independent accumulators.	That a stated conservation invariant (total shares $\leftrightarrow$ total assets, Σ debt $\leftrightarrow$ Σ collateral, Σ rewards $\leftrightarrow$ funded rewards) holds after every state transition.	Confirm if any modeled operation sequence breaks the declared invariant (tracked total diverges from real backing) in the sandbox; refute if the invariant is preserved throughout.

Table 13: Control returning to the test principal before state settles.

Class	Tells	Auditor verifies	Confirm/refute condition
`reentrancy_cex`	External call (including ERC-777/ERC-721 hooks) before state update; no checks-effects-interactions ordering or guard; *view* functions read by integrators during a call frame in which internal state is mid-update (read-only reentrancy).	Whether re-entry - classic or read-only - can observe or act on inconsistent state, and whether pricing/getters called mid-frame return values other contracts trust.	Confirm if re-entry (or a read-only re-entrant price read) lets the test principal double-spend, withdraw twice, or mislead an integrator's pricing in simulation; refute if CEI ordering or a guard blocks it.

Table 14: Access-control gaps on value-bearing state.

Class	Tells	Auditor verifies	Confirm/refute condition
`init_frontrun_funds`	Proxy or upgradeable contract with an `initialize` that sets privileged roles and is not guaranteed atomic with deployment; no initializer guard; unclaimed initialization state.	Whether initialization can be claimed by a party other than the intended deployer, and whether an uninitialized proxy exposes privileged setters.	Confirm if the test principal can call `initialize` (or a privileged setter on an uninitialized proxy) to gain control over funds in the sandbox; refute if initialization is atomic or guarded.
`owner_slot_overwrite`	`delegatecall` into modules whose storage layout may collide with the caller's; unstructured storage without namespacing; an owner/admin slot reachable via a colliding write.	That delegated modules cannot write the caller's owner/admin or critical slots through storage-layout clash.	Confirm if a delegated call path lets the test principal overwrite an owner/admin or fund-controlling slot in simulation; refute if layouts are namespaced and non-colliding.
`zero_role_default`	Role or admin variable left at its default (`address(0)` or unset) after deployment; role checks that pass for the zero address; a mandate never assigned.	Whether any privileged role defaults to an unassigned or zero value that an unprivileged caller can satisfy or claim.	Confirm if an unset/zero-default role admits the test principal to a fund-controlling action in the sandbox; refute if all roles are assigned and zero is rejected.

workflow. Where at-risk funds are demonstrably recoverable and the target is on the signed **target-authorization allowlist**, the finding is additionally eligible for the rescue-and-return lane, which moves value only to the pre-registered recovery multisig. Confirmation in the sandbox never authorizes any on-chain action outside that lane.

17 Main Loop Control

The hunter phase runs as a bounded control loop that coordinates hypothesis generation, probing, coverage tracking, and value-sink review across a rotating council of agents. The loop is governed by three constants: a hard ceiling of `SAFETY_MAX = 120` hunter rounds, a divergence trigger of `DIVERGENT_ROUNDS = 8` consecutive settled rounds, and a council rotation cadence of `COUNCIL_REFRESH_EVERY = 3` rounds. The loop is designed so that a premature `done:true` signal cannot end analysis while material work remains outstanding.

17.1 Termination Predicate

The loop reaches its natural stopping point only when all three conditions below hold simultaneously:

1. **Hypotheses settled.** Every hypothesis is either *confirmed* or *refuted*; none remain in the *investigating* state.
2. **Coverage complete.** The coverage target is met *or* the council has explicitly signaled `done:true`.
3. **Value sinks resolved.** Every identified value sink is either *touched* (analyzed) or *dismissed* with a recorded reason - i.e., the council reports `value sinks reviewed`.

Three **forced-continue** guards prevent an early exit: if `done:true` is asserted while any hypothesis is still *investigating*, the loop continues; if `done:true` is asserted while any value sink remains open, the loop continues; and if a round produces no probes but coverage or sinks remain outstanding, the loop continues. Only when there are no probes *and* every hypothesis, the coverage target, and every value sink are settled does the loop break out of the hunter phase into the divergent pass.

Algorithm 8: Hunter main loop with three stop conditions

```
constants:
  SAFETY_MAX            = 120     # hard ceiling on hunter rounds
  DIVERGENT_ROUNDS      = 8       # consecutive settled rounds -> break to divergent
  COUNCIL_REFRESH_EVERY = 3       # rotate council membership every N rounds

state:
  round        = 0
  settledStreak = 0
  council      = initCouncil()

loop:
  while round < SAFETY_MAX:
```

```
    round = round + 1

    if round mod COUNCIL_REFRESH_EVERY == 0:
        council = refreshCouncil(council)      # rotate perspectives

    probes = council.proposeProbes(state)
    for p in probes:
        result = sandbox.run(p)                # simulation only; no live tx here
        updateHypotheses(result)               # confirmed | refuted | investigating
        updateCoverage(result)
        updateValueSinks(result)               # touched | dismissed | open

    done      = council.signalsDone()
    hypsOpen  = anyHypothesis(state, INVESTIGATING)
    sinksOpen = anyValueSink(state, OPEN)
    covLeft   = not coverageComplete(state)
    noProbes  = (len(probes) == 0)

    # - - three forced-continue guards ----
    if done and hypsOpen:      continue         # (A) done but hyps still investigating
    if done and sinksOpen:     continue         # (B) done but value sinks still open
    if noProbes and (covLeft or sinksOpen):
                               continue         # (C) no probes but work remains

    # - - stop condition: everything settled ----
    settled = (not hypsOpen)
              and (coverageComplete(state) or done)
              and (not sinksOpen)              # 'value sinks reviewed'
    if noProbes and settled:
        settledStreak = settledStreak + 1
    else:
        settledStreak = 0

    if settledStreak >= DIVERGENT_ROUNDS:
        break                                  # exit hunter phase

# - - post-loop pipeline ----
divergentPass(state)                           # adversarial second look
if anyHypothesis(state, CONFIRMED):
    criticReview(state)                        # critic runs only if a confirm exists
recordLessons(state)                           # persist what generalized
return generateVerdict(state)
```

17.2 Post-Loop Pipeline

On exit - whether by the settled-streak break or by reaching `SAFETY_MAX` - control passes through a fixed pipeline. The **divergent pass** takes a deliberately adversarial second look at the settled state to surface hypotheses the primary council may have anchored away from. If and only if at least one hypothesis is *confirmed*, the **critic** reviews those confirmations for soundness and false-positive risk; with no confirmations, the critic is skipped. `recordLessons` then persists the generalizable observations from the run, and `generateVerdict` produces the final audit verdict. Any confirmed finding that reaches the verdict carries only its sandbox evidence and disclosure record into the downstream workflow; nothing in this loop performs

a live transaction, and eligibility for the rescue-and-return lane is decided separately against the signed authorization allowlist.

18 The Rescue-and-Return Lane (Authorized, Gated)

The rescue-and-return lane is the only component of **Mephis** that issues state-changing transactions to Ethereum mainnet. It exists to move **at-risk funds** out of a vulnerable but *authorized* contract and into a pre-registered **recovery multisig**, under mandatory disclosure. It is deliberately *not* an exploitation lane: it never routes value to an operator-chosen or personal address, and its purpose is custody and return, not extraction of profit. Every design choice below narrows what a rescue transaction may do until only one behaviour remains - relocating endangered assets to a fixed **safe harbor** declared in the signed authorization manifest.

The lane is reached only through a chain of gates. A UI gate, `Allow live rescue txs`, must be explicitly enabled. Operator identity - a Settings key or a connected wallet - must hash-match the configured **operator lock**. The target must appear on the **target-authorization allowlist** as a signed manifest entry of the form {`chainId`, `address`, `mandate-ref`, `expiry`}. All three conditions are independent and all are necessary; failing any one refuses the lane before a single transaction is composed.

18.1 Preconditions (Operator + Target + Recovery-Multisig-in-Manifest)

Three preconditions are evaluated before any rescue is even collected as a candidate:

- **Operator identity.** The active signer - a Settings key or a connected wallet - is hashed and compared against the operator lock. A mismatch halts the lane. This proves *who* is acting.
- **Target authorization.** The contract under audit must be present on the signed allowlist for the current `chainId`, with an unexpired `mandate-ref`. This proves the operator is *permitted* to act on *this* contract, and is checked separately from and in addition to operator identity.
- **Recovery multisig in manifest.** A **recovery multisig** address must be configured *and* must appear in the signed authorization manifest as the designated safe harbor. It is never chosen at runtime and never operator-editable mid-audit; it is fixed at the time the manifest is signed. This proves *where* funds may go - and the answer is pre-committed.

`collectRescueRequests` gathers candidates in two ways. A model may explicitly mark `rescue.needed` on a hypothesis or root. Automatically, the system nominates non-refuted hypotheses that carry `plan.steps`, are not already rescue-hashed (deduplication against prior rescues in the same audit), and whose status is one of: probed clean, confirmed, investigating, or an explicit rescue request. Nomination is not authorization - every nominated chain still faces the full policy check below.

18.2 Policy Invariants (`rescuePolicyCheck`)

Before broadcast, each candidate rescue chain passes through `rescuePolicyCheck`. Every invariant must hold; any failure refuses the entire chain. The linchpin is the **recipient-is-recovery-multisig rule**: every address argument that designates a beneficiary - the recipient of a `transfer`, the spender of an `approve` or `permit`, the beneficiary of a `mint`, `withdraw`, `redeem`, `claim`, or `harvest` - must resolve to the recovery

multisig or to a protocol address. A personal address in any such position is categorically refused. This single invariant is what makes the lane a return mechanism rather than an extraction one.

Table 15: All-must-pass invariants enforced by `rescuePolicyCheck`

#	Invariant	Refute condition
1	Live rescue gate is enabled (`Allow live rescue txs`)	Gate off
2	Recovery multisig is set *and* present in the signed manifest	Unset, or not in manifest
3	Chain text is not DoS-like (no denial-of-service or griefing intent)	DoS-like text detected
4	Step count is within 1..8	0 steps, or > 8 steps
5	At most 5 rescue chains per audit	6th chain in the same audit
6	Every call target is in the protocol graph, or is a tracked ERC-20 invoked through an ERC-20 function	Off-graph, untracked target
7	**Every recipient / spender / beneficiary address argument is the recovery multisig or a protocol address**	*Any* personal or non-manifest address in a beneficiary position

Algorithm 9: rescue admission and broadcast (pseudocode, non-runnable)

```
function admitAndBroadcast(audit):
    require gate.allowLiveRescue == true
    require hash(operator.signer) == operatorLock
    require target in allowlist[chainId] and not expired(target.mandate)
    require recoveryMultisig in signedManifest

    candidates <- collectRescueRequests(audit)      # model-marked  auto-nominated
    rescueCount <- 0
    for chain in candidates:
        if rescueHash(chain) already broadcast: continue
        if not rescuePolicyCheck(chain):           # all invariants 1..7
            refuse(chain); continue
        rescueCount <- rescueCount + 1
        if rescueCount > 5: break                  # <= 5 chains per audit

        for step in chain.steps (<= 8):
            if step is view or pure: skip
            snapshot ETH + token balances (before)
            sign(step) with key or wallet           # real funds, real signature
            broadcast(step)
            snapshot ETH + token balances (after)
            attach txHash, valueDelta to hypothesis
        emit disclosureRecord(chain)               # tamper-evident, mandatory
```

`broadcastRescueSteps` skips `view`/`pure` steps, signs each write with the operator key or connected wallet, snapshots ETH and token balances immediately before and after each step, and attaches the resulting

transaction hashes and computed `valueDelta` to the originating hypothesis. It then emits a disclosure record. Crucially, rescue is *independent of simulation funding*: a confirmation obtained against sim-funded balances does not imply the path is affordable with real funds. A flash-funded or sim-funded confirm establishes the vulnerability class; it does not pre-authorize a live path. The live lane uses only real funds and re-checks affordability against them.

18.3 Disclosure Obligation

Every rescue emits a **tamper-evident disclosure record** capturing the target, the mandate reference, the steps taken, the transaction hashes, and the before/after balance snapshots and `valueDelta`. Emission of this record is not optional and is not separable from broadcast - a rescue that could not produce its disclosure record would not proceed. The record initiates a mandatory **responsible-disclosure workflow**: the affected protocol team is notified, and the rescued assets remain under **multi-party custody** in the recovery multisig pending coordinated return to their rightful controller. No single party, including the operator, can unilaterally move funds out of the safe harbor.

> **Note.** Rescue is a last resort, reserved for situations where funds are *actively at risk* and inaction is likely to result in loss. Wherever time allows, the action is coordinated with the protocol team in advance; the multi-party custody and mandatory disclosure exist precisely so that an emergency intervention remains accountable, reversible into the hands of the rightful owner, and never a pretext for unilateral extraction.

19 Outputs and Reporting

Mephis surfaces its reasoning as it runs and consolidates it into an archival report. Throughout, the vocabulary is that of *findings* and *rescues*, not attacks - the outputs are audit artifacts intended for the protocol team and for coordinated disclosure.

19.1 Live Audit Rail

The live rail renders the audit as it unfolds. Round nodes alternate between **REASON** and **PROBES** phases, each carrying the model's thoughts, the current hypothesis statuses, probe call traces, impact summaries, and **funded** tags indicating where a simulation drew on sim or flash funding. The rail lets a human auditor follow the council's deliberation step by step rather than receiving only a terminal verdict.

19.2 Findings List

Findings are collected into a structured list. Each entry records **severity**, **status**, the detection **technique**, supporting **evidence**, the measured `valueDelta`, and - where the rescue-and-return lane acted - the associated **rescue transaction hashes**. The list is the auditable core of the report: every claim ties back to a probe trace or an on-chain snapshot.

19.3 Final Audit Report

`generateVerdict` compiles the final report. Overall risk is derived from the highest-severity *confirmed* finding. The report presents a compact findings summary, renders the reasoning chains, and supports **PDF export**. It may optionally be saved to `/api/audits`, producing a server-side snapshot addressable at `/a/{id}.json` for later reference and coordinated disclosure.

19.4 Contract Brief

Running in parallel to the security audit, the **contract brief** is a plain-language description of what the contract does and who uses it. It deliberately contains *no* security findings; it exists to orient reviewers and to give disclosure recipients context independent of the vulnerability analysis.

20 Limitations

The current implementation carries real constraints, both technical and - importantly - in the placement of its authorization controls. They are stated plainly so that operators do not over-trust the system.

- **Chain scope.** Ethereum mainnet only (`chainId` 1).
- **Runtime.** Client-side; requires an Ollama instance reachable from the browser (`OLLAMA_ORIGINS` must be configured).
- **Source requirement.** Verified source and ABI are required; unverified contracts cannot be audited.
- **Graph bounds.** Up to 14 related contracts in the protocol graph; up to 8 ERC-20s in the impact oracle.
- **Probe and step bounds.** Up to 10 probes per turn; up to 12 steps intended in the hunter prompt; up to 8 rescue steps; up to 5 rescue chains per audit; up to 120 hunter rounds; up to 8 divergent rounds.
- **Profit measurement.** Simulation does not measure native ETH profit; flash funding is simulation-only.
- **Funding semantics.** A sim-funded or flash-funded confirmation does not establish that a path is affordable with real funds.
- **Authorization enforcement.** The operator lock and target allowlist are *client-side* checks, not a server-side ACL.
- **Memory.** Lessons and state persist per-browser in `localStorage` only.
- **Ensemble.** Multi-sample ensembling runs only on round 1.
- **Taxonomy.** The detection taxonomy is optional context for the models, not an exhaustive catalogue.
- **Peer semantics.** The system cannot infer bytecode semantics for unverified peer contracts beyond the ERC-20 surface.

The most consequential of these is the placement of the authorization controls. Because the operator lock and target allowlist are enforced in the client, they protect an honest operator from mistakes but do not withstand a modified client. This is a *known weakness*: a production deployment must move both checks server-side, enforcing operator identity and allowlist membership behind an API that the browser cannot bypass, so that authorization is a property of the service rather than of the page.

21 Responsible Disclosure and Ethics

Mephis is designed for one setting only: auditing contracts the operator *owns* or is *authorized to test* under a written mandate - an engagement letter or an in-scope bug-bounty program. This restriction is not merely a policy statement; it is enforced structurally by two mechanisms acting together. The **operator lock** binds actions to a known identity, and the **target-authorization allowlist** - a signed manifest of {`chainId`, `address`, `mandate-ref`, `expiry`} - restricts analysis to contracts for which a specific, unexpired mandate exists. A contract absent from the allowlist is refused before analysis begins.

Findings do not become public artifacts by default. They flow through **coordinated disclosure**: the affected protocol team is notified, given the contract brief and the evidence, and afforded the opportunity to remediate before wider publication. The reporting pipeline (Section 19) exists to support this workflow, not to broadcast vulnerabilities.

The **rescue-and-return lane** is a *safe-harbor mechanism of last resort*. It engages only where funds are actively at risk, and it routes those funds solely to a pre-registered, multi-party **recovery multisig** declared in the signed manifest - never to an address chosen at runtime and never to a personal account. Every rescue emits a tamper-evident disclosure record and places the recovered assets under multi-party custody pending return to their rightful controller. Denial-of-service actions and the abuse of legitimate owner privilege are explicitly **out of scope**; the policy checks refuse DoS-like chains, and the recipient-is-recovery-multisig invariant forecloses self-dealing.

Stated plainly: operating this system against contracts one is not authorized to test would be **unlawful and unethical**, and lies entirely outside its design intent. The allowlist, the operator lock, the recovery-multisig invariant, and the mandatory disclosure workflow are the technical expression of that boundary. They are not decoration; they are the reason the system can carry sophisticated audit and rescue capabilities without becoming an instrument of extraction.

22 End-to-End Audit Flow

The full loop, from authorization through verdict, proceeds as follows. Terminology throughout is that of findings and rescue, never attack or exploitation.

1. The operator agrees to the authorized-research gate; operator identity must hash-match the operator lock *and* the target must be present on the signed allowlist. Both conditions are necessary.
2. Fetch verified source and ABI; expand the protocol graph by following getters, extras, and source-referenced addresses.
3. Reset the council, the graph, and the probe-seen set; brief the models on prior lessons; start the contract brief in parallel.
4. The analyst maps the protocol; the strategist designs maximum-impact chains as audit hypotheses.
5. Run hunter round 1 with two ensemble samples; merge the unique hypotheses.
6. Apply source focus, auto-advance, and ingestion of peer contracts as the graph warrants.
7. Rank probes by value of information; simulate (optionally with sim or flash funding); compute impact-oracle deltas; demote unproven confirmations; update the graph; record a reflexion memo.

8. Optionally broadcast a rescue under the full policy of Section 18.
9. Every 3 probes, or on any confirmation, re-feed the strategist and composer; merge the fused hypotheses.
10. Repeat until the audit settles, or until 120 hunter rounds. Then run the divergent phase (up to 8 rounds). Then run the critic.
11. Record lessons for future audits and compile the verdict.

This is the system as implemented: a defensive audit loop grounded in simulation, with authorized, disclosure-bound rescue as its only on-chain action.