XBOW

Guide

The Build-Versus-Buy Decision for Autonomous Offensive Security

A practical guide to evaluating control, cost, safety, validation, and long-term ownership

GUIDE

The Build-Versus-Buy Decision for Autonomous Offensive Security

A practical guide to evaluating control, cost, safety, validation, and long-term ownership

As frontier models improve, building an in-house AI penetration testing tool can feel increasingly practical and achievable, especially for enterprises that want tighter control over security and AI use. But there are many factors a team needs to consider before building their own. This guide is intended to help teams consider all aspects of building an AI penetration testing system.

The Build Instinct Is Understandable.

For enterprises, the instinct to build internally is understandable. Security teams already have agreements with AI providers and want control over how AI is used, where data goes, which models are called, and which vendors are allowed into sensitive environments.

Those are legitimate motivations. They reflect the same concerns that should shape any serious enterprise security program: control, accountability, compliance, and cost discipline. However, building internally can also put those same goals at risk if the organization underestimates the operational burden required to sustain them.

AI Models Are Not the Same as AI Penetration Testing.

Frontier models have become exceptionally good at finding vulnerabilities, so it is no surprise that companies are exploring their use in offensive security. But LLMs alone are not a pentesting platform.

Penetration tests have long been the gold standard in testing because they provide the highest-quality results. Unlike scanners, humans testers can apply learned business context, reason through complex attack paths, and validate findings, so results are very low-noise and tightly aligned to real risk. While LLMs excel in some areas they struggle in other important areas for pentesting.

More Vulnerabilities Create More Work.

LLMs can generate a large volume of potential findings, and it can be difficult to discern which are real, exploitable, and important. Without validation and prioritization, AI creates a new AppSec bottleneck: more issues to review, not less risk.

Models Alone Cannot Sustain Long-term Testing at Scale.

Anyone who has spent time working with an LLM knows that performance can degrade as interactions get longer and more complex. The model may lose track of prior context, repeat itself, pursue a weak path too long, or become less coherent as the task expands. Pointing an LLM at a web application may produce useful results. Pointing an LLM at an enterprise portfolio of applications and expecting consistently high-quality outcomes is an entirely different problem.

Safety Is Non-Negotiable

Offensive security has always required boundaries. AI makes that even more important because agents are built to pursue goals, and a goal like "find vulnerabilities" can lead to dangerous territory if the system is not governed. Ungoverned testing can cross scope, create unnecessary load, or take unsafe actions in production.

Source Informs. Runtime Proves.

LLMs review source code. Source review helps teams understand what could go wrong. Runtime testing asks what can actually be exploited in this application, in this environment, under these conditions. Real applications do not behave like clean architecture diagrams. They sit behind WAFs and CDNs or route through API gateways. They depend on permissions, sessions, rate limits, legacy services, fragile integrations, and workflows that only exist in production-like conditions. A weakness that looks serious in code may not be reachable. A weakness that looks minor on its own may become serious when chained through the running application.

Source code, documentation, architecture diagrams, threat models, SAST results, prior pentest reports, bug bounty writeups, incident history, and business priorities can all make autonomous testing sharper. They help guide reconnaissance, focus testing on the areas that matter most, and seed variant analysis. If one weakness appears in one place, the system can use that clue to look for related exploitable patterns elsewhere in the running application.

But context is not proof. Static findings and prior reports point to possible risk. Runtime validation shows whether that risk is reachable, meaningful, and fixable in practice.

The Operational Difficulty of Runtime Penetration Testing

Contextual Differences in Applications

Teams that have built solutions around build-time testing tools like SAST, may assume runtime solutions are similar, but there is a different set of challenges. While static analysis benefits from standardized workflows, runtime penetration testing does not. Every application has its own authentication patterns, authorization models, business logic, custom workflows, nonstandard APIs, legacy dependencies, fragile integrations, and environmental quirks.

Failure Handling and Debugging

The same complexity appears when assessments fail. Someone has to determine whether the failure came from the environment, the model, the orchestration layer, the authentication flow, or the application itself. Then the system has to be improved so the same failure does not repeat across future assessments.

Agent Orchestration

At scale, orchestration becomes one of the hardest problems in autonomous penetration testing. It is not enough to launch agents at a target and let them explore. The platform has to coordinate activity across the attack surface so agents do not duplicate work, collide with one another, exhaust the same paths, or lose track of what has already been tested.

In runtime testing, the edge case is often not the exception. It is the primary use case.

Safety

Safety is not optional in runtime penetration testing. Autonomous systems are goal-directed, and in offensive security, goal-directed behavior can become risky without strict controls. A model asked to find vulnerabilities may push too hard, test the wrong asset, generate unnecessary load, cross scope boundaries, or take actions that are unacceptable in a production environment.

`Offensive capability is only useful if it can be controlled`

An enterprise platform has to make safety part of the execution layer. It must understand scope, enforce policy, constrain techniques, monitor behavior, and know when to stop. It also needs an audit trail so teams can understand what happened, why it happened, and whether the activity stayed within authorized boundaries.

Validation

Finding a possible vulnerability is not the same as proving one exists. This distinction becomes even more important with LLM-driven testing because models are prone to hallucination. They can sound confident while misreading an application, overstating impact, inventing an exploit path, or treating suspicious behavior as confirmed risk. In offensive security, that is not a small problem. A finding that cannot be trusted creates work for the security team and damages confidence in the entire system.

That is why validation cannot be left to the model alone. An enterprise-grade platform needs purpose-built validators that act as a counterbalance to AI reasoning. Those validators have to reproduce the behavior, test exploitability, collect evidence, check scope, remove duplicates, and distinguish real vulnerabilities from model-generated noise.

Taking Advantage of Model Progress

Model progress makes testing better. As models get stronger, they create better attack paths, recover from mistakes more often, and find issues that weaker models miss. That is good news, but it also creates more work for the system around them.

Better models produce more plausible paths. More plausible paths mean more need for constraint, dedupe, validation, cost control, workflow routing, regression testing, and governance. Otherwise, the system just gets better at creating more things for humans to sort through.

There is also no single best model for every part of a pentest. One model may be better at reasoning through code. Another may be better at web interaction. Another may be better at summarizing evidence or planning multi-step attacks. The best platform should be able to use the strongest model for the job, not be trapped inside the strengths and weaknesses of one provider.

Attackers get the same benefits from model progress, but unlike enterprises, there is little to no cost to switch. They do not need full coverage, clean evidence, audit trails, or safe proof patterns. They need one exploitable opening.

Continuous Testing

Continuous testing is far harder than point-in-time testing. A one-time assessment can focus on what exists at a single moment. Continuous testing requires understanding what changed, why it matters, and which parts of the application need to be tested again.

That requires more than rerunning the same assessment on a schedule. An effective system has to identify new attack surfaces, determine which previous tests are still relevant, preserve historical context, avoid redundant execution, and launch focused reassessments that complete quickly. Otherwise, continuous testing becomes expensive repetition: the same work, the same findings, and the same noise delivered over and over again.

The ideal system behaves more like an always-current offensive security layer. It:

- Identifies changes to the attack surface
- Determines which attack paths are newly relevant
- Launches focused incremental assessments
- Completes quickly and efficiently
- Maintains context across time

Integrations and Workflows

Autonomous testing only creates value if its results reach the systems where remediation actually happens. Findings need to flow into ticketing systems, CI/CD pipelines, developer workflows, dashboards, and security reporting processes. If the output is only a PDF that someone has to manually translate into action items, the organization has not achieved scale. It has just made the reporting bottleneck arrive faster.

A platform has to package findings in a way that developers and security teams can use immediately: clear evidence, impact, reproduction steps, remediation guidance, ownership, severity, and status. It also has to fit into existing workflows rather than forcing teams to invent new ones. The point of autonomous testing is not just to find more issues. It is to shorten the path from discovery to remediation.

Enterprise Access Control and Governance

Enterprises also need governance around offensive security tooling. It is not enough for a system to be effective; it has to be controlled. Organizations need to define:

- Who can launch assessments?
- Who can view findings?
- Against which environments?
- Where are results stored?
- Against which internal systems?
- How is sensitive data protected?
- Under what approval process?
- How is auditability maintained?
- How are results segmented?
- How are permissions inherited?

Those questions become more complex as the platform retains historical context over time. Continuous testing requires secure persistence: long-term storage, secure retrieval, role-based access control, multi-tenant isolation, audit logging, data retention policies, encryption, and cross-team visibility controls. Without that foundation, the same context that makes the system powerful can also become a governance risk.

The Total Cost of Ownership

Once a team understands what it actually takes to turn an LLM into a penetration testing solution, the next question is not whether it is possible. It is whether building it internally is the best use of the organization's time, budget, and security talent.

Security teams are already stretched across vulnerability management, incident response, compliance, cloud security, identity and access management, architecture reviews, threat modeling, detection engineering, and traditional penetration testing. An internal AI pentesting initiative does not happen off to the side. It competes directly with work the organization already depends on.

Headcount and Expertise

Teams often underestimate the breadth of expertise required. This is not a small internal tool that one security engineer can maintain on the side. A serious AI penetration testing platform requires offensive security, AI engineering, agent architecture, infrastructure, reliability, runtime orchestration, authentication, evaluation, platform operations, product thinking, and customer support.

Continuous Maintenance Burden

The maintenance burden also never really ends. An internal platform means ongoing responsibility for execution environments, browser infrastructure, container orchestration, runtime isolation, attack tooling, sandboxing, credential management, session persistence, logging, patching, scaling, and storage.

Inefficient Use of LLM Tokens

LLM costs are falling, but high-capability inference still is not cheap, especially for work that requires deep reasoning, tool use, memory, and persistence across a real assessment. Without deliberate control over scope and execution, an LLM-driven system can burn a lot of money doing work that does not reduce risk.

Cost to Prove and Act on Findings

Models can produce a lot of findings. That is useful only if the system can prove which ones are real.

Without validation, the cost falls on humans. AppSec teams have to chase disputed tickets, sort through duplicates, fix weak reproduction steps, rerun failed retests, and decide which findings actually matter. Developers lose time trying to reproduce issues that were never clearly proven. Security teams lose credibility when the output feels noisy or inconsistent.

XBOW: Autonomous Offensive Security Testing, Built For Enterprise Trust

Operating a reliable, secure, scalable, continuously evolving offensive-security platform across real enterprise environments takes time, resources, and expertise.

XBOW gives customers the best of both worlds: the power of frontier models applied to offensive security, with the control, governance, and operational maturity enterprises require. Customers can benefit from autonomous testing without inheriting the full burden of building and maintaining the platform internally.

XBOW customers do not have to staff teams for:

- **Runtime orchestration**
- **Model adaptation**
- **Infrastructure scaling**
- **Authenticated workflow handling**
- **Attack environment maintenance**
- **Offensive AI operations**

XBOW gets better over time because it learns from the work: which attack methods are effective, which checks are reliable, how to stay safe, how to manage changing models, and what kind of evidence security teams trust.

A DIY system only gets better if the company treats it like a real product and keeps investing in it. Otherwise, it starts to fall apart. Models change. Browsers change. Login flows change. Applications change. People leave or move teams. The system gets harder to maintain, and the knowledge behind it fades.

SCHEDULE A DEMO