ThreatModeler®

Top 10 Architectural Flaws Threat Modeling IDENTIFIES

Contents

Introduction

In this eBook, we identify the 10 architectural flaws, or risks, threat modeling identifies. To give a sense of perspective, we classify where each of these flaws fits into the STRIDE framework. STRIDE is an mnemonic for identifying security threats: Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service and Elevation of privilege.

For each flaw we detail its technical impact, and where possible, point out the business implications as well. Also, where possible, we try to offer an estimate for the magnitude of the challenge of protecting against a particular flaw, circumstances under which the flaw tends to occur and some examples of the flaw.

While the goal of this eBook is not to make you feel overwhelmed, it is intended to point out that it's easy to get a false sense of security when it comes to security. Use this eBook as a starting point for where to look for architectural flaws, especially when the architecture changes. Because at the end of the day, threat modeling is a mental exercise in thinking through what you're adversary is going to do. Hopefully this eBook helps you in that regard.

NOTE: This eBook is intended for a technical audience, primarily those involved in day-to-day development.

1 Authenticated Access without Authorization

Applications and services often correctly differentiate unauthenticated and authenticated access to services and functionality by gating functionality with checks on a valid user login and session. This flaw surfaces when an architecture's point of enforcement doesn't also correlate the principal seeking to access functionality or data with authorization prior to granting access. Access may be mitigated on a role-based or capability-/attribute-based scheme.

The technical impact of this flaw is essentially improper horizontal or vertical privilege escalation. The business impact of this flaw is tied to the value of that impersonation. From a STRIDE perspective, this falls under "Spoofing" and almost always "Privilege Escalation". It is undoubtedly one of the top three most common flaws found.

Example of Flaw

Successfully executing a forced-browsing attack (e.g., logging in as User A then replacing request data to access User B's account page) is an exemplary symptom of this flaw. Another is when Service B authenticates requests from Service B but doesn't differentiate user or administrative access.

Command Injection through Inversion of Control

In support of dynamism, applications and services often accept input (data) that it evaluates directly or converts into code to subsequently load/link/execute. This flaw surfaces when an architecture is designed to accept free-form and untrusted input, rather than a known allow-list, and then 'inverts-control' to execute that input either directly or otherwise.

The technical impact of this flaw is often catastrophic, as it grants an attacker the ability to introduce and execute arbitrary code either within (or underneath) the application's security controls or even process space. These technical impacts give the attacker a relatively arbitrary capacity for business impact. Though a struggle, this flaw aligns with "Tampering" in STRIDE.

Example of Flaw

This flaw manifests as many canonical examples in languages that allow direct memory access (e.g., buffer overflows in C/C++) or serialization of objects (e.g., serialization and object- remoting attacks in Java/.NET). Examples of this flaw exist in languages that do not allow direct memory access where string evaluation occurs (e.g., eval() in Python). The flaw also includes "breakouts" where use of a fork/exec based on user input occurs (e.g., exec() in PHP).

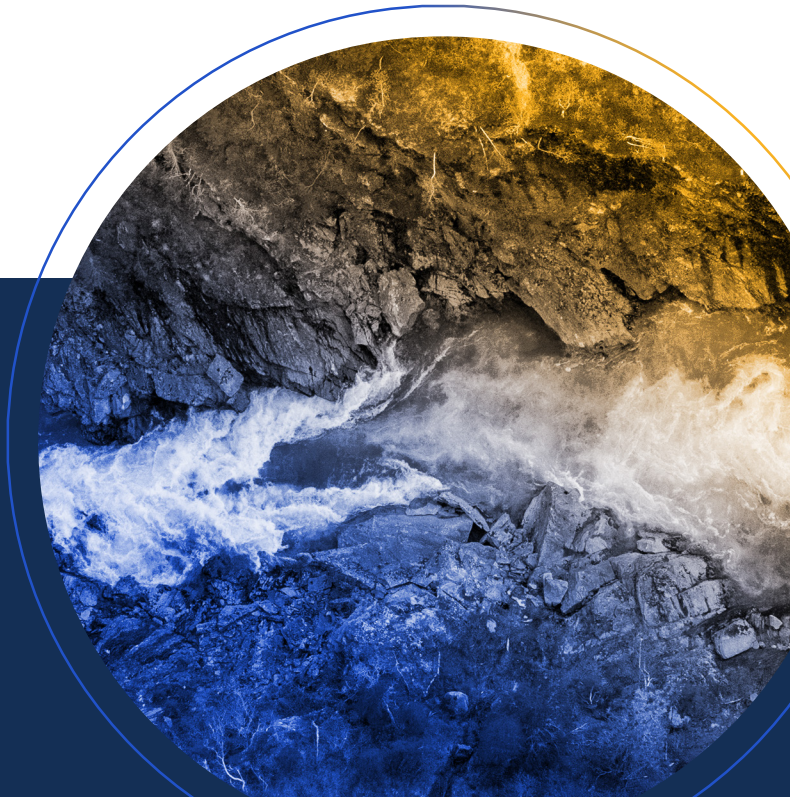
Failure to Protect Integrity in Serial or Persisted Streams

As ecosystems become more decentralized (i.e., federated), the opportunity for central control over integrity and permission becomes less attractive or untenable. More data, including security meta-data such as ACLs, is transported and persisted remotely from trusted systems. This flaw surfaces when an architecture exports data to clients or 3rd-party systems and an attacker can modify that data in transport or as stored.

The technical impact of this finding depends on where, if any place, integrity checks apply. Sometimes an attacker is able to manipulate data but only up to the point of a transaction's verification and clearing, thus limiting its value. The business impact of this flaw is directly related to the data being manipulated. This flaw aligns with "Tampering" in the STRIDE model.

Example of Flaw

Examples of this flaw include manipulation of a browser-based DOM or session data such as a JWT. Manipulation may target user data, such as an "available balance" or security-metadata (such as scope of access or ACLs within a JWT). This flaw may also target serialized request or object stream data, where integrity checks are not applied.



Applying an Incorrect Cryptographic Primitive

Cryptography is exceptionally hard. It's common knowledge that "rolling your own crypto" is folly, but even using expert-provided, well-tested, or certified components can be challenging. This class of flaw occurs when a design employs a control but for a purpose for which it is ineffective. It's not uncommon for engineers to apply integrity checks to data (validating it hasn't changed since write) without a keyed signature indicating who wrote that data. Other situations see designs protect privacy, when they valued integrity or provenance, or otherwise exchanged one capability intended for another of the three.

This class of flaw is both impactful and insidious. Because a control is in place, engineers and security are left with a false sense of security. The technical impact, of course, is that the control is wholly ineffective for its intended purpose. The business impact of this flaw is tied to the failed control objective: provenance, integrity, or privacy. As such, these flaws align with "Spoofing", "Tampering", or "Escalation of Privilege" (where one can read another's data) in STRIDE.

Example of Flaw

Examples of this flaw include using a hash rather than an HMAC and encrypting without first signing or applying a signature. A failure in more subtle distinctions arises when engineers implement a salt rather than a nonce.

Mismatch of Authorization Resolution

Designs may implement an authorization check but not at the same level of resolution between systems or services. For example, authorization may occur through user-level, or even fine-grained attribute-/capability-based access control in one system/service, then role-based access control in the next. In these cases, the second system "loses" resolution necessary to make similarly fine-grained decisions as the first (is the user requesting access to their own data, or another's?). Where architectures propagate attributes through systems allowing finer-grained decision-making capability in a distributed fashion, they may not be properly protected.

The technical impact of this flaw is often that an attacker can coerce the system to do something for which it doesn't have permission, aligning with "Spoofing" or "Privilege Escalation" in STRIDE. The business impact of this corresponds to a breakdown of access control, and often—due to the lack of resolution at the policy enforcement point—carries with it a loss of auditability.

Example of Flaw

Web applications, which authenticate user-specific sessions but use only role-based access control or system-to-system authentication to connect to services (e.g., a database), exhibit this flaw. Message and queuing systems, databases, and other systems that collocate different users', systems', or partners' data often fall prey to this flaw. They authenticate and authorize at a "connection" or "channel" level, while routing/processing messages with their own, more fine-grained accessibility/modifiability is lost. Finally, the affinity of key material with cryptographic primitives may manifest this flaw. Reusing keys or IVs between users would allow each to see or manipulate the others' data.

Refactoring Causes Security Control Collapse

When a security control is properly introduced to an architecture, the work isn't done. This class of flaw emerges when an architecture undergoes change invalidating the effectiveness of a security control. Conceptually, architectural examples that invalidate a security control might include shifting a platform on which an application is provided, enabling user access through a new means, or reusing data in a way it wasn't originally intended.

Like #4, the technical impact of this flaw is the complete ineffectiveness of a control in the face of attack. Business impact includes that same false sense of security, failure commensurate with that which the control protected. May align with any STRIDE category.

Example of Flaw

The canonical example of this flaw is when applications relied on SMS as a second authentication factor, but invalided the "out of band" nature of the SMS by providing users a mobile app. Now, attackers could steal a phone and reset the user's password conveniently by leveraging their MFA control. Reusing previously secret identifiers as public IDs (e.g. CCN, SSN) is another common example.

Confused Deputy, Failure to Trace Distributed Flows

Components within an architecture sometime fail to understand "on behalf of what or whom" they are executing a privileged action. When this occurs, an attacker may "confuse" such a component into conducting malicious actions. The flawed component may not evaluate the appropriate access control or other contextual information as a gating function to the request. Particularly in concert with Flaw #5, the information necessary for the component to make an authorization decision may no longer be available within that scope of a distributed trace.

The technical implication of a confused deputy is akin to misuse of sudo or admin access on an OS. Privilege escalation occurs and the audit trail between the attacker's session may not be easily traced, depending on the auditability of the privileged component and how diligently it tracks on behalf of which callers it takes action.

Example of Flaw

Common examples of this flaw include administrative or account management services, such as exist in customer service or back-office applications. These components may have universal read/write access to customer entities to handle corruption or errors and are intended to be used at the care/discretion of their human operators. However, the services themselves may not put any guardrails on the actual functionality.

Aggregate Data Gains Privilege or Sensitivity

Even when systems diligently label confidential or otherwise sensitive information within their purview, they may not recognize circumstances where the aggregation of certain less sensitive information reaches an equivalent sensitivity or impact to that which they carefully label or protect. For instance, knowing a user's mother's maiden name or "last four" may allow authentication or credential reset, thus reaching an equivalent sensitivity as the credential itself.

Like many flaws documented herein, the technical risk is of an unexpectedly weak security posture. In this case, the exposure of data rather than an ineffective control. Again, the risk to the business is of risk miscalculation due to that unexpected exposure.

Example of Flaw

Examples of this flaw are common in authentication systems, particularly with secondary secrets, as well as in session management (where identifiers may "code for" an authenticated user and access to their data). Similarly, with cryptography, it's evident that a key codes for the ability to encrypt/decrypt/sign, but more subtly, access to an IV or salt may adjust the sensitivity of ciphertext from public to "potentially reversible".

Exporting Privilege to an Untrusted Component

In any distributed system, some components will possess privileged data or functionality while others aren't entrusted with the same capabilities. For instance, store-front applications reserve the privilege of verifying a user has paid before shipping and banking backends validate available funds before making a transfer. This flaw emerges when the system—sometimes for performance reasons—exports such privileged data or operations to a component that can be controlled by an attacker. When exploited, the attacker can remove security controls, such as validations, or modify data and functionality to their benefit.

The technical impact of this flaw aligns with "Tampering" and can amount to data manipulation, bypassing exported validations, authorizations, or other security controls. The business impact is in analog: the customer or partner has been given responsibility the business intended to hold for itself.

Example of Flaw

Client-side (browser or mobile-device) validation is perhaps the most understood manifestation of this architectural flaw. However, in zero-trust architectures, clients or distributed components may be best suited to protect and sign data because they produced it. It's the attacker's prerogative to delete any aspect of the (or the entire) client, to their benefit.

1 Assigning Unwarrented "Trust" to a Process or Component

Threat modelers often draw "trust zones" partitioning their diagrams (and by analog, their systems). There are implied expectations of that boundary: the user is authenticated; the data is encrypted; attackers can observe information in transport; and so forth. "Trust boundaries" are almost never accompanied by an explicit characterization as to what trust entails. This flaw emerges when two components a) communicate across a boundary but possess differing expectations of the security control mitigating that boundary or b) communicate within a boundary but take for granted security properties or posture of the other (i.e., a boundary is missing).

The technical implication of unwarranted trust may apply to any of the STRIDE categories and roughly follow the prior flaw.

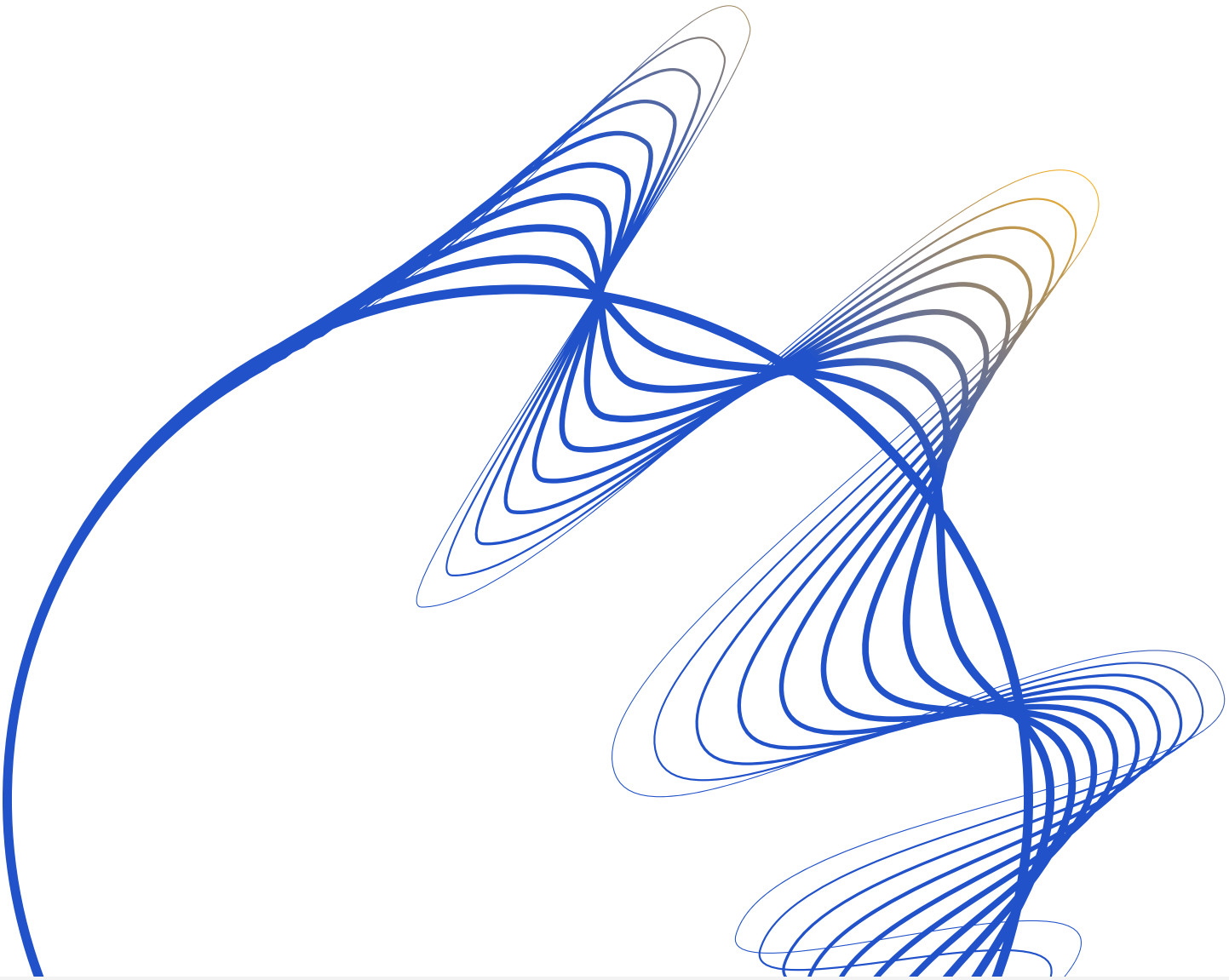
Example of Flaw

Simple infrastructural examples make up the corpus of our understanding of web systems. A firewall lets all web traffic, even malicious, into the webserver. Image registries validate the integrity of components downloaded but typically not their provenance or constituency. Zero-trust architectures may replicate these failures at the application layer, with fancier security controls in place such as any two components can communicate over an encrypted channel once authenticated, but what characterizes "who can call what for which reasons?".

Conclusion

We covered the 10 architectural flaws threat modeling identifies. If after reading this eBook you get the sense that there are a lot of ways a system can fail to protect data/users, we get it. And if you feel like the only way to consistently stay on top of the flaws is by incorporating a repeatable discipline like threat modeling into your development processes, we couldn't agree more.

We hope you found the information in this eBook useful and that it answered many of your questions. But, if you still have questions about how threat modeling can be used to find architectural flaws, we encourage you to contact us here at ThreatModeler. We'll be happy to answer your questions.



For more information, support, or inquiries, please contact us at:

support@threatmodeler.com +1 201 266-0510 threatmodeler.com