aikido

The Mythos-Ready Security Checklist for SaaS CTOs

Sooraj Shah
Dania Durnas

Intro

This practical checklist is designed to help SaaS CTOs harden both their app and company security in the age of agentic AI and prepare for the next generation of models.

Models like Mythos don't introduce entirely new categories of risk. They change the speed and scale at which existing risks show up. Vulnerabilities are discovered faster, patches can be reverse-engineered quickly, and exploits are now generated within hours.

At the same time, agentic AI components, like MCP servers, plugins, agent skills, and tool definitions, expand what you can do. However, they also increase your attack surface in ways that aren't covered by existing security practices.

The CTOs who weather this well will be the ones who get the fundamentals right and build security practices that hold regardless of what model an attacker is running. The core exposure remains the same, including your application code, APIs, dependencies, and access to production systems. What has changed is how quickly issues in those areas can turn into incidents.

So finding vulnerabilities is no longer an issue. The limiting factor is how quickly teams can validate, prioritize, and fix them in production.

Organizations that continuously test their systems, make clear decisions under pressure, and ship fixes quickly will come out on top. Defenders still have an advantage through access to code, runtime behaviour, and system context, but only if they can use that context effectively. This checklist focuses on what helps you find the issues that matter and fix them fast enough to stay ahead. This list isn't exhaustive of all security requirements for CTOs, but about the areas that new models like Mythos directly impact.

What is means to be Mythos-ready

Being Mythos-ready is not about access to a specific model. It shows up in how your team operates when vulnerability discovery becomes cheaper, faster, and more continuous.

Stronger models will help attackers move faster, but capability on its own does not decide the outcome. Attackers are usually working from the outside, and they don't have full context. Defenders already have access to the code, runtime behaviour, dependencies, and internal system design. That context needs to be used to their advantage.

Most security workflows were built for a slower world, with periodic scans, scheduled patch cycles, manual triage, annual pentests, and release windows that assume there is time to wait. That won't work anymore. As volume rises, teams struggle to sift through a flood of findings to figure out what is real. They lose time getting fixes into production.

Teams that handle this well operate differently.

They design for more findings than people can manually process

They assume vulnerability volume will exceed manual triage. Raw findings are filtered, validated, and de-duplicated before they reach engineering. Otherwise, the team gets flooded, engineers lose trust, and the important issues disappear into the queue.

They use the context that attackers lack

They connect code, runtime behaviour, dependency data, and exposure, separating a theoretical issue from something that's truly exploitable in their environment. Without that context, even the strongest tools will feel shallow.

They treat patching as an operational capability

A critical fix can't depend on the next normal release schedule. The team knows who owns the fix, how it gets reviewed, and how it gets shipped. Time from validated issue to production fix is measured and improved.

They build real barriers

Controls that rely on attacker effort become weaker as AI reduces that effort. Scoped access, segmentation, egress controls, hardware-key MFA, signed builds, and isolated credentials matter more because they limit what an attacker can actually do.

They can tell when something has gone wrong

They have useful logs, off-host storage, visibility into outbound traffic, and a way to spot unusual behaviour from users, services, and build systems, beyond simple alerts.

The following checklist turns this operating model into concrete actions across your company, infrastructure, code, application, and AI exposure.

Know what you run

- [ ] **Maintain a continuous inventory of your attack surface**
  You can't patch, segment, or defend what you do not know exists. Using AI, attackers can quickly enumerate your entire exposed attack surface, so inventory needs to be a live practice instead of a periodic audit. This includes your own code and services, third-party dependencies, and cloud resources. Start with your most exposed systems, build toward full coverage, and use agents to keep it current.

- [ ] **Secure your agentic supply chain**
  MCP servers, agent skills, plugins, and tool definitions extend what your team can do with AI, but they also introduce a supply chain risk that your existing security controls were not built for. A compromised MCP server can manipulate agent behavior in ways that are hard to detect and harder to trace. Before connecting any agentic component to your systems, verify its source, review what it can access, and apply the same scrutiny you would to any third-party dependency. For coding agents like Claude Code and Cursor, define and enforce boundaries around which repositories agents can read, what they can write to or execute, and whether they can reach production credentials or systems. Maintain an inventory of every agentic component in use across your team, including ones engineers have pulled in informally, and audit them regularly for changes.

- [ ] **Monitor subdomain takeover opportunities**
  Subdomain takeovers are a favorite way for hackers to listen in on user cookies. This can happen when a CNAME record points to a service you no longer use (e.g., an old S3 bucket that no longer exists). Use a tool to continuously monitor for this risk.

- [ ] **Review and monitor vibe-coded applications**
  AI app builders like Lovable, Bolt, and Replit let anyone in your org spin up a working application without going through engineering. That's happening whether you have a policy for it or not. A vibe-coded internal tool that touches your database or handles customer data is creates a vulnerable attack surface. Require engineering review before any vibe-coded app connects to production data or systems, and keep an inventory of what's been built and by whom.

Reduce what can go wrong

- [ ] **Require phishing-resistant MFA for privileged access**
  For any account with access to production systems, cloud infrastructure, CI/CD, or deployment credentials, require hardware security keys such as Yubikeys. AI-generated phishing attacks are now convincing enough to defeat standard app-based 2FA, and privileged accounts are the primary target. For lower-risk services like internal tools and communication platforms, app-based 2FA via Google Authenticator is acceptable. Avoid SMS-based 2FA in general.

- [ ] **Keep development, staging, and production cloud accounts separate**
  While you could create virtual networks inside your cloud accounts to keep staging, QA, and production separate, you'll end up continually managing user access rights for new devs and QA teams. We recommend keeping development, staging, QA, UAT, and production infrastructure in completely separate cloud accounts. Network segmentation and isolation reduce the blast radius of a security incident or breach event and are increasingly an expectation from regulators and auditors. All cloud providers offer unified billing, so that's one less headache. There are also techniques for cascading control policies and service guardrails from top down within the cloud service provider accounts and environments, making each environment more secure and consistently configured.

- [ ] **Restrict deployment credentials by IP address**
  CI/CD systems get hacked all the time (read about a real-life example). When you give deployment credentials to your CI/CD systems, make sure to lock them to specific IP addresses as an extra layer of defense. A compromised CI/CD pipeline is particularly dangerous in an environment where autonomous attack tools can move from initial access to full exfiltration, compromising hundreds of repositories in just minutes.

- [ ] **Close common web-exposed paths**
  CSP (Content Security Policy) headers can protect you from common cross-site scripting (XSS) attacks by providing an additional security layer that controls which dynamic resources are allowed to load. That prevents attackers from injecting scripts into your web pages. Setting these correctly closes a class of vulnerabilities that automated scanners actively look for and which agentic AI exploits easily.

- [ ] **Scope AI agent permissions**
  Treat coding agents and MCP servers like production users. Define what they can read, write, execute, and access. Block access to production credentials by default, and log agent actions. An agent with broad permissions and no audit trail is an unmonitored insider with code access.

- [ ] **Keep development, staging, and production cloud accounts separate**
  Don't commit secrets, passwords, or tokens directly into your codebase. Handle sensitive information separately to prevent accidental sharing or exposure. This allows for clear separation between your environments (e.g., development, staging, and production). If you have coding agents that can read and write to your repositories, make sure they don't have access to production secrets. Agents that can escalate their own permissions are a real risk.

- [ ] **Use a VPN for your entire team**
  You can never fully trust the network you use (especially when traveling), so encrypt everything by default. As a bonus, if your VPN offers a static IP address, you can whitelist critical access to administrative tools and services by using conditional access policies.

- [ ] **Conduct regular access reviews**
  Even though you've set up an onboarding checklist, some users might still end up with broader access rights than they really need. If one of these users gets hacked, those extra access rights become an outsized risk. Regular access reviews mitigate these risks.

Control your software supply chain

- [ ] **Use lockfiles to protect your supply chain**
  If you don't use lockfiles, any time you build your application, you'll pull in the latest versions of all references to open-source packages and libraries. Lockfiles enumerate the packages pulled in during the build process, making your app more secure against supply-chain attacks and namespace collision attacks on your open-source dependencies, which have been increasing over the last couple of years. Lockfiles also protect against slopsquatting, where AI coding tools hallucinate package names that attackers pre-register with malicious code.

- [ ] **Monitor your software dependencies**
  Applications are built using dozens of third-party libraries. Many packages and libraries have dependencies on other packages and libraries. A single flaw in any of these sets of packages and libraries could put your entire application at risk. This covers known CVEs in your third-party libraries and vulnerabilities with documented entries in databases like NVD and GHSA. Also track packages approaching end of life, because as support drops off, the exploit risk rises.

- [ ] **Check your code for malware**
  Software supply chain attacks have been accelerating for years. Malicious packages are exceptionally dangerous because attackers act fast once they've established a foothold, and AI-driven attacks compress that time to mere minutes. Note that centralized CVE databases are updated too slowly and won't keep you safe from these kinds of attacks. They have been falling behind for some time and are currently overwhelmed, even before the onslaught of thousands of new exploits of previously undetected bugs and vulnerabilities.

- [ ] **Secure your developer devices as seriously as your infrastructure**
  Developer machines hold cloud credentials, publish tokens, SSH keys, and direct access to source code, so they are increasingly the targets of supply-chain attacks. Deploy an endpoint agent across your developer fleet through your existing MDM to inspect and block risky packages, IDE extensions, and AI tools before they install. It also gives you visibility into which AI tools are running across your team's machines.

Find and validate issues

- [ ] **Enforce a secure code review checklist**
  Always prioritize security during coding, and that goes for pull requests as well. Adjust the checks you carry out depending on where the code is. Dealing with user management is one thing; dealing with business workflows and structures is another. Use your common sense, but also make yourself familiar with typical security issues and logic flaws. For AI-written code, apply the same scrutiny to AI-written code that you would to code from a junior engineer you haven't worked with before. LLM-generated code ships faster, in higher volume, and with consistent defect patterns that are often hard to spot in manual code reviews.

- [ ] **Run AI pentesting against your application**
  Pentesting catches what emerges when your components interact in a running system. This includes broken access control like IDORs, chained auth flaws, and business logic issues that SAST can't catch. Pentesting takes nothing for granted and checks the most basic assumptions, as well as all of your infrastructure. Autonomous agents can run attack simulations against controlled production-like environments, chain multi-step exploit paths, validate findings through real exploitation, and then generate the requisite fix PRs. Tests run complete in a matter of hours. Tests run complete in a matter of hours.

- [ ] **Run AI-assisted vulnerability discovery on your own codebase**
  Attackers now have access to AI tools that can scan your code for exploitable flaws faster than any human researcher can. The defender's advantage is to do the same thing first. Point an AI agent at your own codebase and not just at pentest time, but as a regular practice, with an AI SAST tool (although white-box AI pentesting can cover this too). The goal is to find the things your static analysis tools miss before someone else does. Fight AI with AI.

- [ ] **Check your LLMs for the most common exploits**
  If you're implementing LLMs into your product or internal tooling, test them for the most common exploits before exposing them to users. Prompt injection, indirect injection via retrieved content, and excessive agency are the classes of vulnerability most likely to matter in production. Your threat model changes every time your agent's tools or data sources change, so this is another ongoing process.

- [ ] **Establish a security review gate for AI-generated code**
  AI coding tools ship code faster than human review cycles were designed to handle. The volume is the problem as much as the quality, where even a low defect rate produces a lot of vulnerable code when the output is high enough. One in five organizations suffers a serious incident due to AI-generated code. To prevent this, add a review gate specifically for AI-generated code before it reaches production. This doesn't have to be manual. LLM-assisted review can do the heavy lifting as long as someone or something is looking at the output before it ships. This doesn't have to be manual. LLM-assisted review can do the heavy lifting as long as someone or something is looking at the output before it ships.

- [ ] **Set up a bug bounty program with AI triaging**
  Encourage white hat hackers to report vulnerabilities through a bug bounty program. Offer rewards to make it worth their while. We also strongly recommend that you set up a bug bounty platform to get easy access to skilled ethical hackers. With AI, researchers are now finding more real bugs faster, and the volume of AI-assisted reports has increased sharply, too. Consider adding AI triage and validation to scale the program if submissions are high.

Fix fast and recover

- [ ] **Treat patching as a continuous pipeline**
  AI tools in malicious hands can diff a vendor security patch, understand what it fixes, and generate a working exploit in hours. That means your release and deployment processes need to be able to ship a security fix on the same day it's available, rather than in a week or two. Audit how long it actually takes your team to go from "critical patch available" to "running in production," and work to reduce the time to under 24 hours.

- [ ] **Know your tolerance for downtime before a crisis forces the decision**
  Patching faster means taking things offline sooner and more often. That's a business decision as much as a technical one, and it needs to be made in advance of an active incident. Work out with your leadership team what acceptable downtime looks like for each critical system, and make sure your deployment process can execute an emergency patch without requiring a board meeting to approve it. When possible, the best practice is to implement zero-downtime deployment architectures and rolling deployment techniques.

- [ ] **Off-host tamper-resistant logging**
  Store logs off-host and write-once. If an attacker can modify or delete your logs, you lose the ability to reconstruct what happened. You should be able to answer what changed, who changed it, and what data was touched.

- [ ] **Monitor outbound traffic and unusual service behavior**
  Monitoring helps catch if there's an AI attacker in your infrastructure. Track unusual outbound connections, service account behavior, DNS activity, and build-system access. An AI's lateral movement is fast enough that you may only catch it in the logs after the fact.

aikido

Set up Aikido to automate many of these security checks.

Sign up in less than 3 minutes