API Security Best Practices: A Technical Dialog

Junior Developer: Hi, I'm working on our new Flask backend, and I want to make sure I'm handling security correctly. What are the most important things to focus on for API security?

Senior Engineer: That's a crucial topic. Let's break it down into four key best practices: Authentication, Authorization, Input Validation, and Rate Limiting.

Junior Developer: Okay, Authentication is verifying who the user is, right?

Senior Engineer: Exactly. Never trust that a request is from who it says it is without verification. For a modern API, this usually means implementing token-based authentication. A common approach is using JSON Web Tokens (JWTs). The user logs in with their credentials, receives a signed token, and then includes that token in the header of every subsequent request to prove their identity. Never expose API keys on the client-side.

Junior Developer: Got it. So once they're authenticated, what's Authorization?

Senior Engineer: Authorization is about what an authenticated user is allowed to do. Just because a user is logged in doesn't mean they should have access to everything. For example, a standard user might be authorized to view their own profile data but not an administrator's. We implement this by checking the user's roles or permissions after we've authenticated them.

Junior Developer: That makes sense. What about Input Validation?

Senior Engineer: This is a big one. You should always treat any data coming from a client as untrusted. Never pass raw user input directly to a database query or a system command. This is how vulnerabilities like SQL Injection or command injection happen. In Python, use libraries like Pydantic or Marshmallow to define the expected data structure, type, and format. If the incoming data doesn't match the schema, reject the request immediately.

Junior Developer: So we're essentially creating a strict "allow list" for data. The last one was Rate Limiting?

Senior Engineer: Yes. Rate limiting protects your API from being overwhelmed by too many requests, either from a malfunctioning script or a deliberate Denial-of-Service (DoS) attack. By limiting the number of requests a single user or IP address can make in a given timeframe—say, 100 requests per minute—you ensure the API remains stable and available for all users. You can implement this in Flask using extensions like Flask-Limiter.

Junior Developer: Authentication, Authorization, Input Validation, and Rate Limiting. That's a great framework. Thanks for clarifying!