ThreatModeler

WHITEPAPER

Ten Architectural Flaws Worth Finding

The design-level risks that live in your architecture, not in your code, and the ones code-level tools tend to miss.

THE SHORT VERSION

The Flaws That Live in the Architecture

Finding code-level flaws has become cheap. A frontier model can return a long list of them in minutes, and scanners have done a version of this for years. The risks in this guide are different. They live in the architecture, in the gap between what a system does and what it was intended to do, and tools that only read what is already in the code routinely miss them.

This guide catalogs ten architectural flaws that threat modeling finds. Each is classified against STRIDE for perspective, with its technical impact, its likely business impact, and an example of how it shows up in real systems. STRIDE is the standard mnemonic for security threats: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege.

None of these flaws announce themselves. Several hide behind a control that is visibly in place but quietly ineffective, which is its own kind of risk: a false sense of security. Catching them takes a clear picture of intended design, kept current as the architecture changes. That is the step you can move but cannot skip, and it is what ThreatModeler® Nexus™ grounds every model in.

Written for a technical audience: the developers and architects who make and review design decisions day to day. Use it as a place to start looking, especially when the architecture changes.

WHY THEY GET MISSED

Designed In, Not Written In

A code scanner reasons about what is present in the code. Prompt-based AI does much the same: it reads what is there and lists what could go wrong with it. Both are useful, and both share one limit. They do not carry a model of what the system was supposed to do, so they cannot see the things that are missing.

Architectural flaws sit exactly there. A control can be present and still wrong for its purpose. An authorization check can pass and still grant access it should not. A trust boundary can be drawn and never say what trust means. These are decisions about intent, placement, and context, and they only surface when a model holds intended design alongside the threats and controls.

INTENT, NOT JUST PRESENCE

The question moves from what vulnerabilities exist to what the system was meant to do, and what breaks if that intent is violated.

CONTEXT ACROSS COMPONENTS

A threat in one component is often mitigated, or worsened, by another upstream or downstream. Read in isolation, that context is lost.

CHANGE INVALIDATES CONTROLS

A refactor, a new access path, or reused data can quietly collapse a control that was sound when it shipped.

A RECORD YOU CAN DEFEND

Decisions held in one place can be queried and proven, instead of reconstructed from memory after the fact.

ThreatModeler Nexus grounds each model against the Secure Design Graph, a connected, queryable record of components, data flows, trust boundaries, threats, and controls. That grounding is what turns a guess read from code into intended design, and what lets the work find what is missing rather than only document what is already there.

01 Authenticated Access without Authorization

STRIDE · SPOOFING · ELEVATION OF PRIVILEGE

Many systems correctly separate unauthenticated from authenticated access by gating on a valid login and session, then stop. The flaw appears when the enforcement point never correlates the authenticated principal with authorization for the specific resource or action. Access should resolve on a role, capability, or attribute scheme, not on the fact of being logged in.

Impact. Improper horizontal or vertical privilege escalation; business cost tracks the value of the impersonation. Among the most common flaws found.

Seen as. A forced-browsing attack, where a user logs in as themselves and then swaps request data to reach another user's account; or a service that authenticates a caller but never distinguishes user from administrative access.

02 Command Injection through Inversion of Control

STRIDE · TAMPERING

To stay dynamic, applications accept input and then evaluate it, or convert it into code to load, link, or run. The flaw appears when the design accepts free-form, untrusted input instead of a known allow-list, then inverts control to execute it.

Impact. Often catastrophic. The attacker can introduce and run arbitrary code inside, or beneath, the application's controls, which puts business impact largely in their hands.

Seen as. Buffer overflows in C and C++, object-deserialization attacks in Java and .NET, string evaluation such as eval() in Python, or fork and exec breakouts such as exec() in PHP.

03 Failure to Protect Integrity in Serial or Persisted Streams

STRIDE · TAMPERING

As systems federate, central control over integrity becomes impractical, and more data, including security metadata such as access-control lists, is carried and stored away from trusted systems. The flaw appears when an architecture exports data to clients or third-party systems and an attacker can modify it in transit or at rest.

Impact. Depends on where integrity checks apply, if anywhere. Sometimes manipulation is bounded by a transaction's verification step, and sometimes it is not. Business cost tracks the data being changed.

Seen as. Tampering with a browser DOM or a session token such as a JWT, altering an available balance or the scope of access encoded in that token, or modifying serialized object streams that carry no integrity check.

04 Applying an Incorrect Cryptographic Primitive

STRIDE · SPOOFING · TAMPERING · ELEVATION OF PRIVILEGE

Cryptography is hard, and even well-tested, certified components fail when used for a purpose they do not serve. Engineers apply an integrity check without a keyed signature that says who wrote the data, or protect privacy when they needed integrity or provenance, trading one property for another.

Impact. Insidious, because a control is visibly in place while being ineffective for its actual objective. The false sense of security is the real risk.

Seen as. Using a plain hash where an HMAC is required, encrypting without first signing, or reaching for a salt where a nonce is needed.

05 Mismatch of Authorization Resolution

STRIDE · SPOOFING · ELEVATION OF PRIVILEGE

A design may check authorization, but at different resolutions across services. Fine-grained, attribute or capability-based control in one service hands off to coarse role-based control in the next, which loses the detail needed to ask whether the user is reaching their own data or someone else's.

Impact. An attacker can coerce the system into acting beyond its permission, and the loss of resolution at the enforcement point often takes auditability with it.

Seen as. A web app that authenticates user sessions but reaches its database with system-to-system credentials only; queues and shared datastores that authorize at the connection level while routing per-user data underneath; or keys and IVs reused across users.

06 Refactoring Causes Security Control Collapse

STRIDE · ANY STRIDE CATEGORY

Introducing a control is not the end of the work. This flaw emerges when later change quietly invalidates a control that was once effective: a platform shift, a new access path, or data reused in a way it was not designed for.

Impact. The control becomes ineffective exactly when it is relied on, with the same false sense of security and a failure scaled to whatever the control protected.

Seen as. SMS used as a second factor, then undone when the same device gains a companion app that can reset the password; or a once-secret identifier such as an account or social-security number repurposed as a public ID.

07 Confused Deputy, Failure to Trace Distributed Flows

STRIDE · ELEVATION OF PRIVILEGE

A component sometimes fails to understand on whose behalf it is performing a privileged action, so an attacker can confuse it into acting maliciously. Paired with mismatched authorization resolution, the context needed to decide may no longer exist within the distributed trace.

Impact. Comparable to misuse of administrative access on an operating system. Privilege escalation occurs, and the trail back to the attacker's session may not be traceable.

Seen as. Account-management or back-office services with broad read and write access to customer records, intended for careful human use but built with no guardrails on the function itself.

08 Aggregate Data Gains Privilege or Sensitivity

STRIDE · INFORMATION DISCLOSURE

Even systems that carefully label sensitive data can miss the moment when several pieces of low-sensitivity information combine to reach the sensitivity of what they protect. A mother's maiden name plus a last-four can be enough to reset a credential, reaching the sensitivity of the credential itself.

Impact. An unexpectedly weak posture through exposure rather than a broken control, and a risk that is miscalculated because the exposure was never modeled.

Seen as. Secondary secrets in authentication and credential reset; session identifiers that stand in for an authenticated user; or access to an IV or salt that moves ciphertext from public to potentially reversible.

09 Exporting Privilege to an Untrusted Component

STRIDE · TAMPERING

In any distributed system, some components hold privileged data or operations that others are not entrusted with. This flaw emerges when the system, often for performance, exports that privilege to a component an attacker controls, who can then remove validations or alter data to their benefit.

Impact. Maps to tampering: bypassed validations, manipulated data, and security controls the business meant to keep for itself now handed to the client or partner.

Seen as. Client-side validation in a browser or mobile app is the familiar case; in zero-trust designs, distributed components asked to protect and sign their own data can be edited or deleted entirely by the attacker.

10 Assigning Unwarranted Trust to a Process or Component

STRIDE · ANY STRIDE CATEGORY

Threat modelers draw trust zones, but the boundary rarely says what trust actually entails. The flaw emerges when two components communicate across a boundary with different expectations of the control protecting it, or communicate within a boundary while taking each other's posture for granted because no boundary was drawn.

Impact. Can apply to any STRIDE category, and tends to track the flaw above it.

Seen as. A firewall that admits all web traffic, including malicious, to the web server; image registries that verify integrity but not provenance; or zero-trust layers where any two authenticated components may talk, with nothing characterizing who may call what, and why.

THE PATTERN, AND THE ANSWER

Catch What's Missing

Read together, the ten share a root. The control is often present; the context that would make it effective is not. Intent, placement, and trust go unstated, so the flaw hides behind something that looks like security. That is why a list of code findings, however fast it arrives, does not close the gap.

ThreatModeler Nexus works against this directly. The System Mapping Agent builds the architecture, from design artifacts or by reading the code of a system already in production. The Secure Design Graph grounds that picture in intended design, capturing trust boundaries, control placement, and authorization intent as one connected, queryable model. The Graph Agent then reasons over it to find what is missing, and the Reporting Agent turns the result into something you can defend to the board.

ONE SOURCE OF TRUTH

Components, data flows, trust boundaries, threats, and controls in a single model, so context is never read in isolation.

THE SAME ANSWER EVERY TIME

Grounded in the Graph rather than a one-off prompt, the model is repeatable and traceable, with a record behind every decision.

CURRENT AS THE DESIGN CHANGES

Refactors and new paths update the model, so a control that collapses is surfaced instead of assumed sound.

GOVERNED BY DESIGN

AI operates within a governed, deterministic framework, on your architectural context and approved security content, to produce explainable, auditable outcomes.

Code-trained tools find what's there. Grounding in design is how you find what's missing.

ThreatModeler

WHERE TO START

Find the flaws that live in the design.

See how ThreatModeler Nexus grounds every model in intended design, against the Secure Design Graph, so the architectural risks in this guide surface instead of hiding.

TALK WITH OUR TEAM

threatmodeler.ai

For more information, support, or inquiries:

Email
support@threatmodeler.com

Phone
+1 201 266-0510

Web
threatmodeler.ai