BINARY DEFENSE

Threat-Informed Detection Engineering (TIDE)

A Field Guide to Engineering Threat-Informed Detection Pipelines

Build detections like software. Validate like attackers. Deploy like DevOps.

Author:
Binary Defense ARC Labs &
Detection Engineering Team

Contact:
arclabs@binarydefense.com

Table of Contents

What This Is All About

Modern cybersecurity operations sit at the intersection of intelligence, engineering, and automation.

Detection teams face a paradox: while the industry often equates progress with producing more detections, true security maturity comes from building better ones. Many organizations still rely on static content and manual processes designed for breadth, not precision. This more often the not leads to noise instead of insight

Threat-Informed Detection Engineering (TIDE) emerged at Binary Defense as a response to that imbalance.

It reimagines detection creation not as an act of rule-writing, but as a disciplined engineering practice that is grounded in threat intelligence, threat modeling and software development principles.

This guide introduces the foundational philosophy and implementation mechanics of TIDE, emphasizing its integration with a Detection-as-Code (DaC) model.

Part I
Understanding Threat-Informed Detection Engineering

From Art to Engineering

For years, detection work was seen as a syntax game, focused on who could write the query or master a SIEM's search language. But knowing how to write detections is not the same as knowing why to write them. TIDE redefines the field around purpose and precision, treating each detection as an engineered threat model built from intelligence, validated through evidence, and tuned for impact.

While this skill remains valuable, modern adversaries operate with automation, obfuscation, and adaptive behaviors that demand something more precise.

TIDE reframes detection engineering through an evidence-based lens. It asks engineers to think like researchers: to begin each detection with a hypothesis, gather supporting evidence from available telemetry, design an experiment to validate it, and only then deploy the logic into production.

In this model, detections become scientific artifacts, each one traceable to a threat model, test results, and validation evidence.

Intelligence as a Design Input

Threat intelligence serves as the cornerstone of TIDE. Rather than passively consuming reports, detection engineers translate adversary tradecraft into structured detection requirements.

This translation process answers 5 essential questions:

1. What is the goal and objective of the attack?
2. What is the impact of the attack?
3. What tactics, techniques, and procedures (TTPs) are they likely to employ?
4. Which TTPs are associated with high impact events?
5. How can those behaviors be observed in our existing telemetry?

This intelligence-led approach ensures that engineering resources are spent where they matter most, on adversary behaviors with proven relevance, rather than on theoretical or outdated attack vectors.

The Role of Threat Modeling in Detection Engineering

In threat-informed detection engineering, a model is never a guess. The process begins by constructing a detailed attack model that represents how a real adversary would attempt to achieve their objectives within a specific environment. This model defines not only the tactics and techniques an attacker might use, but also the order of operations, toolsets, and decision points along the way.

Once the model is established, our engineers emulate the attack in controlled conditions, executing each phase of the intrusion as the adversary would. This controlled experimentation serves a critical purpose: it generates raw evidence (the impact of events and telemetry artifacts) that reveals how a real-world attack manifests in observable data. These observations allow us to identify which events are truly meaningful and which are merely background noise.

From those observations, we craft formal detection hypotheses. Each hypothesis follows a structured logic pattern that connects attacker behavior to its observable evidence. For example: If an adversary executes technique T1543.003 (Create or Modify System Process) to escalate privileges, then we should observe specific patterns of common living off the land binaries within system event logs, accompanied by correlated process lineage behaviors.

This approach mirrors the scientific method by being measurable, repeatable, and grounded in empirical observation. The model is then converted into code, validated against recorded telemetry from our emulation, and refined until the detection produces consistent, explainable results.

In practice, threat modeling within TIDE is not a theoretical exercise; it's a feedback-driven engineering discipline.

We build the model, execute the attack, study the telemetry, and codify what truly matters.

Each detection is grounded in observed behavior, validated through experimentation, and designed to produce signal over noise, ensuring that the detections we deploy are not superfluous.

Part II
Detection-as-Code:
Turning the TIDE on Adversaries

The Engineering Analogy

Software development achieved reliability and scalability by embracing practices like version control, continuous integration, and automated testing. Detection engineering is now undergoing the same transformation.

Detection-as-Code applies DevOps principles to security content, treating detections as first-class software objects.

This approach enables:

- Version Control: Each detection exists as a file in a Git repository, complete with history, author, and change justification.
- Peer Review: Pull requests allow engineers to review logic, ensure coverage, and minimize redundancy.
- Schema Enforcement: YAML or JSON schemas enforce consistency across detections.
- CI/CD Pipelines: Automated validation and deployment pipelines ensure that only tested detections reach production systems.

Why Code Matters

When detections are written as code, they become:

- Auditable: Every modification is recorded.
- Testable: Logic can be validated against known data.
- Reusable: Templates and functions can be modularized across environments.
- Deployable: Automation replaces manual configuration, ensuring consistency at scale.

This structure transforms what was once a reactive SOC process into a continuous engineering lifecycle.

Part III
Implementing TIDE:
The Detection-as-Code Workflow

Step 1 — Define Threat Priorities

The first phase in threat-informed detection engineering is contextualization.

Engineers identify the adversaries and techniques that pose the greatest risk to the organization based on intelligence analysis and operational telemetry.

Inputs include threat intelligence reports, internal incident data, and ATT&CK technique mappings.

The goal is not to cover the entire ATT&CK matrix, but to prioritize based on observed likelihood and impact.

Deliverables at this stage include a structured YAML file defining detection requirements and a coverage matrix mapping existing detections to ATT&CK techniques.

Checklist

- Map top adversaries and TTPs
- Identify required telemetry (Sysmon, EDR, cloud logs, etc.)
- Document detection requirements and known gaps

Step 2 — Formulate the Threat Model

Each detection in TIDE originates from a well-defined threat model, not from intuition or guesswork.

This model connects an adversary's intent to their observable behaviors and identifies the telemetry that will expose those actions within the environment.

Binary Defense engineers use the GOST framework (Goal, Objective, Strategy, and Tactics) to structure this process.

GOST provides a way to think through attacker intent before writing any detection.

Applying the GOST Model in Practice

Let's walk through a practical example that translates a threat model into a validated detection with an attack scenario. An adversary deploys Cobalt Strike payloads by installing a malicious Windows service to achieve persistence and enable command-and-control.

GOST Breakdown

GOST	
Goal	Maintain long-term access and control of the compromised system.
Objective	Establish persistence on the host through a new Windows service that auto-starts with elevated privileges.
Strategy	Use a signed or unsigned binary to install a Windows service that launches a Cobalt Strike beacon on system startup.
Tactics	T1543.003 — Create or Modify System Process: Windows Service T1059.001 — Command and Scripting Interpreter: PowerShell T1071.001 — Application Layer Protocol: Web (C2 beacon)

Impact Measurement

After defining the model, engineers identify impact events or observable artifacts that prove the attack occurred and indicate risk severity.

These include:

- Creation of a new Windows service (EventID 7045) linked to an unsigned or suspicious executable.
- Service binary paths that reference non-standard directories (e.g., %TEMP%, \Users\, or AppData).
- Command-line activity involving service creation utilities (sc create, New-Service, InstallUtil).
- Subsequent network connections from the service process to external or uncommon destinations.

By measuring these impact events across telemetry, detection engineers can quantify both fidelity (true positives vs noise) and coverage (how many environments this technique could affect).

Data and Telemetry Requirements

The next step is to identify which telemetry sources are needed to observe this behavior in a reliable, low-noise way.

Required Data Sources:

- Windows System Logs (EventID 7045) — Service creation events
- Sysmon Logs (EventID 1, 17) — Process creation and service registration
- EDR Telemetry — Command-line arguments, parent/child relationships, file metadata
- PowerShell Logging — ScriptBlock and transcription logs for service creation commands
- Network Logs — Outbound connection records, proxy logs, or EDR network events

Key Fields:

ServiceName, ImagePath, SignatureValidation, CommandLine, ParentProcess, FileHash, UserAccount, DestinationDomain, DestinationPort

This phase clarifies what data is essential for accuracy, ensuring that detections are written against known telemetry reality, not hypothetical fields.

From Model to Hypothesis

Once the data is defined, the model is converted into a formal detection hypothesis. A TIDE hypothesis is a testable statement that links adversary behavior to observable evidence.

Hypothesis:

- If an adversary installs a persistent backdoor by creating a Windows service using an unsigned or abnormally located executable,and that service process later establishes outbound connections, then this behavior indicates likely malicious persistence.

This becomes the blueprint for the detection logic written in the next step of the workflow.

Checklist

- Write hypothesis for each prioritized TTP
- Define data requirements for validation

Step 3 — Build Detection Logic

Using standardized schemas such as Sigma, engineers express detection logic in structured YAML:

```yaml
title: Suspicious Unsigned Service Creation
id: bd-t1543-001
description: Detects creation of a Windows service linked to unsigned or abnormally locat
status: experimental
logsource:
  product: windows
  service: system
detection:
  selection_service_creation:
    EventID: 7045
  suspicious_path:
    ImagePath|contains:
      - "\\users\\"
      - "\\temp\\"
      - "\\appdata\\"
  unsigned_binary:
    SignatureValidation: null
  condition: selection_service_creation and (suspicious_path or unsigned_binary)
fields:
  - EventID
  - ServiceName
  - ImagePath
  - SignatureValidation
tags:
  - attack.persistence
  - attack.t1543.003
  - attack.t1059.001
```

Each detection file includes metadata such as author, validation date, ATT&CK mapping, and data dependencies.The logic is peer-reviewed before merging into the main repository branch, ensuring quality control and shared understanding across teams.

Checklist

- Detection logic written using YAML schema
- Metadata includes ATT&CK ID, data sources, and validation notes
- Peer review completed and approved

Step 4 — Validate Through Testing

Validation confirms that a detection behaves as intended in both simulated and real-world conditions.

You can use adversary emulation frameworks like Atomic Red Team and Caldera to execute specific ATT&CK techniques.

Detection triggers are captured, fidelity measured, and false positives analyzed.

Unit and integration tests are automated using Python or Go test harnesses.

Example:

```
    def test_service_creation(fixture_logs, detection):
        matched = detection.match(fixture_logs["unsigned_service_creation"])
        assert matched is True
```

Validation results are stored in validation.yaml, forming a permanent record of test evidence.

Checklist

- ATT&CK technique emulated successfully
- Detection fired under expected conditions
- False positives documented and tuned
- Validation results archived

Step 5 — Deploy via CI/CD

Deployment occurs automatically through continuous integration pipelines.

When a new detection is merged into the main branch, the pipeline executes validation, testing, and deployment to the appropriate environment, whether that's a SIEM, EDR, or XDR platform.

Example pipeline (simplified):

```
stages:
  - validate
  - test
  - deploy
validate:
  script: python validate_schema.py
test:
  script: pytest tests/
deploy:
  script: ./deploy_to_siem.sh
  when: branch == 'main'
```

Deployment metadata (including version, timestamp, and owner) is recorded for full traceability.

If the detection fails validation, the pipeline automatically halts or rolls back the previous stable version.

Checklist

- CI pipeline integrated with repo
- Validation and test gates configured
- Deployment metadata logged
- Rollback tested successfully

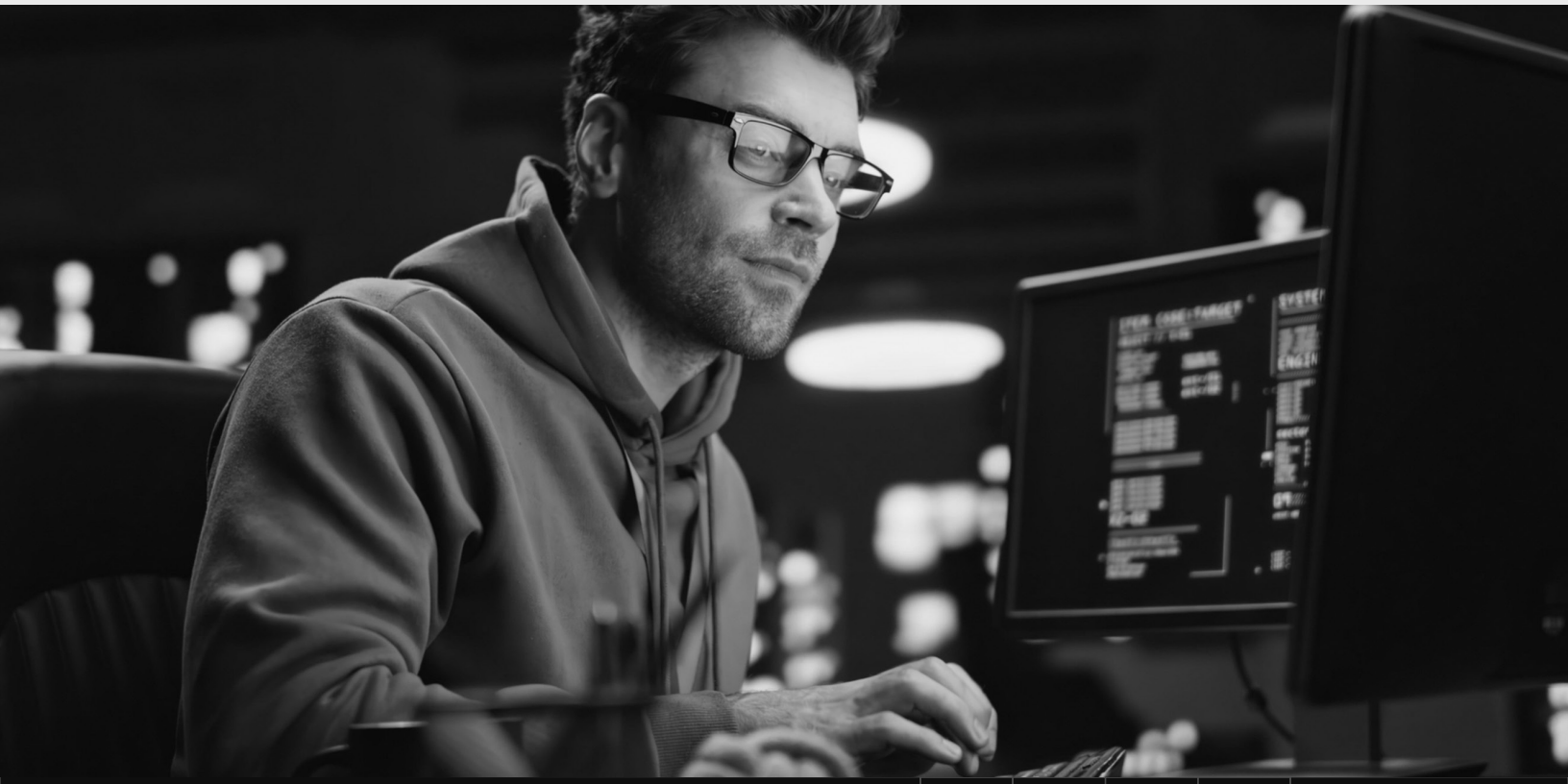
Step 6 — Measure and Refine

Continuous measurement ensures that detections remain effective over time.

Metrics such as fidelity, coverage, mean time to detect (MTTD), and drift rate are automatically collected.

Metric	Description	Target
Fidelity Score	Ratio of true positives to total alerts	≥ 90%
Coverage Score	% of relevant ATT&CK techniques covered	Growing quarterly
MTTD	Time from telemetry ingestion to alert	≤ 5 minutes
Drift Rate	Detection degradation due to telemetry changes	≤ 10% per quarter

This data informs tuning, gap analysis, and roadmap planning. Each iteration of the TIDE process increases precision while reducing analyst fatigue.



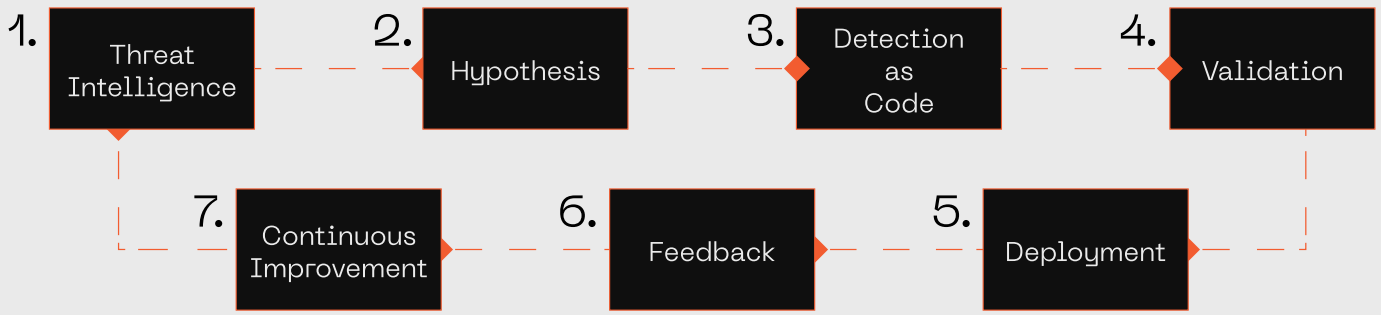
Part IV
Practical Implementation Example

```
/detections
  /persistence
    /t1543_003_service_creation
      detection.yaml
      validation.yaml
      tests/
        test_detection.py
      docs/
        readme.md
/coverage
  attack_matrix.csv
/tests
  fixtures/
  scripts/
```

This directory layout demonstrates how each detection exists as a self-contained unit — complete with code, documentation, and validation.

Such modularity allows organizations to scale detection engineering efforts without sacrificing traceability or quality.

Part V
The Continuous Improvement Loop



Over time, this cycle cultivates a high-fidelity detection ecosystem capable not only of identifying threats but of learning from them.

Part VI
Integration and Tooling Recommendations

Integration	Function
Threat Intel Platform to Git	Auto-populate detection backlog from new TTPs
CI/CD to SIEM/EDR	Continuous deployment and rollback automation
Case Management to Git	Feed alert outcomes into tuning process
ATT&CK Navigator	Map coverage and identify maturity trends

Recommended tool stack:
GitHub/GitLab for code management, Sigma for schema, Pytest for validation, Atomic Red Team for emulation, and Jenkins or GitHub Actions for CI/CD orchestration.

Conclusion

Threat-informed detection engineering represents a paradigm shift in how defenders conceptualize and construct their detection capabilities.

By aligning intelligence with engineering discipline, it moves the practice of detection from an art form to an empirical science that is governed by data, process, and continuous validation.

When coupled with Detection-as-Code, TIDE offers not just a methodology but a framework for sustainable, scalable defense.

It empowers teams to:

- Build detections aligned to real adversary behavior.
- Validate with evidence, not assumptions.
- Deploy through automation with minimal human friction.
- Measure performance with transparency and precision.

The result is a living system that learns, adapts, and strengthens with each iteration.

Detection becomes not a static rulebook, but a dynamic reflection of the threat landscape itself.

In short:
TIDE transforms threat intelligence into engineered resilience.

BINARY DEFENSE

At Binary Defense, our Detection Engineering team builds detections with precision, adaptability, and intent. By applying a threat-informed, Detection-as-Code approach, we continuously refine detections based on real adversary behaviors and validation results from live environments.

This engineering discipline powers our Managed Detection and Response (MDR) service, giving clients a defense program that evolves in step with the threat landscape. Every alert, every detection, every response is backed by intelligence, validated through automation, and guided by human expertise.

Binary Defense MDR isn't reactive security; it's engineered protection that learns, adapts, and outthinks attackers.

Visit: binarydefense.com
Explore our research: ARC Labs
Watch: ThreatTalk Webinar Series
Contact: info@binarydefense.com