SCIENCE TECH XPLORE

SECURING THE ENTERPRISE API

FRONT-END PATTERNS · DDOS DEFENSÉ · FIPS-GRADE MULESOFT

VENKATA PAVAN KUMAR GUMMADI

MuleSoft Certified Integration Architect

Securing the Enterprise API

Front-End Patterns, DDoS Defense, and FIPS-Grade MuleSoft Architecture

Venkata Pavan Kumar Gummadi

MuleSoft Certified Integration Architect • Platform Architect • Developer Enterprise API Security & Integration Architect

Published by
ScienceTech Xplore

Securing the Enterprise API Front-End Patterns, DDoS Defense, and FIPS-Grade MuleSoft Architecture

First Published 2023 by ScienceTech Xplore

ISBN 978-93-49929-05-0

ScienceTech Xplore

www.sciencetechxplore.org

Printed and Bounded by
ScienceTech Xplore, India

About the Author

Venkata Pavan Kumar Gummadi is a MuleSoft Technical Architect and technology leader with more than eighteen years of experience in system architecture, digital transformation, and enterprise innovation across insurance, healthcare, telecommunications, and financial services. He has led global teams of more than twenty-five engineers building scalable, cloud-native, API-led platforms — including Wealth InFocus, a cloud and API-driven wealth-communications platform built on MuleSoft — that integrate AI, real-time analytics, and data-driven decision systems.

His research has appeared in peer-reviewed journals including Computer Fraud & Security and the Journal of Information Systems Engineering and Management, on topics from high-volume MuleSoft batch processing to API lifecycle management and API design. He holds three MuleSoft certifications — Certified Integration Architect, Platform Architect, and Developer and a Bachelor of Technology in Engineering. He lives in Connecticut.

Preface

Why This Book?

This book grew out of a conversation I had with a payments-team engineer who had just discovered that every one of her company's internal APIs was reachable from the public internet — not through a breach, but through a misconfigured gateway that had been in production for eight months. The data was fine; nobody had noticed. Yet.

After eighteen years building and securing integration platforms, I have noticed the same three problems appear in almost every enterprise. First, teams treat security as a phase that follows development rather than a constraint that shapes it. Second, they conflate authentication with authorization, and authorization with auditing — three distinct controls that must all be present. Third, they underestimate the operational surface: a well-secured API can still be taken offline by a sustained Layer 7 flood if nobody planned for it.

The Three Problems This Book Addresses

This book addresses all three problems in a structured way. Part One builds the mental model: trust boundaries, the threat landscape, and the gateway patterns that protect availability. Part Two goes deeper into identity, cryptography, secrets management, and Zero Trust. Part Three covers the operational and organizational controls: observability, compliance, DevSecOps, and container security. Each chapter ends with concrete, implementable guidance, not aspirational advice.

The examples throughout the book use MuleSoft Anypoint Platform, which is the integration platform I know best. But the patterns in Part One and Part Two are equally applicable to teams using Kong, AWS API Gateway, Azure API Management, or any other modern gateway technology. The MuleSoft-specific material is concentrated in Chapter 5 and threaded through the examples in later chapters.

Why I Wrote This Book

The API is the front door of the modern enterprise and in most organizations, that door is quietly unlocked. Over more than eighteen years architecting integration and digital-transformation platforms across insurance, healthcare, telecommunications, and financial services, I have designed and operated API systems that move billions of dollars in transactions and millions of customer communications a day. I have also seen, again and again, how easily the very qualities that make these platforms powerful speed, flexibility, automation, and reach become the qualities that make them dangerous when **security** is left for last.

This book grew out of a conversation I had with a payments-team engineer who had just discovered that every one of her company's internal APIs was reachable from the public internet

not through a breach, but through a misconfigured gateway that had been in production for eight months. The data was fine; nobody had noticed. Yet. That moment captures the gap this book is written to close: the distance between an API that appears secure and one that provably is.

My perspective is grounded in practice and in research. I have led global teams of more than twenty-five engineers building cloud-native, API-led platforms that integrate real-time analytics and data-driven decision systems, and I have published peer-reviewed work on high-volume MuleSoft batch processing, API lifecycle management, and API design in journals including Computer Fraud & Security and the Journal of Information Systems Engineering and Management. Everything in these pages has been pressure-tested against real systems and real adversaries, not just whiteboards.

After all those years building and securing integration platforms, I keep seeing the same three problems in almost every enterprise. First, teams treat security as a phase that follows development rather than a constraint that shapes it. Second, they conflate authentication with authorization, and authorization with auditing — three distinct controls that must all be present. Third, they underestimate the operational surface: a well-secured API can still be taken offline by a sustained Layer 7 flood if nobody planned for it.

How This Book Is Organized

This book addresses all three problems in a structured way. Part One builds the mental model: trust boundaries, the threat landscape, and the gateway patterns that protect availability. Part Two goes deeper into identity, cryptography, secrets management, and Zero Trust. Part Three covers the operational and organizational controls: observability, compliance, DevSecOps, and container security. Each chapter ends with concrete, implementable guidance, not aspirational advice.

The examples throughout the book use MuleSoft Anypoint Platform, which is the integration platform I know best. But the patterns in Part One and Part Two are equally applicable to teams using Kong, AWS API Gateway, Azure API Management, or any other modern gateway technology. The MuleSoft-specific material is concentrated in Chapter 5 and threaded through the examples in later chapters.

Who This Book Is For

This book is written for integration architects and senior engineers who are responsible for the security posture of API platforms. It assumes you are comfortable reading XML and YAML configuration files, that you understand the HTTP request-response cycle at a protocol level, and that you have at least a passing familiarity with OAuth 2.0. It does not assume prior deep experience with cryptography, compliance frameworks, or incident response.

How to Read This Book

Read Chapters 1 through 6 in order. They build on each other and establish the reference architecture that the rest of the book defends. Chapters 7 through 15 can be read in any order depending on what is most pressing for your team. The five appendices are reference material: the glossary, the configuration cookbook, the design-review workbook, the case studies, and the threat-control catalog.

The code snippets and configuration examples throughout the book are illustrative. Always test security controls in a non-production environment before deploying them, and always validate that the controls actually work — the difference between a configured control and a working control is the difference between security and the appearance of it.

Good luck, and stay curious.

— Venkata Pavan Kumar Gummadi, Connecticut

Contents

Chapter 1
Foundations & the Security-First Mindset

> *" "Architecture is the decisions you wish you could change later but can't."*
>
> *" I heard a principal architect say that in a hallway in 2016, and I've never found a better definition. Security decisions are the ones you can change least of all.*

1.1 What "enterprise architecture" actually means when something breaks

People love to draw enterprise architecture as a tidy four-layer cake — business, application, data, technology. It's a fine diagram for a steering committee. It is useless at 2 a.m. when the payments API is returning 503s and the on-call engineer is trying to figure out whether it's the gateway, the load balancer, a downstream core banking system, or a junior dev who pushed a bad config at 5 p.m. on a Friday.

So let me give you the version that's survived contact with reality. In a real enterprise, the architecture you care about for security is the set of **trust boundaries** and the **flows that cross them**. Everything else is decoration.

A trust boundary is any line where the assumptions change. Traffic on one side is "the public internet, assume hostile." Traffic on the other side is "inside the DMZ, authenticated but not yet authorized." Cross another line and you're in "the core network, this caller has already proven who it is." Every time data crosses one of those lines, something should be checked, and that check is where security lives or dies.

Here's the mental model I sketch on every whiteboard:

Notice that this is not the four-layer cake. It's a sequence of checkpoints. The discipline of security-first architecture is, simply: **know your boundaries, and never let a request skip a checkpoint.**

"***War story***. A retailer I worked with had a beautiful four-layer architecture diagram framed on the wall. They also had an internal "admin" API that a partner integration could reach directly because someone had punched a firewall rule year earlier "temporarily." The diagram showed no such path. The attackers found it in about a week. The lesson isn't "draw better diagrams." It's that the diagram and the firewall rules had drifted apart, and nobody owned the drift.

1.2 Security as a constraint, not a feature

The single biggest shift I want you to make is this: stop treating security as a feature you add and start treating it as a **constraint you design within** — the same way you treat latency budgets or cost.

When latency is a constraint, you don't ask "should this call be fast?" at the end. You design the data flow so it *can* be fast. Security works exactly the same way. If you ask "is this secure?" in the final review, you've already lost, because the cheap moments to make good decisions are gone. The token format, where state lives, whether the gateway can even *see* the payload it's supposed to inspect — those are decided in the first week.

I use three lenses throughout this book, and I'll keep coming back to them:

- **Confidentiality** — can the wrong person read it? (encryption in transit and at rest, tokenization, masking)
- **Integrity** — can the wrong person change it, or can it be changed without anyone noticing? (signatures, validation, idempotency, audit)
- **Availability** — can a legitimate user actually use it when they need to? (this is where DDoS, rate limiting, circuit breakers, and capacity live)

That third one — availability — is the one architects under-weight and the one this book leans into. A system that's perfectly confidential and perfectly tamper-proof and *down* has failed its users completely. Chapters 3 and 4 are largely about availability under hostile conditions.

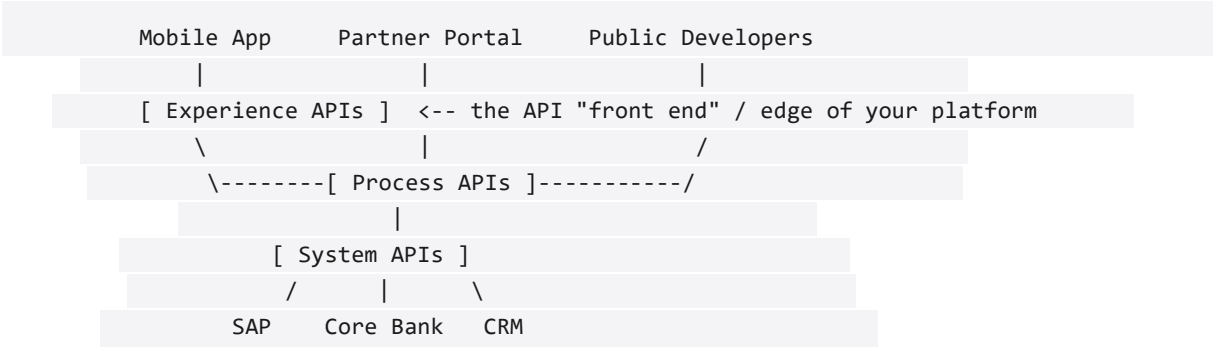
1.3 The API-led layering model

Most modern enterprise integration is organized — at least aspirationally — into three kinds of APIs. MuleSoft popularized this language with **API-led connectivity**, but the idea predates the marketing and you'll see it under other names.

• **System APIs** sit closest to the systems of record. They unlock data from SAP, a core banking platform, Salesforce, a mainframe. They change rarely and they're owned by the team that owns the underlying system.

• **Process APIs** orchestrate. They combine and shape data from multiple system APIs to model a business process — "open an account," "submit a claim." They contain the logic that would otherwise sprawl across every consumer.

• **Experience APIs** are the front door. They're tailored to a specific consumer: the mobile app, the partner portal, the public developer program. This is the layer your external traffic actually touches.



Why does this matter for security? Because **the layer where you put a control determines how much it costs and how well it scales.** The experience layer — what I'll call the **API front end** for the rest of the book — is where you concentrate the controls that deal with *the outside world's behavior*: authentication of external callers, rate limiting, caching, request validation, and the resilience patterns that keep a misbehaving consumer from taking down a system of record.

That's the throughline of Chapter 3. The front end is your blast door.

"***Gotcha***. A common mistake is putting business authorization deep in the system API "because that's where the data is," while leaving the experience API wide open behind a single shared API key. Now every consumer effectively has the keys to the kingdom and you're relying on the deepest, slowest, least-scalable layer to make decisions it doesn't have the context to make well. Push coarse controls (is this caller allowed to talk to me at all? are they over their quota?) to the front; keep fine controls (can this specific user see this specific record?) close to the data. Don't collapse them into one place.

"Defense in depth" gets repeated until it's noise. Let me make it concrete with a single request and show you every layer that should touch it.

A customer taps "View Statement" in a mobile banking app. Here's the gauntlet that request should run:

- **TLS termination at the edge.** The connection is TLS 1.2 at minimum, ideally 1.3, with a modern cipher suite. Anything weaker is refused. (We'll get strict about this in Chapter 5 when FIPS enters the picture.)
- **Edge / WAF inspection.** A web application firewall checks for obvious injection and known-bad signatures. A bot-management layer scores the client.
- **Rate limiting / quota at the gateway.** Is this client over its allowance? (Chapter 3.)
- **Authentication.** The request carries an OAuth 2.0 access token — most commonly a JWT issued by the org's identity provider. The gateway validates the signature, issuer, audience, and expiry *before* the payload reaches any business logic.
- **Authorization (coarse).** Does this token's scope even permit "read statements"?
- **Schema / threat validation.** Is the request shaped the way the contract says? Reject oversized payloads, unexpected fields, wrong content types.
- **Routing to the experience API**, which orchestrates calls to process and system APIs over mutual TLS (mTLS) inside the network.
- **Authorization (fine).** The process/system layer confirms this *user* owns *this* account.
- **Data layer** returns the statement; sensitive fields are masked or tokenized as policy requires.
- **Audit.** The whole thing is logged with a correlation ID so that, after the fact, you can prove who saw what.

No single one of those is sufficient. The WAF misses novel attacks. Tokens get stolen. Rate limits can be evaded with distributed clients. The point of depth is that an attacker has to beat *all* of them, and each layer is cheap insurance against the failure of the others.

I'll be honest about a tension here: every layer adds latency and operational complexity. Security-first does **not** mean "turn everything to maximum." It means making the trade-off deliberately, at design time, with the boundaries in front of you. A public, unauthenticated marketing API and an internal payments API do not deserve the same gauntlet.

1.5 The CIA triad meets the API front end

Let me tie the abstract triad to the concrete patterns this book covers, so you can see where we're going:

Property	Threats	Front-end patterns (Ch. 3–4)	Platform controls (Ch. 5)
Confidentiality	Eavesdropping, token theft, data leakage	TLS everywhere, field masking, careful caching (don't cache private data!)	FIPS-validated crypto, secrets management, TLS config
Integrity	Injection, tampering, replay	Schema validation, JWT signature checks, idempotency keys	Message signing, secure properties
Availability	DDoS, traffic spikes, cascading failure	Rate limiting, spike control, HTTP caching, circuit breakers, bulkheads	Runtime tuning, clustering, autoscaling

If you remember nothing else from this chapter, remember the bottom-right of that table reads "availability," because that's the column most architects forget to staff. We're going to staff it heavily.

1.6 Standards and reference points you should know

You don't need to memorize these, but you should recognize them when a security reviewer cites them:

- **OWASP Top 10 (2021 edition)** — the general web application risk list. Broken Access Control sits at #1.
- **OWASP API Security Top 10 (2019 edition)** — the API-specific list (BOLA, broken auth, excessive data exposure, lack of resources & rate limiting, etc.). We use it as our backbone in Chapter 2.
- **NIST SP 800-53** — the control catalog U.S. federal systems map to; useful even in the private sector as a checklist.
- **FIPS 140-2** — the U.S. federal standard for cryptographic modules. This is *the* standard you'll be asked to comply with in regulated and government-adjacent work, and it's what enterprise platform documentation overwhelmingly targets. Chapter 5 is built around it.
- **PCI DSS** — if you touch cardholder data, this is non-negotiable, and several of its requirements (network segmentation, strong crypto, rate-limited admin access) map directly onto the patterns here.[19]

1.7 How to read the rest of this book

Chapter 2 makes the threats vivid, because you can't defend what you can't picture. Chapter 3 is the heart of the front-end toolbox — the four or five patterns you'll use every day.

Chapter 4 zooms in on the availability nightmare, DDoS, and shows how the Chapter 3 patterns combine into a defense. Chapter 5 gets hands-on with MuleSoft and FIPS, with the screenshots and configuration. Chapter 6 assembles all of it into one architecture and a checklist.

Keep that boundary diagram from §1.1 in your head the whole way through. Every pattern we discuss is really just a better answer to the question: *what should happen when a request crosses this line?*

Chapter 1 takeaways

- Architecture, for security purposes, is **trust boundaries and the flows that cross them** — not a layer-cake diagram.
- Treat security as a **constraint you design within**, decided early, not a feature added late.
- Use the **API-led layering** (system / process / experience) and concentrate outside-world controls at the **experience/front-end** layer.
- **Defense in depth** means an attacker must beat every layer; no single control is trusted alone.
- **Availability** (DDoS, rate limiting, resilience) is the most under-staffed leg of the CIA triad — and the focus of this book.

Next: [Chapter 2 — The Threat Landscape & Anatomy of Attacks](02-threat-landscape.md)

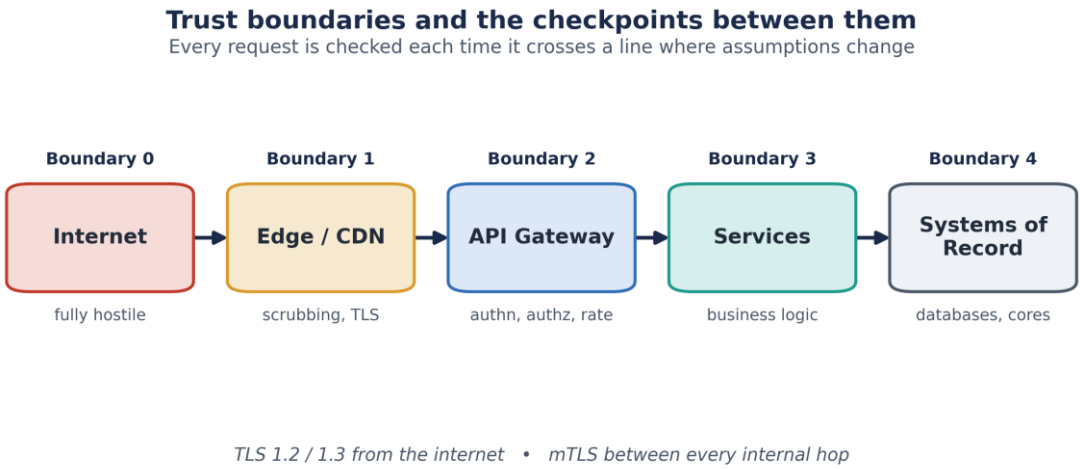


Figure 1-1. Trust boundaries and the checkpoints between them

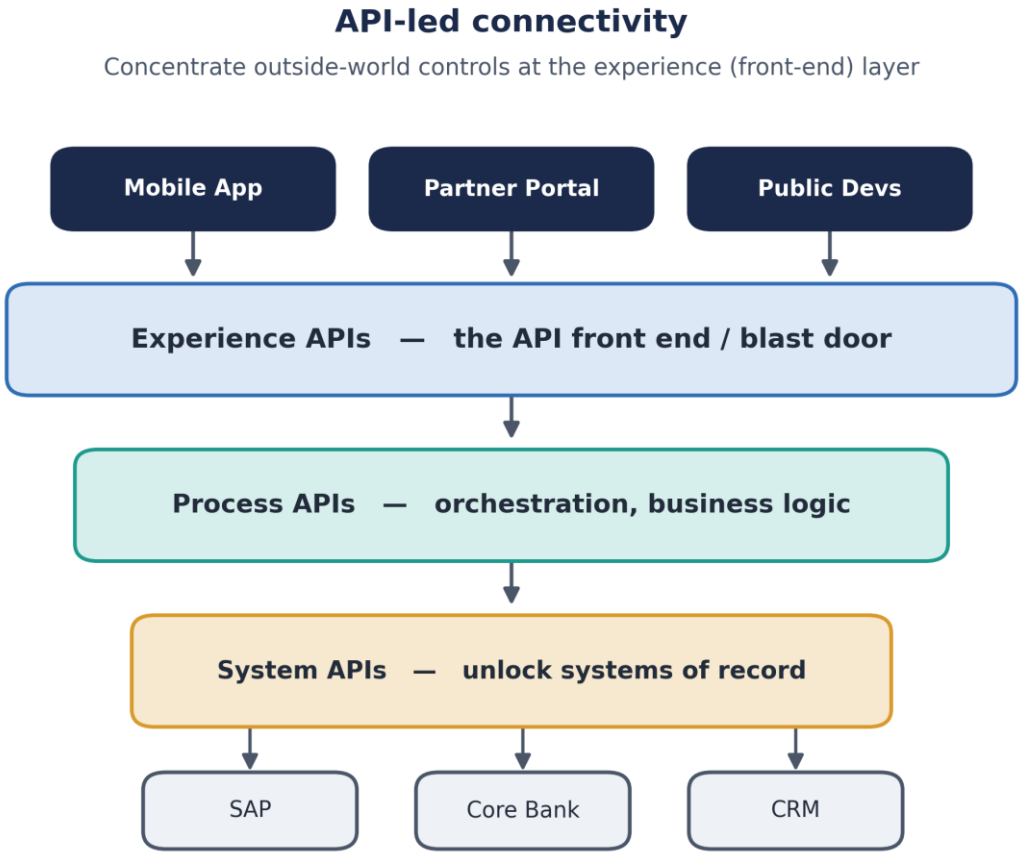


Figure 1-2. API-led connectivity layering

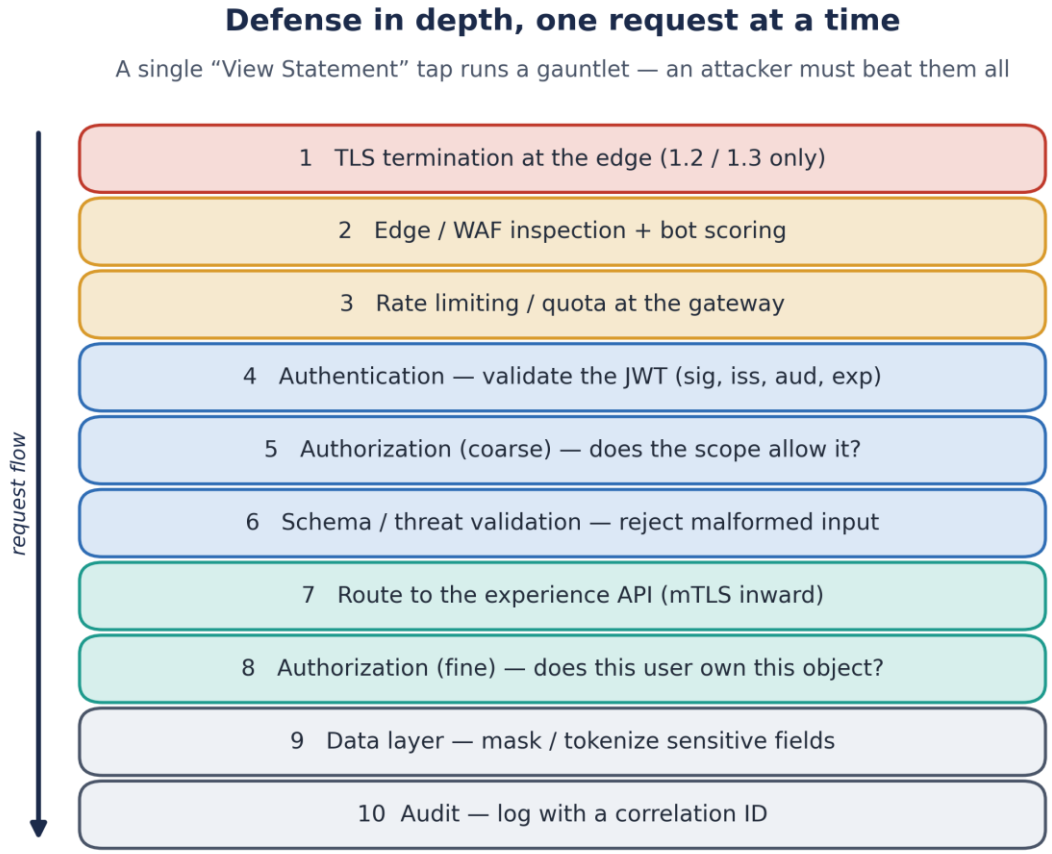


Figure 1-3. Defense in depth — a single request's security gauntlet

Chapter 2
The Threat Landscape & Anatomy of Attacks

“ You will never design a good control for an attack you don't understand. This chapter is the "know your enemy" chapter. I'm going to walk you through how the attacks that matter to enterprise APIs actually work, with enough mechanical detail that the defenses in Chapters 3–5 feel obvious rather than arbitrary.

2.1 Who's actually attacking you

Before the taxonomy, a reality check on adversaries, because the right defense depends on who you're defending against.

- **Opportunistic scanners.** The background radiation of the internet. Automated tools probing every public IP for known holes. They're not after *you*; you're just an IP. The fix is hygiene: patch, don't expose admin endpoints, don't run defaults.
- **Credential-stuffers and abusers.** They have a giant list of leaked username/password pairs and they'll throw them at your login. Or they'll abuse a legitimate-looking endpoint — scraping prices, mass-creating accounts, draining a coupon. This is where rate limiting and bot management earn their keep.
- **Targeted attackers.** Someone who wants *your* data specifically. They'll chain small flaws — an over-sharing endpoint here, a missing authorization check there — into a real breach. Defense in depth is aimed squarely at these.
- **Volumetric / DDoS actors.** Sometimes extortion, sometimes a competitor, sometimes a disgruntled person with a booter subscription. They don't want your data; they want you *down*. Chapter 4 is for them.

Keep these archetypes in mind. A control that stops a scanner cold (patching) does nothing against a DDoS, and vice versa.

2.2 The backbone: OWASP API Security Top 10 (2019)

The API-specific list everyone designs against is the **OWASP API Security Top 10, 2019 edition**. I'll group them and explain the ones that bite hardest.

ID	Name	One-line description
API1	Broken Object Level Authorization (BOLA)	The API trusts an ID in the request and returns *someone else's* object.
API2	Broken User Authentication	Tokens, sessions, or login flows that can be bypassed or forged.
API3	Excessive Data Exposure	The API returns more fields than the client needs and trusts the client to filter.
API4	Lack of Resources & Rate Limiting	No limits, so a client can exhaust CPU, memory, or downstream capacity.
API5	Broken Function Level Authorization	A regular user can call an admin function just by knowing the path.
API6	Mass Assignment	The API blindly binds client input to internal objects, letting a caller set fields they shouldn't.
API7	Security Misconfiguration	Defaults, verbose errors, missing headers, open cloud storage.
API8	Injection	Untrusted input reaches an interpreter — SQL, NoSQL, OS, LDAP.
API9	Improper Assets Management	Forgotten old versions, "shadow" and "zombie" APIs still running.
API10	Insufficient Logging & Monitoring	You got breached and found out from a journalist.

Let me dig into the ones that I see cause real damage and that connect directly to the patterns later in the book.

2.3 Anatomy: Broken Object Level Authorization (API1 / BOLA)

This is the single most common serious API flaw I find, and it's almost boring how simple it is.

Your API has an endpoint:

```
GET /accounts/{accountId}/statements
```

A logged-in user, Alice, has account `100245`. Her app calls:

```
GET /accounts/100245/statements
Authorization: Bearer <Alice's valid token>
```

The server checks the token — it's valid, it's Alice — and returns the statements. Looks fine. But what happens when Alice (or an attacker who stole Alice's token, or just a curious user)

sends:

```
GET /accounts/100246/statements
Authorization: Bearer <Alice's valid token>
```

If the code does this:

```
account = db.getAccount(pathParam.accountId)   // 100246
return account.statements
```

...then Alice just read her neighbor's bank statements. The token was *authenticated* (we know it's Alice) but the request was never *authorized* (does Alice own 100246?). Authentication answers "who are you." Authorization answers "are you allowed." Conflating them is the BOLA trap.

The fix is unglamorous and must live close to the data:

```
account = db.getAccount(pathParam.accountId)
if account.ownerId != token.subject:
    return 403 Forbidden
return account.statements
```

> "***Why the gateway can't fully save you here***. The API front end can do a lot, but it usually doesn't know that account 100246 belongs to Bob. Object-level authorization needs business context, so it belongs in the process/system layer (recall §1.3). This is the canonical example of a fine-grained control that you must not push to the edge. The edge handles "is this caller allowed to talk to me and within their quota"; the depths handle "does this user own this object."

2.4 Anatomy: Injection (API8)

Injection is old and refuses to die. The mechanism: untrusted input is concatenated into a command that an interpreter then executes.

Classic SQL injection. A search endpoint builds a query like this:

```
query = "SELECT * FROM customers WHERE name = '" + input + "'"
```

A normal caller sends `input = "Smith"`. An attacker sends:

```
' OR '1'='1
```

...and the query becomes `SELECT * FROM customers WHERE name = '' OR '1'='1'`, which returns every customer. Worse payloads append `; DROP TABLE` or use `UNION SELECT` to pull data out of other tables.

The fix is **parameterized queries / prepared statements**, full stop. Never build a query by string concatenation:

```
query = "SELECT * FROM customers WHERE name = ?"
db.execute(query, [input])     // the driver binds input as data, never as code
```

At the **front end**, you add a layer of insurance: schema validation that rejects inputs that don't match the expected shape (a name field shouldn't contain `'` and SQL keywords), payload size limits, and a WAF that catches known injection signatures. None of those *replace* parameterized queries — they're depth, not the primary control. I want to be clear about that, because I've watched teams buy a WAF and think they were done. The WAF buys you time and catches the lazy attacks. The prepared statement is what actually closes the hole.

2.5 Anatomy: Broken Authentication (API2) and token theft

The dominant pattern for API auth is **OAuth 2.0** with bearer tokens, very often **JWT** (JSON Web Tokens) carrying the access claims, issued by an identity provider (Okta, Ping, Azure AD, ForgeRock, Keycloak, etc.).

A JWT looks like three base64url chunks separated by dots: `header.payload.signature`. The header says how it was signed; the payload carries claims (`sub`, `iss`, `aud`, `exp`, scopes); the signature lets you verify it wasn't tampered with.

Here's how authentication gets broken in practice:

- **The "none" algorithm trap.** Early/naïve JWT libraries would honor a header that said `"alg": "none"` — i.e., *no signature required*. An attacker crafts a token with whatever claims they like, sets `alg` to `none`, and some servers accepted it. The fix: pin the accepted algorithms server-side; never trust the token's own header to tell you how

much to trust it.

• **Weak signature verification.** Accepting `HS256` (symmetric, signed with a shared secret) where you expected `RS256` (asymmetric), letting an attacker who knows the public key sign tokens. Fix: pin the algorithm *and* the key.

• **No expiry / no revocation.** Tokens that live forever. Steal one and you're in until heat death. Fix: short-lived access tokens (minutes), refresh tokens with rotation, and a revocation path.

• **Tokens in the wrong place.** Putting access tokens in URLs (they land in logs, browser history, referer headers). Fix: `Authorization` header only.

• **Credential stuffing on the login.** Not a token flaw at all — just throwing leaked passwords at the login endpoint until one works. Fix: rate limiting (Ch. 3), MFA, and breached-password detection.

The front-end job is to validate every token *before* the request touches business logic: signature, issuer (`iss`), audience (`aud`), expiry (`exp`), and required scopes. We'll configure exactly this with a MuleSoft JWT Validation policy in Chapter 5.

"***Gotcha***. A surprising number of teams validate the token's signature but never check aud (audience). That means a token your identity provider minted for a different application is happily accepted by yours. Always check that the token was meant for you.

2.6 Anatomy: Excessive Data Exposure (API3) and Mass Assignment (API6)

These two are mirror images.

Excessive Data Exposure is when the API returns the whole internal object and relies on the client to show only the relevant bits. A mobile app shows your name and balance; the JSON response also includes `internalRiskScore`, `ssn`, and `fraudFlags` because the developer just serialized the database row. The data's right there in the response body — the "filtering" was only happening in the UI. Attackers read the raw response.

Fix: define explicit output contracts (DTOs / response schemas). Return only the fields the contract promises. Validate responses against the schema, ideally at the front end.

Mass Assignment is the input-side version. The client sends a JSON body and the server binds it straight onto an internal object:[27]

```
PATCH /users/me
{ "displayName": "Alice", "role": "admin" }
```

If the code does `user.updateFrom(requestBody)` and `role` is a bindable property, Alice just promoted herself. Fix: explicit allow-lists of which fields a given endpoint may set; never blind-bind.

Both of these are arguments for **schema validation at the front end as a contract**, not a suggestion — a theme we operationalize in Chapter 3.

2.7 Anatomy: Lack of Resources & Rate Limiting (API4)

This is the bridge into the rest of the book, so I'll spend a moment on it.

Every API call consumes resources: CPU to parse, memory to hold the payload, a thread while it waits on a database, a connection in a downstream pool. If there's no limit on how fast or how large requests can be, a single client — malicious or just buggy — can consume all of it and starve everyone else. This is the *availability* leg of the triad under attack, and it has several flavors:

- **Request flooding.** Too many requests per second. (Defense: rate limiting / throttling, Ch. 3.)
- **Heavy queries.** A few requests that each ask for a year of transactions across all accounts, unpaginated. (Defense: enforce pagination, cap page sizes, set query timeouts.)
- **Large payloads.** Multi-megabyte uploads to an endpoint that expected a few KB. (Defense: payload size limits at the gateway.)
- **Recursive / amplifying inputs.** A GraphQL query nested 30 levels deep, or an XML payload crafted to expand exponentially ("billion laughs"). (Defense: depth/complexity limits, disable DTD/external entity expansion.)
- **Slow clients.** Connections that send a byte every few seconds to hold a thread open (Slowloris). (Defense: connection/read timeouts at the edge.)

The thing to internalize: **rate limiting isn't a nice-to-have, it's a correctness property of any internet-facing API.** An API without resource limits is a system that *will* go down; the only question is whether it's an attacker or your own success that does it.

2.8 Anatomy: the volumetric attack (DDoS)

I'll preview DDoS here and dedicate Chapter 4 to defense.

A **Denial of Service** attack makes a system unavailable. A **Distributed** Denial of Service uses many sources at once so you can't just block one IP. They come in three broad layers:

- **Volumetric (L3/L4).** Raw bandwidth or packet floods — UDP floods, amplification attacks (DNS, NTP, memcached reflection where a small spoofed request triggers a huge response toward the victim). Measured in Gbps/Mpps. These saturate your pipe before traffic even reaches your application. You cannot absorb these yourself; you need an upstream scrubbing provider / CDN.
- **Protocol (L3/L4).** Attacks that exhaust connection-tracking resources — SYN floods, for example, that open half-connections until the table is full.
- **Application layer (L7).** This is the sneaky one. Requests that look *legitimate* — real HTTP GETs to a real, expensive endpoint — but at a rate designed to exhaust your application or database. Far lower bandwidth, much harder to distinguish from real users. This is where your front-end patterns (rate limiting, caching, circuit breakers) do real work, because the CDN can't always tell the difference but your application logic can.

```
L3/L4 volumetric  ->  saturates the pipe        ->  defended UPSTREAM (CDN/scrubber)
L3/L4 protocol    ->  exhausts connection state ->  defended at EDGE / load balancer
L7 application    ->  exhausts app/DB           ->  defended at GATEWAY + RUNTIME (this book)
```

The single most important mental shift for DDoS: **you defend it in layers, and the layer that stops a given attack is determined by where the attack does its damage.** Trying to stop a 200 Gbps UDP flood at your Mule runtime is like trying to stop a flood with a window screen. Conversely, a clever L7 attack sails right through the CDN and only your application-aware controls will catch it.

2.9 The "shadow API" problem (API9) — the breach you don't see coming

I want to end on the threat that causes breaches *despite* good controls: forgotten APIs.

You build `/v2/accounts` with all the controls in this book. Great. But `/v1/accounts` is still running — it predates your JWT policy, it has a BOLA flaw, and nobody's looked at it in two years. Or a developer stood up a "temporary" debug endpoint that returns config including

a database password, and it never came down. These **shadow** (undocumented) and **zombie** (deprecated but live) APIs are where targeted attackers love to live, because your shiny new controls aren't on them.

The defenses are organizational as much as technical: an API inventory/catalog that's actually maintained, automated discovery scanning your own IP ranges for endpoints, a deprecation process that *removes* old versions on a schedule, and gateway configuration that refuses to route to anything not in the catalog. We'll wire the catalog discipline into the reference architecture in Chapter 6.

"***War story***. The worst breach I ever cleaned up wasn't a clever exploit. It was a /internal/health endpoint that, for debugging, dumped the full application configuration — including database credentials — and had been exposed to the internet through a misconfigured ingress rule for fourteen months. No injection, no stolen token.

2.10 Broken function level authorization (API5)

API5 is when a **regular user invokes an admin function** because authorization checked authentication but not role/privilege for that *operation*.

```
DELETE /admin/users/12345
Authorization: Bearer <valid customer token>
```

If the endpoint only checks "token valid" and not "token has admin scope or role," any logged-in customer can delete users.

Defense:

- Separate admin API base path with stricter policies (IP allowlist, MFA step-up, admin scope)
- Enforce `admin:*` or role claim at gateway **and** application
- Deny by default; explicit allow per function

"***War story***. A "hidden" admin endpoint /internal/v1/cache/clear had no auth because "it's internal." It was reachable through the same load balancer as public APIs after a routing misconfiguration. Attackers cleared caches repeatedly causing performance collapse. Function-level auth + network segmentation would have stopped it.

2.11 Security misconfiguration (API7) — the checklist of shame

The OWASP category that catches everything you forgot:

Misconfiguration	Real example
Default credentials	admin/admin on management console
Verbose errors	Stack traces with SQL in response body
Unnecessary HTTP methods	TRACE enabled
Missing security headers	No X-Content-Type-Options
Open cloud storage	S3 bucket public-read with API exports
CORS * with credentials	Browser-based token theft
Debug endpoints in prod	/actuator/env exposing secrets
Directory listing enabled	Config backups downloadable
Outdated TLS	TLS 1.0 still enabled "for legacy client"

Run **infrastructure and API configuration audits** quarterly — automated scanners plus human review of "temporary" exceptions.

2.12 Threat modeling quick reference — STRIDE per API layer

Layers	Spoofing	Tampering	Repudiation	Info disclosure	DoS	Elevation
CDN/Edge	Fake client IP headers	—	—	Cache leaks	Volumetric	—
Gateway	Stolen JWT	Unsigned	No audit	Token in logs	Rate limit bypass	Scope escalati
Experience API	—	Bad input	—	Over-sharing	Expensive orchestrati	—
Process	—	—	—	BOLA	Downstream fan-out	Mass assignme
System	mTLS bypass	SQL injection	—	DB dump	Heavy queries	DB cred

Use this grid in design reviews (expanded in Chapter 14).

Chapter 2 takeaways

- Match defenses to **adversaries**: scanners, abusers, targeted attackers, and volumetric actors each need different controls.
- The **OWASP API Security Top 10 (2019)** is the backbone. The heavy hitters: **BOLA**, **broken auth**, **excessive data exposure**, and **lack of rate limiting**.
- **BOLA** is fixed with object-level authorization close to the data — the front end can't do it for you.
- **Injection** is fixed with parameterized queries; the WAF and schema validation are depth, not the cure.
- **Authentication** is OAuth2/JWT — validate signature, `iss`, `aud`, `exp`, and scope *before* business logic.
- **Lack of rate limiting** is an availability bug that guarantees an eventual outage. **DDoS** is defended in layers, each at the place it does damage.
- Don't forget **shadow/zombie APIs** — maintain an inventory and decommission old versions.

Next: [Chapter 3 — API Front-End Design Patterns](03-frontend-patterns.md)

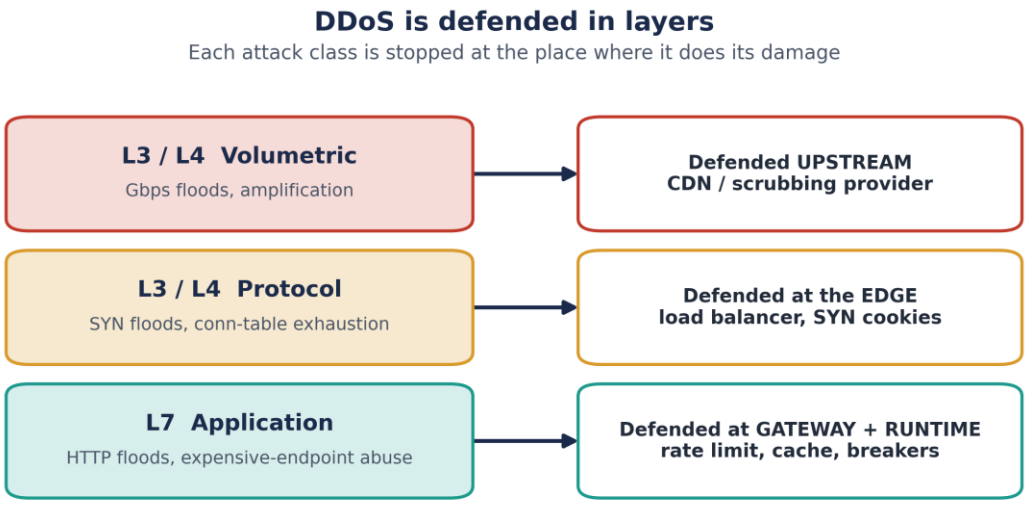


Figure 2-1. DDoS attack classes and the layer where each is absorbed.

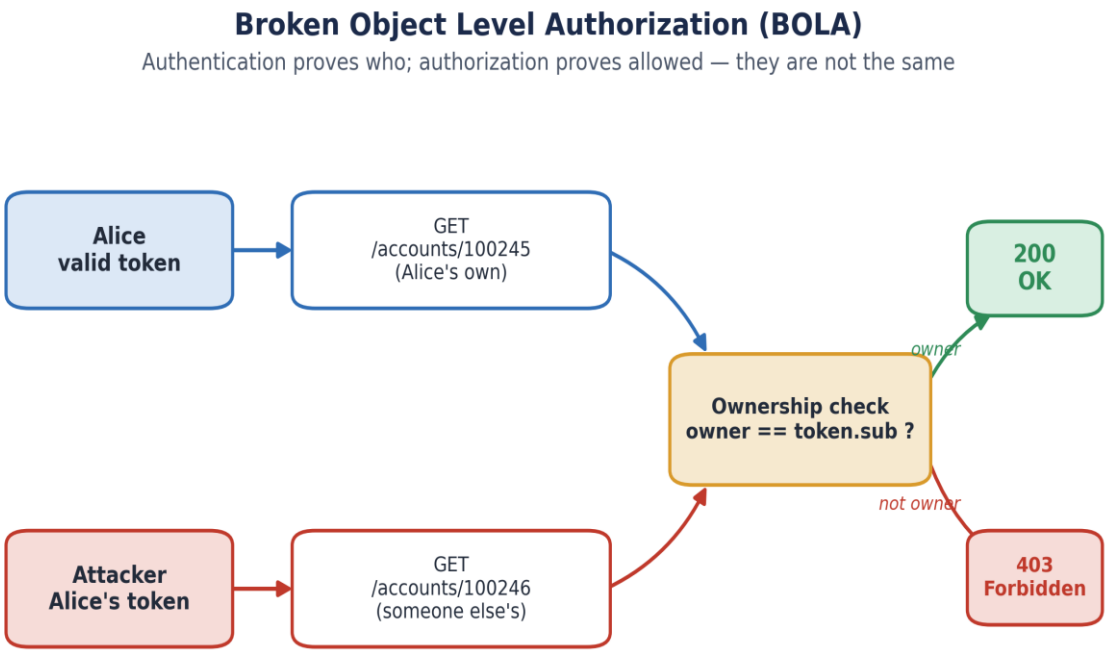


Figure 2-2. Broken Object Level Authorization (BOLA)

Chapter 3
API Front-End Design Patterns

“ This is the chapter I'd keep if I could only keep one. The API front end — the experience layer, the gateway edge — is where a handful of well-understood patterns do an enormous amount of work: they protect availability, smooth out load, contain failure, and enforce the rules of engagement with the outside world. Master these five and you've solved most of the day-to-day battle.

We'll cover, in order:

- **Rate limiting & throttling** (with the spike-control variant)
- **HTTP caching**
- **Circuit breaker**
- **Bulkhead & timeouts** (the circuit breaker's siblings)
- **Policy-driven front ends** — how all of the above get applied declaratively at the gateway

I'll use MuleSoft Anypoint API Manager policies for the concrete configuration because that's our running platform, but I'll call out the vendor-neutral idea each time so it transfers to Apigee, Kong, NGINX, or whatever you run.

3.1 Rate limiting & throttling

The idea

Rate limiting caps how many requests a client may make in a window of time. It's the front-end answer to OWASP API4 (lack of resources & rate limiting) and a frontline DDoS control. There are two flavors people lump together, and the difference matters:

- **Rate limiting (hard):** Over the limit? The request is **rejected**, typically with HTTP `429 Too Many Requests`. Used to protect the system and to enforce commercial quotas.
- **Throttling (soft):** Over the limit? The request is **delayed/queued** and processed when capacity frees up, rather than rejected outright. Smooths bursts at the cost of latency.

You also choose a **scope**: per client/API key, per user, per IP, or global. Per-client is the workhorse — it isolates a noisy or hostile consumer without punishing everyone.

The algorithms (know these, you'll be asked)

```
Fixed window:    count requests in each clock-aligned window (e.g. each minute).
                 Simple. Flaw: a burst at 00:59 and 01:00 doubles the effective rate.

Sliding window:  count requests over the trailing N seconds from "now."
                 Smoother, no boundary doubling. Slightly more state to track.

Token bucket:    a bucket holds N tokens, refilled at a steady rate; each request
                 spends one. Allows controlled bursts up to bucket size, then steady rate.
                 The most common choice for APIs.

Leaky bucket:    requests enter a queue that drains at a fixed rate. Great for throttling
                 (smoothing) rather than hard rejection.
```

For a public API I almost always reach for **token bucket** when I want to permit reasonable bursts, and a **sliding window** counter when commercial quota accuracy matters (you don't want to give a customer 2x their paid rate at minute boundaries).

Configuring it in MuleSoft (Rate Limiting & Spike Control policies)

Anypoint API Manager ships two relevant policies out of the box:

- **Rate Limiting** — hard cap; rejects with `429` over the limit. There's also **Rate Limiting - SLA-based**, which reads the limit from the SLA tier the client's application is registered under (so "Gold" clients get more than "Silver").
- **Spike Control** — a throttling/smoothing policy that queues bursts and lets them through over time rather than rejecting, with a configurable delay and number of retries.

"***Figure 3.1*** — Screenshot: In Anypoint Platform, go to **API Manager (select your API) Policies Add policy**. The policy gallery appears with categories down the left ("**Compliance," "Quality of Service," "Security," "Transformation**"). Under Quality of Service you'll see tiles for Rate Limiting, Rate Limiting - SLA-based, and Spike Control.

"***Figure 3.2*** — Screenshot: Selecting Rate Limiting opens the configuration pane. The key fields are Number of Reqs (e.g. 100), Time Period (e.g. 1), and Time Unit (dropdown: Second/Minute/Hour). Below that, Expose Headers is a checkbox (leave it on — see below), and there's an optional Cluster behavior toggle.

A hard rate-limiting policy configured to 100 requests/minute, expressed as the policy's effective configuration, looks like this:

```
policy: rate-limiting
configuration:
  rateLimits:
    - maximumRequests: 100
      timePeriodInMilliseconds: 60000   # 1 minute
  exposeHeaders: true                    # emit X-RateLimit-* headers
  clusterizable: true                    # share counters across the cluster
```

Spike Control, for smoothing rather than rejecting:

```
policy: spike-control
configuration:
  maximumRequests: 50           # allowed within the window
  timePeriodInMilliseconds: 1000
  delayTimeInMillis: 250        # how long a queued request waits before retry
  delayAttempts: 3              # retries before giving up with 429
  queuingLimit: 200             # max requests held in the queue
  exposeHeaders: true
```

Tell the client the truth: rate-limit headers

A rate limiter that silently rejects is hostile to your good clients. **Always emit standard headers** so well-behaved consumers can back off:

```
HTTP/1.1 200 OK
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 37
X-RateLimit-Reset: 1685620800
```

And when you do reject, be explicit and tell them when to come back:

```
HTTP/1.1 429 Too Many Requests
Retry-After: 22
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 0
```

```
{ "error": "rate_limited", "message": "Quota exceeded. Retry after 22 seconds." }
```

`Retry-After` is the single most important courtesy you can extend. Clients that respect it will self-regulate; clients that ignore it identify themselves as candidates for harder treatment.

Choosing limits without guessing

People agonize over the "right" number. Here's the pragmatic method:

- Pull a week of production traffic. Find the 99th-percentile request rate per client. That's your *normal ceiling*.
- Set the limit at roughly **2–3x** that p99 to absorb legitimate bursts.
- Layer it: a generous **per-client** limit, a tighter **per-IP** limit (to catch a single host hammering with many keys), and a **global** ceiling sized to what your downstream systems can actually survive.
- Watch your 429 rate after rollout. A nonzero-but-low rate is healthy; a spike means either an attack or a limit set too low.

"***Gotcha*** — distributed counters. If your gateway runs as a cluster (it should), a naïve per-node counter lets a client get N× the limit by spreading requests across N nodes. Make sure the limiter shares state across the cluster (in MuleSoft, the clusterizable/cluster toggle; elsewhere, a shared Redis). I've seen "100/min" quietly become "600/min" because of six un-clustered nodes.

"***War story***. A client of mine got hit with a credential-stuffing run against their login API. No rate limit. The attack wasn't huge — maybe 40 requests/second — but each login attempt did a bcrypt hash (deliberately slow, ~100ms) and the thread pool drowned. The whole platform went down, not from bandwidth but from work amplification. A 5/minute-per-IP limit on the login endpoint specifically would have made the attack a non-event. Rate-limit your expensive endpoints harder than your cheap ones.

3.2 HTTP caching

The idea

The cheapest request is the one you never make. **Caching** stores a response and serves it again without re-running the work. For the front end this does two things at once: it cuts latency for users *and* it shields your back end from load — which makes it a genuine availability and DDoS-mitigation control, not just a performance trick. If a hot endpoint is cached at the edge, a flood of identical GETs never reaches your origin.

The HTTP caching model in one page

HTTP has a built-in caching protocol. You should use it rather than inventing your own. The server controls caching with response headers:

```
Cache-Control: public, max-age=300        # cacheable by anyone for 5 minutes
Cache-Control: private, max-age=60         # only the end-user's browser, 1 minute
Cache-Control: no-store                    # never cache (use for sensitive/personal data!)
ETag: "v3-9a8b7c"                          # an opaque version identifier for this response
Last-Modified: Tue, 30 May 2024 12:00:00 GMT
Vary: Accept, Authorization                # cache key must include these request headers
```

Two mechanisms work together:

- **Freshness** (`max-age`): within the window, serve the cached copy with zero origin contact.
- **Validation** (`ETag` / `If-None-Match`): when freshness expires, the cache asks the origin "still `"v3-9a8b7c"`?" The origin replies `304 Not Modified` (tiny, no body) if nothing changed. You skip regenerating and re-sending the payload.

The security rules of caching (read this twice)

Caching is where well-meaning teams leak data. Burn these in:

- **Never cache personal or sensitive responses on a shared cache.** A bank balance cached at a shared CDN can be served to the wrong user. Use `Cache-Control: private` (browser only) or `no-store` for anything user-specific.
- **Get `Vary` right.** If a response differs by `Authorization` or `Accept-Language`, the cache key must include it, or User A gets User B's response. When in doubt for authenticated content, don't share-cache it.

• **Cache the cacheable.** Reference data, product catalogs, public content, config that changes hourly — perfect. Per-user account data — not on a shared cache.

• **Watch cache poisoning.** If an unkeyed input (a weird header) can change the cached response, an attacker can poison the entry for everyone. Keep cache keys tight and validate inputs.

"Gotcha. The most common caching incident I see isn't a performance problem — it's Cache-Control: public, max-age=... accidentally applied to an authenticated endpoint, and a shared cache serving one customer's data to the next. When you turn on edge caching, audit every route for what it's allowed to cache. Default to no-store and opt routes in deliberately.

Configuring it in MuleSoft (HTTP Caching policy)

Anypoint API Manager has an **HTTP Caching** policy under the **Quality of Service** category.

"***Figure 3.3*** — Screenshot: In the Add policy gallery, choose HTTP Caching. The config pane shows a Caching Strategy (Object Store v2 by default), a Cache key expression (a DataWeave expression, default keyed on method + path + query), a Maximum cache entry TTL field, and toggles like Follow HTTP Caching directives respecting origin server Cache-Control and Don't cache when response is an error.

A typical config for a public reference-data API:

```
policy: http-caching
configuration:
  # Build the cache key from method, path, and the query string.
  cacheKey: "#[attributes.method ++ ':' ++ attributes.requestPath ++ ':' ++ attributes.queryString]"
  maxCacheEntryTtl: 300            # seconds
  httpCachingProfile: respect-cache-headers   # honor origin Cache-Control/ETag
  cacheableStatuses: [200, 203, 300, 301, 410]
  invalidationHeaderName: "X-Cache-Invalidate"
```

The crucial choice is the **cache key**. Default it on method + path + query for public data. For anything that varies by user you must either include the user/token in the key (which fragments the cache and reduces its value) or — better — **don't cache it here at all** and push per-user caching to the client with `private, max-age`.

"***War story*** — caching as DDoS armor. A media client of ours kept falling over whenever a story went viral: tens of thousands of identical GETs for the same article hit the origin and the database melted. We put a 60-second edge cache on the article endpoint. Same viral spike the next month: the origin saw a few requests per minute instead of thousands per second, because the cache absorbed the identical traffic. A 60-second cache turned a catastrophic spike into a non-event. Caching is one of the highest-leverage availability controls you have.

3.3 Circuit breaker

The idea

Distributed systems fail in a particularly nasty way: a slow dependency turns into a *cascading* failure. Your API calls a downstream system that's struggling. Each call now takes 30 seconds to time out instead of 50ms. Threads pile up waiting. Your thread pool exhausts. Now *you're* down too — not because you failed, but because you politely waited for something that was never going to answer. Then *your* callers pile up, and the failure climbs the stack.

The **circuit breaker** pattern (popularized by Michael Nygard's *Release It!* and embodied in libraries like Netflix Hystrix and Resilience4j) stops the bleeding. It wraps calls to a dependency and watches them. It has three states:

- **Closed:** normal. Calls pass through. The breaker counts failures.
- **Open:** the failure rate crossed the threshold. Stop calling the dependency entirely — **fail fast** (return an error or a fallback immediately, in microseconds, instead of waiting 30s for a timeout). This is the key move: failing fast frees your threads and protects *you*, and it also gives the struggling dependency room to recover instead of being hammered while it's down.
- **Half-open:** after a cool-off period, let a single trial request through. If it succeeds, close the circuit (recovered). If it fails, snap back open and wait again.

What "fail fast" should return

When the circuit is open you have choices, in rough order of preference:

- **A graceful fallback** — cached/last-known-good data, a default, a degraded-but-useful response. ("Recommendations unavailable" instead of a broken page.)
- **A clear error fast** — `503 Service Unavailable` with `Retry-After`, returned in milliseconds.

The worst option, and the default if you do nothing, is to hang and time out — punishing the user *and* spreading the failure.

Configuring it (Resilience4j-style and on the gateway)

In application code (Java/Spring with Resilience4j, the go-to after Hystrix went into maintenance), the config is declarative:

```
resilience4j:
  circuitbreaker:
    instances:
      coreBankingService:
        slidingWindowType: COUNT_BASED
        slidingWindowSize: 20           # look at the last 20 calls
        failureRateThreshold: 50        # open if >=50% failed
        slowCallRateThreshold: 80       # a "slow" call counts as a failure...
        slowCallDurationThreshold: 2s   # ...if it takes longer than 2s
        waitDurationInOpenState: 30s    # cool-off before half-open
        permittedNumberOfCallsInHalfOpenState: 3
        minimumNumberOfCalls: 10        # don't trip on a tiny sample
```

On the integration layer, **MuleSoft's `until-successful` and a configured `circuit-breaker` in the resilience/retry strategy** play the same role, often paired with **timeouts** and a **fallback flow**. Conceptually:

```
<try>
  <http:request config-ref="CoreBanking_HTTPS" path="/accounts/{id}"
                responseTimeout="2000"/>   <!-- fail fast: 2s, not 30s -->
  <error-handler>
    <on-error-continue type="HTTP:TIMEOUT, HTTP:CONNECTIVITY">
      <!-- fallback: serve last-known-good from object store -->
      <os:retrieve key="lkg-account-#[vars.id]" objectStore="lkgStore"/>
    </on-error-continue>
```

```
  </error-handler>
</try>
```

The two numbers that matter most and that everyone gets wrong:

- **Timeout** must be *aggressive*. The whole point is to not wait. A 2-second timeout on a call that normally takes 50ms is generous. A 30-second default timeout means the breaker never gets to do its job before your threads are gone.
- `waitDurationInOpenState` (cool-off) must be long enough to let the dependency actually recover — usually tens of seconds, not one. Too short and you keep poking a sick system.

> "***Gotcha.*** A circuit breaker with the platform's default timeout is theater. I've reviewed dozens of "we have circuit breakers" architectures where the breaker was wired up correctly but the underlying HTTP call still had a 30-second (or infinite!) timeout. The breaker opens long after the thread pool is already exhausted. Set the timeout first, then the breaker.

3.4 Bulkheads & timeouts — the circuit breaker's siblings

Two more resilience patterns belong in the same toolbox, because a circuit breaker alone isn't enough.

Bulkhead. Named after a ship's watertight compartments: a breach in one doesn't sink the whole vessel. In software, you isolate resources per dependency so that one slow downstream can't consume *all* your threads/connections. Give the flaky reporting service its own pool of, say, 10 threads; when it's slow, those 10 saturate and calls to it queue or reject — but the 90 threads serving the payments path are untouched.

```
resilience4j:
  bulkhead:
    instances:
      reportingService:
        maxConcurrentCalls: 10     # reporting can never eat more than 10
      paymentsService:
        maxConcurrentCalls: 50
```

Timeouts everywhere. Every network call must have a timeout. No exceptions. An untimed call is a thread you might never get back. Set them based on the dependency's real p99 plus headroom — not a copy-pasted default.

Together the resilience trio works like this: **timeouts** bound how long any single call can hurt you, **bulkheads** bound how much of your capacity one dependency can consume, and the **circuit breaker** stops calling a dependency that's clearly down. Use all three; they cover different failure shapes.

3.5 Policy-driven front ends

The idea that ties the chapter together

Everything above — rate limiting, caching, auth, validation — could be written into each API's code. **Don't.** The power of a modern API front end is that these are applied as **policies**: declarative, reusable units of cross-cutting behavior, attached to APIs at the gateway, managed centrally, and changeable without redeploying the service.

This is the operational heart of "security as a constraint, not a feature" from Chapter 1. When security is a policy you attach, you can:

- Apply a consistent baseline (TLS, JWT validation, rate limit) to *every* API by default.
- Change a limit or rotate a rule across the whole estate in minutes, no code change, no redeploy.
- Prove compliance — "show me every API and its attached policies" is one screen, not an archaeology project.
- Let platform/security own the policies while product teams own the business logic.

How it works in MuleSoft (API Manager + automated policies)

In Anypoint, you manage policies per API in **API Manager**, and you can enforce baseline policies across many APIs with **Automated Policies**.

"**Figure 3.4** — Screenshot: API Manager API Administration lists every managed API with columns for Name, Version, Status (Active), and a count of Applied policies. This is your at-a-glance compliance view — any API showing 0 policies is a red flag.

"**Figure 3.5** — Screenshot: The Policies tab for a single API shows an ordered list of applied policies (e.g. 1. JWT Validation, 2. Rate Limiting, 3. HTTP Caching) with up/down arrows. Order matters — authentication runs before rate limiting runs before caching. Drag to reorder; the gateway executes top-to-bottom on the request path.

"**Figure 3.6** — Screenshot: Under API Manager Automated Policies, you define policies (e.g. JWT Validation, IP allowlist, Rate Limiting) that are automatically applied to all APIs in the environment matching a runtime/gateway version. New APIs inherit the security baseline the moment they're deployed — no one can forget to add it.

Policy ordering — the part people get wrong

The order of front-end policies isn't cosmetic; it's a security property. Here's the order I apply on a typical external-facing experience API, and *why*:

```
1. IP allowlist / blocklist        (cheapest rejection first; drop known-bad before any work)
2. JWT / OAuth token validation    (authenticate before doing anything expensive)
3. Rate limiting (per client)      (now that we know WHO, enforce their quota)
4. Header / schema validation      (reject malformed/oversized payloads early)
5. HTTP caching                    (serve cached responses without hitting the backend)
6. (route to backend)
```

The principle: **reject as early and as cheaply as possible.** You don't want to spend a database lookup on a request you're going to throw away for being over its rate limit. And you must authenticate *before* you rate-limit *per client*, because you can't key a per-client limit on a client you haven't identified yet. (Pre-auth, you can still rate-limit per IP — and you should, to protect the auth step itself.)

"**Gotcha**. I once saw caching placed before authentication "for performance." The result: the gateway served cached, authenticated-user data to unauthenticated callers, because the cache hit short-circuited the auth check entirely. Authentication and authorization must run before anything that can return data. Order your policies like a security checkpoint, not a performance optimization.

A note on custom policies

The built-in policies cover most needs, but you can also write **custom policies** for Anypoint (packaged with the Mule SDK / `mule-http-policy` archetype) when you need

something specific — a bespoke header transform, a proprietary auth scheme, a regional data rule. The discipline is the same: keep the logic in a reusable policy, not sprinkled through application code.

3.6 Idempotency keys — integrity under retry

Mobile clients retry on timeout. Without **idempotency**, a payment API might charge twice.

```
POST /payments
Idempotency-Key: 550e8400-e29b-41d4-a716-446655440000
Authorization: Bearer ...
{ "amount": 100, "account": "100245" }
```

Gateway or service stores key + response for 24 hours. Duplicate key with same body return cached response (200/201), don't re-execute.

Security angle: idempotency keys prevent **replay-induced duplicate transactions**— an integrity control, not just UX.

Implement in process layer with object store or Redis; gateway can pass through header.

3.7 API versioning and deprecation — security of the lifecycle

Shadow APIs (Chapter 2, API9) thrive when versioning is informal.

```
/v1/accounts  — deprecated 2022-01, sunset 2024-06, NO JWT policy
/v2/accounts  — current, full policy stack
```

Secure lifecycle:

- Announce deprecation in response header: `Sunset: Sat, 01 Jun 2024 00:00:00 GMT`
- Monitor `/v1` traffic; contact remaining consumers
- **Remove route** at sunset — don't leave zombie
- Automated policies apply to new versions by default

3.8 GraphQL-specific front-end controls

If your experience layer exposes GraphQL:

Control	Setting
Max query depth	10
Max complexity score	500 (calibrate)
Disable introspection in prod	Yes (or auth-gated)
Persisted queries only	For public mobile apps
Rate limit by query cost	Not just request count

GraphQL's flexibility is an attack surface — one query can fan out to dozens of downstream calls (DoS).

3.9 Policy testing and promotion workflow

Before promoting policy changes to Production automated policies:

```
1. Apply to Sandbox API with test client
2. Negative tests: no token, bad token, over limit, bad schema
3. Positive tests: golden path per SLA tier
4. Load test rate limit threshold
5. Change ticket + approval
6. Apply to Production automated policy
7. Monitor policy_violation metrics 30 min post-change
```

"**Figure 3.7** — Screenshot: API Manager Policy test console or external tool (Postman collection run) showing pass/fail matrix for six test cases. Attach to change ticket as evidence.

• **Rate limiting** (hard `429`) and **throttling/spike control** (soft, queued) are mandatory availability controls. Pick an algorithm (token bucket is the default), scope per-client, emit `X-RateLimit-*` and `Retry-After` headers, share counters across the cluster, and limit expensive endpoints harder.

• **HTTP caching** is a security/availability control as much as a speed one — but get `Cache-Control`, `Vary`, and cache keys right, and **never** share-cache personal data.

• **Circuit breakers** stop cascading failure by **failing fast**; they're only as good as the **aggressive timeout** behind them. Pair with **bulkheads** (resource isolation) and **timeouts everywhere**.

• **Policy-driven front ends** make all of the above reusable, centrally managed, and provable. **Order matters**: reject cheap-and-early, authenticate before rate-limiting per client, and never cache ahead of auth.

Next: [Chapter 4 — Defending APIs from DDoS Attacks](04-ddos-defense.md)

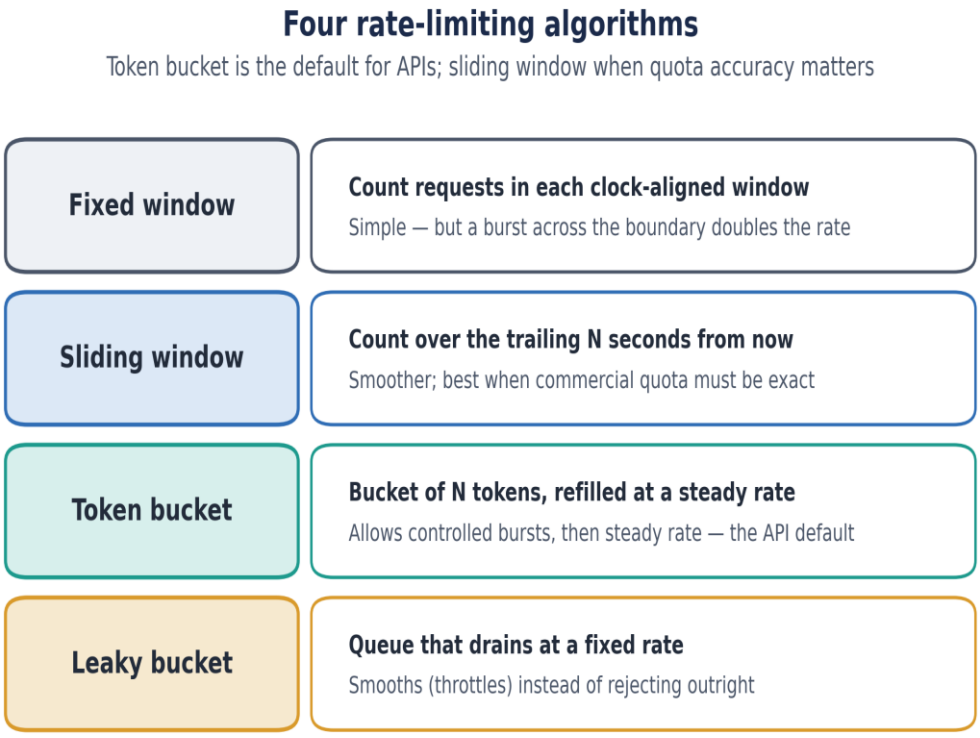


Figure 3-1. The four rate-limiting algorithms.

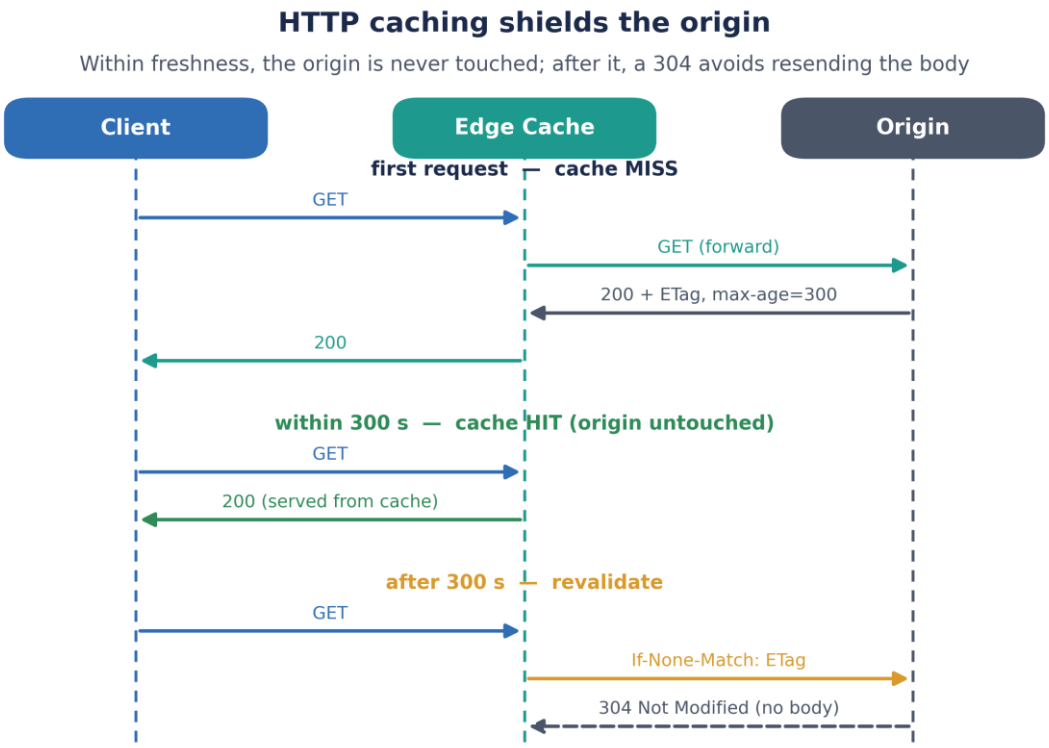


Figure 3-2. HTTP caching across three scenarios.

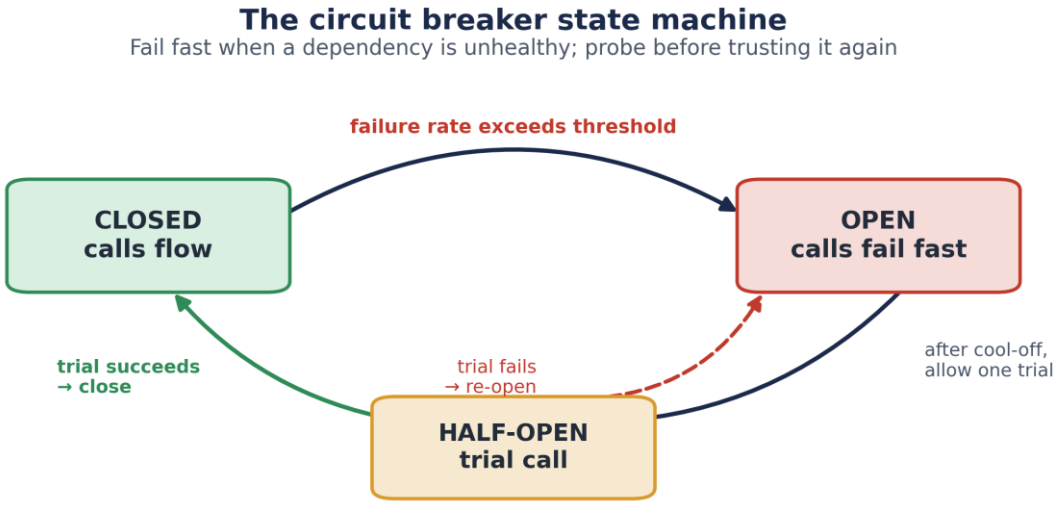


Figure 3-3. The circuit breaker state machine.

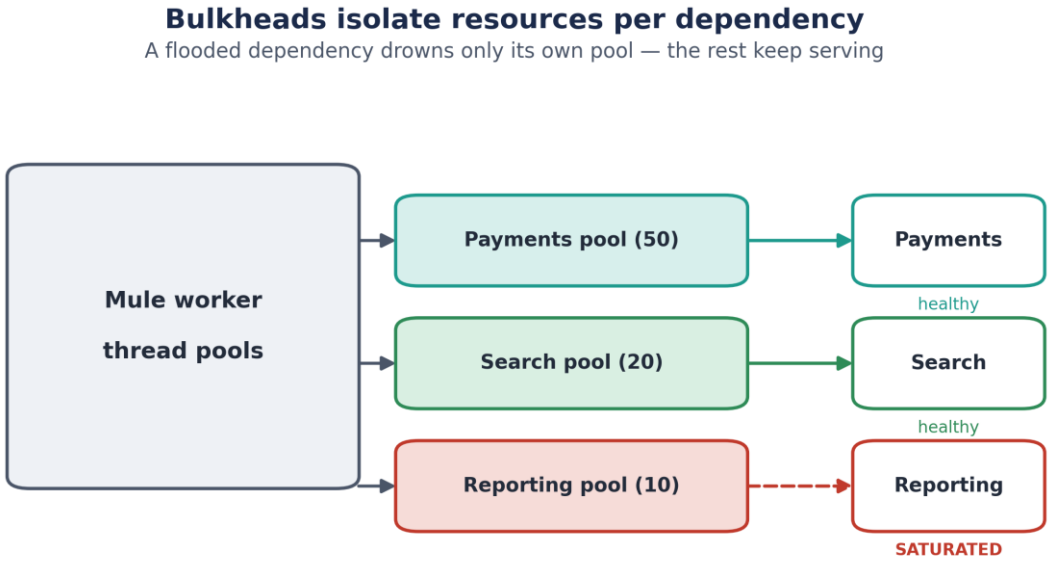


Figure 3-4. Bulkhead isolation.

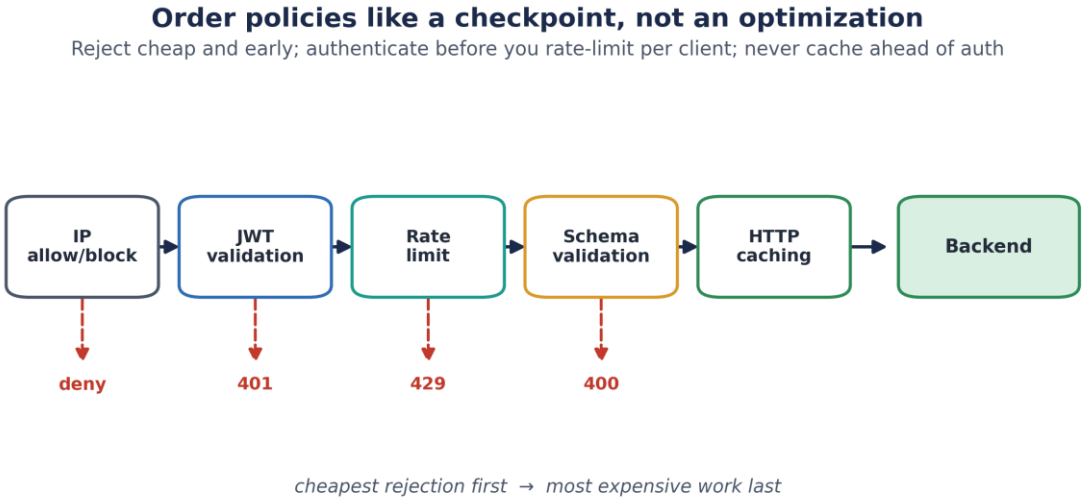


Figure 3-5. Correct gateway policy ordering.

Chapter 4
Defending APIs from DDoS Attacks

" A DDoS is the purest attack on availability. The attacker doesn't want your data — they want your users locked out. This chapter is a layered playbook: from the network edge down to the runtime, who stops what, and how to tune the front-end patterns from Chapter 3 so they actually hold under a flood instead of falling over with everything else.

4.1 First, the hard truth about where DDoS is won

You cannot defend a large volumetric DDoS at your own infrastructure. I'll say it plainly because architects waste enormous effort here: if 300 Gbps of garbage is aimed at your network, your firewall, your load balancer, and your Mule runtime are already underwater before any clever application logic runs. The pipe is full. Game over.

So the first principle of DDoS defense is **defend each layer at the place where the attack does its damage, and push volumetric defense upstream of your own infrastructure entirely.**

Your architecture's job is twofold: (1) **contract or deploy upstream scrubbing** so volumetric floods never reach you, and (2) **build the L7 defenses yourself**, because those attacks look like real traffic and only your application-aware controls can tell the difference.

4.2 Layer 1 — the upstream edge (where you stop the big floods)

Sit a **CDN / DDoS-scrubbing provider** in front of everything. Common choices are Cloudflare, Akamai, AWS Shield (with CloudFront/ALB), Azure DDoS Protection (with Front Door), Google Cloud Armor, and Fastly. What you get:

- **Anycast absorption.** Your traffic is spread across a global network with terabits of capacity. A flood gets diluted across hundreds of points of presence instead of hitting one data center.

• **Automatic L3/L4 mitigation.** SYN floods, UDP/amplification reflection, and malformed-packet attacks are scrubbed before they reach your origin — often without you doing anything.

• **Origin hiding.** This is critical and often skipped: **lock your origin so it only accepts traffic from the CDN.** If attackers learn your origin IP, they bypass all that protection and hit you directly. Use the CDN's IP allowlist on your firewall/security groups, and rotate the origin IP if it leaks.

> **"Gotcha — the leaked origin.** Teams put Cloudflare in front of their API, feel safe, and leave the origin reachable on its public IP. Attackers find the real IP (old DNS records, certificate transparency logs, an email header) and attack it directly, sailing past the CDN. The CDN only protects what it sits in front of. Firewall your origin to the CDN's published ranges and verify it with a direct curl from outside — if you can reach the origin without going through the CDN, so can the attacker.

4.3 Layer 2 — the edge / load balancer (protocol attacks and first triage)

At your own perimeter (the load balancer / reverse proxy / ingress), you handle protocol-level exhaustion and do coarse triage:

• **SYN cookies** to survive SYN floods without a connection table that fills up.

• **Connection limits and timeouts.** Cap concurrent connections per source IP. Set aggressive **read/write timeouts** to kill **Slowloris**-style slow attacks that hold connections open by trickling bytes.

• **TLS offload with sane limits.** TLS handshakes are CPU-expensive; an attacker forcing endless renegotiations can exhaust CPU. Cap handshake rates; disable client-initiated renegotiation.

• **Geo/IP filtering** for coarse, cheap rejection if your API is regional and you're under attack from regions you don't serve. (Use sparingly — it's blunt.)

NGINX is the canonical example, and the config reads like the Chapter 3 patterns applied at the edge:

```
# Limit connections per client IP
limit_conn_zone $binary_remote_addr zone=perip:10m;
# Limit request rate per client IP (10 req/s, burst of 20)
limit_req_zone  $binary_remote_addr zone=reqs:10m rate=10r/s;

server {
    limit_conn perip 20;                # max 20 simultaneous conns per IP
    limit_req  zone=reqs burst=20 nodelay;

    client_body_timeout   5s;           # kill slow body senders (Slowloris)
    client_header_timeout 5s;
    send_timeout          10s;
    keepalive_timeout     15s;

    client_max_body_size  1m;           # reject oversized payloads early
}
```

Notice this is just **rate limiting + timeouts + payload caps** — the same patterns from Chapter 3, applied one layer further out. The patterns compose across layers; that's the whole point of defense in depth for availability.

4.4 Layer 3 — the API gateway (L7 defense, where your patterns earn their keep)

This is the layer where application-layer DDoS — the HTTP floods that look like real requests — is actually stopped. Everything in Chapter 3 now gets tuned specifically for hostile load.

Rate limiting, tuned for attack

Your normal rate limits (§3.1) are the first wall. Under attack, the tuning that matters:

- **Layer your scopes.** Per-IP (catches a single host), per-client/API-key (catches a compromised consumer), and a **global ceiling** sized to what your back end can survive. The global ceiling is your circuit of last resort — better to `429` some legitimate traffic than to let the database die and serve *no one*.
- **Tighten on expensive endpoints.** Login, search, report generation, anything that does real work. An L7 attacker will target your most expensive endpoint precisely because it amplifies their effort. (Recall the bcrypt war story in §3.1 — 40 req/s took a platform down.)

- **Fail cheap.** A `429` must cost you almost nothing to produce. Make sure the rejection happens *before* auth lookups, DB hits, or downstream calls (policy ordering, §3.5).

Spike control / throttling

Hard limits reject; **spike control** (§3.1) smooths. Under a bursty attack, spike control queues the surge and drains it at a rate your back end tolerates, so a momentary spike doesn't translate into a momentary outage. Combine them: spike control to smooth legitimate bursts, hard rate limits as the absolute ceiling.

Caching as a flood absorber

This is one of the most effective and underused L7 DDoS defenses. If an attacker floods a **cacheable** endpoint, a front-end cache (§3.2) serves the flood from memory and your origin never sees it. The viral-article war story from Chapter 3 is exactly an unintentional DDoS that caching neutralized. Audit your hot, public, read-heavy endpoints and make sure they're cached — that converts "thousands of identical requests per second hammering the DB" into "a handful per cache-TTL."

The catch: caching only helps for **cacheable** requests. An attacker who varies the query string (`?cachebuster=<random>`) on every request defeats a naïve cache. Mitigations: ignore unknown query params in your cache key, normalize keys, and rate-limit cache *misses* specifically (a high miss rate from one source is itself a signal).

WAF and bot management

A **Web Application Firewall** at the gateway/edge inspects L7 traffic and blocks requests matching attack signatures and behavioral rules. For DDoS specifically, the useful WAF features are:

- **Rate-based rules** ("if any IP exceeds N requests in M minutes, block for T minutes") — automated, dynamic rate limiting.
- **Bot scoring / managed challenges** — present a JavaScript or CAPTCHA challenge to suspicious clients; real browsers pass, simple flood scripts don't. (Use challenges, not hard blocks, to avoid locking out real users behind a shared NAT.)
- **Anomaly detection** — flag traffic that deviates from your normal baseline (sudden geographic shift, unusual user-agents, no `Referer`).

Circuit breakers and bulkheads — surviving the part that gets through

No defense is perfect; some attack traffic will reach your services. The resilience patterns (§3.3–3.4) determine whether that means *degradation* or *collapse*:

- **Circuit breakers** keep an overwhelmed downstream from taking down everything that calls it. If the database is saturated, the breaker opens and callers fail fast (or serve last-known-good from cache) instead of piling up.
- **Bulkheads** ensure the attacked endpoint can't consume every thread. If the attacker floods `/search`, the bulkhead caps search at its allotted pool; `/payments` keeps working. You degrade one feature instead of losing the platform.

This is the difference between "search was slow for 20 minutes" and "we were completely down for 20 minutes." Same attack, vastly different blast radius, decided entirely by whether you isolated resources.

4.5 Layer 4 — the runtime and back end (last line)

At the application/runtime layer:

- **Right-size and autoscale.** Horizontal headroom buys time. Autoscaling on request rate / CPU can absorb moderate L7 floods (until cost or downstream limits bite). Don't rely on it alone — autoscaling a stateless gateway is easy, but it just pushes load onto a database that *can't* scale as fast.
- **Protect the data tier.** The database is usually the real bottleneck and the thing an L7 attack is ultimately trying to exhaust. Connection pools with hard caps (so the app can't open unlimited DB connections), query timeouts, read replicas for read-heavy load, and the caching layer in front all protect it.
- **Shed load gracefully.** When saturated, prefer returning `503` with `Retry-After` quickly to queuing forever. A fast, honest "come back in 30 seconds" preserves capacity for some users; an infinite queue serves no one.

4.6 Detection and response — you can't defend what you can't see

Mitigation without detection is luck. Wire up:

- **Baseline metrics:** requests/sec, error rate, p99 latency, `429`/`503` rate, cache hit ratio, downstream connection-pool usage, per-IP and per-client request counts. Know your normal so you can recognize abnormal.

• **Alerts on the right signals:** a sustained spike in request rate *combined with* rising latency and error rate is the classic DDoS signature. A spike in traffic with *stable* latency might just be legitimate success — don't cry wolf.

• **Dashboards that distinguish layers:** is the pain at the edge (bandwidth), the gateway (429 storm), or the data tier (connection pool maxed)? That tells you which lever to pull.

• **A runbook, written before the attack.** Who can raise the CDN's "under attack" mode? Who can tighten rate limits in API Manager? Who decides to geo-block? Decide this on a calm Tuesday, not during the incident.

"Figure 4.1 — Screenshot: In Anypoint Monitoring, an API dashboard shows time-series tiles for Request volume, Response time (p99), Failure rate, and Policy violations (429s). During an L7 attack you'll see request volume and 429s spike together while p99 climbs — that correlated pattern is your signal to escalate to the DDoS runbook.

"War story — the calm playbook. The best-run incident I was part of barely felt like an incident. The team had rehearsed: when the alert fired, on-call flipped the CDN to "under attack" challenge mode, tightened the per-IP rate limit from 100 to 20 per minute through API Manager (a config change, no redeploy — that's the payoff of policy-driven front ends from §3.5), and watched the dashboards. The attack was an L7 HTTP flood; within four minutes the challenge page and the tighter limit had shed 90% of it, the database connection pool dropped back under 50%, and real users barely noticed. No heroics. Just layers, a runbook, and the ability to change a policy in seconds.

4.7 The DDoS defense checklist

A condensed, in-order checklist you can lift into a design review:

```
EDGE / UPSTREAM
[ ] CDN / scrubbing provider in front of all public APIs
[ ] Origin firewalled to CDN ranges only (verified from outside) [ ]
L3/L4 volumetric mitigation enabled (provider-managed)
[ ] "Under attack" / challenge mode available and someone can flip it

PERIMETER / LB
[ ] Per-IP connection limits + aggressive read/header timeouts (Slowloris)
[ ] SYN cookies; TLS handshake rate caps; client renegotiation disabled [
] Payload size caps at the edge

GATEWAY (L7)
[ ] Layered rate limits: per-IP, per-client, global ceiling
[ ] Expensive endpoints rate-limited harder (login/search/reports)
[ ] Spike control for burst smoothing
[ ] Caching on hot public read endpoints (cache key ignores unknown params) [ ]
WAF rate-based rules + bot challenge for suspicious clients
[ ] Policy order: reject cheap & early (IP -> authn -> rate -> validate -> cache)

RUNTIME / DATA
[ ] Circuit breakers with aggressive timeouts on every downstream
[ ] Bulkheads isolating per-dependency resources
[ ] DB connection-pool caps + query timeouts; read replicas / cache in front
[ ] Autoscaling with awareness of the non-scalable data tier
[ ] Graceful load shedding (fast 503 + Retry-After) over infinite queueing

DETECTION / RESPONSE
[ ] Baseline metrics + alerts on volume+latency+error correlation
[ ] Layer-aware dashboards (edge vs gateway vs data)
[ ] Written, rehearsed runbook with named owners
```

4.8 Provider-specific notes

Provider	Volumetric	L7 / WAF	Origin protection
Cloudflare	Automatic; "Under Attack" mode	Managed rules, bot fight	Authenticated origin pulls, IP allowlist
AWS Shield Advanced	+ CloudFront / ALB	AWS WAF integration	Security groups on origin
Azure Front Door	DDoS Protection plan	WAF policy on AFD	Private origin in VNet
Akamai	Prolexic scrubbing	Kona Site Defender	Origin lockdown

Regardless of vendor, verify: **origin IP not discoverable**, **rate limits at gateway** as second line, **runbook names who flips what**.

4.9 Cost and capacity planning under attack

DDoS defense has a **cost dimension**:

- CDN/scrubbing bandwidth charges
- Autoscaling compute during L7 flood
- SIEM ingest spike from WAF logs

Budget for **baseline + 3x traffic** headroom. Negotiate DDoS response SLA with provider in enterprise contract. Document finance contact for SEV-1 attacks that may require emergency capacity purchase.

Chapter 4 takeaways

- DDoS is defended **in layers**, each at the place the attack does damage. **Volumetric (L3/L4) must be stopped upstream** by a CDN/scrubber — you can't absorb it yourself.
- **Lock your origin to the CDN**, or the protection is bypassable.
- **L7 application floods** are your job: layered **rate limiting**, **spike control**, **caching as a flood absorber**, **WAF/bot challenges**, and **circuit breakers + bulkheads** to keep what gets through from cascading.
- Tighten limits on **expensive endpoints**, make rejections **cheap and early**, and protect the **data tier** explicitly — it's the real bottleneck.
- **Detection + a rehearsed runbook** turn a crisis into a non-event. Policy-driven front ends let you re-tune in seconds, no redeploy.

Next: [Chapter 5 — MuleSoft Security & FIPS 140-2](05-mulesoft-fips.md)

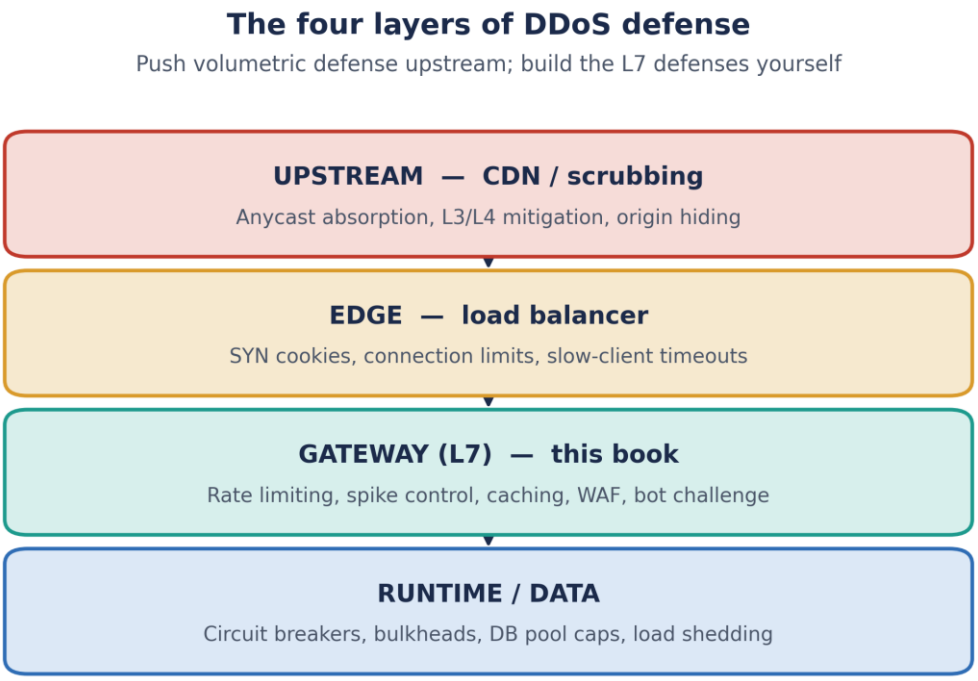


Figure 4-1. The four layers of DDoS defense.

Chapter 5
MuleSoft Security & FIPS 140-2

" This is the hands-on chapter. Everything before it was about what to defend and where to put the controls.
This chapter is about how to configure them on the platform I know best — MuleSoft Anypoint Platform — including how to run Mule in FIPS 140-2 mode, which is the requirement that shows up in every government and regulated - industry RFP I've ever seen.

I'll walk through the platform's security model first, then the policies we keep referencing, then the FIPS configuration step by step. Every screen is described so you can find it in your own console.

Platform versions referenced here: Mule Runtime **4.4.x** and Anypoint Platform. Treat the specifics as version-bound: exact console screens, policy names, default cipher lists, and the certified cryptographic-provider version all vary by runtime release and by deployment model (CloudHub, Runtime Fabric, or on-premises). If you're on a different 4.x minor, the screens and property names are the same in spirit — but always check your release notes for the exact certified provider version your build ships with, and validate the policy names against your own API Manager before quoting them in a design document.

5.1 The Anypoint security model — the pieces and who owns them

Before touching a policy, understand the platform's security architecture. It has four layers, and confusing them is how you end up with a "secure" API that anyone with a browser can reach.

Access Management is the front door to the platform. In regulated environments this is non-negotiable:

- **SSO via SAML 2.0** federated to your corporate IdP (Okta, Azure AD, Ping, etc.). No local Anypoint passwords floating around.
- **MFA enforced** at the IdP for anyone with platform access.

• **Least-privilege roles.** "Organization Administrator" is not a default for developers. Typical split: *API Owners* manage their APIs, *Deployers* push to non-prod, a small *Platform Admin* group owns prod and Access Management.

"**Figure 5.1 — Screenshot:** Access Management Users shows the user list with columns for Name, Email, Identity Provider (e.g. "Corporate SAML"), and Assigned Roles. The Invite users button is top-right. Every user should show your corporate IdP, not "Anypoint Credentials."

"**Figure 5.2 —** Screenshot: Access Management Roles lists built-in roles (Organization Administrator, API Manager Environment Administrator, Deployer, Viewer, …) and any custom roles you've created. Click a role to see its granular permissions (e.g. "Create APIs in this environment," "Apply policies," "Deploy applications").

Environment isolation is simpler than people make it. Create separate environments (`dev`, `test`, `prod`), bind each to its own set of runtimes/workers, and **never** share secrets across them. A `dev` client credential must not work against `prod`. API Manager policies are per-environment — your prod rate limits and JWT validation don't accidentally inherit from a dev sandbox where someone turned security off to debug.

"**Figure 5.3 — Screenshot:** The Environment dropdown in the top navigation bar of Anypoint Platform, expanded to show Sandbox, Development, Test, Production. The currently selected environment is highlighted. Every API Manager and Runtime Manager screen is scoped to whichever environment is selected here — switching environments is switching worlds.

5.2 Client applications and credentials — the API consumer side

When an external system calls your API through the managed front end, it does so as a **client application** registered in API Manager, with credentials (client ID / client secret) and optionally an **SLA tier** that governs its rate limit.

"**Figure 5.4 —** Screenshot: API Manager Client Applications lists registered consumers with Name, Client ID (partially masked), Contact email, and Date created. The Create application button is top-right.

"**Figure 5.5 — Screenshot:** Creating a client application: fields for Application name, Description, Contact email, and an APIs section where you add which APIs this client may call and at what SLA tier (e.g. Bronze / Silver / Gold). The SLA tier ties directly to the Rate Limiting - SLA-based policy from Chapter 3.

A client application record looks like:

```
Application name:  PartnerPortal-Prod
Client ID:         a1b2c3d4-e5f6-7890-abcd-ef1234567890
Client Secret:     ••••••••••••••••  (shown once at creation — store in a vault)
SLA tier:          Gold  →  500 requests/minute
Authorized APIs:   customer-experience-api v1.0
```

Security practices for client credentials:

- Issue credentials per application, per environment. Never share a prod credential across five different partner systems.
- Rotate secrets on a schedule and on any suspected compromise. API Manager supports regenerating a client secret without changing the client ID.
- Pair client credentials with **JWT validation** (§5.4) so the credential alone isn't enough — the caller must also present a valid token from your IdP.

5.3 Runtime security — TLS, keystores, and secure properties

The Mule worker itself must be hardened. Three areas matter most.

TLS configuration

Every inbound and outbound HTTPS connection needs a TLS context with modern settings. For FIPS and non-FIPS alike, that means **TLS 1.2 minimum** (1.3 preferred where supported), strong cipher suites, and valid certificates.

In a Mule 4 flow, TLS is configured via a `tls:context` and referenced by your HTTP listener or requester:

```
<tls:context name="Inbound_TLS_Context">
  <tls:key-store type="jks"
                 path="keystores/api-gateway.jks"
                 keyPassword="${secure::tls.keyPassword}"
                 password="${secure::tls.storePassword}"/>
 <tls:trust-store path="keystores/truststore.jks"
                  password="${secure::tls.trustPassword}"
                  type="jks"/>
</tls:context>

<http:listener-config name="HTTPS_Listener">
  <http:listener-connection protocol="HTTPS"
                            host="0.0.0.0"
                            port="8081"
                            tlsContext="Inbound_TLS_Context"/>
</http:listener-config>
```

For outbound calls to downstream systems, enable **mTLS** (mutual TLS) inside your network — the runtime presents a client certificate and the downstream verifies it. This is Boundary 3 in the Chapter 1 diagram.

"**Figure 5.6 — Screenshot:** In Runtime Manager (select application) Settings Properties, the Properties tab shows key-value pairs. Sensitive values like tls.keyPassword are entered here for CloudHub, or supplied via your secrets manager for on-prem/hybrid. The Secure Properties toggle (a lock icon next to the value field) marks a property as encrypted at rest in the platform.

Secure properties — stop checking secrets into Git

This one is embarrassingly common. Teams hard-code database passwords, API keys, and keystore passwords in `config.yaml` and commit them. Mule's **Secure Properties** module encrypts values at build time so only the runtime with the right decryption key can read them.

Step 1 — generate an encryption key (once per environment, store the key in your vault):

```
# Mule 4.4 secure-properties-tool, packaged with the runtime
java -cp secure-properties-tool.jar \
  com.mulesoft.tools.SecurePropertiesTool \
  string encrypt AES CBC <your-encryption-key> "mySecretPassword123"
# Output: ![rOXRkV...encrypted blob...]
```

Step 2 — put the encrypted value in your config:

```
# config-prod.yaml
db:
  password: "![rOXRkV...encrypted blob...]"

tls:
  keyPassword: "![aBcDeF...encrypted blob...]"
  storePassword: "![gHiJkL...encrypted blob...]"
```

Step 3 — reference with the `secure::` prefix in your Mule config:

```
<db:config name="Database_Config">
  <db:generic-connection url="jdbc:postgresql://db.internal:5432/accounts"
                         user="mule_svc"
                         password="${secure::db.password}"/>
</db:config>
```

Step 4 — supply the decryption key at runtime (never in the artifact):

```
# On-prem / standalone: in conf/wrapper.conf or as a JVM arg
wrapper.java.additional.N=-Dmule.key=<your-encryption-key>

# CloudHub: Runtime Manager → Settings → Properties
#   Key: mule.key    Value: <your-encryption-key>   (mark as Secure Property)
```

"Gotcha. The encryption key (mule.key) and the encrypted values travel separately. The artifact in your artifact repository contains only ciphertext. The key lives in Runtime Manager or your deployment pipeline's secret store. If someone steals the Git repo, they get encrypted blobs they can't use without the key. This is table stakes do it on every project.

Logging hygiene

Mule's default logging is verbose. Before prod, audit your log statements and DataWeave transforms to ensure you're not printing:

- Full request/response bodies containing PII or cardholder data
- `Authorization` headers or tokens
- Secure property values (Mule masks `${secure::...}` in logs by default — good — but custom log statements can still leak)

Set log levels to `INFO` in prod; `DEBUG` only on demand, time-boxed, with log scrubbing.

5.4 API Manager policies — the security baseline

This is where the Chapter 3 patterns become concrete MuleSoft configuration. I'll walk through the four policies that form the **security baseline** I attach to every external-facing API, in the order from §3.5.

Policy 1 — IP Allowlist (optional, for known consumers)

If your API is consumed only by known partners with fixed egress IPs, an allowlist is the cheapest rejection you'll ever get.

"**Figure 5.7 — Screenshot:** In the policy gallery under Security, select IP Allowlist. The config pane has a text area for IP addresses (one per line, CIDR supported, e.g. 203.0.113.0/24) and a toggle Include X-Forwarded-For (enable this if the gateway sits behind a load balancer/CDN and the real client IP arrives in that header).

```
policy: ip-allowlist
configuration:
  ips:
    - "203.0.113.10"
    - "203.0.113.0/24"
    - "198.51.100.50"
  includeForwardedFor: true
```

Policy 2 — JWT Validation

The workhorse authentication policy. Validates the bearer token before the request reaches your API implementation.

> *"**Figure 5.8** — Screenshot: Select **JWT Validation** from the Security policy gallery. The config pane shows:*

> *" - **JWT Origin** — dropdown: `httpBearerAuthenticationHeader` (reads `Authorization: Bearer <token>`)*

> *" - **Signing Method** — dropdown: `rsa`, `hmac`, or `jwks` (choose `jwks` for tokens from a modern IdP)*

> *" - **JWKS URL** — text field, e.g. `https://login.yourorg.com/.well-known/jwks.json`*

> *" - **Audiences** — text field, your API's expected `aud` claim, e.g. `https://api.yourorg.com`*

> *" - **Claims Validation** — expandable section for required claims (`exp`, `iss`, custom scopes)*

```
policy: jwt-validation
configuration:
 jwtOrigin: httpBearerAuthenticationHeader
 signingMethod: jwks
 jwksUrl: "https://login.yourorg.com/.well-known/jwks.json"
audiences:
  - "https://api.yourorg.com"
 claimsValidation:
  - claim: "exp"
      required: true
  - claim: "iss"
      required: true
      value: "https://login.yourorg.com"
    - claim: "scope"
      required: true
      regex: ".*read:accounts.*"
```

This policy rejects, with `401 Unauthorized`, any request whose token is missing, expired, wrongly signed, issued by the wrong IdP, or missing required scopes — *before* your flow executes a single connector call.

Policy 3 — Rate Limiting (SLA-based)

Tie the limit to the client's registered SLA tier (§5.2).

> *"**Figure 5.9** — Screenshot: Select **Rate Limiting - SLA-based** under Quality of Service. The config shows SLA tiers you've defined for this API (e.g. **Bronze: 50/min**, **Silver: 200/min**, **Gold: 500/min**) and the **Expose Headers** checkbox.*

```
policy: rate-limiting-sla-based
configuration:
  exposeHeaders: true
  clusterizable: true
```

The per-tier numbers are set when you define SLA tiers on the API itself (**API Manager (API) SLA Tiers**).

Policy 4 — Message Logging (for audit, not for secrets)

Log request metadata for security monitoring — correlation ID, client ID, path, response code, latency — without logging bodies or tokens.

"Figure 5.10 — Screenshot: Message Logging policy under Compliance. Config fields: Conditional (DataWeave expression to filter which requests to log), Message (what to log — use metadata only), Category (log category name).

```
policy: message-logging
configuration:
  conditional: "#[attributes.method != 'OPTIONS']"
  message: >-
    clientId=#[attributes.headers['x-client-id'] default 'unknown']
    method=#[attributes.method]
    path=#[attributes.requestPath]
    status=#[attributes.statusCode]
    durationMs=#[vars.requestDuration]
```

```
  category: "api.audit"
```

Applying the baseline with Automated Policies

Rather than attaching these four policies to every API by hand (and inevitably forgetting one), use **Automated Policies** to enforce the baseline platform-wide.

"Figure 5.11 — Screenshot: API Manager Automated Policies Create automated policy. Select JWT Validation, configure it once with your org's JWKS URL and audience, set the scope to All APIs in Production environment, and save. Every API deployed to Production from this point inherits JWT validation automatically. Repeat for Rate Limiting and Message Logging.

The resulting policy order on any managed API should read:

```
1. IP Allowlist          (if applicable)
2. JWT Validation
3. Rate Limiting (SLA)
4. Message Logging
5. HTTP Caching          (if applicable — see Chapter 3)
```

5.5 FIPS 140-2 — what it is and why enterprises require it

FIPS 140-2 (*Federal Information Processing Standard 140-2*, "Security Requirements for Cryptographic Modules") is the U.S. government standard that specifies what a cryptographic module must do to be considered secure enough for federal use. It is the standard that:

- U.S. federal agencies require (under FISMA and related mandates).
- Most U.S. state governments, defense contractors, and financial institutions reference in their procurement and audit requirements.
- MuleSoft documents and certifies against for its FIPS-enabled runtime offering.

A "FIPS 140-2 validated" module is one that has been tested by an accredited lab (a CMVP-listed lab) and listed on the NIST CMVP validation list. You don't get to *claim* FIPS compliance by using AES — you must use a **validated module** running in **FIPS mode**.

"A note on 140-2 versus 140-3. A successor standard, FIPS 140-3, has been finalized and the CMVP has begun validating modules against it; over time 140-3 validations will supersede 140-2 ones. For most enterprise integration work today, FIPS 140-2 is still the requirement you will see in RFPs, procurement language, and platform documentation — but treat the version number as a moving target. Confirm with your MuleSoft account team and your auditor which standard your build's cryptographic provider is validated against, and design so that swapping to a 140-3-validated provider is a configuration change, not a re-architecture.

What changes when you enable FIPS mode on Mule:

- The JVM's JCE (Java Cryptography Extension) provider is replaced with a **FIPS-validated provider** (in Mule 4.4, this is the Bouncy Castle FIPS-certified provider, `bc-fips` / `bctls-fips`).
- Only **FIPS-approved algorithms** are available. Non-approved algorithms (MD5, DES, RC4, non-approved EC curves, etc.) are **rejected at runtime** with an error.
- TLS cipher suites are restricted to the FIPS-allowed set (AES-GCM, AES-CBC with approved key sizes, approved key exchange).
- Random number generation uses a FIPS-approved DRBG.

What does **not** change: your flows, your connectors, your policies. FIPS is a runtime crypto substrate. Your API still does the same things; it just can't accidentally use a weak algorithm.

"Gotcha — FIPS is not a policy toggle. You can't flip a switch in API Manager and be FIPS-compliant. It requires a FIPS-enabled Mule runtime build, a FIPS JCE provider installed, the mule.security.model property set, and — critically — every cryptographic operation in your application and its dependencies must use FIPS-approved algorithms. A single connector or custom Java class calling MessageDigest.getInstance("MD5") will blow up in FIPS mode. Audit your dependencies.

5.6 Enabling FIPS 140-2 on Mule Runtime — step by step

I'll cover both deployment models: **on-premises / hybrid** (you manage the JVM) and **CloudHub** (MuleSoft manages the worker, with a FIPS-capable offering).

Prerequisites

- Mule Runtime **4.4.x** (or the specific FIPS-certified build your MuleSoft account team provides). Confirm with your account rep or the release notes that your build includes the FIPS provider packaging.
- FIPS-validated JCE provider JARs on the runtime classpath (typically bundled with the FIPS-certified distribution; if not, your account team supplies them).
- All TLS configurations in your apps must use **FIPS-approved cipher suites** (AES-128-GCM, AES-256-GCM, AES-128-CBC, AES-256-CBC with TLS 1.2+).
- All certificate key sizes must meet FIPS minimums (RSA ≥ 2048 bits, EC ≥ P-256).

Step 1 — Verify you have the FIPS runtime distribution

"**Figure 5.12 — Screenshot:** In Anypoint Platform Runtime Manager Server Groups / Servers (for on-prem/hybrid) or the CloudHub deployment screen, check the Mule Version field. A FIPS-certified deployment will show a build tagged for FIPS (your account team or release notes will identify the exact build number). If you're on a standard build, request the FIPS distribution from MuleSoft before proceeding.

Step 2 — Install the FIPS JCE provider (on-prem / hybrid)

On a standard Mule standalone installation, the FIPS provider JARs go into the runtime's `lib` directory:

```
<mule-home>/
  lib/
    bc-fips-<version>.jar          # Bouncy Castle FIPS core provider
    bctls-fips-<version>.jar     # Bouncy Castle FIPS TLS provider
  conf/
    wrapper.conf                   # where we'll set the FIPS property
```

"**Figure 5.13 — Screenshot:** A file-browser view of <mule-home>/lib/ showing the two FIPS JARs alongside the standard Mule JARs. If these JARs are absent, you're not on a FIPS distribution — stop and get the right build.

Step 3 — Set the FIPS security model property

This is the switch. Add the following JVM argument to `wrapper.conf`:

```
# In <mule-home>/conf/wrapper.conf
# Find the last wrapper.java.additional.N line and add the next number:

wrapper.java.additional.20=-Dmule.security.model=fips140-2
```

The property `mule.security.model=fips140-2` tells the Mule runtime to:

- Register the FIPS JCE provider as the sole crypto provider.
- Reject any cryptographic operation that isn't FIPS-approved.
- Restrict TLS handshakes to FIPS-approved cipher suites.

"**Figure 5.14 — Screenshot:** A text editor showing wrapper.conf with multiple wrapper.java.additional.N lines. The last one reads wrapper.java.additional.20=-Dmule.security.model=fips140-2. This is the single most important line in the whole FIPS setup.

Step 4 — Configure the FIPS provider registration (if required by your build)

Some Mule FIPS distributions require an explicit `java.security` override to register the Bouncy Castle FIPS provider. Check your distribution's README; if needed, add a custom security properties file:

```
# custom-java.security (path referenced via wrapper.java.additional)
security.provider.1=org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
security.provider.2=org.bouncycastle.jsse.provider.BouncyCastleJsseProvider
# Remove or deprioritize the default SunJCE provider
```

And reference it:

```
wrapper.java.additional.21=-Djava.security.properties==/path/to/custom-java.security
```

(The `==` syntax *replaces* the default `java.security` rather than merging — which is what you want in FIPS mode, so non-FIPS providers can't sneak in.)

Step 5 — Update TLS configurations to FIPS-approved cipher suites

In your Mule TLS contexts and any explicit cipher-suite configuration, restrict to the approved set. For TLS 1.2 in FIPS mode, the safe list is:

```
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
```

In Mule 4.4 you can constrain cipher suites on the HTTP listener:

```
<http:listener-config name="HTTPS_Listener_FIPS">
  <http:listener-connection protocol="HTTPS"
                host="0.0.0.0"
                port="8081"
                tlsContext="Inbound_TLS_Context">
    <tls:revocation-check enabled="true"/>
  </http:listener-connection>
</http:listener-config>
```

And verify your keystore certificates use approved key sizes:

```
# Check a certificate's key size
keytool -list -v -keystore api-gateway.jks -storepass <password> \
  | grep "Key size"
# Must show >= 2048 for RSA, or P-256/P-384 for EC
```

Step 6 — CloudHub FIPS configuration

On CloudHub, MuleSoft offers FIPS-capable worker images for regulated customers. The configuration differs because you don't edit `wrapper.conf` directly:

- **Request FIPS workers** from your MuleSoft account team when provisioning the org (this is an account-level setting, not self-service in all regions).
- In **Runtime Manager Deploy Application**, select the **FIPS-certified Mule version** from the version dropdown.
- Set the FIPS property via application properties:
- Deploy. The CloudHub FIPS worker already has the FIPS JCE provider on the classpath; the property activates it.

"**Figure 5.15 —** Screenshot: Runtime Manager (application) Settings Properties tab. A property row shows Key:mule.security.model, Value:fips140-2. The secure-property lock icon is not needed for this value (it's not a secret, it's a mode flag).

Step 7 — Verify FIPS mode is active

After restart/deploy, confirm the runtime is actually in FIPS mode. Check the Mule log on startup:

```
INFO  ... MuleRuntime - Security model: fips140-2
INFO  ... MuleRuntime - Registered security provider: BouncyCastleFipsProvider
```

If you see `Security model: default` instead, the property didn't take — go back to Step 3.

Negative test — prove FIPS is actually enforcing. Deploy a trivial flow that deliberately calls a non-FIPS algorithm:

```
<flow name="fips-negative-test">
  <http:listener path="/fips-test" config-ref="HTTPS_Listener_FIPS"/>
  <java:invoke-static class="java.security.MessageDigest"
                      method="getInstance(String)"
                      target="digest">
    <java:args><![CDATA[#[{ arg0: 'MD5' }]]]></java:args>
  </java:invoke-static>
</flow>
```

In FIPS mode this must fail with an exception like `NoSuchAlgorithmException: MD5 not available`. If it succeeds, FIPS is **not** active — stop and fix before going to prod.

Positive test — call a FIPS-approved endpoint over TLS 1.2 with an approved cipher:

```
# From a client machine, verify the TLS handshake uses a FIPS cipher
openssl s_client -connect your-api.yourorg.com:443 -tls1_2 \
  -cipher 'ECDHE-RSA-AES256-GCM-SHA384' </dev/null 2>/dev/null \
  | openssl x509 -noout -subject -dates
# Should succeed and show your certificate details
```

Step 8 — Audit your application for non-FIPS algorithms

Walk this checklist against every app deploying to the FIPS runtime:

```
[ ] No MD5, SHA-1 (for signatures), DES, 3DES, RC4 anywhere in code or config
[ ] No RSA keys < 2048 bits; no EC curves below P-256
[ ] All TLS contexts use TLS 1.2+ with FIPS-approved cipher suites
[ ] Secure Properties encryption uses AES (it does by default in Mule 4.4)
[ ] Third-party connectors/JARs don't call non-FIPS algorithms internally
    (test each connector in FIPS mode — see negative test above)
[ ] OAuth/JWT signing algorithms are RS256/ES256 (approved), not HS256
    with a weak secret
[ ] Database connection encryption (if JDBC over TLS) uses approved suites
```

"War story. A government contractor enabled FIPS on their Mule workers and every deployment started failing on startup. Root cause: a legacy JMS connector's internal library called MessageDigest.getInstance("SHA-1") during initialization. SHA-1 is fine for checksums in normal mode but isn't approved for digital signatures in FIPS mode, and the BC FIPS provider rejected it. The fix was upgrading the connector to a version that used SHA-256. The lesson: test every application in FIPS mode in a lower environment before flipping prod. Don't assume "we don't use MD5 anywhere" — your dependencies might.

5.7 Any point Platform hardening checklist

Beyond FIPS, this is the full platform hardening list I run on every enterprise engagement:

```
ACCESS MANAGEMENT
[ ] SSO/SAML federated to corporate IdP; no local Anypoint passwords
[ ] MFA enforced at the IdP for all platform users
[ ] Least-privilege roles; "Organization Administrator" limited to 2-3 people
[ ] Service accounts for CI/CD pipelines, not personal accounts

ENVIRONMENTS
[ ] Separate dev / test / prod environments with isolated runtimes [ ]
No shared secrets across environments
[ ] Prod deployments require approval (change-management integration)

API MANAGER
[ ] Automated Policies: JWT Validation on all prod APIs
[ ] Automated Policies: Rate Limiting (SLA-based) on all prod APIs [ ]
Automated Policies: Message Logging (audit metadata) on all prod APIs [ ]
```

```
IP Allowlist on APIs with known, fixed consumers
[ ] API catalog complete; no shadow/zombie APIs (see Chapter 2, API9)

RUNTIME
[ ] TLS 1.2+ on all inbound and outbound connections
[ ] mTLS for service-to-service inside the network
[ ] Secure Properties for all secrets; mule.key in vault, not in Git
[ ] Log levels INFO in prod; no bodies/tokens in logs
[ ] FIPS 140-2 mode enabled where required (§5.6)
[ ] FIPS negative test passed for every application

MONITORING
[ ] Anypoint Monitoring dashboards per API (volume, latency, errors, 429s)
[ ] Alerts on error-rate and latency spikes (see Chapter 4 runbook)
[ ] Audit logs retained per organizational policy
```

5.12 Anypoint MQ and object store security

Anypoint MQ (managed messaging) carries sensitive payloads between flows.

```
[ ] TLS for client connections
[ ] Queue-level access via client app credentials
[ ] Encrypt message payloads containing PII before publish (application-level)
[ ] Dead-letter queues monitored — don't accumulate secrets in DLQ messages
[ ] Message TTL set — don't retain sensitive data indefinitely
```

Object Store v2 (used by HTTP Caching, idempotency):

```
[ ] Encrypt sensitive cached data only if unavoidable — prefer not caching PII
[ ] TTL aligned with data sensitivity
[ ] Cluster-scoped stores for prod; no cross-environment store sharing
```

5.13 Runtime Manager and deployment pipeline security

```
[ ] Deploy permissions limited to CI service account + named deployers
[ ] Prod deploy requires manual approval in pipeline
[ ] Application artifacts from trusted Nexus/Exchange only
[ ] No manual upload of unsigned JAR to prod workers
[ ] Properties diff reviewed on deploy (catch accidental prod secret in dev property)
```

"Figure 5.16 — Screenshot: Runtime Manager Deployments Deployment history showing version, deployer identity (service account), timestamp. Auditors sample this for change control evidence.

5.14 Exchange and API catalog governance

Anypoint Exchange is your API asset registry:

- Publish RAML/OAS with security schemes documented (`oauth_2_0`, scopes)
- Tag APIs: `PCI`, `PII`, `Public`, `Internal`
- Link to API Manager instance and SLA tiers
- Version deprecation status visible

Shadow APIs won't appear in Exchange — run **automated discovery** (scan your gateway routes vs Exchange inventory) monthly.

5.15 FIPS troubleshooting guide

Symptoms	Likely cause	Fix
NoSuchAlgorithmException: MD5	Dependency uses MD5	Upgrade connector; replace hash
SSLHandshakeException	Non-FIPS cipher offered	Restrict cipher suite list
InvalidKeyException RSA 1024	Weak key in keystore	Reissue 2048+ cert
Startup works, FIPS not active	Property typo	Verify mule.security.model=fips140-2 in logs
Intermittent JWT fail after IdP rotation	JWKS cache stale	Lower cache TTL; force refresh

Keep a **FIPS smoke test** in CI that deploys to FIPS worker and runs negative MD5 test on every release.

Chapter 5 takeaways

- Anypoint security is four layers: **Access Management**, **environment isolation**, **API Manager policies**, and **runtime hardening**. Don't confuse them.
- **Client applications** get per-app credentials and SLA tiers; pair with JWT so the credential alone isn't enough.
- **Secure Properties** keep secrets out of Git; the decryption key travels separately via `mule.key`.
- The **security baseline** is four automated policies: IP Allowlist (if applicable) JWT Validation Rate Limiting (SLA) Message Logging.
- **FIPS 140-2** replaces the JVM crypto provider and rejects non-approved algorithms. Enable with `-Dmule.security.model=fips140-2`, use FIPS-approved TLS ciphers and key sizes, and **test every app** with a negative test before prod.

- Run the **hardening checklist** (§5.7) on every engagement.

Next: [Chapter 6 — Reference Architecture & Secure Delivery Playbook](06-reference-architecture.md)

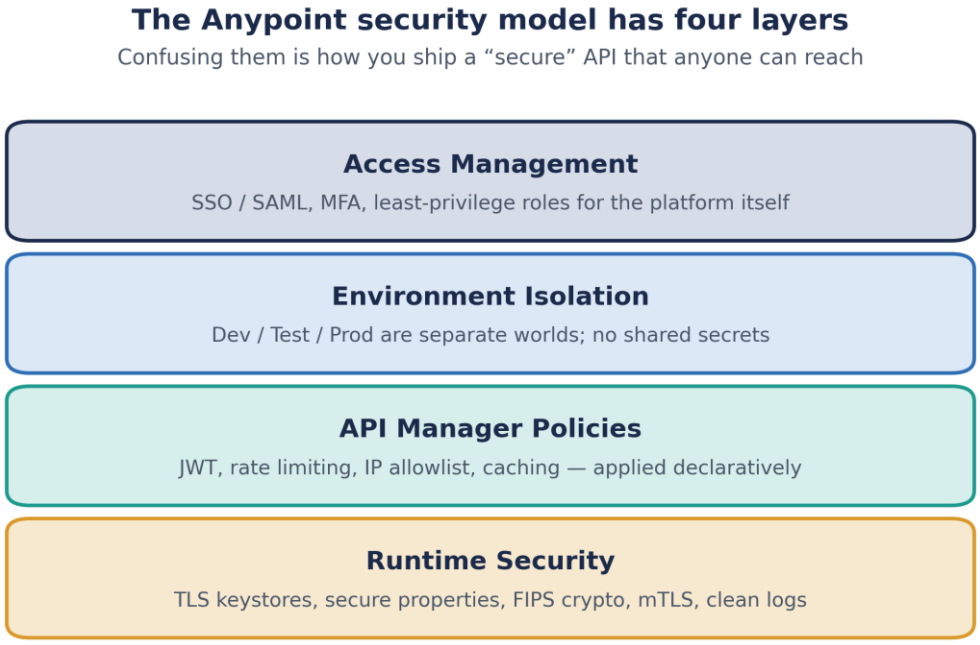


Figure 5-1. The Anypoint Platform security model.

Chapter 6
Reference Architecture & Secure Delivery Playbook

"We've walked the threats, the front-end patterns, the DDoS layers, and the MuleSoft/FIPS configuration. This final chapter assembles all of it into one reference architecture you can take into a design review, a threat-modeling or a Monday-morning production cutover. Then a delivery playbook — the and the sequence — so the architecture doesn't stay a diagram.

6.1 The reference architecture — one picture

This is the architecture I draw when a new enterprise API program asks "what does secure look like?" It's built from every pattern in this book, mapped to the trust boundaries from Chapter 1.

Read it top to bottom as a request's journey. Every horizontal band is a layer that adds controls. Every vertical `[ BOUNDARY ]` is a line where assumptions change.

6.2 Component-by-component — what each box does for security

CDN / DDoS edge (Boundary 0 → 1)

- Terminates TLS 1.2/1.3 from the internet.
- Absorbs L3/L4 volumetric DDoS (Chapter 4, §4.2).
- Hides origin IP; firewall origin to CDN ranges only.
- Optional WAF managed rules and bot challenges for coarse L7 filtering.
- **Does not** do per-user authorization — that's downstream.

API Manager policy front end (Boundary 1 → 2)

The managed gateway. Every external request passes through the policy stack from Chapter 3 and Chapter 5:

Orders	Policy	Security property	Chapter
1	IP Allowlist	Reject unknown sources cheaply	5.4
2	JWT Validation	Authentication (who?)	5.4
3	Rate Limiting (SLA)	Availability / quota (OWASP API4)	3.1
4	Spike Control	Burst smoothing	3.1
5	Schema / Threat Validation	Integrity (reject malformed)	2.4, 2.6
6	HTTP Caching	Availability (shield back end)	3.2
7	Message Logging	Audit / detect (OWASP API10)	5.4

Automated Policies (§5.4) enforce items 2–4 and 7 on every prod API by default.

Experience APIs (Boundary 2)

- Tailored to each consumer channel (mobile, partner, public).
- **Coarse authorization** — "does this token's scope permit this resource class?"• Response shaping — return only the fields the contract promises (OWASP API3 defense).
- **No business-object authorization here** — that's deeper.

Process APIs (Boundary 3)

- Orchestrate across system APIs.
- **Fine-grained authorization** — BOLA checks live here (Chapter 2, §2.3): "does this user own this account?"
- **Circuit breakers + bulkheads + timeouts** on every downstream call (Chapter 3,§3.3–3.4).
- mTLS to system APIs.

System APIs (Boundary 3 → 4)

- Unlock data from systems of record.
- Parameterized queries only — no injection (Chapter 2, §2.4).
- Explicit input/output DTOs — no mass assignment, no excessive data exposure (Chapter 2, §2.6).

- mTLS or encrypted JDBC to the data tier.
- FIPS-validated TLS where required (Chapter 5, §5.6).

Systems of record (Boundary 4)

- The crown jewels. Network-segmented; not internet-reachable.
- Accept connections only from known system API runtimes (IP + mTLS).• Encrypted at rest; access logged.

6.3 Threat-model walkthrough — "View Statement" end to end

Let's trace a single user action through this architecture and show where each threat from Chapter 2 is stopped. Alice opens her banking app and taps **View Statement**.

The happy path

```
1. Mobile app obtains OAuth2 access token (JWT) from corporate IdP via authorization code + PKCE.
2. App calls:  GET /accounts/100245/statements
               Authorization: Bearer <JWT>
               X-Client-Id: <registered client app ID>

3. CDN: TLS 1.2 terminated. Traffic is clean (no volumetric attack active).
Request forwarded to origin (API gateway).

4. API Manager policies execute in order:
 a. IP Allowlist: N/A (mobile users have dynamic IPs) — skipped.
   b. JWT Validation: signature verified against JWKS. iss, aud, exp checked.
      Scope includes "read:statements".  Authenticated.
   c. Rate Limiting: Alice's client app is Gold tier (500/min). She's at 12/min.
   d. Spike Control: no burst. Pass through.
   e. Schema Validation: GET with path param — no body to validate. Path param is numeric.
   f. HTTP Caching: statements are user-specific → Cache-Control: private. Cache MISS.
   (If this were a public exchange-rate endpoint, a cache HIT would skip steps g onward.) g.
  Message Logging: metadata logged (client, path, timestamp). No body, no token.

5. Request routed to Customer Experience API (experience layer).
Coarse check: token scope "read:statements" covers this resource.

6. Experience API calls Account Process API over mTLS.
   Process API performs BOLA check:
     account 100245 → owner = Alice (token.sub).  Authorized.
  Process API calls Core Banking System API over mTLS.
     Circuit breaker: CLOSED (downstream healthy). Timeout: 2s.

7. System API executes parameterized query:
     SELECT * FROM statements WHERE account_id = ?  [100245]
 Returns DTO with only { date, description, amount } — no internal fields.

8. Response flows back up. Experience API shapes final JSON.
  Cache-Control: private, no-store set on response.
   200 OK returned to Alice in ~180ms.
```

Now the attacks — where they're stopped

Attack	Where it's stopped	How
Stolen token, wrong account (BOLA)— attacker uses Alice's token to request account 100246	Process API (step 6)	account.owner != token.sub→403 Forbidden
Forged token (broken auth) — attacker crafts a JWT with alg: none	API Manager JWT policy (step 4b)	Signature verification fails →401
Token for wrong audience — valid token but minted for a different app	API Manager JWT policy (step 4b)	aud mismatch →401
Credential stuffing on login — 10,000 password guesses/sec	CDN bot challenge + per-IP rate limit on /oauth/token	Challenge blocks scripts; rate limit caps at 5/min per IP
DDoS — volumetric — 200 Gbps UDP flood	CDN (step 3)	Scrubbed upstream; origin never sees it
DDoS — L7 flood on /statements	Rate limiting (4c) + caching for public endpoints (4f) + circuit breaker if DB saturates (step 6)	429s shed load; cache absorbs identical GETs; breaker fails fast
SQL injection in a search param	Schema validation (4e) + parameterized queries (step 7)	Bad input rejected at gateway; prepared statements prevent execution
Excessive data exposure — response includes SSN	System API DTO (step 7)	Only contracted fields returned; SSN never leaves the DB layer
Mass assignment — attacker sends {"role":"admin"} on profile update	Schema validation (4e) + explicit field allow-list in process API	Unknown fields rejected
Shadow API — attacker finds old /v1/statements without JWT policy	API catalog + automated policies	Old version is decommissioned; all active APIs inherit JWT via automated policy
Secret in logs — token appears in application log	Message Logging policy (4g) + log hygiene (§5.3)	Policy logs metadata only; app log levels audited

Every row maps to a control we configured in Chapters 3–5. That's the point of defense in depth — no single row is the *only* defense, but together they cover the OWASP API Top 10.

6.4 The secure delivery playbook

An architecture diagram is worth nothing if it doesn't survive contact with a delivery team. This is the sequence I run — the order matters because later steps depend on earlier ones.

Phase 1 — Platform foundation (week 1–2)

```
[ ] Provision Anypoint org with SSO/SAML to corporate IdP
[ ] Enforce MFA at the IdP
[ ] Create environments: Sandbox, Development, Test, Production
[ ] Define roles: Platform Admin (2-3 people), API Owners, Deployers, Viewers
[ ] Contract/provision CDN with DDoS protection in front of public endpoints
[ ] Firewall origin to CDN IP ranges; verify with external curl
[ ] Request FIPS-certified Mule runtime if required (§5.6)
[ ] Generate per-environment Secure Properties encryption keys; store in vault
```

Exit criteria: A developer can log in via SSO, see only their environment, and the CDN sits in front of a health-check endpoint with origin locked down.

"Figure 6.1 — Screenshot: API Manager Automated Policies showing three active automated policies: JWT Validation (Production), Rate Limiting SLA (Production), Message Logging (Production). The Scope column reads "All APIs in Production." This is the screen you show in the design review to prove the baseline is enforced.

Phase 2 — Security baseline policies (week 2–3)

```
[ ] In API Manager Production: create Automated Policy → JWT Validation
      (org JWKS URL, audience, required claims)
[ ] Create Automated Policy → Rate Limiting (SLA-based)
[ ] Create Automated Policy → Message Logging (metadata only)
[ ] Define SLA tiers on the API template: Bronze / Silver / Gold with limits
[ ] Document the policy order (§3.5) and publish to the API team wiki
[ ] Create a "hello world" API, deploy to Production, verify all three
      automated policies appear on it without manual attachment
```

Exit criteria: Deploy a blank API to Production and it automatically has JWT + rate limiting + audit logging. No human can forget.

Phase 3 — First experience API (week 3–5)

```
[ ] Design API contract (RAML/OAS) with explicit request/response schemas
[ ] Register API in API Manager; bind to Production environment
[ ] Implement experience API with:
      - Coarse scope check
      - Response DTO (no excessive data exposure)
      - mTLS to process layer
[ ] Register client applications for each consumer; assign SLA tiers
[ ] Configure IP Allowlist if consumer has fixed egress
[ ] Add HTTP Caching policy on any public, cacheable endpoints
```

```
[ ] Negative test: call without token → 401; call with wrong aud → 401;
    call over rate limit → 429 with Retry-After
[ ] Positive test: legitimate flow end-to-end with correlation ID in logs
```

Exit criteria: One production experience API serving real traffic, passing negative security tests, with audit logs flowing.

Phase 4 — Process and system APIs (week 5–8)

```
[ ] Implement process APIs with BOLA checks on every object-level endpoint
[ ] Implement system APIs with parameterized queries and output DTOs
[ ] Add circuit breakers + timeouts + bulkheads on every downstream call
[ ] Enable mTLS between all layers
[ ] Encrypt all secrets with Secure Properties
[ ] If FIPS: deploy to FIPS runtime; run negative test (§5.6 Step 7);
    audit all connectors for non-FIPS algorithms
[ ] Load test to establish p99 latency and set rate limits at 2-3x (§3.1)
```

Exit criteria: Full three-layer API passes a BOLA test (request another user's object 403), a load test without breaching rate limits, and (if applicable) a FIPS negative test.

Phase 5 — DDoS readiness and monitoring (week 8–9)

```
[ ] Configure Anypoint Monitoring dashboards per API (§4.6)
[ ] Set alerts: request volume + latency + error rate correlation
[ ] Write and rehearse the DDoS runbook (§4.6 war story)
[ ] Verify CDN "under attack" mode can be activated by on-call
[ ] Verify rate limits can be tightened in API Manager without redeploy
[ ] Run a tabletop exercise: "200K req/s on /search" — who does what?
```

Exit criteria: On-call engineer can, in under 5 minutes, tighten rate limits and flip CDN challenge mode using only the runbook.

Phase 6 — Governance and ongoing (continuous)

```
[ ] Maintain API catalog; quarterly review for shadow/zombie APIs (Ch. 2, API9)
[ ] Rotate client secrets on schedule
[ ] Review Automated Policies when IdP/JWKS URLs change
[ ] Annual pen test scoped to the API surface
[ ] Post-incident review template ready (even if you haven't had one yet)
[ ] Onboard new APIs via a standard template that inherits the baseline
```

6.5 The one-page security review checklist

Print this. Bring it to every design review. Every box should be checked or explicitly marked N/A with a reason.

```
── PLATFORM ──────────────────────────────────────────
[ ] SSO/SAML + MFA for Anypoint Platform access
[ ] Least-privilege roles assigned
[ ] Separate environments; no shared prod/dev secrets
[ ] FIPS 140-2 enabled on runtime (if required) with negative test passed

─── EDGE / DDoS
──────────────────────────────────────────────[ ] CDN/DDoS
protection in front of all public APIs
[ ] Origin firewalled to CDN only (verified externally) [ ]
DDoS runbook written and rehearsed

─── API FRONT END (every external API) ──────────────────
[ ] JWT Validation (automated policy) — iss, aud, exp, scope checked
[ ] Rate Limiting (SLA-based, automated policy) — per-client + global ceiling
[ ] Spike Control on burst-prone endpoints
[ ] HTTP Caching on public read endpoints (NOT on personal data)
[ ] Schema validation — request and response contracts enforced
[ ] Message Logging — metadata only, no bodies/tokens
[ ] Policy order correct: reject cheap → authn → rate → validate → cache [ ]
Client applications registered per consumer, per environment

─── API IMPLEMENTATION ──────────────────────────────────
[ ] BOLA checks on every object-level endpoint (process/system layer)
[ ] Parameterized queries — no string-concatenated SQL
[ ] Output DTOs — no excessive data exposure
[ ] Input allow-lists — no mass assignment
[ ] Circuit breakers + aggressive timeouts + bulkheads on downstreams
[ ] mTLS between all internal layers
[ ] Secure Properties for all secrets; mule.key in vault

─── OPERATIONS ──────────────────────────────────────────
[ ] Monitoring dashboards + correlated alerts (volume + latency + errors)
[ ] API catalog current; no shadow/zombie APIs
[ ] Log levels INFO in prod; audit logs retained per policy
[ ] Pen test scheduled; post-incident review template ready
```

6.6 What we didn't cover (honest scope)

This book is deliberately focused. A few things it touches but doesn't fully treat — because each could be its own book:

- **Identity architecture** (full OAuth2/OIDC flows, token lifecycle, refresh rotation, consent). We configured JWT validation; designing the IdP is out of scope.

- **Data classification and tokenization** (PCI scope reduction, field-level encryption in the data tier). We said "mask and tokenize"; implementing it is domain-specific.
- **Container/Kubernetes security** for Mule on RTF or on-prem K8s. The patterns transfer; the K8s hardening (pod security policies, network policies, secrets operators) is a parallel track.
- **Compliance frameworks end-to-end** (SOC 2, PCI DSS audit prep, FedRAMP). We mapped controls to FIPS and OWASP; an audit is a program, not a config change.

If you're building in one of those areas, take the trust-boundary model and the defense-in-depth discipline from Chapter 1 and apply them there too. The thinking is the same even when the tooling changes.

6.7 Closing

Security in enterprise architecture is not a project with an end date. It's a property you maintain — the same way you maintain uptime. The patterns in this book are not new. Rate limiting is decades old. Circuit breakers were named in 2007. FIPS 140-2 was published in 2001. What's changed is the stakes: every enterprise is now an API company, the attack surface is the same surface your customers use, and the integration layer — the place where I started this book — is the choke point where you either absorb the chaos or pass it straight through to systems that were never built to face the internet.

Put the controls at the boundaries. Make them policies, not prayers. Test the negative cases. Write the runbook before you need it.

Build the blast door. Then sleep.

Chapter 6 takeaways

- The **reference architecture** maps every pattern to a trust boundary: CDN policy front end experience process system data.
- The **threat-model walkthrough** shows how BOLA, broken auth, injection, DDoS, and shadow APIs are each stopped at a specific layer.

- The **delivery playbook** is six phases: platform baseline policies first API full layering DDoS readiness governance.
- The **one-page checklist** (§6.5) is your design-review companion.
- Security is a **property you maintain**, not a project you finish.

6.8 Where to go from here — Part II

Part I gave you the whole picture: the mindset, the threats, the front-end patterns, the DDoS playbook, the MuleSoft/FIPS configuration, and the reference architecture that ties them together. For most teams, implementing what's in these six chapters is a year of good, hard work.

Part II goes deeper, one security domain at a time. Each chapter takes a box from the reference architecture in §6.1 and opens it up:

- The IdP box becomes [Chapter 7 — Identity & Access Management](07-identity-access-management.md).
- The "TLS / mTLS everywhere" lines become [Chapter 8 — Transport Security, Encryption & Key Management](08-transport-encryption.md).
- The Secure Properties and `mule.key` discussion becomes [Chapter 9 — Secrets Management & Configuration Security](09-secrets-management.md).
- The trust-boundary discipline becomes [Chapter 10 — Zero Trust Architecture](10-zero-trust.md).
- The schema-validation and WAF controls become [Chapter 11 — API Threat Protection](11-api-threat-protection.md).
- The Message Logging policy and monitoring become [Chapter 12 — Observability, SIEM & Incident Response](12-observability-incident-response.md).
- The "FIPS where required" and audit threads become [Chapter 13 — Compliance & Governance](13-compliance-governance.md).
- The delivery playbook becomes [Chapter 14 — Secure SDLC & DevSecOps](14-secure-sdlc-devsecops.md).
- The internal service-to-service mesh becomes [Chapter 15 — Microservices, Containers & Service Mesh Security](15-microservices-container-security.md).

Read the ones your current project needs. They assume you've absorbed Part I, but otherwise stand alone.

Next: [Chapter 7 — Identity & Access Management](07-identity-access-management.md)

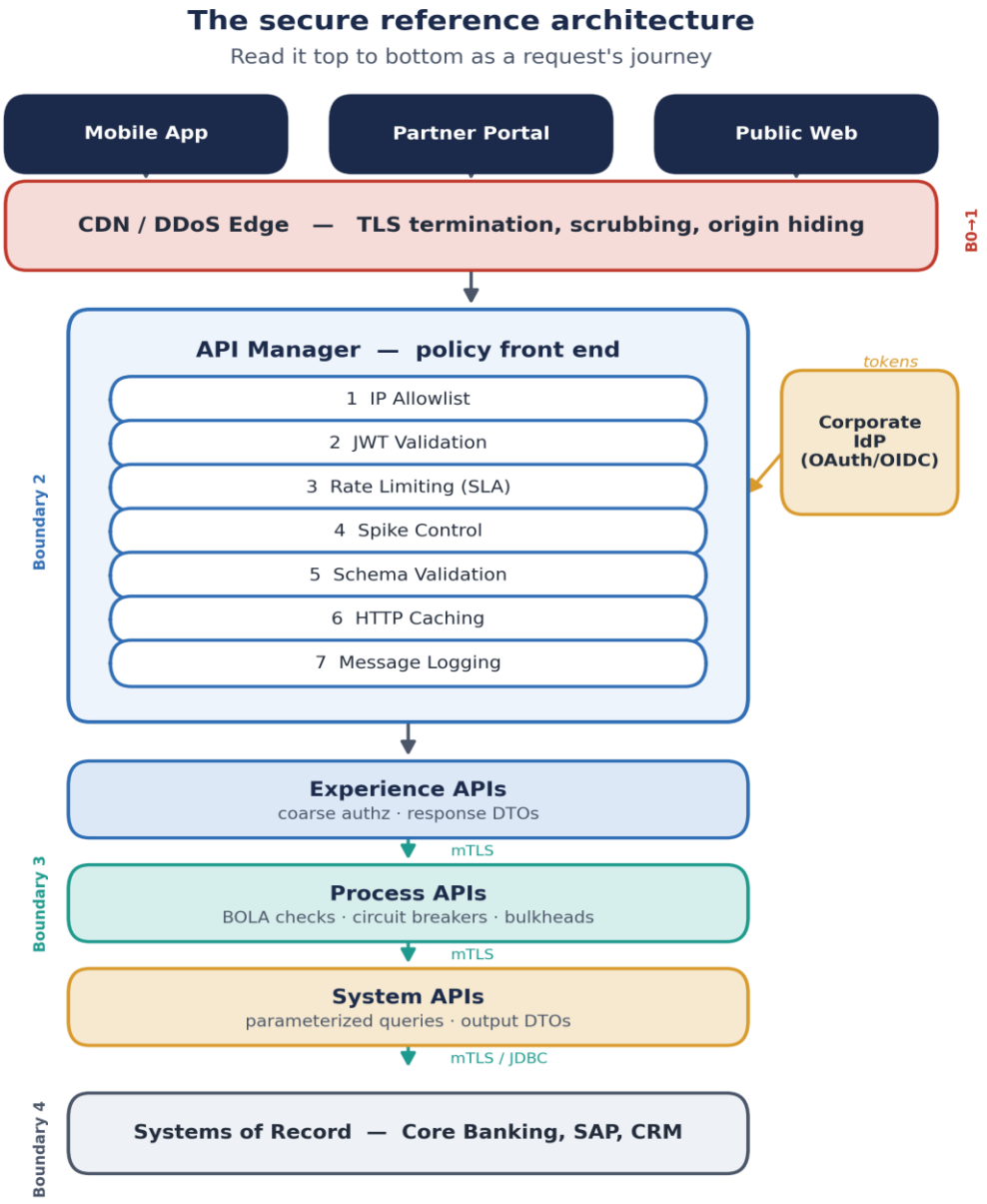


Figure 6-1. The secure reference architecture.

Chapter 7
Identity & Access Management

“ Part I told you where authentication happens — at the API front end, logic. This chapter opens the IdP box and explains how identity actually enterprise: OAuth 2.0, OpenID Connect, JWT internals, token lifecycle, and that keep breaking auth despite everyone thinking they've "done

OAuth 2.0 (RFC 6749), OpenID Connect 1.0, PKCE (RFC 7636), and JWT (RFC 7519) are the baseline. If your IdP is Okta, Azure AD, PingFederate, ForgeRock, or Keycloak, the flows are the same even when the console labels differ.

7.1 Authentication vs. authorization — again, because teams still mix them up

Authentication (AuthN): prove *who you are*. Username/password, MFA, certificate, biometric.

Authorization (AuthZ): given who you are, *what are you allowed to do?* Scopes, roles, claims, object-level permissions.

In API security:

Layers	AuthN	AuthZ
API Gateway (front end)	Validate JWT signature, issuer, audience, expiry	Coarse scope check: "does token include read:accounts?"
Process API	Token already validated upstream	Fine-grained: "does this user own account 100245?" (BOLA)
System API	mTLS proves the calling service	Data-level: "return only fields this role may see"

The gateway answers AuthN. It can do coarse AuthZ. It cannot do BOLA — that needs business context (Chapter 2, §2.3).

7.2 OAuth 2.0 — the framework everyone uses

OAuth 2.0 is an **authorization framework**, not an authentication protocol. It lets a client obtain limited access to a resource on behalf of a user (or itself) without handing the client the user's password.

The four roles

```
Resource Owner    — the user (or system) who owns the data
Client            — the app requesting access (mobile app, partner portal, batch job)
Authorization Server — the IdP that issues tokens (login.yourorg.com)
Resource Server   — your API that protects resources (api.yourorg.com)
```

Grant types you must understand

Grant	Use case	Guidance
Authorization Code + PKCE	User-facing apps (mobile, SPA, web)	**Default choice.** PKCE is mandatory for public clients.
Client Credentials	Machine-to-machine, no user	Service accounts, batch jobs, partner backends
Refresh Token	Extending sessions without re-login	Short-lived access tokens + rotated refresh tokens
Resource Owner Password	Legacy username/password direct to token endpoint	**Deprecated.** Do not use for new apps.
Implicit	SPA getting token in URL fragment	**Deprecated.** Use Auth Code + PKCE instead

> **"Gotcha.** I still find password grant in production configs from 2018 projects nobody migrated. It trains users to type credentials into third-party apps and bypasses MFA. Kill it.

Authorization Code + PKCE — the flow that matters

PKCE (Proof Key for Code Exchange) stops an attacker who intercepts the authorization code from exchanging it for tokens — because they don't have the `code_verifier` that only the legitimate client generated.

```
1. Client generates code_verifier (random 43–128 chars)
and code_challenge = BASE64URL(SHA256(code_verifier))

2. Client redirects user to Authorization Server:
GET /authorize?
     response_type=code
     &client_id=mobile-app
     &redirect_uri=https://app/callback
     &scope=openid read:accounts
     &state=random-csrf-token
     &code_challenge=<challenge>
     &code_challenge_method=S256

3. User logs in (with MFA). Authorization Server shows consent screen.

4. Authorization Server redirects back:
   GET https://app/callback?code=AUTH_CODE&state=random-csrf-token
  Client verifies state matches (CSRF protection).

5. Client exchanges code for tokens (back-channel, not in browser):
  POST /token
   grant_type=authorization_code
   &code=AUTH_CODE
   &redirect_uri=https://app/callback
   &client_id=mobile-app
   &code_verifier=<original verifier>

6. Authorization Server returns:
 {
   "access_token": "eyJ...",
   "token_type": "Bearer",
   "expires_in": 3600,
   "refresh_token": "def502...",
   "id_token": "eyJ..."        // if scope included openid
   }
```

"Figure 7.1 — Screenshot: Your IdP's application registration screen (Okta: Applications Create App Integration OIDC Native; Azure AD: App registrations New registration Mobile and desktop). Key fields: Redirect URIs (exact match required), Grant types (Authorization Code only), PKCE required toggle if available.

Client Credentials — machine-to-machine

No user. The client authenticates with its own `client_id` + `client_secret` (or certificate) and receives an access token scoped to what that service account is allowed to do.

```
POST /token
grant_type=client_credentials
&client_id=batch-reconciliation-svc
&client_secret=<secret>
&scope=read:transactions write:reconciliation
```

Use for: nightly batch jobs, partner backend-to-backend, internal microservices calling each other through the gateway. Pair with **mTLS** where the client secret alone isn't enough (Chapter 8).

7.3 OpenID Connect — authentication on top of OAuth

OIDC adds an **identity layer**: an `id_token` (a JWT) that tells the client *who* logged in, plus a standard UserInfo endpoint.

Token	Purpose	Audience
Access token	Call APIs	Resource server (your API gateway)
ID token	Identity claims about the user	Client app only — **never send to APIs**
Refresh token	Get new access tokens	Client + Authorization Server only

The `id_token` carries claims like `sub` (subject/user ID), `email`, `name`, `auth_time`, `acr` (authentication context — did they use MFA?). Your **API gateway validates the access token**, not the id_token.

Standard OIDC scopes:

- `openid` — required to get an id_token
- `profile`, `email`, `address`, `phone` — optional identity claims

API-specific scopes you define: `read:accounts`, `write:payments`, `admin:users`. Register them in your IdP and document them in your API catalog.

7.4 JWT internals — what the gateway actually validates

A JWT is `header.payload.signature`, each part base64url-encoded.

Header:

```
{ "alg": "RS256", "typ": "JWT", "kid": "abc123" }
```

Payload (claims):

```
{
  "iss": "https://login.yourorg.com",
  "sub": "user-uuid-12345",
  "aud": "https://api.yourorg.com",
  "exp": 1685624400,
  "iat": 1685620800,
  "scope": "read:accounts read:statements",
  "client_id": "mobile-banking-app",
  "acr": "urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport"
}
```

Signature: RS256 = RSA-SHA256 over `header.payload`, signed with the IdP's private key. Your gateway verifies with the public key from the JWKS endpoint.

Validation checklist (every API must enforce all of these)

```
[ ] Signature valid (fetch JWKS from iss/.well-known/jwks.json, match kid)
[ ] iss matches expected issuer URL exactly
[ ] aud matches this API's audience (or contains it, per your IdP's format)
[ ] exp is in the future (allow small clock skew, e.g. 60 seconds)
[ ] nbf (if present) is in the past
[ ] Required scopes present for the requested resource
[ ] alg is in your allow-list (RS256, ES256 — never "none", never unexpected HS256)
```

"**War story.** A pen test found our API accepted tokens with aud: "https://api.other-division.com" because we validated signature and expiry but never aud. Tokens minted for a low-trust internal app worked against our customer-facing API. One line in the JWT policy fixed it. Check aud.

JWKS rotation

IdPs rotate signing keys. Your gateway must:

- Fetch JWKS from `https://login.yourorg.com/.well-known/openid-configuration` → `jwks_uri`
- Cache keys with a TTL (e.g. 24 hours)
- On signature failure with unknown `kid`, refresh JWKS immediately and retry once

MuleSoft JWT Validation policy handles this when you point it at the JWKS URL (Chapter 5, §5.4).

7.5 Token lifecycle — the decisions that prevent long-lived breaches

Access token lifetime

Short. 5–15 minutes for high-sensitivity APIs (banking, healthcare). Up to 60 minutes for lower-risk internal APIs. Short lifetime limits damage from a stolen token.

Refresh tokens

Longer-lived (hours to days), used only to obtain new access tokens. **Rotate on every use:** when a refresh token is exchanged, the IdP issues a new refresh token and invalidates the old one. If an attacker replays a stolen refresh token after the legitimate client has refreshed, the IdP detects reuse and revokes the whole token family.

Revocation

OAuth 2.0 Token Revocation (RFC 7009): `POST /revoke` with the token. Use on logout, password change, and account compromise. Your IdP should support it; your apps should call it.

Token storage on clients

Client type	Where to store	Never
Mobile (native)	OS secure storage (Keychain, Keystore)	LocalStorage, plain files
SPA	Memory only; use refresh via httpOnly cookie if BFF pattern	localStorage, sessionStorage for access tokens
Server-side app	Server session or encrypted server-side store	Embed in client-side JavaScript
M2M	Vault / secrets manager	Source code, config files in Git

7.6 SAML vs. OIDC — when you see both

In large enterprises you often have **both**:

- **SAML 2.0** — still dominant for **workforce SSO** (employees logging into Anypoint Platform, Salesforce, internal portals). XML-based, browser POST bindings, IdP-initiated flows.
- **OIDC** — dominant for **customer and partner APIs**, mobile apps, and modern SPAs. JSON/JWT-based, simpler for developers.

They solve different problems. SAML federates *human login to applications*. OIDC/OAuth federates *API access for apps and users*.

Typical pattern:

```
Employee → SAML SSO → Anypoint Platform console (Chapter 5, §5.1)
Customer → OIDC + PKCE → Mobile app → JWT access token → API Gateway
Partner  → Client Credentials or OIDC → Partner portal → API Gateway
```

Don't try to make SAML talk directly to your REST APIs. Use the IdP's OIDC layer (most enterprise IdPs expose both) for API tokens.

"Figure 7.2 — Screenshot: Anypoint Access Management Identity Providers Add identity provider SAML 2.0. Fields: Identity Provider Name, Issuer URL, IdP Sign-in URL, IdP Certificate (paste X.509). This federates employee login — separate from API JWT validation.

7.7 Multi-factor authentication (MFA)

MFA is baseline for workforce access and increasingly expected for customer-facing high-risk operations (payments, profile changes, password reset).

Factors:

- Something you know (password)
- Something you have (TOTP app, hardware key, SMS — SMS is weaker, avoid for high assurance)
- Something you are (biometric — device-bound)

Step-up authentication: normal session uses password + TOTP; initiating a wire transfer requires re-authentication. OIDC supports this via `acr` (Authentication Context Class Reference) and `max_age` parameters on the authorize request.

```
GET /authorize?...&acr_values=urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport
          &max_age=0    // force fresh login
```

Your API can require a minimum `acr` in the access token for sensitive endpoints.

7.8 Common authentication attacks and defenses

Attack	Mechanism	Defense
Token theft	XSS steals token from browser storage	httpOnly cookies or memory-only; short access token TTL
Authorization code interception	Malicious app registers same redirect URI	Exact redirect URI match; PKCE
CSRF on OAuth callback	Attacker tricks victim into completing flow with attacker's code	state parameter validated by client
JWT alg:none	Forge token with no signature	Pin allowed algorithms server-side
JWT key confusion	RS256 token verified with HS256 using public key as secret	Pin algorithm; use JWKS not shared secrets for RS256
Credential stuffing	Leaked passwords tried at login	Rate limit login endpoint; MFA; breached-password detection
Refresh token replay	Stolen refresh token used after rotation	Refresh token rotation + family revocation
Open redirect	redirect_uri validation bypass	Strict exact-match allow-list
Scope escalation	Client requests admin scope, server grants it	IdP consent screen; scope allow-list per client registration

7.9 Configuring JWT validation on MuleSoft — recap and extend

From Chapter 5, the baseline policy. Extended for production:

```
policy: jwt-validation
configuration:
 jwtOrigin: httpBearerAuthenticationHeader
 signingMethod: jwks
 jwksUrl: "https://login.yourorg.com/.well-known/jwks.json"
 jwksCacheTtlInMinutes: 1440
 audiences:
  - "https://api.yourorg.com"
  - "api://customer-experience"    # if your IdP uses resource URIs
 claimsValidation:
  - claim: "exp"
    required: true
```

```
- claim: "iss"
  required: true
  value: "https://login.yourorg.com"
- claim: "scope"
  required: true
  regex: ".*read:accounts.*"      # per-API: adjust per route via conditional policy
```

For different APIs requiring different scopes, use **conditional policy execution** or separate API instances in API Manager rather than one regex trying to cover everything.

"Figure 7.3 — Screenshot: JWT Validation policy Claims Validation section expanded, showing rows for exp (required), iss (required, value filled), scope (required, regex). The JWKS URL field at the top is populated with your IdP's endpoint.

7.10 Identity architecture patterns for the enterprise

Pattern A — Central IdP, API gateway as resource server

```
[Clients] → [API Gateway validates JWT] → [Services trust gateway]
                  ↑
            [Corporate IdP]
```

Simplest. Gateway is the only JWT validator. Internal services trust that anything reaching them passed the gateway (mTLS + network segmentation).

Pattern B — BFF (Backend for Frontend)

```
[SPA] → [BFF server] → [API Gateway] → [Services]
          ↑
    holds tokens server-side;
    SPA gets session cookie only
```

Solves the SPA token storage problem. BFF exchanges OIDC tokens and holds refresh tokens server-side. SPA never sees a bearer token.

Pattern C — Service mesh with workload identity

```
[Service A] → [mesh mTLS + SPIFFE ID] → [Service B]
```

No user JWT — services authenticate each other with workload certificates (Chapter 10, Chapter 15). User context propagated as signed internal headers after gateway validation.

7.11 IAM checklist for API programs

```
IDENTITY PROVIDER
[ ] Corporate IdP with MFA for workforce (SAML/OIDC to platforms)
[ ] Separate OIDC application registrations per client app (mobile, web, partner)
[ ] Authorization Code + PKCE for all user-facing clients
[ ] Client Credentials for M2M with secrets in vault
[ ] Password and Implicit grants disabled

TOKENS
[ ] Access token TTL ≤ 15 min (high sensitivity) or ≤ 60 min (internal)
[ ] Refresh token rotation enabled
[ ] Token revocation on logout and password change
[ ] aud, iss, exp, scope validated at gateway on every request

CLIENT SECURITY
[ ] Redirect URIs exact-match allow-list
[ ] Tokens not in localStorage (SPA) or plain files (mobile)
[ ] state parameter on all OAuth flows

API GATEWAY
[ ] JWT Validation policy on every external API (automated policy)
[ ] JWKS URL configured with cache and kid refresh
[ ] Algorithm allow-list (RS256, ES256 only)
[ ] Per-API scope requirements documented and enforced

OPERATIONS
[ ] Client secret rotation schedule
[ ] IdP signing key rotation monitored (JWKS kid changes)
[ ] Failed auth / token validation metrics alerted
```

7.12 API keys vs. OAuth tokens — when each is appropriate

You still see **API keys** (`X-API-Key: abc123`) on internal and partner APIs. They're simpler but weaker than OAuth.

Mechanism	Strength	Weakness	Use when
API key	Easy to implement	No expiry by default; one key = full access if leaked	Low-risk internal tools, dev sandboxes
OAuth client credentials	Scoped, revocable, auditable	More moving parts	Partner M2M, production
JWT access token	Short-lived, carries identity + scopes	Validation complexity	User-facing and high-assurance

If you must use API keys in production:

- One key per consumer, per environment
- Store hashed server-side (like passwords)
- Rotate on schedule; instant revoke path
- **Never** the only control on sensitive data — pair with IP allowlist or mTLS
- Migrate path to OAuth documented

MuleSoft client ID/secret in API Manager is an API-key-style model. Treat the **client secret** like a password: vault storage, rotation, never in mobile app binaries.

7.13 Token introspection (RFC 7662)

Some architectures use **opaque access tokens** instead of JWTs. The resource server can't validate locally — it calls the IdP **introspection endpoint**:

```
POST /introspect
token=<access_token>
Authorization: Basic <client_id:secret>

Response:
{ "active": true, "scope": "read:accounts", "sub": "user-123", "exp": 1685624400 }
```

Pros: Instant revocation (token inactive as soon as IdP marks it). **Cons:** Latency and availability dependency on IdP every request.

Hybrid pattern: JWT for most calls (fast local validation); introspection for high-risk operations or when `jti` appears on a revocation list.

Cache introspection results for **seconds only** — not minutes — or revocation is meaningless.

7.14 Federation and multi-tenant IdP patterns

Large enterprises often have **multiple IdPs** after mergers:

```
Partner users  → Partner IdP (federated SAML/OIDC) → Your IdP (token exchange) → Your APIs
Employees      → Corporate AD / Entra ID → Your APIs
Customers      → Customer CIAM (Auth0, Okta Customer Identity) → Your APIs
```

Token exchange (RFC 8693) — swap a token from one issuer for a token your APIs trust. Configure trust relationship carefully; validate original token before issuing new one.

Multi-tenant APIs: `tenant_id` claim in JWT; enforce in process layer so tenant A never reads tenant B's data (BOLA at tenant scope).

```
if token.tenant_id != resource.tenant_id:
    return 403
```

7.15 Session management for web and mobile

Concern	Web	Mobile native
Session fixation	Regenerate session ID post-login	N/A (token-based)
Logout	Revoke refresh token + clear server	Revoke tokens; clear Keychain
Idle timeout	15–30 min for banking web	App backgrounding → require re-auth for
Concurrent	Policy per product	Device binding optional

Refresh token in httpOnly cookie (BFF pattern): SPA never touches refresh token; BFF handles OIDC silent refresh. Prevents XSS token theft from browser storage.

7.16 Identity operations — runbook snippets

Compromised user account

```
1. IdP: disable account / force password reset
2. IdP: revoke all refresh tokens for sub
3. If JWT not introspected: wait for access token exp (≤15 min) OR publish jti to deny list at gateway
4. Review audit logs for sub since compromise estimate
5. Notify user per policy
```

Compromised client application secret

```
1. API Manager: regenerate client secret
2. Notify partner via secure channel
3. Monitor old secret usage — should drop to zero
4. If abuse detected: disable client application until partner confirms
```

IdP signing key rotation

```
1. IdP publishes new key in JWKS (new kid)
2. Gateway refreshes JWKS cache (automatic or forced)
3. Old kid tokens expire naturally within access token TTL
4. Monitor jwt_validation_failures metric during rotation window
```

"**Figure 7.4 — Screenshot:** IdP Signing Keys admin page showing primary and secondary keys with kid values and rotation schedule. Cross-check kids match what your gateway JWKS cache shows.

Chapter 7 takeaways

- **OAuth 2.0** authorizes; **OIDC** authenticates. Use **Authorization Code + PKCE** for user apps; **Client Credentials** for M2M.
- Validate **every** JWT claim that matters: signature, `iss`, `aud`, `exp`, scopes. Never trust `alg` from the token header.
- **Short access tokens**, **rotated refresh tokens**, **revocation on logout**.
- **SAML** for workforce SSO to consoles; **OIDC/JWT** for API access.
- MFA and step-up auth for high-risk operations; rate-limit the login endpoint.

Previous: [Chapter 6 — Reference Architecture](06-reference-architecture.md) · Next: [Chapter 8 — Transport Security, Encryption & Key Management](08-transport-encryption.md)

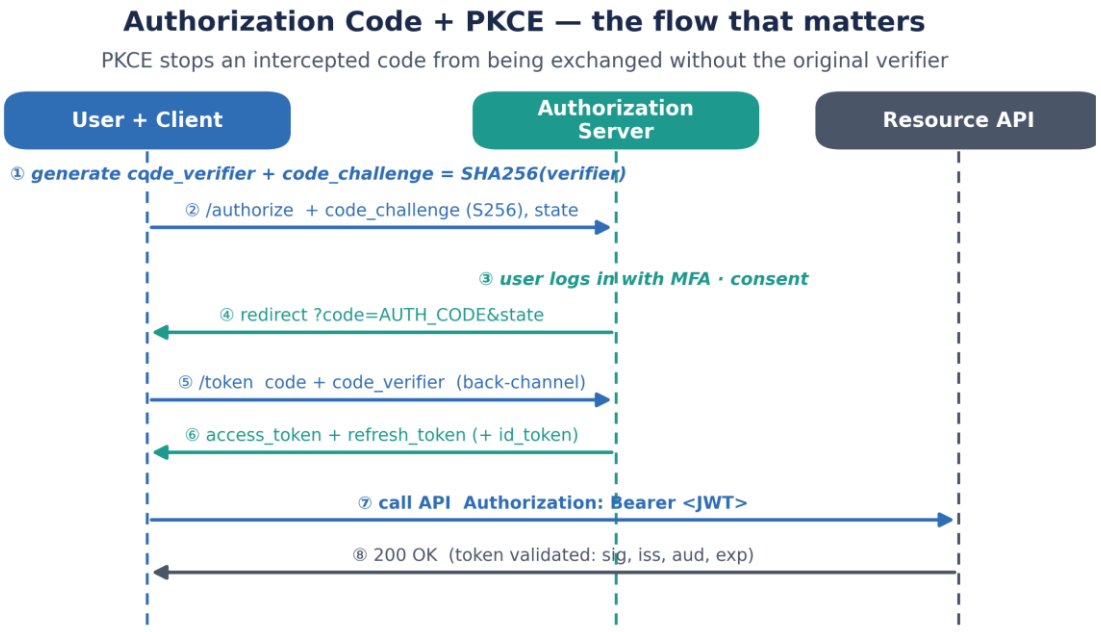


Figure 7-1. Authorization Code + PKCE flow.

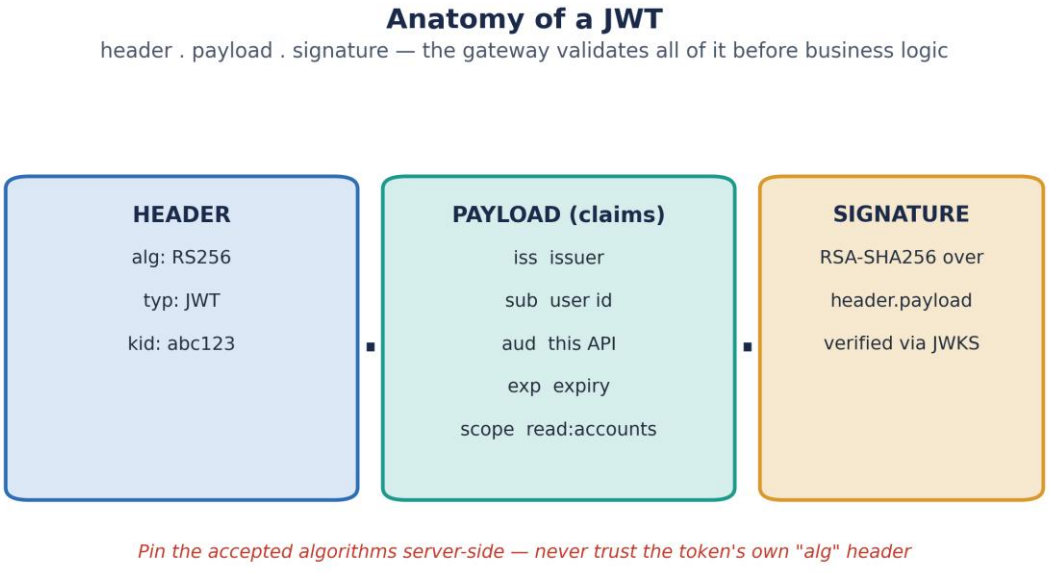


Figure 7-2. Anatomy of a JWT.

Chapter 8
Transport Security, Encryption & Key Management

“ Confidentiality and integrity depend on cryptography done correctly — not "HTTPS enabled" in a checkbox. This chapter covers TLS 1.2 and 1.3, mutual service-to-service trust, encryption at rest, tokenization, and the key-discipline that makes all of it auditable.

8.1 TLS — what "HTTPS" actually guarantees

Transport Layer Security gives you three properties on the wire:

- **Confidentiality** — eavesdroppers can't read the payload
- **Integrity** — tampering is detected
- **Server authentication** — the client verifies it's talking to the real server (and with mTLS, the server verifies the client too)

TLS does **not** authenticate the *user*. That's JWT/OAuth upstream. TLS authenticates the *endpoint*.

TLS versions

Versions	Status
SSL 3.0, TLS 1.0, TLS 1.1	**Prohibited.** PCI DSS and every major browser have deprecated them.
TLS 1.2	Minimum acceptable everywhere. Required for FIPS 140-2 environments (Chapter 5).
TLS 1.3	Preferred where supported. Faster handshake, removes weak cipher suites by design.

Configure your edge, load balancer, API gateway, and Mule runtime to **disable TLS 1.0/1.1** and prefer 1.3 with 1.2 fallback.

8.2 Cipher suites — what to allow and what to block

A cipher suite names the combination of key exchange, authentication, encryption, and MAC algorithms. A sound baseline:

Allow (TLS 1.2):

```
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
```

Allow (TLS 1.3):

```
TLS_AES_256_GCM_SHA384
TLS_AES_128_GCM_SHA256
TLS_CHACHA20_POLY1305_SHA256
```

Block:

```
Anything with NULL, EXPORT, DES, 3DES, RC4, MD5
RSA key exchange without forward secrecy (TLS_RSA_WITH_*)
Anonymous DH (aNULL)
```

Forward secrecy (PFS): prefer **ECDHE** key exchange so that compromising the server's long-term private key doesn't decrypt past sessions.

> **"Gotcha.** Default Java/Mule cipher lists often still include legacy suites for "compatibility." In a FIPS deployment (Chapter 5), the FIPS provider restricts these automatically. In non-FIPS, explicitly configure the allowed list — don't rely on defaults.

Testing your TLS configuration

```
# Quick check with openssl
openssl s_client -connect api.yourorg.com:443 -tls1_2 \
  -cipher 'ECDHE-RSA-AES256-GCM-SHA384' </dev/null 2>&1 | grep "Cipher"
```

```
# Broader audit — use ssllabs.com SSL Test or testssl.sh
testssl.sh --protocols --cipher-permutations https://api.yourorg.com
```

Target grade **A or A+** on SSL Labs. Anything below means you're offering weak suites or missing HSTS.

8.3 HTTP Strict Transport Security (HSTS)

TLS protects only if the client uses HTTPS. HSTS tells browsers: "always use HTTPS for this domain for the next N seconds."

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

Set at the CDN or load balancer for all public API hostnames. Submit to the HSTS preload list for browser built-in enforcement.

APIs consumed by mobile apps and server clients should **reject plain HTTP** at the gateway — don't offer a redirect from HTTP to HTTPS on the API port; refuse the connection.

8.4 Mutual TLS (mTLS) — service-to-service trust

In the reference architecture (Chapter 6), internal traffic between experience process system APIs uses **mTLS**. Both sides present certificates; both verify the other.

Why mTLS inside the network:

- Network segmentation is not sufficient (lateral movement after a breach)
- IP allowlists don't prove identity (spoofing, compromised host)
- mTLS gives each service a cryptographic identity

Certificate model for internal services

Approach	Pros	Cons
Shared internal CA	Simple PKI you control	Compromise of CA affects all
Per-service certificates from CA	Revocable per service	CA ops overhead
SPIFFE/SPIRE (Chapter 10, 15)	Short-lived, automated rotation	Requires mesh infrastructure

Mature shops use an internal CA (often HashiCorp Vault PKI, or cloud CA services) or SPIFFE for Kubernetes workloads.

MuleSoft mTLS configuration

```
<!-- Outbound: present client cert to downstream -->
<tls:context name="Outbound_mTLS">
 <tls:key-store type="jks"
                path="keystores/service-client.jks"
                keyPassword="${secure::tls.keyPassword}"
                password="${secure::tls.storePassword}"/>
<tls:trust-store path="keystores/internal-ca-trust.jks"
                  password="${secure::tls.trustPassword}"/>
</tls:context>
```

```
<http:request-config name="Process_API_Client">
  <http:request-connection protocol="HTTPS"
                           host="process-api.internal"
                          port="443"
                           tlsContext="Outbound_mTLS"/>
</http:request-config>
```

"Figure 8.1 — Screenshot: Runtime Manager Settings TLS for a CloudHub worker, or on-prem tls:context in Anypoint Studio's Global Elements view showing Key Store and Trust Store paths configured.

8.5 Certificate lifecycle

Certificates expire. Expired certs cause outages that look like attacks.

Lifecycle stages

```
1. Generate key pair (2048-bit RSA minimum, or P-256 EC)
2. Create CSR (Certificate Signing Request)
3. Submit to CA (public: DigiCert, Let's Encrypt; internal: your PKI)
4. Install certificate + chain on endpoint
5. Monitor expiry (alert at 30, 14, 7 days)
6. Renew and redeploy before expiry
7. Revoke if private key compromised (CRL/OCSP)
```

Let's Encrypt

Free, automated, 90-day certificates. Ideal for public-facing dev/test and some production edges. Use **cert-manager** (Kubernetes) or your CDN's ACME integration for automatic renewal.

Internal certificates

Longer validity (1–2 years) but **must** be in a tracking system. I've seen more production outages from expired internal certs than from expired public ones — because nobody put them in the monitoring tool.

```
# Check cert expiry from command line
echo | openssl s_client -connect api.yourorg.com:443 2>/dev/null \
  | openssl x509 -noout -dates -subject
```

8.6 Encryption at rest

Data on disk — databases, file stores, message queues, logs, backups — must be encrypted separately from TLS. TLS only protects data *in transit*.

Layers

Layers	Mechanism	Typical
Disk / volume	Full-disk or volume encryption	AWS EBS encryption, Azure SSE, LUKS on-
Database	Transparent Data Encryption	SQL Server TDE, Oracle TDE, PostgreSQL
Application-level	Field-level encryption before	Encrypt PII columns with KMS-managed
Object storage	Server-side encryption (SSE-S3,	Default on in major clouds

Field-level encryption

For highly sensitive fields (SSN, account numbers, health data), encrypt at the application layer before the value hits the database. The DB stores ciphertext; only the app with the KMS key can decrypt.

```
plaintext SSN → AES-256-GCM encrypt with DEK → store ciphertext + wrapped DEK
DEK wrapped by KMS master key (KEK) → store wrapped DEK alongside ciphertext
```

This limits blast radius: a DBA with DB access sees ciphertext, not SSNs.

8.7 Tokenization vs. encryption

Encryption is reversible with the key. **Tokenization** replaces sensitive data with a non-sensitive surrogate (token) that has no mathematical relationship to the original.

```
PAN 4111-1111-1111-1111  →  token tok_8f3a9c2b  (stored in token vault)
```

The token vault maps `tok_8f3a9c2b` real PAN. Systems that don't need the real PAN only ever see the token. PCI scope reduction often drives tokenization for cardholder data.

In API responses, return tokens instead of real identifiers where the client doesn't need the raw value.

8.8 Key management — the part audits ask about

Cryptography is only as good as key handling. Enterprise practice centers on:

Key hierarchy

Envelope encryption: DEK encrypts data; KEK encrypts DEK. Rotate DEKs frequently; rotate KEK rarely with re-wrap of DEKs.

Where keys live

Option	Use case
Cloud KMS (AWS KMS, Azure Key Vault, GCP Cloud KMS)	Cloud-native apps, envelope encryption
HSM (Hardware Security Module)	FIPS 140-2 Level 3, PCI, government
HashiCorp Vault Transit	On-prem or multi-cloud, API-based crypto
Mule Secure Properties key	Application config secrets only — not a general KMS

"Gotcha. Developers sometimes use Secure Properties encryption (Chapter 5, Chapter 9) as a general-purpose KMS. It's for config values in deployment artifacts, not for encrypting millions of database records at runtime. Use a real KMS for data encryption.

Key rotation

Key type	Rotation frequency	Process
TLS certificates	Before expiry (automate)	ACME / cert-manager
API client secrets	90 days or on compromise	IdP rotation + vault update
DEKs	Per policy or on compromise	Re-encrypt data with new DEK
KEK / KMS master	Annually or on policy	Re-wrap DEKs, no data re-encrypt
JWT signing keys	IdP-managed, JWKS kid rotation	Gateway refreshes JWKS

Document every rotation in your key-management policy. Auditors ask for the procedure, not just the fact that you rotate.

8.9 FIPS 140-2 and transport crypto (tie-in to Chapter 5)

In FIPS mode, your TLS and at-rest crypto must use **FIPS-validated modules** and **approved algorithms**:

- TLS 1.2+ with approved cipher suites only
- AES-128/256, RSA ≥2048, EC P-256/P-384
- Approved DRBG for random number generation
- No MD5, SHA-1 for signatures, DES, RC4

The Mule FIPS configuration from Chapter 5 applies here at the runtime layer. Your load balancer and CDN must also be configured for FIPS-approved TLS if you're in a fully FIPS environment — check vendor documentation for FIPS mode on F5, AWS GovCloud, etc.

8.10 Transport & encryption checklist

```
TLS (PUBLIC-FACING)
[ ] TLS 1.0/1.1 disabled; TLS 1.2 minimum, 1.3 preferred
[ ] Strong cipher suites only; weak/legacy suites explicitly removed
[ ] HSTS enabled with max-age ≥ 1 year
[ ] Certificate expiry monitoring with alerts
[ ] SSL Labs grade A or A+

mTLS (INTERNAL)
[ ] mTLS between API layers (experience → process → system)
[ ] Internal CA or SPIFFE for service identity
[ ] Trust stores contain only required CAs (no default public CA trust for internal)

ENCRYPTION AT REST
[ ] Database TDE or volume encryption enabled
[ ] Object storage SSE enabled (KMS-managed keys)
[ ] Field-level encryption for highest-sensitivity PII [
] Tokenization for PCI cardholder data where applicable
```

```
KEY MANAGEMENT
[ ] Keys in KMS/HSM, not in application code or Git
[ ] Key rotation procedures documented and tested
[ ] Separation of duties: developers can't access production keys
[ ] FIPS-validated modules where required (Chapter 5)
```

8.11 Building an internal PKI — practical steps

For mTLS at scale you need an **internal Certificate Authority**.

Option A — HashiCorp Vault PKI

```
vault secrets enable pki
vault secrets tune -max-lease-ttl=87600h pki
vault write pki/root/generate/internal \
  common_name="yourorg.internal" ttl=87600h
vault write pki/roles/mule-service \
  allowed_domains="mule.internal" allow_subdomains=true max_ttl=720h
vault write pki/issue/mule-service common_name="payments.mule.internal"
```

Automate renewal: sidecar or init container renews cert at 80% of TTL.

Option B — Cloud CA

- AWS Private CA
- Azure Private Link / internal CA services
- Google Cloud CAS

Integrate with cert-manager in Kubernetes for automatic pod certificate injection.

CA hierarchy best practice

Never issue service certs directly from root. Compromise of intermediate is recoverable; root compromise is catastrophic.

8.12 TLS on the API gateway and load balancer — configuration patterns

NGINX (edge)

```
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:...;
```

```
ssl_prefer_server_ciphers off;   # TLS 1.3 ignores this
ssl_session_cache shared:SSL:10m;
ssl_session_timeout 1d;
ssl_stapling on;
ssl_stapling_verify on;
```

AWS ALB

- Security policy: `ELBSecurityPolicy-TLS-1-2-2017-01` minimum; prefer `TLS-1-2-Ext-2018-06` or TLS 1.3 policies
- Attach ACM certificate; enable HTTPHTTPS redirect at ALB

Mule listener

Ensure `tls:context` on every `http:listener-connection` with `protocol="HTTPS"` in non-dev environments. Mixed HTTP/HTTPS in prod is a finding waiting for an auditor.

8.13 Data masking and redaction in transit and logs

Encryption protects confidentiality on the wire; **masking** limits exposure in payloads and logs.

Techniques	Example
Truncation	****-****-****-1234 for PAN display
Tokenization	Return tok_abc instead of real identifier
Format-preserving encryption	FPE for legacy systems needing format match
Dynamic masking	Role-based: teller sees last 4; fraud team sees full with audit

Apply masking in **system/process layer** before response leaves trust zone. Gateway can add masking policy for known fields via DataWeave transform policy if centralized.

8.14 Cryptographic agility

Algorithms age. Design for **agility** — ability to switch algorithms without rewriting every service.

```
- Abstract crypto behind platform APIs (JCE, KMS) — not custom implementations
- Version encryption envelope: { "v": 1, "alg": "AES-256-GCM", "ciphertext": "..." }
- Document approved algorithm list; review annually
- FIPS shops: track CMVP validation status of provider versions
```

When 3DES and SHA-1 were deprecated, teams with hard-coded algorithms suffered longest migrations.

Chapter 8 takeaways

- **TLS 1.2+** with strong ciphers and **HSTS** for public APIs; disable legacy protocols.
- **mTLS** for service-to-service identity inside the network — IP allowlists are not enough.
- **Certificate lifecycle** must be monitored; expired certs cause outages.
- **Encryption at rest** is separate from TLS; use TDE, field-level encryption, and tokenization as appropriate.
- **Keys** belong in KMS/HSM with documented rotation — not in code or config repos.

Previous: [Chapter 7 — Identity & Access Management](07-identity-access-management.md) · Next: [Chapter 9 — Secrets Management & Configuration Security](09-secrets-management.md)

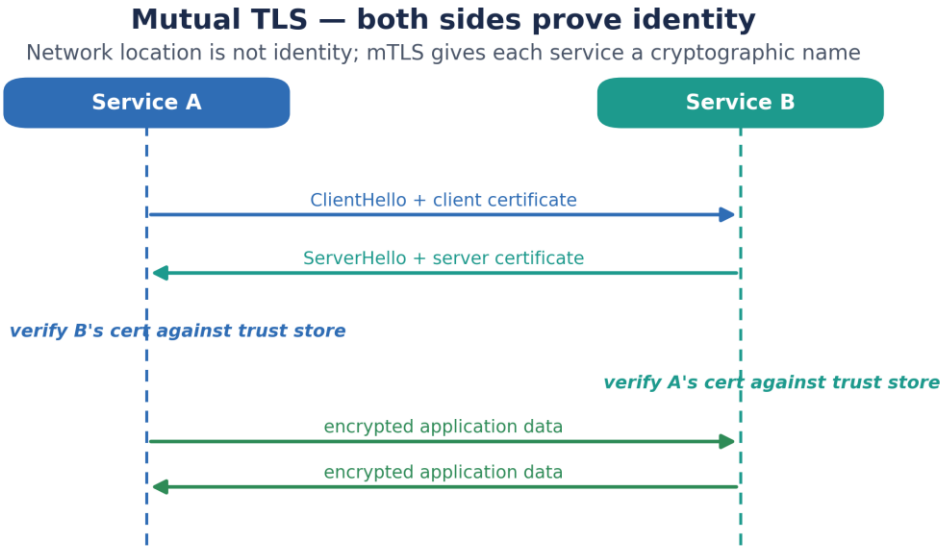


Figure 8-1. Mutual TLS handshake.

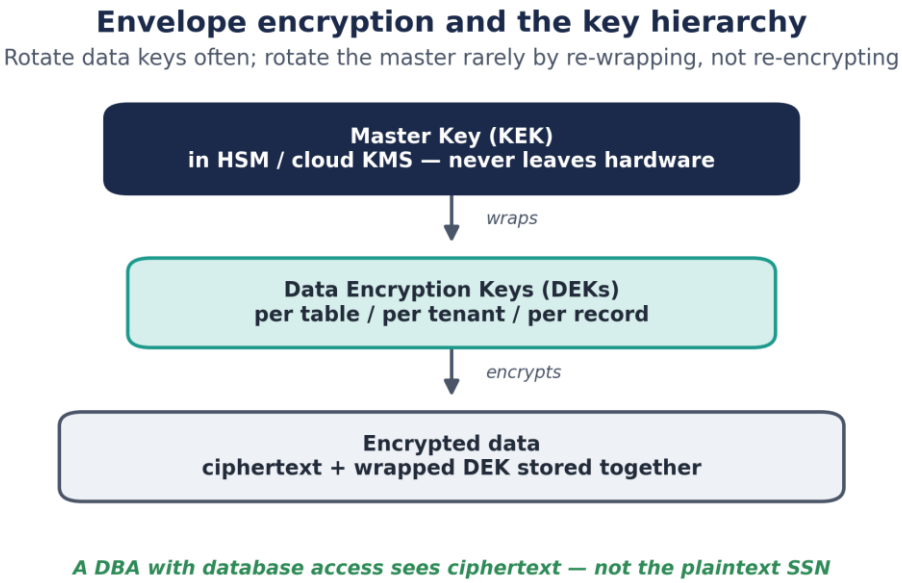


Figure 8-2. The key hierarchy: KEK to DEK to data.

Chapter 9
Secrets Management & Configuration Security

“ A leaked repository should not be a breach. This chapter is about where secrets live, how they get to runtime without passing through Git, how you rotate them without a weekend outage, and how MuleSoft Secure Properties fits into a broader vault strategy — not replaces it.

9.1 What counts as a secret

Anything that grants access or decrypts data:

- Database passwords and connection strings
- API client secrets and OAuth credentials
- TLS private keys and keystore passwords
- Encryption keys (`mule.key`, KMS credentials)
- Third-party API keys (payment gateways, SMS providers)
- SSH keys and service account passwords
- JWT signing keys (IdP-side — but apps sometimes wrongly hold these)

Not secrets (but often misclassified): hostnames, port numbers, non-sensitive feature flags, public API URLs. Don't encrypt everything — it adds ops burden without security gain.

“War story. A contractor cloned a repo to a personal GitHub "for backup." The repo had prod-db-password=... in config.yaml. GitHub secret scanning didn't catch it because it wasn't a known API key format. The password was valid for eleven months until a routine pen test found the public fork. Rotateeverything; don't rely on repo privacy.

Anti-pattern	Why it fails	What to do instead
Secrets in environment variables in CI	Logs are widely accessible	Masked injection, secret stores
Hard-coded in DataWeave/Java	Visible in artifact, logs,	${secure::} or vault lookup
Shared "integration" password for 12	No accountability, can't rotate	Per-service credentials
Secrets in Docker image layers	Image registry leak	Runtime mount or init container

9.2 The anti-patterns that cause breaches

Anti-pattern	Why it fails	What to do instead
Secrets in application.properties in Git	Repo leak = full compromise	Vault + runtime injection
Same password dev and prod	Dev laptop compromise → prod access	Per-environment secrets

9.3 Secrets management architecture

Principles:

- **Git stores references and ciphertext, never plaintext secrets.**
- **CI/CD authenticates to vault with short-lived identity** (OIDC to AWS/Azure, not long-lived access keys in Jenkins).
- **Runtime fetches secrets at startup or on demand** — not baked into the artifact.
- **Audit every secret access.**

9.4 HashiCorp Vault — the on-prem/multi-cloud standard

Vault is the most common dedicated secrets platform in enterprises that aren't all-in on one cloud.

Core concepts

Concepts	Meaning
Secret engine	Backend that stores/generates secrets (KV, database, PKI, AWS)
Path	secret/data/prod/mule/db-password
Policy	ACL: which identity can read/write which paths
Auth method	How clients prove identity (AppRole, Kubernetes, LDAP, cloud IAM)

KV secrets (static)

```
# Write
vault kv put secret/prod/mule/db password='...' host='db.internal'
```

```
# Read (from CI or runtime with appropriate policy)
vault kv get -field=password secret/prod/mule/db
```

Dynamic database credentials

Vault generates a **short-lived** DB user/password valid for 1 hour, then revokes it. Even if leaked, the window is tiny.

```
vault read database/creds/mule-app-role
# Key                Value
# lease_duration     1h
# username           v-token-mule-abc123
# password           A1b2C3d4-...
```

Configure Mule's DB connector with credentials fetched at startup via a small bootstrap script or custom property provider.

AppRole for Mule (on-prem)

```
1. Enable AppRole auth in Vault
2. Create policy: read secret/data/prod/mule/*
3. Create role bound to policy; get role_id + secret_id
4. Mule startup: authenticate → get token → read secrets → set as system properties
5. secret_id delivered via secure channel, not in Git
```

9.5 Cloud-native secrets

AWS Secrets Manager

```
# CloudHub / EC2 / ECS: IAM role grants read
# Mule property via AWS SDK or sidecar injection
aws secretsmanager get-secret-value --secret-id prod/mule/db-password
```

Integrate with **AWS Parameter Store** for non-rotating config; Secrets Manager for credentials with automatic rotation Lambdas.

Azure Key Vault

```
az keyvault secret show --vault-name prod-kv --name mule-db-password
```

Anypoint on Azure often uses Key Vault for `mule.key` and connector credentials via Runtime Fabric integration.

GCP Secret Manager

Similar pattern; workload identity for GKE pods to access secrets without static keys.

Pick one vault per environment. Don't scatter secrets across three systems.

9.6 MuleSoft Secure Properties — deep dive

Chapter 5 introduced Secure Properties. Here's the full operational picture.

What it does

Encrypts property **values** at build time with AES (CBC in Mule 4.4). Runtime decrypts with `mule.key`. Artifact in Nexus/Git contains only `![ciphertext]`.

Build-time encryption

```
java -cp secure-properties-tool.jar \
  com.mulesoft.tools.SecurePropertiesTool \
  string encrypt AES CBC <encryption-key> "ProductionDbP@ssw0rd"

# Output: ![kX9f2Q8...]
```

```
# src/main/resources/config-prod.yaml
db:
  host: "db.prod.internal"
  user: "mule_svc"
  password: "![kX9f2Q8...]"
```

Runtime decryption

```
<db:generic-connection password="${secure::db.password}"/>
```

```
# JVM arg or CloudHub property
mule.key=<encryption-key>
```

Per-environment keys

Environment	Encryption key	Config file
dev	dev-key-...	config-dev.yaml
test	test-key-...	config-test.yaml
prod	prod-key-...	config-prod.yaml

Never reuse prod key in dev. A dev artifact with prod ciphertext decrypted by a leaked dev key is a common mistake.

Limits of Secure Properties

- Not a rotation system — re-encrypt and redeploy to change values
- Key in `mule.key` is itself a secret — must live in vault/Runtime Manager, not Git
- Not for high-volume runtime data encryption — config only
- Team members with `mule.key` + artifact can decrypt offline — access control matters

When to use Secure Properties vs. Vault

Scenario	Secure Properties	Vault
DB password in Mule app	(with key in Runtime Manager)	(preferred for rotation)
50 connector credentials	Gets unwieldy	
Dynamic DB creds		
CI/CD pipeline secrets		
TLS keystore passwords		

Hybrid (common): Secure Properties for non-rotating connector config bundled in the deployable; Vault for database passwords and API keys that rotate.

9.7 Secret rotation without downtime

Static secret rotation

```
1. Generate new secret in vault (version 2)
2. Configure downstream to accept BOTH old and new (dual-write period)
3. Update consumers to use new secret
4. Revoke old secret after TTL buffer
5. Remove dual-accept on downstream
```

For databases: create second DB user with same grants, update Mule config, deploy, drop old user.

Automated rotation (AWS Secrets Manager example)

Lambda rotates RDS password updates secret triggers Mule app refresh (or next connection pool recycle). Requires app that reconnects on auth failure or periodic pool refresh.

Certificate rotation

Automate with cert-manager (K8s) or CDN ACME. For internal certs, calendar alerts at T-30 days.

9.8 Configuration security beyond secrets

Environment separation

```
config-dev.yaml    → dev secrets, dev endpoints
config-test.yaml   → test secrets, test endpoints
config-prod.yaml   → prod secrets, prod endpoints
```

Deploy pipeline selects profile by environment. **Block prod profile deploy to non-prod workers** and vice versa.

Feature flags vs. secrets

Use a feature flag service (LaunchDarkly, Unleash, cloud-native) for toggles. Don't hide "secret" feature enablement in obfuscated config — it's not security.

Infrastructure as Code secrets

Terraform, ARM, CloudFormation: never plaintext in `.tf` files. Use:

```
# Terraform — reference vault
data "azurerm_key_vault_secret" "db" {
  name         = "mule-db-password"
  key_vault_id = azurerm_key_vault.prod.id
}
```

Scan IaC with **Checkov**, **tfsec**, or **Snyk IaC** in CI (Chapter 14).

9.9 Access control and audit

Role	Vault access
Developer	Read dev secrets only
CI/CD pipeline	Read deploy-target secrets; short-lived token
Mule runtime identity	Read prod app secrets only
Platform admin	Break-glass prod; MFA + ticket required
Auditor	Read audit log, not secrets

Enable **audit logging** on Vault and cloud KMS. Log: who, what path, when, success/fail. Ship to SIEM (Chapter 12).

9.10 Secrets checklist

```
STORAGE
[ ] No plaintext secrets in Git (scan with git-secrets, trufflehog, GitHub secret scanning)
[ ] Central vault (Vault, AWS SM, Azure KV) for all environments
[ ] Per-environment keys and credentials — no sharing dev/prod

MULESOFT
[ ] Secure Properties for bundled config; mule.key in Runtime Manager / vault [
] ${secure::} prefix for all sensitive properties
[ ] Separate encryption keys per environment

ROTATION
[ ] Rotation schedule documented (90 days for API secrets, cert expiry automated)
[ ] Rotation tested in lower environment
[ ] Break-glass procedure for emergency revocation
```

```
CI/CD
[ ] Pipeline uses OIDC/short-lived tokens to vault — no long-lived keys in Jenkins
[ ] Secret values masked in build logs

GOVERNANCE
[ ] Audit log enabled on vault
[ ] Least-privilege policies per application identity
[ ] Offboarding revokes vault access same day
```

9.11 Break-glass and emergency access

When production is down and vault is unreachable, **break-glass** accounts exist — heavily controlled:

```
- Stored in physical safe or HSM-sealed envelope
- MFA + two-person rule to use
- Automatic alert to security team on use
- Mandatory review within 24 hours
- Credentials rotated immediately after incident
```

Break-glass is not a substitute for vault HA. Design vault for availability (HA cluster, cloud SLA).

9.12 Secrets in CI/CD — GitHub Actions and Jenkins patterns

GitHub Actions — OIDC to AWS (no static keys)

```
permissions:
  id-token: write
  contents: read
steps:
  - uses: aws-actions/configure-aws-credentials@v2
    with:
      role-to-assume: arn:aws:iam::123456789:role/github-mule-deploy
      aws-region: us-east-1
  - run: aws secretsmanager get-secret-value --secret-id prod/mule-key
```

Jenkins — Credentials Binding

```
withCredentials([string(credentialsId: 'mule-prod-key', variable: 'MULE_KEY')]) {
  sh 'mvn deploy -Dmule.key=$MULE_KEY'
}
```

Never `echo $MULE_KEY`. Use credential masking plugins.

9.13 Developer laptop hygiene

Secrets leak from laptops constantly:

```
[ ] Full-disk encryption (BitLocker, FileVault)
[ ] No prod secrets on developer machines — use dev/sandbox only
[ ] git-secrets or pre-commit hooks block commits with patterns
[ ] MDM enforces screen lock, remote wipe
[ ] VPN/ZTNA for access to internal tools — not for carrying prod DB passwords offline
```

Chapter 9 takeaways

- **Secrets never belong in Git** in plaintext — use vault + ciphertext or runtime injection.
- **Secure Properties** encrypts Mule config at rest; `mule.key` is itself a secret in Runtime Manager or vault.
- **Dynamic secrets** from Vault shrink blast radius for database access.
- **Rotate** with dual-credential windows to avoid downtime.
- **Audit** every secret read; least-privilege per workload identity.

Previous: [Chapter 8 — Transport Security](08-transport-encryption.md) · Next: [Chapter 10 — Zero Trust Architecture](10-zero-trust.md)

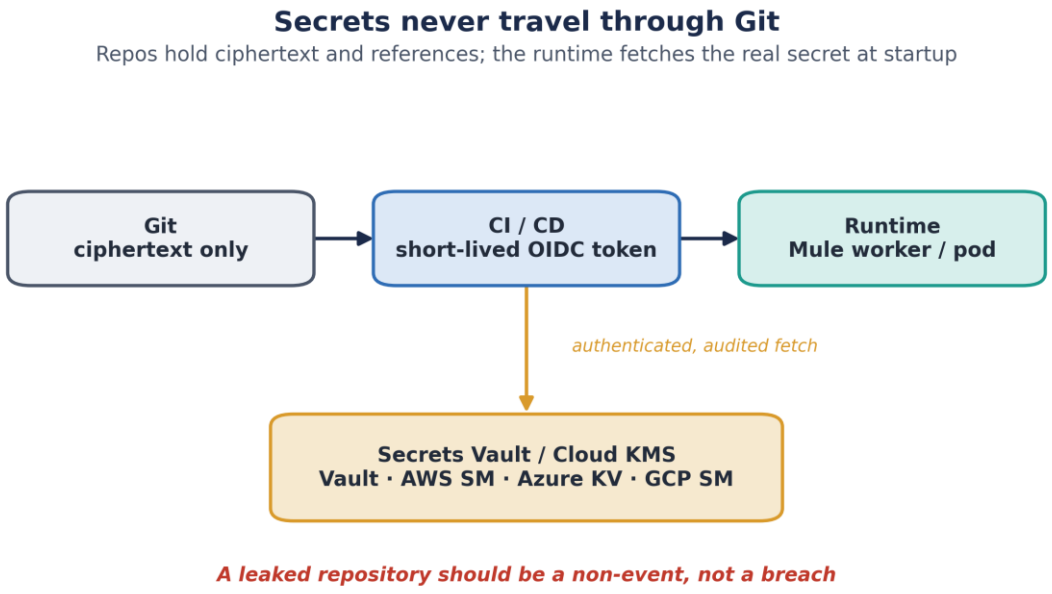


Figure 9-1. Secrets management — vault-mediated fetch.

Chapter 10
Zero Trust Architecture

“ "Never trust, always verify." That phrase is on every vendor slide — and most implementations still look like a VPN with extra steps. This chapter makes NIST SP 800-207 practical for API-led enterprises: what zero trust actually changes in your architecture, and how it maps to the gateway, mTLS, and identity patterns from Part I.

10.1 What zero trust is — and what it isn't

Zero trust is a security model that assumes **no implicit trust** based on network location. Being "inside the corporate network" does not mean you're trusted. Every access request is authenticated, authorized, and encrypted based on identity and context.

It is not:

- Rip out your firewall and buy one product
- "Zero trust network access" (ZTNA) client VPN replacement only
- A project with an end date

It is:

- Identity as the perimeter
- Least-privilege access continuously evaluated
- Microsegmentation so breach doesn't mean lateral free roam
- Assume breach — design so compromise is contained

NIST SP 800-207 (published 2020) defines three core principles:

- **All data sources and computing services are resources.**
- **All communication is secured regardless of network location.**

• **Access to resources is granted per session**, based on identity, device state, and context — with least privilege.

10.2 The old model vs. zero trust

Castle and moat (legacy)

Once an attacker is inside (phishing, compromised laptop, supply chain), they move laterally with minimal friction.

Zero trust (target state)

Every request — from employee laptop, partner API, or microservice — goes through policy evaluation. Network segment helps but doesn't grant trust.

10.3 NIST logical components — mapped to your stack

NIST component	Role	Your implementation
Policy Engine (PE)	Decides allow/deny	API Gateway policies, IdP authz, service mesh
Policy Administrator (PA)	Sets up communication	Gateway routing, mesh config, ZTNA broker
Policy Enforcement Point (PEP)	Enforces decision	Gateway, sidecar proxy, WAF
Identity Manager	Creates/attributes identity	Okta, Azure AD, Ping
Device posture	Health of requesting	MDM integration, Conditional Access
Data access policy	Rules for data	ABAC, scopes, BOLA checks

Your **API Manager front end** is a PEP. Your **IdP** is the identity manager. Your **JWT validation + scope policies** are the policy engine for APIs.

10.4 Zero trust for APIs — the practical slice

APIs are how modern enterprises expose capability. Zero trust for APIs means:

1. No anonymous internal APIs

"Internal" doesn't mean "no auth." Every service-to-service call uses **mTLS or workload identity** (SPIFFE) plus optional user context propagation.

```
External:  JWT validated at gateway
Internal:  mTLS between services + signed internal context header
```

2. Default deny

New API is **not routable** until registered in API Manager with policies attached (Chapter 3, §3.5 automated policies). Firewall default deny between segments.

3. Least privilege scopes

Client gets minimum scopes: `read:accounts` not `admin:*`. Review quarterly.

4. Continuous validation

Short token TTL. Re-check device posture for sensitive ops (Conditional Access). Session risk signals from IdP (impossible travel, new device).

5. Encrypt everything

TLS 1.2+ external; mTLS internal (Chapter 8). No "trusted VLAN" with plain HTTP.

10.5 Microsegmentation

Split the network into small zones; restrict traffic between zones to required ports and identities.

Implementation:

- Cloud: security groups / NSGs / VPC subnets per tier
- On-prem: firewall rules + VLANs
- Kubernetes: **NetworkPolicies** denying all except labeled pods
- Service mesh: traffic only through sidecar with mTLS

Microsegmentation **slows** lateral movement; it doesn't replace identity checks.

10.6 Workload identity — SPIFFE and SPIRE

SPIFFE (Secure Production Identity Framework for Everyone) defines a standard for workload identity: a **SPIFFE ID** like `spiffe://yourorg.com/ns/prod/sa/payments-api`.

SPIRE is the reference implementation that issues **short-lived SVID certificates** (often 1 hour) to workloads — pods, VMs, bare metal.

```
Pod starts → SPIRE agent → attests workload (K8s SA, node attestation)
       → Issues X.509 SVID → Sidecar uses for mTLS to other services
```

Why it matters for zero trust:

- No long-lived shared certificates in config
- Identity is bound to the workload, not the IP
- Automatic rotation

SPIRE is production-ready and integrated with Istio, Envoy, and HashiCorp Consul. Mule on Kubernetes can participate via sidecar mesh (Chapter 15).

10.7 Identity-aware proxy (IAP) and ZTNA

ZTNA products (Zscaler Private Access, Palo Alto Prisma Access, Cloudflare Access) replace "VPN = full network access" with "proxy evaluates identity per application."

Pattern:

```
User → ZTNA client → Policy check (IdP + device) → Specific internal app only
```

For **API developer access** to internal tools (Anypoint, Swagger UI in non-prod), use IAP instead of broad VPN. Reduces attack surface.

10.8 Conditional Access (device and context)

Microsoft Entra ID (Azure AD), Okta, and Ping support **Conditional Access** policies:

- Require MFA
- Require managed/compliant device
- Block legacy auth
- Block locations outside allow-list
- Require approved client app

Map to APIs by having the IdP embed `acr` / `amr` claims in tokens; gateway rejects tokens that don't meet bar for sensitive routes.

```
# Pseudocode: gateway conditional policy
if path matches "/payments/*" and token.acr != "mfa" → 403
```

10.9 Zero trust maturity — realistic stages

Stage	Characteristics
1 — Traditional	VPN, perimeter firewall, trust internal network
2 — Hybrid	SSO everywhere, API gateway with JWT, some mTLS
3 — Advanced	Automated policies, microsegmentation, vault for secrets
4 — Optimized	SPIFFE/mesh, continuous authz, device posture, ZTNA replacing VPN

Most enterprises are **stage 2–3**. Stage 4 is aspirational for regulated leaders. Don't let perfect block good — **stage 2 done well** beats **stage 4 on a slide deck**.

10.10 Applying zero trust to the reference architecture

Re-read Chapter 6's diagram through a zero-trust lens:

Components	Zero-trust control
CDN	TLS, DDoS, origin lockdown — not trust, but availability
API Gateway	PEP: JWT, scopes, rate limits, schema validation
Experience API	Coarse authz; no trust of client input

Component	Zero-trust control
Process API	BOLA; mTLS from gateway; circuit breakers
System API	mTLS; parameterized queries; DTOs
Database	No direct access from DMZ; credentials from vault
IdP	Central identity; MFA; Conditional Access
Employees	SAML/OIDC to consoles only — not API backdoors

Remove any path that bypasses the gateway ("temporary" firewall rules, direct DB access from dev laptops in prod).

10.11 Zero trust checklist

```
IDENTITY
[ ] Every human access via SSO + MFA
[ ] Every API access via OAuth2/JWT or mTLS — no "internal = open"
[ ] Conditional Access for sensitive apps and API scopes

NETWORK
[ ] Microsegmentation: gateway / app / data tiers separated [ ]
Default deny between segments; allow-list only required ports [ ]
No flat "trusted" corporate network for servers

WORKLOAD
[ ] mTLS or SPIFFE between services
[ ] Short-lived credentials; no shared service passwords

POLICY
[ ] API automated policies on all prod APIs (JWT, rate limit, audit)
[ ] Least-privilege scopes per client application
[ ] Default deny for new APIs until registered and policy-bound
```

```
ASSUME BREACH
[ ] Lateral movement limited by segmentation + workload identity
[ ] Audit logs to SIEM (Chapter 12)
[ ] Incident runbook tested
```

Chapter 10 takeaways

- **Zero trust** = no implicit trust by network location; verify identity and context per request.
- Your **API gateway** is a Policy Enforcement Point; **JWT + mTLS** are the core mechanisms.
- **Microsegmentation** and **workload identity (SPIFFE)** contain breach blast radius.
- **ZTNA/IAP** replaces broad VPN for human access to internal tools.
- Aim for **stage 2–3 done well** before chasing vendor "zero trust platform" completeness.

Previous: [Chapter 9 — Secrets Management](09-secrets-management.md) · Next: [Chapter 11 — API Threat Protection](11-api-threat-protection.md)

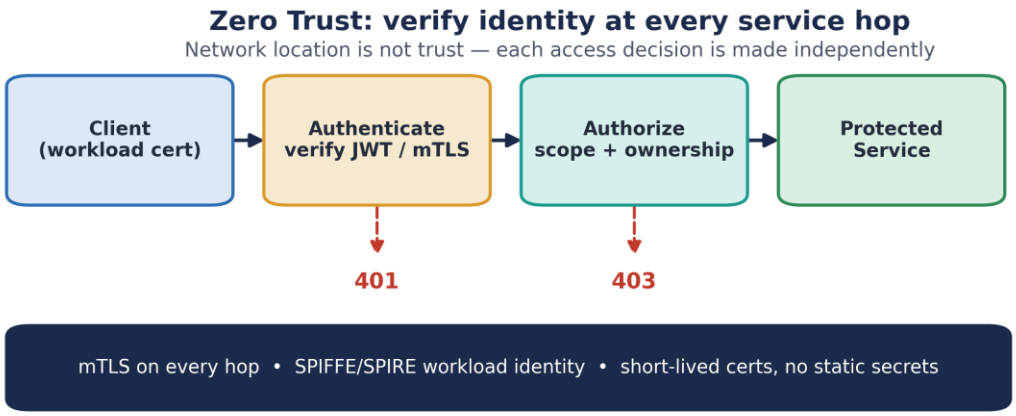


Figure 10-1. Zero Trust — every hop independently verified.

Chapter 11
API Threat Protection

“ Rate limiting stops abuse; this chapter stops malice. Input validation, schema

enforcement, WAF tuning, bot management, and security headers — the controls that

address OWASP API8 (injection), API6 (mass assignment), API3 (excessive data

exposure), and API7 (misconfiguration) at the front end and in application code.

11.1 Defense in depth for application threats

```
Layer 1: Edge WAF           — known signatures, rate-based rules, geo blocks
Layer 2: Gateway policies   — schema validation, size limits, threat protection
Layer 3: API implementation — parameterized queries, DTOs, BOLA
Layer 4: Data tier          — least-privilege DB users, encryption
```

No layer alone is sufficient. WAF misses novel payloads; app code without gateway validation gets overwhelmed by garbage traffic before it can reject.

11.2 Schema validation as a security contract

An API contract (OpenAPI 3.0 / RAML 1.0) is not documentation — it's a **security boundary**. The gateway enforces it before the payload touches business logic.

What to enforce

Check	Blocks
Required fields present	Incomplete attacks, parser confusion
Types correct (string, integer, enum)	Type confusion injection
maxLength, minLength	Buffer-style abuse, oversized payloads
pattern (regex)	Characters that enable injection
additionalProperties: false	Mass assignment of unknown fields
maxItems on arrays	Billion-laughs style expansion

Check	Blocks
Content-Type matches body	MIME confusion

Example OpenAPI constraint

```
paths:
 /accounts/{id}:
  patch:
      requestBody:
      content:
          application/json:
          schema:
            type: object
            additionalProperties: false
            properties:
                displayName:
                  type: string
                  maxLength: 100
                  pattern: '^[a-zA-Z0-9 \-\.]+$'
                preferredLanguage:
                  type: string
                  enum: [en, es, fr]
            # note: no "role", no "accountId" — mass assignment blocked
```

"Figure 11.1 — Screenshot: API Manager Add policy Security Request Validation (or Schema Validation depending on 4.4.x policy name). Upload RAML/OAS or paste JSON Schema; enable Validate request headers and Validate request body.

MuleSoft: Request Validation policy

Reject with `400 Bad Request` and a generic error body — don't echo which field failed in prod (helps attackers iterate).

11.3 Injection family — mechanics and fixes

SQL injection

Attack: `' OR '1'='1` in a query parameter.

Fix: Prepared statements only.

```
// WRONG
stmt.execute("SELECT * FROM users WHERE id = '" + input + "'");

// RIGHT
PreparedStatement ps = conn.prepareStatement("SELECT * FROM users WHERE id = ?");
ps.setString(1, input);
```

Mule Database connector: use parameterized queries in SQL or DataWeave `#[payload.id]` binding — never string concat.

NoSQL injection

Attack: `{"$gt": ""}` in JSON body to MongoDB query.

Fix: Validate schema (reject `$` keys in user input); use typed query APIs; never pass raw JSON to query builder.

Command / OS injection

Attack: `; rm -rf /` in filename passed to shell.

Fix: Never call shell with user input. Use library APIs. Allow-list filenames.

LDAP injection

Attack: `*)(uid=*))(|(uid=*` in login filter.

Fix: Escape LDAP special chars; parameterized LDAP APIs.

XML / XXE

Attack: External entity in XML body references `file:///etc/passwd`.

Fix: Disable external entities in XML parser (Mule XML module: secure processing features). Prefer JSON APIs.

Header injection

Attack: CRLF in header value response splitting.

Fix: Reject `\r` and `\n` in header inputs; validate redirect URLs.

11.4 Web Application Firewall (WAF) — tuning for APIs

WAFs (AWS WAF, Azure WAF, Cloudflare, F5, Imperva) sit at the edge and match request patterns against rule sets.

Managed rule groups

- OWASP Core Rule Set (CRS) 3.x
- Vendor bot management
- Known bad inputs, SQLi, XSS signatures

API-specific WAF strategy

Do	Don't
Enable OWASP CRS in detection first	Enable block mode day one without tuning
Exclude JSON false positives on known-good	Disable WAF on /api/* "because it breaks"
Rate-based rules (IP > N req/min → block)	Rely on WAF alone for SQLi
Log every blocked request to SIEM	Return verbose WAF error to client

Common API false positives

- Large JSON payloads with base64 (look like XSS)
- SQL keywords in legitimate JSON (`"description": "SELECT your plan"`)•
  Special characters in OAuth tokens in headers

Tune: create exclusions for specific parameter paths after reviewing blocked requests in count mode.

```
// AWS WAF — rate-based rule (example)
{
  "Name": "ApiRateLimit",
  "Statement": {
    "RateBasedStatement": {
      "Limit": 2000,
      "AggregateKeyType": "IP"
```

```
    }
  },
  "Action": { "Block": {} }
}
```

11.5 Bot management

Not all automated traffic is malicious — but credential stuffing, scraping, and L7 DDoS are.

Common techniques:

Technique	What it does
JavaScript challenge	Browser must execute JS; simple scripts fail
CAPTCHA (hCaptcha, reCAPTCHA)	Human verification on suspicious login
Device fingerprinting	Score client consistency
Behavioral analysis	Request timing, mouse patterns (web)
TLS fingerprinting	JA3/JA3S — bots often have distinct TLS handshakes

Apply bot management on:

- Login / token endpoints
- Account creation
- Password reset
- High-value read APIs under scrape attack

API **machine clients** (partner M2M) bypass via client credentials + IP allowlist — don't challenge known good bots.

11.6 Security headers

HTTP response headers that harden clients:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
X-Content-Type-Options: nosniff
X-Frame-Options: DENY
Content-Security-Policy: default-src 'none'; frame-ancestors 'none'
Cache-Control: no-store                   # for sensitive API responses
Referrer-Policy: strict-origin-when-cross-origin
```

For **JSON APIs** consumed by mobile/server clients, CSP matters less than `Cache-Control: no-store` on authenticated responses and `nosniff`.

MuleSoft: set via **HTTP Headers policy** or in API implementation response transformers.

```
policy: http-headers
configuration:
  responseHeaders:
    - key: X-Content-Type-Options
      value: nosniff
    - key: Cache-Control
      value: no-store
```

11.7 Payload and complexity limits

Limit	Typical value	Stops
Max body size	1 MB (adjust per API)	Large payload DoS
Max URL length	2 KB	Log abuse
Max header size	8 KB	Header bomb
GraphQL depth	10 levels	Recursive query abuse
GraphQL complexity score	budget per query	Expensive resolver chains
XML entity expansion	disabled	Billion laughs
Request timeout	5–30 s	Slowloris at app layer

Configure at gateway first (cheap reject), reinforce at server.

11.8 Response security — stopping data leakage

Error handling

```
// PROD — generic
{ "error": "invalid_request", "correlation_id": "abc-123" }
```

```
// NOT prod — don't expose
{ "error": "org.postgresql.util.PSQLException: column \"ssn\"..." }
```

Map internal exceptions to stable external codes. Log details server-side with correlation ID.

Response filtering

System layer returns DTO with 5 fields, not 50-column DB row (Chapter 2, API3). Optionally validate response against output schema at gateway if policy supports it.

11.9 API threat protection checklist

```
INPUT
[ ] OpenAPI/RAML schema enforced at gateway
[ ] additionalProperties: false on mutating endpoints
[ ] Max body/header/url limits configured
[ ] Parameterized queries — zero string-concat SQL

WAF / EDGE
[ ] OWASP CRS enabled; tuned from detection to block
[ ] Rate-based WAF rules on public APIs
[ ] Bot management on auth endpoints

HEADERS
[ ] HSTS, nosniff, no-store on sensitive responses
[ ] No verbose errors in production

CODE
[ ] XXE disabled in XML parsers
[ ] OS/command injection paths eliminated
[ ] Output DTOs; no raw entity serialization

MONITORING
[ ] WAF blocks logged to SIEM
[ ] Spike in 400/403 correlated with attack patterns
```

11.10 OWASP API Security Top 10 — control mapping (2019)

Map each risk to your front-end and app controls for design reviews:

OWASP API	Primary control (this book)	Layer
API1 BOLA	Object-level authz in process API	Application
API2 Broken auth	JWT validation, MFA, PKCE	Gateway + IdP
API3 Excessive data exposure	Output DTOs, response schema	Application
API4 Rate limiting	Rate Limiting + Spike Control policies	Gateway
API5 Broken function auth	Scope checks per route	Gateway + app
API6 Mass assignment	Schema additionalProperties: false	Gateway
API7 Misconfiguration	Automated policies, hardened defaults	Platform
API8 Injection	Prepared statements + WAF + schema	App + edge
API9 Improper assets mgmt	API catalog, decommission zombies	Governance
API10 Insufficient logging	Message Logging + SIEM	Gateway + ops

Bring this table to every security review. Blank cells are gaps.

11.11 Content-Type and MIME confusion attacks

Attack: Send `Content-Type: application/json` with XML body (or vice versa) to confuse parser and bypass validation.

Defense:

- Reject requests where `Content-Type` doesn't match body parser used
- Strict allow-list: `application/json`, `application/xml` only where needed
- Reject missing `Content-Type` on mutating requests

```
# Gateway policy concept
allowedContentTypes:
  - application/json
  - application/xml   # only if API supports XML
```

11.12 CORS — often misconfigured, occasionally exploited

Cross-Origin Resource Sharing is a browser control, not server-to-server. But misconfiguration enables token theft via malicious sites.

```
# DANGEROUS
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

# BETTER — explicit origin allow-list
Access-Control-Allow-Origin: https://app.yourorg.com
```

APIs consumed only by mobile and server clients may not need CORS at all. If you don't need browser cross-origin access, don't enable it.

11.13 Penetration testing scope for APIs

Annual pen test for critical APIs should include:

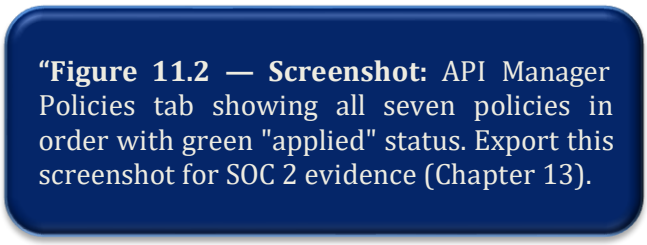
```
[ ] BOLA on every object-ID endpoint
[ ] Auth bypass (no token, bad token, wrong aud, expired)
[ ] Injection on all text inputs
[ ] Rate limit bypass (distributed IPs, header rotation)
[ ] Mass assignment on PATCH/POST
[ ] Information disclosure in errors and OPTIONS
[ ] HTTP method tampering (PUT on read-only resource)
[ ] Shadow API discovery (version fuzzing, path brute)
```

Provide testers: OpenAPI spec, test accounts per role, staging mirroring prod policies.

11.14 API threat protection in MuleSoft — policy stack example

Full **security** policy order for a regulated external API:

```
1. IP Blocklist (known bad actors)
2. JWT Validation
3. Rate Limiting - SLA-based
4. Spike Control
5. Request Validation (RAML/OAS)
6. HTTP Headers (security headers on response)
7. Message Logging (audit metadata)
```



Chapter 11 takeaways

- **Schema validation** at the gateway is a security control, not optional documentation.
- **Injection** is fixed in code (prepared statements); WAF is depth.
- **Tune WAF** in detection before block; exclude API false positives surgically.
- **Bot management** on login and abuse-prone endpoints; allow-list M2M partners.
- **Generic errors** externally; **correlation IDs** internally.

Previous: [Chapter 10 — Zero Trust](10-zero-trust.md) · Next: [Chapter 12 — Observability, SIEM & Incident Response](12-observability-incident-response.md)

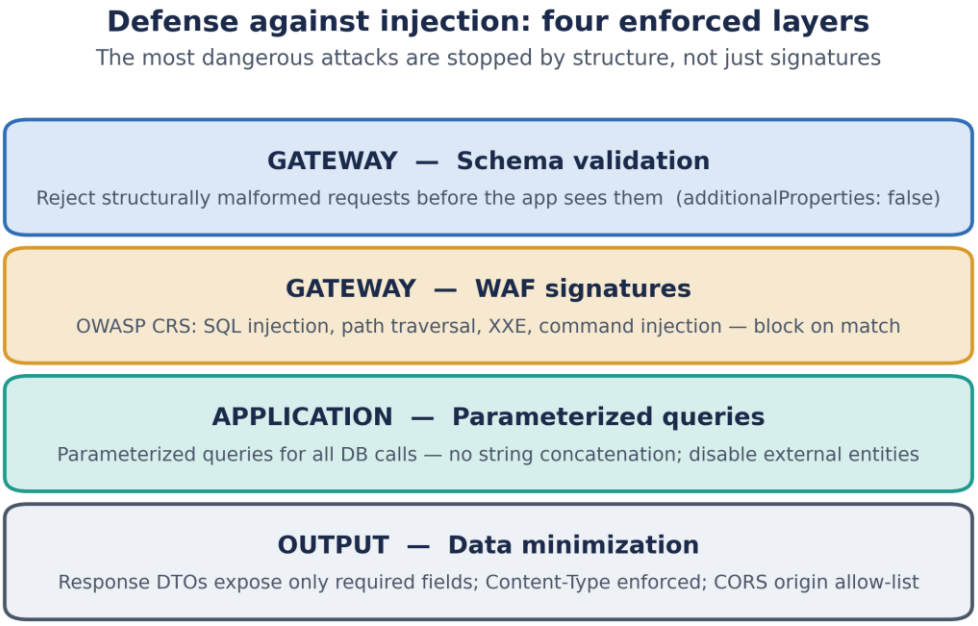


Figure 11-1. Four enforced layers of injection defense.

Chapter 12
Observability, SIEM & Incident Response

" You cannot defend what you cannot see. Chapter 5 configured Message Logging; this chapter builds the full picture: what to log, what to metric, how to get it into a SIEM, how to detect attacks, and how to respond when the pager fires at 2 a.m. without making everything worse.

12.1 The three pillars — applied to API security

Pillar	Security use	Tools
Logs	Audit trail, forensics, compliance	Splunk, Elastic, Azure Sentinel, Datadog
Metrics	Detection, SLOs, rate anomalies	Prometheus, CloudWatch, Anypoint Monitoring
Traces	Attack path reconstruction, latency	Jaeger, Zipkin, OpenTelemetry, Anypoint Visualizer

They complement each other. Logs tell you *what happened*; metrics tell you *something changed*; traces tell you *where in the chain it broke*.

12.2 Logging strategy — log security events, not everything

What to log (every API request metadata)

```
timestamp, correlation_id, client_id, user_sub (from JWT),
method, path, status_code, duration_ms, source_ip,
policy_violation (rate_limit, jwt_fail, schema_fail),
environment, api_version
```

What NOT to log

- Full `Authorization` header or JWT body

- Request/response bodies with PII, PAN, passwords
- Decrypted secure properties
- Session tokens, refresh tokens

"Gotcha. "Log everything for debug" in prod is how credentials end up in Splunk with a 7-year retention. Structured metadata only; body logging behind a time-boxed debug flag with automatic redaction.

Correlation ID

Generate at gateway (or accept `X-Correlation-ID` from client if trusted); propagate through every downstream call via header. Every log line includes it. Incident investigation starts with one ID.

```
<!-- Mule: set and propagate -->
<set-variable variableName="correlationId"
              value="#[attributes.headers.'x-correlation-id' default uuid()]"/>
<http:request headers="#[{'X-Correlation-ID': vars.correlationId}]"/>
```

Log format

Structured JSON — not grep-unfriendly plain text.

```
{
  "ts": "2024-06-15T14:32:01Z",
  "level": "INFO",
  "correlation_id": "f47ac10b-58cc-4372-a567-0e02b2c3d479",
 "event": "api_request",
  "client_id": "mobile-banking",
  "sub": "user-uuid-12345",
  "method": "GET",
  "path": "/accounts/100245/statements",
  "status": 200,
  "duration_ms": 142,
  "source_ip": "203.0.113.45"
}
```

12.3 Security-specific events to log

Event	Why
auth_failure (JWT invalid, expired, wrong aud)	Brute force, token theft, misconfig
authz_failure (403)	BOLA attempts, privilege escalation
rate_limit_exceeded (429)	Abuse, DDoS
schema_validation_failure (400)	Injection probes
waf_block	Attack signatures
admin_action (policy change, secret access)	Insider threat, audit
certificate_expiry_warning	Prevent outage

Alert on **spikes** and **new patterns**, not every single 401.

12.4 Metrics for security monitoring

Golden signals (adapted for APIs)

Metrics	Alert when
Request rate	>> baseline p99 (DDoS, scrape)
Error rate (5xx)	Sustained increase
401/403 rate	Spike vs baseline (attack, misdeploy)
429 rate	Sustained high (abuse or limit too low)
p99 latency	Degradation under load (L7 attack)
JWT validation failures	Spike
Downstream circuit breaker open	Dependency attack or failure
Cache hit ratio drop	Cache bypass attack (random query strings)

Any point Monitoring

"Figure 12.1 — Screenshot: Anypoint Monitoring API dashboard for a production API. Tiles: Request Count, Response Time, Errors, Policy Violations. Create custom alert: Policy Violations > 100 in 5 min PagerDuty webhook.

Export to enterprise observability via **Anypoint API Analytics** API or log drain to Splunk/Datadog agent on worker.

12.5 SIEM — centralizing and detecting

SIEM (Security Information and Event Management) aggregates logs, correlates events, and fires alerts.

Common enterprise options: Splunk Enterprise/Cloud, Microsoft Sentinel, IBM QRadar, Elastic Security, Sumo Logic.

Data sources to ingest

```
API gateway / Message Logging policy output
WAF logs (Cloudflare, AWS WAF)
IdP logs (login success/fail, MFA challenges)
CDN access logs
Vault audit log
Cloud audit (CloudTrail, Azure Activity Log)
Mule application logs (JSON)
```

Example detection rules

Rule name	Logic	Severity
API credential stuffing	>50 auth_failure from same IP in 5 min	High
BOLA probe	>20 403 on /accounts/* from one sub	Medium
DDoS indicator	request_rate > 3x baseline AND latency up	High
Token replay	same JWT jti from multiple IPs	High
Off-hours admin	policy change outside business hours	Medium
Geo anomaly	token used from 2 countries < 1 hour apart	Medium

Tune thresholds from **your** baseline — don't copy vendor defaults blindly.

MITRE ATT&CK mapping (optional maturity)

Map detections to techniques (T1110 Brute Force, T1190 Exploit Public-Facing Application) for coverage reporting in mature security programs.

12.6 Distributed tracing for security incidents

When investigating "user X saw wrong data" or "request took 30s," traces show the hop-by-hop path:

```
Gateway (2ms) → Experience API (45ms) → Process API (1200ms) → DB (1180ms)
```

OpenTelemetry (OTel) is the dominant tracing standard. Mule 4 supports tracing exporters; service mesh (Istio) adds automatic span propagation.

Security use: prove whether a request reached the system API without gateway auth (shadow path detection).

12.7 Incident response playbook

Phases (NIST SP 800-61)

```
1. Preparation      – runbooks, contacts, tools ready
2. Detection          – alerts, user report, pen test
3. Analysis           – scope, timeline, correlation ID
4. Containment        – block IP, tighten rate limits, revoke tokens
5. Eradication        – patch, remove shadow API, rotate secrets
6. Recovery           – restore service, monitor closely
7. Post-incident      – blameless review, control improvements
```

API-specific containment actions (fastest first)

Action	Time	Effect
CDN "under attack" / challenge mode	1–2 min	Shed L7 bots
Tighten rate limit in API Manager	2–5 min	No redeploy
IP block at WAF	2–5 min	Stop known source
Revoke client application secret	5 min	Stop compromised partner
Revoke user sessions / tokens at IdP	5–10 min	Stop stolen user access
Disable API route / policy deny-all	5 min	Nuclear option for one API

Action	Time	Effect
Rotate DB credentials from vault	15–30	Stop data exfil if creds leaked

"War story. We contained an L7 scrape by tightening SLA rate limit from 500 to 50/min in API Manager while the CDN team enabled challenge mode. No code deploy. Attack traffic dropped 85% in four minutes; legitimate mobile users unaffected because their app backed off on 429. Preparation beat heroics.

Incident communication template

```
SEV: [1-3]
Status: [Investigating | Contained | Resolved]
Impact: [which APIs, which customers]
Start time (UTC):
Actions taken:
Next update: [time]
Correlation IDs / sample: [for internal]
```

12.8 Retention and compliance

Log type	Typical retention
API access audit	1–7 years (regulated)
Security events	1 year minimum
Debug / trace	7–30 days
WAF raw	90 days

GDPR: logs with personal data (`sub`, IP) need lawful basis and retention limits; consider pseudonymization of `sub` in long-term archive.

PCI DSS 4.0: log access to cardholder data environment; retain 12 months, 3 months immediately available.

12.9 Observability checklist

```
LOGGING
[ ] Structured JSON logs with correlation_id on every request
[ ] No tokens, passwords, or PII bodies in logs
[ ] Message Logging policy on all prod APIs (metadata only)
[ ] Security events: auth_fail, authz_fail, rate_limit, schema_fail

METRICS
[ ] Dashboards: volume, latency, errors, 401/403/429, policy violations
[ ] Alerts on correlated spikes (volume + errors + latency)
[ ] Baseline established for anomaly detection

SIEM
[ ] Gateway, WAF, IdP, vault audit ingested
[ ] Detection rules for credential stuffing, BOLA probes, DDoS patterns
[ ] Runbook linked from alert notifications

INCIDENT RESPONSE
[ ] Written playbook with named roles
[ ] Containment actions tested (rate limit tighten, CDN challenge)
[ ] Post-incident review template
[ ] Tabletop exercise annually
```

12.10 Log aggregation architecture

A typical pipeline:

```
Mule worker → JSON log file / stdout
     → Fluent Bit / Filebeat agent
     → Kafka or cloud log ingest
     → Splunk / Elastic / Sentinel
     → Dashboards + detection rules
     → PagerDuty / Opsgenie on alert
```

CloudHub: configure log forwarding to your SIEM per MuleSoft documentation for your contract tier. **RTF/K8s:** DaemonSet collectors on every node.

Normalize field names (`source_ip`, not `clientIp` in one system and `remote_addr` in another) for cross-source correlation.

12.11 Tabletop exercise scenario — API credential leak

Run this annually with platform, security, and API owners:

```
Scenario: Partner client secret found in public GitHub repo at 14:00 UTC.
Facilitator asks:
  - Who revokes the secret? (API Manager admin)
  - Who notifies partner? (API owner)
  - Who checks SIEM for abuse since secret entered repo? (SOC)
  - Who rotates downstream credentials if partner had excessive scope? (Platform)
  - Who files incident ticket and post-mortem? (Incident commander)
Target: containment actions decided in < 15 minutes tabletop time.
```

Document gaps. Fix runbook. Retest next quarter.

12.12 SLA and SLO for security operations

Metric	Target (example)
Time to detect credential leak (automated)	< 1 hour
Time to revoke compromised client secret	< 30 min
Time to tighten rate limits under attack	< 5 min
Critical vulnerability patch (internet-facing)	< 7 days
Pen test finding remediation (high)	< 30 days

Security SLOs belong on the same dashboard as availability SLOs — executives see both.

Chapter 12 takeaways

- Log **metadata and security events**, not secrets or bodies.
- **Correlation IDs** across every hop — investigations depend on them.
- **SIEM rules** tuned to your baseline; alert on spikes and patterns.
- **Contain fast** with gateway/CDN controls before redeploying code.
- **Retention** meets compliance without hoarding PII forever.

Previous: [Chapter 11 — API Threat Protection](11-api-threat-protection.md) · Next: [Chapter 13 — Compliance & Governance](13-compliance-governance.md)

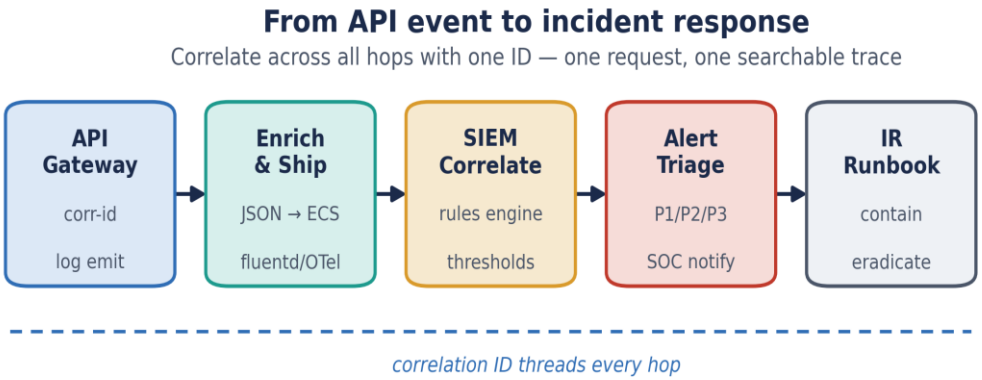


Figure 12-1. From API event to incident response.

Chapter 13
Compliance & Governance

“ Auditors don't care about your circuit breaker diagram. They care whether you can

show that controls exist, operate, and are monitored. This chapter maps the frameworks

you'll actually encounter — PCI DSS 4.0, GDPR, HIPAA, SOC 2, NIST CSF 1.1 — to the

technical patterns in this book, so one architecture satisfies many checklists.

13.1 The unified control model

Build **one set of controls**; map them to many frameworks. Duplicating per audit is how teams drown.

This book's control	PCI	GDPR	HIPAA	SOC 2	NIST CS
JWT + MFA	Req 8	Art 32	§164.312(d)	CC6.1	PR.AC
TLS 1.2+ / mTLS	Req 4	Art 32	§164.312(e)	CC6.7	PR.DS-
Rate limiting	Req 6, 11	Art 32	§164.312(a)	CC6.6	DE.CM
Encryption at rest	Req 3	Art 32	§164.312(a)(2)(iv)	CC6.1	PR.DS-
Logging + SIEM	Req 10	Art 30, 33	§164.312(b)	CC7.2	DE.AE
Secrets in vault	Req 3, 8	Art 32	§164.312(d)	CC6.1	PR.AC-
FIPS 140-2	Req 3, 4 (gov)	—	—	—	PR.DS
API inventory	Req 6, 12	Art 30	§164.316	CC3.2	ID.AM

13.2 PCI DSS 4.0

Payment Card Industry Data Security Standard version 4.0 is the version enterprises should now be planning and designing toward. Several of its requirements are phased in on future-dated effective dates, so confirm the applicable version and the effective date of each requirement with your assessor before you rely on it — the PCI SSC updates the standard periodically, and the version in force is whatever your acquiring bank and assessor hold you to.153

Scope reduction for APIs

Goal: cardholder data (CHD) never touches your API if possible.

```
Preferred:  Mobile app → tokenization provider → your API sees token only
Acceptable: API → encrypted CHD → PCI-scoped segment
Avoid:      API stores PAN in your database
```

Requirements that hit APIs directly

Req	Summary	Your implementation
3	Protect stored CHD	Tokenization; no PAN in logs
4	Encrypt transmission	TLS 1.2+; no SSL
6	Secure development	SAST/DAST, change control (Ch 14)
8	Identify and authenticate	MFA for admin; unique IDs
10	Log and monitor	SIEM, audit trails (Ch 12)
11	Test security	Pen test, vulnerability scans
12	Policies	API governance, risk assessment

Network segmentation (Req 1)

CDE (cardholder data environment) isolated. API in DMZ terminates TLS; only tokenization service in CDE talks to HSM.

13.3 GDPR

General Data Protection Regulation (EU 2016/679) — applies if you process EU residents' data, regardless of where servers sit.

Articles that shape API design

Article	Requirement	API implication
5	Data minimization	DTOs return minimum fields (API3)
6	Lawful basis	Document why each API processes data

Articl	Requirement	API implication
17	Right to erasure	Delete APIs; cascade to systems
20	Data portability	Export endpoint with standard format
25	Privacy by design	Security from architecture (Part I)
32	Security of processing	Encryption, resilience, testing
33	Breach notification	72 hours — IR playbook (Ch 12)
35	DPIA	High-risk processing needs assessment

Technical measures (Art 32)

"Pseudonymisation and encryption" — tokenize identifiers in non-prod; encrypt PII at rest (Chapter 8).

Logging: IP and `sub` in logs may be personal data — retention limits and access controls apply.

13.4 HIPAA Security Rule

Health Insurance Portability and Accountability Act — U.S. healthcare PHI.

Relevant safeguards

Safeguard	Type	API control
Access control	Administrative	RBAC, JWT scopes, BOLA
Audit controls	Technical	Message Logging, SIEM
Integrity	Technical	TLS, signatures, schema validation
Transmission security	Technical	TLS 1.2+, mTLS
Encryption	Addressable	Encrypt PHI at rest

BAA (Business Associate Agreement) required with cloud vendors processing PHI (AWS, Azure offer BAAs). MuleSoft on CloudHub: confirm BAA coverage with MuleSoft/Salesforce legal for your contract.

13.5 SOC 2 Type II

Service Organization Control 2 — auditor attestation that controls operate over time (6–12 month period). Customers request SOC 2 reports from SaaS and API providers.

Trust Services Criteria

Category	API relevance
Security (CC)	Access control, encryption, monitoring — core of this book
Availability (A)	DDoS, rate limiting, circuit breakers
Processing Integrity (PI)	Schema validation, idempotency
Confidentiality (C)	TLS, field encryption, least privilege
Privacy (P)	GDPR overlap — notice, retention

Your API platform team may need to **produce evidence**: policy screenshots, access reviews, pen test reports, change tickets for prod deploys.

13.6 NIST Cybersecurity Framework 1.1

The **NIST Cybersecurity Framework** remains one of the most widely adopted control taxonomies in enterprise security; this book maps to **CSF 1.1**, whose five functions are the vocabulary most organizations still use. (NIST continues to revise the framework, so verify which revision your security organization has standardized on — the function-level mapping below holds regardless of the point release.) CSF 1.1 organizes controls into five functions:

```
Identify  → asset inventory, API catalog, risk assessment
Protect   → access control, encryption, training
Detect    → SIEM, monitoring, anomaly detection
Respond   → incident playbook
Recover   → backups, DR, comms
```

Map your reference architecture (Chapter 6) to CSF for executive reporting. This book aligns to **CSF 1.1**; if your organization has adopted a later revision of the framework, the five-function structure and the control mappings here still carry over.

13.7 API governance — making compliance sustainable

API catalog / registry

Every API registered with:

- Owner and business purpose
- Data classification (public, internal, confidential, PCI, PHI)
- Auth model and scopes
- SLA tier and rate limits
- Linked policies in API Manager
- Deprecation date for old versions

“Figure 13.1 — Screenshot: Anypoint Exchange showing API assets with API version, status (published/deprecated), classification tag, and linked API Manager instance. Shadow APIs are what's not in this catalog.

Change management

Prod API policy changes and deployments through ticket + approval. Automated policies reduce drift but changes to automated policies still need review.

Access reviews

Quarterly: who has Anypoint org admin? Which client applications are still active? Revoke stale credentials.

Risk assessment

Annual (or on major change): threat model per critical API (STRIDE worksheet); link to controls.

13.8 Evidence collection for audits

Auditors ask for **artifacts**. Prepare:

```
[ ] Network diagram (reference architecture §6.1)
[ ] API inventory export from Exchange/API Manager
[ ] Automated policy screenshots (JWT, rate limit, logging)
[ ] Sample SIEM report: auth failures, admin actions
[ ] Penetration test executive summary (last 12 months)
[ ] Vulnerability scan results and remediation tickets
[ ] FIPS configuration evidence (wrapper.conf, negative test log) if applicable
[ ] Secrets management policy and vault audit sample
[ ] Incident response tabletop notes
[ ] SOC 2 report from cloud/IdP vendors
```

Automate evidence where possible — scheduled exports beat scrambling before audit week.

13.9 Compliance checklist

```
GOVERNANCE
[ ] API catalog complete; classification tags applied
[ ] Quarterly access and client credential reviews [ ]
Change management for prod policy and deploys [ ]
Annual threat model for critical APIs

PCI (if in scope)
[ ] CHD tokenized or scoped to minimal segment
```

```
[ ] TLS 1.2+; no PAN in logs
[ ] ASV scans and pen test current

GDPR (if EU data)
[ ] Data minimization in API contracts
[ ] Retention documented; erasure process tested
[ ] DPIA for high-risk processing
[ ] Breach notification procedure < 72 hours

HIPAA (if PHI)
[ ] BAA with processors
[ ] Audit logging on PHI access APIs
[ ] Encryption addressable implementation documented

SOC 2 / NIST
[ ] Controls mapped to Trust Services / CSF 1.1
[ ] Evidence repository maintained year-round
```

Chapter 13 takeaways

- **One control set**, many framework mappings — don't rebuild per audit.
- **PCI:** shrink scope via tokenization; TLS and logging are table stakes.
- **GDPR:** minimization, lawful basis, breach IR — shaped in API design.
- **HIPAA / SOC 2 / NIST CSF 1.1** align to the same technical controls in this book.
- **Governance** (catalog, access reviews, evidence) makes compliance repeatable.

Previous: [Chapter 12 — Observability & IR](12-observability-incident-response.md)
·Next: [Chapter 14 — Secure SDLC & DevSecOps](14-secure-sdlc-devsecops.md)

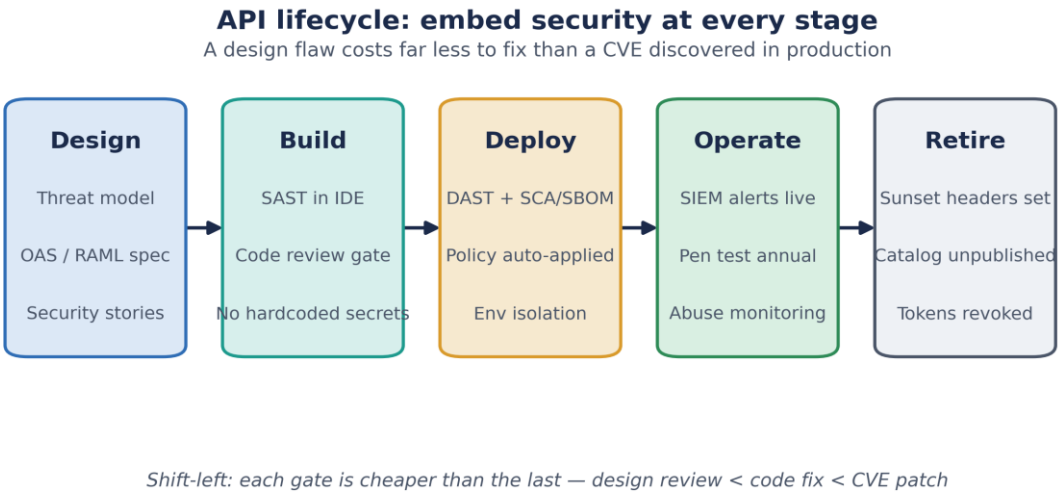


Figure 13-1. API lifecycle security gates.

Chapter 14
Secure SDLC & DevSecOps

" Security at the end of a release is a gate that everyone hates and nobody respects. Shift left — build security into design, code, build, and deploy so that unsafe changes are caught before they reach Production. This chapter covers threat modeling, SAST/DAST/SCA, supply chain security, secure CI/CD, and IaC scanning.

14.1 Shift left — what it means in practice

```
Traditional:  Design → Code → Test → Security review → Prod
                                          ↑
                                  bottleneck, finds bugs late

Shift left:   Threat model → Secure code → SAST in commit →
              SCA in build → DAST in staging → Policy gate → Prod
                    ↑                              ↑
              cheap to fix                  auto-block on critical
```

Cost curve: fixing a vulnerability in design costs hours; in production costs incidents, fines, and reputation.

14.2 Threat modeling — STRIDE for APIs

Do this **before** you build, for every new experience API and major change.

STRIDE

Threat	Question	Example API threat
Spoofing	Can attacker fake identity?	Forged JWT, stolen client secret
Tampering	Can data be modified in transit?	No TLS, weak integrity
Repudiation	Can actions be denied?	Missing audit logs
Information disclosure	Can data leak?	BOLA, excessive fields, verbose errors

Threat	Question	Example API threat
Denial of service	Can service be taken down?	No rate limit, expensive endpoint
Elevation of privilege	Can user gain admin?	Mass assignment, broken function auth

Lightweight process (1–2 hour workshop)

```
1. Draw data flow diagram (from Chapter 6 template)
2. List assets (data, credentials, availability)
3. Walk each STRIDE category per component
4. Map existing controls from this book
5. File backlog items for gaps
6. Attach diagram to API catalog entry (Chapter 13)
```

Common tools: Microsoft Threat Modeling Tool, OWASP Threat Dragon, IriusRisk (enterprise).

14.3 Secure coding standards

Publish a **short, enforceable** standard for integration developers:

```
AUTH
- Never implement custom crypto; use platform JWT policies and TLS
- Never log tokens or passwords

INPUT
- Parameterized queries only
- Validate all input against schema; additionalProperties: false

OUTPUT
- Return DTOs; never serialize ORM entities to API response
- Generic error messages in prod

SECRETS
- No secrets in Git; use Secure Properties or vault
- Rotate client secrets per policy

DEPENDENCIES
- No new library without SCA scan approval
```

Review in PR checklist — not a 50-page document nobody reads.

14.4 SAST — Static Application Security Testing

SAST analyzes **source code** without running it.

Tool	Languages	Integration
SonarQube	Java, JS, many	CI pipeline, quality gate
Checkmarx, Veracode	Enterprise multi-language	CI, policy gate
Semgrep	Custom rules, YAML	Fast PR scans
CodeQL (GitHub)	Multi	GitHub Actions

What SAST catches in Mule/Java projects

- SQL string concatenation
- Hard-coded passwords
- Weak crypto (`MD5`, `DES`)
- XPath injection
- Insecure deserialization

```
# GitHub Actions — Semgrep example
- uses: returntocorp/semgrep-action@v1
  with:
    config: p/owasp-top-ten
```

Quality gate: block merge on **critical/high** new findings; track medium in backlog.

14.5 DAST — Dynamic Application Security Testing

DAST attacks a **running** application in staging.

Tool	Notes
OWASP ZAP	Open source; CI baseline scan
Burp Suite Enterprise	Commercial; deep crawl
Invicti, Acunetix	Automated scanning

Run against **staging** that mirrors prod policies (JWT, rate limit). Schedule weekly; full scan before major releases.

API-specific DAST

Import OpenAPI spec into ZAP for targeted endpoint coverage. Test BOLA by swapping object IDs with another user's token.

14.6 SCA — Software Composition Analysis

90%+ of code is dependencies. SCA finds known CVEs in libraries.

Tool	Integration
Snyk	CI, IDE, repo monitor
OWASP Dependency-Check	Maven/Gradle CI
GitHub Dependabot	Automated PRs
Mend (WhiteSource)	Enterprise policy

Mule projects

Scan `pom.xml` transitive dependencies. Mule runtime bundles its own libs — also monitor **MuleSoft security advisories** for connector and runtime CVEs.

Policy: no critical CVE in prod without compensating control or patch plan within SLA (e.g. 7 days critical, 30 days high).

14.7 Supply chain security

High-profile incidents like the npm ecosystem compromises and Log4j sharply raised the stakes here.

Controls

```
[ ] Pin dependency versions in pom.xml / package-lock.json — no floating ranges in prod
[ ] Verify artifact checksums from Maven Central / corporate Nexus
[ ] Private artifact repository (Nexus, Artifactory) — don't pull direct from internet in prod build
[ ] Sign deployment artifacts; verify signature at deploy
[ ] SBOM (Software Bill of Materials) — CycloneDX export from CI (increasingly expected by customers)
[ ] Review new connectors and community modules before prod
```

Log4j lesson

Transitive dependencies hide vulnerabilities. SCA + rapid patch process matter more than perfect code review.

14.8 Secure CI/CD pipeline

Pipeline secrets

- CI authenticates to vault via **OIDC** (GitHub Actions AWS/Azure) — no long-lived keys in Jenkins env vars
- `mule.key` for prod encrypt step injected at deploy job only
- Mask all secret outputs in logs

Environment promotion

```
dev  → automatic on main branch (after SAST/SCA)
test → automatic or daily
prod → manual approval + change ticket + DAST green in staging
```

Separate credentials per environment — pipeline can't deploy prod artifact with dev `mule.key`.

14.9 Infrastructure as Code (IaC) scanning

Terraform, ARM, CloudFormation define networks, firewalls, IAM — misconfigurations are vulnerabilities.

Tool	Checks
Checkov	Terraform, ARM, K8s manifests
tfsec	Terraform
Snyk IaC	Multi-cloud

```
checkov -d terraform/ --framework terraform
# FAIL: Security group allows 0.0.0.0/0 on port 22
```

Scan IaC in CI before apply. Block public S3 buckets, open SSH, missing encryption flags.

14.10 Security testing in the API lifecycle

Phase	Activity
Design	Threat model (STRIDE), data classification

Build	SAST, SCA on every PR
Test	Integration tests for authz (BOLA negative cases)
Staging	DAST, load test rate limits
Pre-prod	Pen test for critical APIs (annual)
Prod	Continuous monitoring (Chapter 12), policy drift detection

Automated authz tests (example)

```
@Test
void bolA_userCannotAccessOtherAccount() {
  String aliceToken = tokenFor("alice");
  Response r = given().auth().oauth2(aliceToken)
    .get("/accounts/BOB_ACCOUNT_ID/statements");
  assertThat(r.statusCode()).isEqualTo(403);
}
```

14.11 DevSecOps roles

Role	Responsibility
API team	Secure code, schema, tests
Platform team	Automated policies, gateway baseline, FIPS
Security team	Standards, tool config, pen test, IR
SRE	Monitoring, incident runbook

Security team **enables** — provides templates, policies, gates — not sole gatekeeper on every deploy.

14.12 Secure SDLC checklist

```
DESIGN
[ ] Threat model for new/changed APIs documented
[ ] Data classification assigned in catalog

CODE
[ ] Secure coding standard in PR template
[ ] SAST on every build; block critical
[ ] SCA on dependencies; CVE SLA defined
[ ] No secrets in Git (pre-commit hook: git-secrets)

BUILD / DEPLOY
[ ] CI uses OIDC to vault — no static prod keys [
] Artifact signing; SBOM generated
[ ] Prod deploy requires approval + change record
[ ] IaC scanned before apply
```

```
TEST
[ ] DAST weekly on staging
[ ] BOLA negative tests in automated suite
[ ] Annual pen test for critical APIs

OPERATE
[ ] Policy drift detection (API without automated policies = alert)
[ ] Post-incident fixes tracked to closure
```

Chapter 14 takeaways

- **Shift left:** threat model early; automate SAST/SCA/DAST in pipeline.
- **SCA** is non-negotiable — most vulnerabilities are in dependencies.
- **Secure CI/CD:** OIDC to vault, signed artifacts, prod approval gates.
- **IaC scanning** prevents cloud misconfig before deploy.
- **BOLA tests** in CI prove authz works, not just authn.

Previous: [Chapter 13 — Compliance & Governance](13-compliance-governance.md) ·Next: [Chapter 15 — Microservices, Containers & Service Mesh Security](15-microservices-container-security.md)

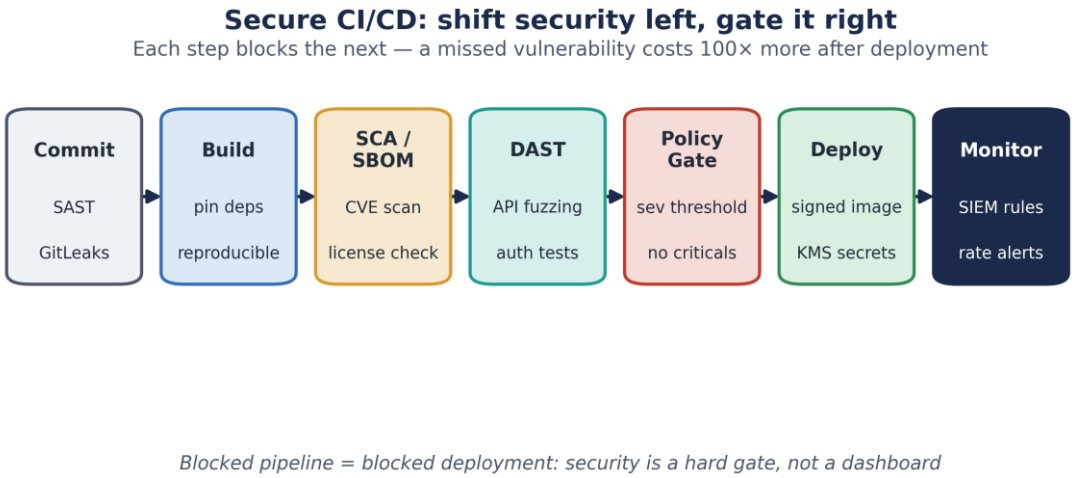


Figure 14-1. Secure CI/CD pipeline — shift left, gate right.

Chapter 15
Microservices, Containers & Service Mesh Security

“ Mule runtimes on CloudHub, Runtime Fabric, or Kubernetes — plus the microservices they call — need hardening beyond "we deployed in Docker." This chapter covers container image security, Kubernetes controls, and service mesh mTLS as the answer to zero-trust between services (Chapter 10), closing Part II and the book.

15.1 The runtime landscape

Enterprise integration deploys on:

Platform	MuleSoft option	Notes
SaaS PaaS	CloudHub	MuleSoft manages worker; limited OS access
Container PaaS	CloudHub 2.0, RTF on K8s	Your cluster or MuleSoft-managed
Kubernetes	RTF, self-managed Mule	Full K8s security applies
VM / bare metal	On-prem Mule standalone	OS hardening, wrapper.conf

Security responsibility **shifts right** as you move down — CloudHub abstracts the OS; on K8s you own the node, network policy, and pod security.

15.2 Container image security

Build practices

```
# GOOD patterns
FROM mule:4.4.0-java17    # pin digest when possible
USER mule                           # non-root
COPY --chown=mule:mule app.jar /
# NO secrets in ENV or COPY
```

```
BAD:  ENV DB_PASSWORD=prod123
BAD:  COPY keystore.jks /opt/secrets/   # bake secrets into image
BAD:  USER root
BAD:  FROM mule:latest                  # floating tag
```

Image scanning

Scan every image in CI with **Trivy**, **Snyk Container**, or **Aqua**:

```
trivy image yourorg/mule-payments-api:1.2.3 --severity HIGH,CRITICAL
```

Block deploy on critical OS/package CVE without patch plan.

Registry

Private registry (ECR, ACR, Harbor); immutable tags for prod; delete old vulnerable tags.

15.3 Kubernetes security fundamentals

RBAC

Least-privilege **ServiceAccounts** per namespace/app. No `cluster-admin` for app teams.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mule-payments-api
  namespace: integration-prod
```

NetworkPolicies — default deny

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: deny-all-ingress
 namespace: integration-prod
spec:
 podSelector: {}
 policyTypes: [Ingress, Egress]
---
# Allow only gateway → mule pods on 8081
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: allow-gateway-to-mule
spec:
```

```
  podSelector:
   matchLabels:
       app: mule-payments-api
  ingress:
   - from:
       - podSelector:
         matchLabels:
             app: api-gateway
       ports:
       - port: 8081
```

Pod Security

Pod Security Standards (replaced PodSecurityPolicy in K8s 1.25+):

- **restricted** profile for integration namespaces: no root, drop capabilities, read-only root filesystem where possible

```
apiVersion: v1
kind: Namespace
metadata:
  name: integration-prod
  labels:
    pod-security.kubernetes.io/enforce: restricted
```

Secrets in Kubernetes

Use **Kubernetes Secrets** backed by external secrets operator (ESO) syncing from Vault/AWS SM — not plaintext in Git manifests.

```
# External Secrets Operator pattern
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
 name: mule-db-password
spec:
 secretStoreRef:
  name: vault-backend
 target:
  name: mule-db-secret
 data:
  - secretKey: password
      remoteRef:
      key: secret/prod/mule/db
        property: password
```

Mount as volume env in Mule pod; rotate at source in Vault.

15.4 Runtime Fabric (RTF) — Mule on Kubernetes

MuleSoft **Runtime Fabric** deploys Mule workers into your K8s cluster with an RTF agent managing lifecycle.

Security considerations:

- RTF namespace isolated with NetworkPolicies
- Ingress only through corporate ingress controller / API gateway
- `mule.key` and connector secrets via K8s secrets from vault sync
- Worker pods use dedicated ServiceAccount; RTF agent has minimal RBAC
- Enable **Anypoint Monitoring** and log drain to SIEM

*"**Figure 15.1** — Screenshot: **Runtime Manager Fabrics (your fabric) Resources** showing worker allocation per environment. Verify **Production** workers run in a separate namespace from **Development**.*

15.5 Service mesh — mTLS everywhere

A **service mesh** (Istio, Linkerd, Consul Connect) injects a **sidecar proxy** (Envoy) alongside each pod. All traffic flows through sidecars; the mesh provides:

- **Automatic mTLS** between services
- **Traffic policy** (retries, timeouts — complements circuit breakers)
- **Observability** (metrics, traces)
- **Authorization policies** (L4/L7 allow rules)

Istio PeerAuthentication

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: integration-prod
spec:
  mtls:
    mode: STRICT    # reject plain text
```

AuthorizationPolicy — zero trust between services

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: mule-payments-only-from-gateway
 namespace: integration-prod
spec:
 selector:
  matchLabels:
      app: mule-payments-api
 action: ALLOW
 rules:
  - from:
      - source:
          principals: ["cluster.local/ns/ingress/sa/api-gateway"]
```

Only the gateway's service identity can call the payments Mule app — even inside the cluster.

Mule + mesh

Mule container listens on localhost; Envoy handles inbound mTLS and forwards plain HTTP to Mule on loopback — or configure Mule for mTLS directly if not using sidecar ingress to app.

15.6 SPIFFE/SPIRE with mesh (tie-in Chapter 10)

Istio 1.14+ integrates with **SPIRE** for workload attestation. SPIRE issues SVIDs; Istio uses them for certificate identity. Short-lived certs (hours) replace year-long cert rotation pain.

When to adopt: large K8s estates with many microservices and strict zero-trust mandates. Smaller Mule-on-RTF deployments often start with Istio **STRICT mTLS** and mesh CA before full SPIRE.

15.7 API gateway vs. service mesh

Concern	API Gateway (Chapter 3–5)	Service mesh
North-south (internet → cluster)	Primary	Ingress gateway (Istio Gateway)
East-west (service → service)	Optional	Primary
JWT validation	Yes	Usually at ingress only
Rate limiting	Yes	Basic; gateway better
mTLS internal	Manual config	Automatic
Developer ownership	Platform/API team	Platform/SRE

Pattern: Gateway handles external JWT, rate limits, WAF integration; mesh handles internal mTLS and fine service-to-service policy.

15.8 CloudHub vs. self-managed — security split

Control	CloudHub	Self-managed K8s/VM
OS patching	MuleSoft	You
TLS termination	Configurable	You
Network segmentation	Shared responsibility	You
FIPS workers	Request from account team	Your FIPS JVM
Compliance attestations	MuleSoft SOC 2	Your SOC 2 + vendor

For regulated workloads, **RTF on private K8s** or **hybrid** is common — data plane in your VPC, control plane in Anypoint.

15.9 Container and K8s checklist

```
IMAGE
[ ] Non-root user; no secrets in image layers [ ]
Pinned base image; scanned in CI (Trivy/Snyk) [ ]
Private registry; immutable prod tags

KUBERNETES
[ ] RBAC least privilege per namespace
[ ] NetworkPolicy default deny + explicit allow
[ ] Pod Security Standards (restricted) enforced
[ ] Secrets from vault via External Secrets Operator — not in Git

MULE / RTF
[ ] Prod namespace isolated from dev
[ ] mule.key via K8s secret / vault sync
[ ] Ingress only through corporate gateway

SERVICE MESH (if deployed)
[ ] STRICT mTLS between services
[ ] AuthorizationPolicy per critical workload
[ ] Traces exported to observability stack

GOVERNANCE
[ ] Node image / K8s version patch cadence
[ ] RTF/Mule runtime version aligned with security advisories
```

15.10 Closing Part II — the full stack

You now have the depth behind Part I's reference architecture:

Part I chapter	Part II depth
Ch 1–2 Mindset & threats	Ch 7 IAM, Ch 10 Zero Trust, Ch 11 Threat protection
Ch 3 Front-end patterns	Ch 11 Validation, Ch 12 Detection
Ch 4 DDoS	Ch 12 IR, Ch 11 WAF
Ch 5 MuleSoft/FIPS	Ch 8 Crypto, Ch 9 Secrets, Ch 15 Runtime
Ch 6 Reference arch	Ch 13 Compliance, Ch 14 SDLC, Ch 15 Mesh

Security is not one control — it's **layers of discipline** across identity, transport, secrets, policy, code, runtime, and governance. Maintain them the way you maintain uptime: continuously, with owners, with evidence.

Chapter 15 takeaways

- **Container images:** non-root, no baked secrets, scan in CI.
- **Kubernetes:** NetworkPolicies, Pod Security Standards, RBAC, external secrets.
- **Service mesh** delivers **automatic mTLS** and east-west zero trust.
- **Gateway + mesh:** gateway for north-south JWT and rate limits; mesh for internal identity.
- **RTF/CloudHub choice** shifts who owns OS, network, and compliance evidence.

15.11 Hardening checklist — Linux host and VM (on-prem Mule)

For Mule standalone on VM (still common in many enterprises):

```
[ ] OS patched within vendor SLA
[ ] SSH key-only; no root login
[ ] Firewall: only 443 from LB, management from jump host only
[ ] SELinux or AppArmor enforcing where supported
[ ] Separate disk for logs; log rotation configured
[ ] Mule runs as non-root dedicated user
[ ] Antivirus/EDR per corporate standard
[ ] CIS benchmark scan quarterly
```

15.12 Disaster recovery and security

DR sites must have **same security controls** as primary — not a soft target.

```
[ ] DR APIs behind same policy templates (exported from API Manager)
[ ] Secrets replicated to DR vault region
[ ] TLS certs valid for DR DNS names
[ ] Firewall rules synced
[ ] DR failover tested annually including auth and rate limit behavior
```

Previous: [Chapter 14 — Secure SDLC & DevSecOps](14-secure-sdlc-devsecops.md) · [Appendix A](appendix-a-glossary-standards.md)

End of book.

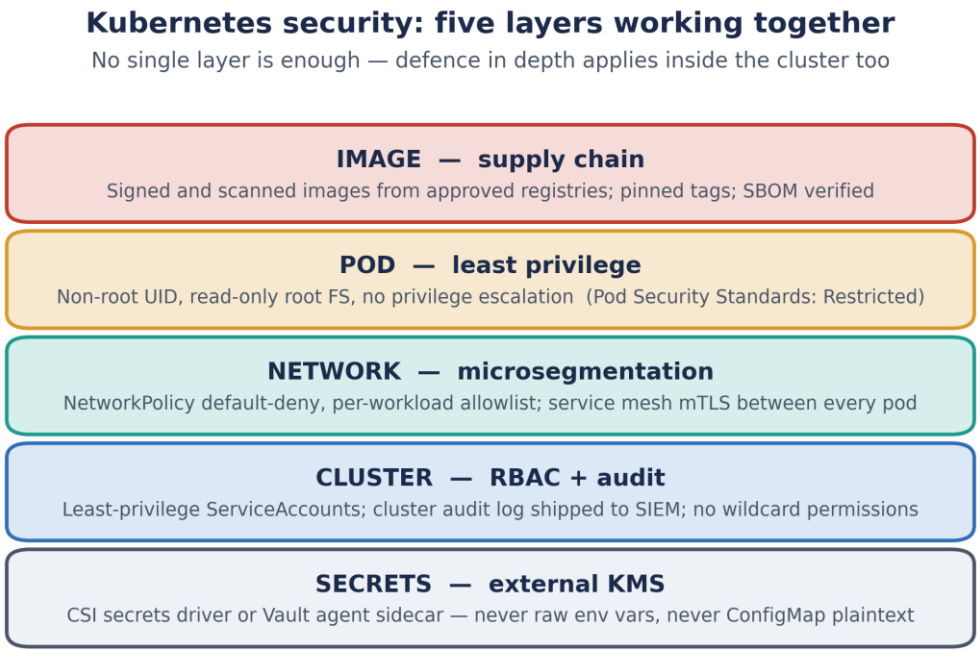


Figure 15-1. Kubernetes security layers.

Appendix A — Glossary, Standards Reference & Quick Sheets

" Reference material to bookmark for audits, onboarding, and design reviews.

A.1 Glossary

Terms	Definition
ABAC	Attribute-Based Access Control — decisions from user, resource, and environment
ACL	Access Control List
API Gateway	Managed front door enforcing policies before traffic reaches services
BFF	Backend for Frontend — server that holds tokens for SPAs
BOLA	Broken Object Level Authorization (OWASP API1)
CDE	Cardholder Data Environment (PCI)
CHD	Cardholder Data (PAN, CVV, etc.)
CORS	Cross-Origin Resource Sharing — browser cross-domain policy
CSRF	Cross-Site Request Forgery
DEK	Data Encryption Key
DDoS	Distributed Denial of Service
DTO	Data Transfer Object — explicit API input/output shape
FIPS 140-2	U.S. federal standard for cryptographic modules
HSM	Hardware Security Module
HSTS	HTTP Strict Transport Security
IAM	Identity and Access Management
IdP	Identity Provider
JWT	JSON Web Token
JWKS	JSON Web Key Set — public keys for JWT verification
KEK	Key Encryption Key (master/wrapping key)
mTLS	Mutual TLS — both sides present certificates
OIDC	OpenID Connect — identity layer on OAuth 2.0

Terms	Definition
OWASP	Open Web Application Security Project
PAN	Primary Account Number (payment card)
PCI DSS	Payment Card Industry Data Security Standard
PEP	Policy Enforcement Point (NIST zero trust)
PHI	Protected Health Information (HIPAA)
PII	Personally Identifiable Information
PKCE	Proof Key for Code Exchange (RFC 7636)
PKI	Public Key Infrastructure
RBAC	Role-Based Access Control
SBOM	Software Bill of Materials
SCA	Software Composition Analysis
SIEM	Security Information and Event Management
SOC 2	Service Organization Control 2 audit framework
SPIFFE	Secure Production Identity Framework for Everyone
SPIRE	SPIFFE Runtime Environment
SVID	SPIFFE Verifiable Identity Document
TLS	Transport Layer Security
WAF	Web Application Firewall
ZTNA	Zero Trust Network Access

A.2 Standards and RFCs referenced

Documents	Title / relevance
RFC 6749	OAuth 2.0 Authorization Framework
RFC 6750	OAuth 2.0 Bearer Token Usage
RFC 7636	PKCE
RFC 7519	JSON Web Token (JWT)
RFC 7662	OAuth 2.0 Token Introspection
RFC 8693	OAuth 2.0 Token Exchange
RFC 7009	OAuth 2.0 Token Revocation
OpenID Connect 1.0	Identity layer on OAuth 2.0
OWASP API Security Top 10 (2019)	API risk categories

Document	Title / relevance
OWASP Top 10 (2021)	Web application risks
NIST SP 800-207	Zero Trust Architecture
NIST CSF 1.1	Cybersecurity Framework
NIST SP 800-53 Rev 5	Security controls catalog
PCI DSS v4.0	Payment card security
FIPS 140-2	Cryptographic module validation
SAML 2.0	XML-based federation for SSO

A.3 Cipher suite quick reference (TLS 1.2 / FIPS-friendly)

```
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
```

TLS 1.3 (when FIPS mode allows in your stack):

```
TLS_AES_256_GCM_SHA384
TLS_AES_128_GCM_SHA256
```

A.4 HTTP status codes — security semantics

Code	Meaning	Security use
400	Bad Request	Schema validation failure
401	Unauthorized	Missing/invalid authentication
403	Forbidden	Authenticated but not authorized (BOLA)
404	Not Found	Optional: hide existence of resource
429	Too Many Requests	Rate limit / abuse
503	Service Unavailable	Circuit open, load shed, maintenance

Use **401** vs **403** consistently: 401 = "who are you?"; 403 = "I know who you are; you can't."

A.5 JWT claims reference

Claim	Name	Validate?
iss	Issuer	Yes — exact match
sub	Subject (user ID)	Yes — for audit
aud	Audience	Yes — must include this API
exp	Expiry	Yes
nbf	Not before	If present
iat	Issued at	Optional skew check
scope	OAuth scopes	Yes — per endpoint
client_id	Calling application	Log and rate-limit key
acr	Auth context (MFA)	For step-up routes
jti	JWT ID	Revocation lists

A.6 Master policy order reference card

Print and post in platform team wiki:

```
NORTH-SOUTH (external API)
  1. IP blocklist / allowlist
  2. JWT validation (iss, aud, exp, scope)
  3. Rate limiting (SLA-based)
  4. Spike control
  5. Request/schema validation
  6. HTTP caching (public reads only)
  7. Security response headers
  8. Message logging (metadata)

EAST-WEST (internal)
  1. mTLS or mesh STRICT mode
  2. Service authorization policy
  3. Timeouts + circuit breaker
  4. Audit log with correlation ID
```

A.7 Data classification guide

Class	Examples	Controls
Public	Marketing content, public rates	TLS, rate limit
Internal	Employee directory	SSO, internal network
Confidential	Customer PII, account data	JWT, encryption, audit
Restricted	PCI CHD, PHI	Tokenization, FIPS, segmentation

Tag every API in Exchange with one class. Policies escalate with class.

A.8 Incident severity matrix

SEV	Criteria	Example
1	Active breach, prod down, CHD/PHI exposure	Live data exfiltration
2	Major degradation, attack contained but ongoing	DDoS mitigated, investigating
3	Minor impact, single API, workaround exists	One partner rate limited incorrectly
4	Cosmetic, no security impact	Dashboard typo

SEV 1–2: wake people up. SEV 3: next business day unless worsens.

A.9 Audit evidence index

Control area	Evidence artifact	Chapter
Authentication	JWT policy screenshot, IdP MFA config	5, 7
Rate limiting	SLA tier config, 429 logs	3, 4
Encryption	TLS scan, FIPS negative test log	5, 8
Secrets	Vault audit sample, no secrets in Git scan	9
Logging	SIEM dashboard, retention policy	12
SDLC	SAST/SCA reports, pen test summary	14
Compliance	Control mapping worksheet	13

A.10 Further reading

- OWASP API Security Project documentation
- NIST SP 800-207 Zero Trust Architecture
- MuleSoft Security Best Practices (Anypoint documentation)•

Release It! — Michael Nygard (resilience patterns)

- PCI DSS v4.0 Quick Reference Guide
- OpenID Connect Core 1.0 specification

Appendix B — Security Configuration Cookbook

“ Copy-paste-ready configuration for the controls discussed throughout the book. Every

recipe is annotated so you understand why, not just what. Adapt hostnames, key sizes,

and limits to your environment.

B.1 MuleSoft API Manager — full policy stack

B.1.1 JWT Validation (production)

```
policy: jwt-validation
configuration:
 jwtOrigin: httpBearerAuthenticationHeader
 signingMethod: jwks
 jwksUrl: "https://login.yourorg.com/.well-known/jwks.json"
jwksCacheTtlInMinutes: 1440
 skipClientIdValidation: false
 clientIdExpression: "#[attributes.headers['client_id']]"
validateAudClaim: true
 mandatoryAudClaim: true
 supportedAudiences: "https://api.yourorg.com"
 mandatoryExpirationClaim: true
 mandatoryNotBeforeClaim: false
 validateCustomClaim: true
 mandatoryCustomClaims:
  iss: "#[vars.expectedIssuer]"
  scope: "#['read:accounts']"
```

Why each line: `jwksUrl` lets the gateway fetch rotating signing keys; `validateAudClaim` stops tokens minted for other apps; `mandatoryExpirationClaim` rejects never-expiring tokens; custom claims pin issuer and scope.

B.1.2 Rate Limiting — fixed

```
policy: rate-limiting
configuration:
  rateLimits:
    - maximumRequests: 100
      timePeriodInMilliseconds: 60000
  exposeHeaders: true
  clusterizable: true
```

B.1.3 Rate Limiting — SLA-based with tiers

```
# Define tiers on the API (API Manager -> SLA Tiers)
slaTiers:
 - name: Bronze
  maximumRequests: 50
  timePeriodInMilliseconds: 60000
 - name: Silver
  maximumRequests: 200
  timePeriodInMilliseconds: 60000
 - name: Gold
  maximumRequests: 500
    timePeriodInMilliseconds: 60000
```

```
# Apply the SLA-based policy
policy: rate-limiting-sla-based
configuration:
 exposeHeaders: true
 clusterizable: true
```

B.1.4 Spike Control (smoothing)

```
policy: spike-control
configuration:
  maximumRequests: 50
  timePeriodInMilliseconds: 1000
  delayTimeInMillis: 250
  delayAttempts: 3
  queuingLimit: 200
  exposeHeaders: true
```

B.1.5 IP Allowlist

```
policy: ip-allowlist
configuration:
  ips:
    - "203.0.113.0/24"
```

```
    - "198.51.100.50"
  includeForwardedFor: true
```

B.1.6 HTTP Caching (public reads)

```
policy: http-caching
configuration:
  cacheKey: "#[attributes.method ++ ':' ++ attributes.requestPath ++ ':' ++ attributes.queryString]"
  maxCacheEntryTtl: 300
  httpCachingProfile: respect-cache-headers
  cacheableStatuses: [200, 203, 300, 301, 410]
  invalidationHeaderName: "X-Cache-Invalidate"
```

B.1.7 HTTP Headers (security response headers)

```
policy: http-headers
configuration:
  responseHeaders:
    - key: Strict-Transport-Security
      value: "max-age=31536000; includeSubDomains"
    - key: X-Content-Type-Options
      value: nosniff
    - key: X-Frame-Options
      value: DENY
    - key: Cache-Control
      value: no-store
    - key: Referrer-Policy
      value: strict-origin-when-cross-origin
```

B.1.8 Message Logging (audit metadata only)

```
policy: message-logging
configuration:
  loggingConfigurations:
    - itemName: request-audit
      itemData: >-
        clientId=#[attributes.headers['client_id'] default 'unknown']
        method=#[attributes.method]
        path=#[attributes.requestPath]
        correlationId=#[correlationId]
      category: api.audit
      level: INFO
      firstSection: true
      secondSection: false
```

B.2 Mule runtime — TLS and FIPS

B.2.1 Inbound TLS context

```
<tls:context name="Inbound_TLS_Context">
  <tls:key-store type="jks"
                 path="keystores/api-gateway.jks"
                 keyPassword="${secure::tls.keyPassword}"
                 password="${secure::tls.storePassword}"/>
  <tls:trust-store path="keystores/truststore.jks"
                   password="${secure::tls.trustPassword}"
                   type="jks"/>
  <tls:context enabledProtocols="TLSv1.2,TLSv1.3"
               enabledCipherSuites="TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_RSA_WITH_AES_1
28_GCM_SHA256"/>
</tls:context>
```

B.2.2 Outbound mTLS context

```
<tls:context name="Outbound_mTLS">
  <tls:key-store type="jks"
                 path="keystores/service-client.jks"
                 keyPassword="${secure::tls.keyPassword}"
                 password="${secure::tls.storePassword}"/>
  <tls:trust-store path="keystores/internal-ca-trust.jks"
                   password="${secure::tls.trustPassword}"/>
</tls:context>
```

B.2.3 FIPS activation (`wrapper.conf`)

```
wrapper.java.additional.20=-Dmule.security.model=fips140-2
 wrapper.java.additional.21=-Djava.security.properties==/opt/mule/conf/custom-java.security


# custom-java.security
security.provider.1=org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider
security.provider.2=org.bouncycastle.jsse.provider.BouncyCastleJsseProvider
```

B.2.4 FIPS negative test flow

```
<flow name="fips-negative-test">
  <http:listener path="/fips-test" config-ref="HTTPS_Listener_FIPS"/>
  <java:invoke-static class="java.security.MessageDigest"
                      method="getInstance(String)"
                      target="digest">
    <java:args><![CDATA[#[{ arg0: 'MD5' }]]]></java:args>
  </java:invoke-static>
```

```
  <set-payload value="UNEXPECTED: MD5 succeeded - FIPS NOT active"/>
</flow>
```

Expected in FIPS mode: `NoSuchAlgorithmException: MD5 not available`.

B.3 Secure Properties

B.3.1 Encrypt a value

```
java -cp secure-properties-tool.jar \
  com.mulesoft.tools.SecurePropertiesTool \
  string encrypt AES CBC "${ENCRYPTION_KEY}" "ProductionDbP@ssw0rd"
# -> ![kX9f2Q8...]
```

B.3.2 Reference in config

```
db:
  host: "db.prod.internal"
  user: "mule_svc"
  password: "![kX9f2Q8...]"
```

```
<db:generic-connection url="jdbc:postgresql://db.internal:5432/accounts"
                       user="${db.user}"
                       password="${secure::db.password}"/>
```

B.3.3 Supply the key at runtime

```
# On-prem
wrapper.java.additional.22=-Dmule.key=${MULE_KEY}
# CloudHub: Runtime Manager -> Settings -> Properties (mark Secure)
#   mule.key = <key>
```

B.4 NGINX edge hardening

```
# TLS
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384;
ssl_prefer_server_ciphers off;
ssl_session_cache shared:SSL:10m;
ssl_stapling on;
ssl_stapling_verify on;
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;

# Rate + connection limits
limit_conn_zone $binary_remote_addr zone=perip:10m;
limit_req_zone  $binary_remote_addr zone=reqs:10m rate=10r/s;
```

```
server {
    limit_conn perip 20;
    limit_req  zone=reqs burst=20 nodelay;
    client_body_timeout   5s;
    client_header_timeout 5s;
    send_timeout          10s;
    keepalive_timeout     15s;
    client_max_body_size  1m;
 # Block plain HTTP on the API port (no redirect)
 if ($scheme = http) { return 426; }}
```

B.5 AWS WAF — rate-based rule

```
{
  "Name": "ApiRateLimit",
  "Priority": 1,
  "Statement": {
    "RateBasedStatement": {
      "Limit": 2000,
      "AggregateKeyType": "IP"
    }
  },
  "Action": { "Block": {} },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "ApiRateLimit"
  }
}
```

B.6 Resilience4j (downstream protection)

```
resilience4j:
 circuitbreaker:
  instances:
      coreBankingService:
      slidingWindowType: COUNT_BASED
      slidingWindowSize: 20
      failureRateThreshold: 50
      slowCallRateThreshold: 80
      slowCallDurationThreshold: 2s
      waitDurationInOpenState: 30s
      permittedNumberOfCallsInHalfOpenState: 3
    minimumNumberOfCalls: 10
 bulkhead:
  instances:
```

```
      reportingService:
      maxConcurrentCalls: 10
      paymentsService:
      maxConcurrentCalls: 50
 timelimiter:
  instances:
      coreBankingService:
      timeoutDuration: 2s
```

B.7 OpenAPI security schema (input validation)

```
openapi: 3.0.3
paths:
 /accounts/{id}:
  patch:
      security:
      - oauth2: [write:accounts]
      requestBody:
      required: true
      content:
          application/json:
          schema:
            type: object
            additionalProperties: false
            properties:
                displayName:
                  type: string
                  maxLength: 100
                  pattern: '^[a-zA-Z0-9 \-\.]+$'
                preferredLanguage:
                  type: string
                  enum: [en, es, fr]
components:
 securitySchemes:
  oauth2:
      type: oauth2
      flows:
      authorizationCode:
          authorizationUrl: https://login.yourorg.com/authorize
          tokenUrl: https://login.yourorg.com/token
          scopes:
          read:accounts: Read account data
            write:accounts: Modify account data
```

B.8 HashiCorp Vault — AppRole for Mule

```
# Enable AppRole
vault auth enable approle

# Policy: read prod mule secrets
vault policy write mule-prod - <<EOF
path "secret/data/prod/mule/*" { capabilities = ["read"] }
EOF

# Role
vault write auth/approle/role/mule-prod \
  token_policies="mule-prod" token_ttl=1h token_max_ttl=4h

# Get role_id (safe to bundle) and secret_id (deliver securely)
vault read auth/approle/role/mule-prod/role-id
vault write -f auth/approle/role/mule-prod/secret-id
```

B.9 Kubernetes — default-deny + allow

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: default-deny
 namespace: integration-prod
spec:
 podSelector: {}
 policyTypes: [Ingress, Egress]
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: allow-gateway-to-mule
 namespace: integration-prod
spec:
 podSelector:
  matchLabels: { app: mule-payments-api }
 policyTypes: [Ingress]
 ingress:
  - from:
      - podSelector:
        matchLabels: { app: api-gateway }
      ports:
        - port: 8081
```

B.10 Istio — STRICT mTLS + authorization

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
 name: default
 namespace: integration-prod
spec:
 mtls:
  mode: STRICT
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: payments-only-from-gateway
 namespace: integration-prod
spec:
 selector:
  matchLabels: { app: mule-payments-api }
 action: ALLOW
 rules:
  - from:
    - source:
            principals: ["cluster.local/ns/ingress/sa/api-gateway"]
```

B.11 GitHub Actions — OIDC to cloud (no static keys)

```
permissions:
 id-token: write
 contents: read
jobs:
 deploy:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: aws-actions/configure-aws-credentials@v2
        with:
        role-to-assume: arn:aws:iam::123456789012:role/github-mule-deploy
        aws-region: us-east-1
    - name: Fetch mule.key
        run: echo "MULE_KEY=$(aws secretsmanager get-secret-value \
             --secret-id prod/mule-key --query SecretString --output text)" >> $GITHUB_ENV
    - name: SAST
        uses: returntocorp/semgrep-action@v1
        with: { config: p/owasp-top-ten }
    - name: Build & deploy
        run: mvn -B deploy -Dmule.key=$MULE_KEY
```

Appendix C
Security Design-Review Workbook

“ Templates and worksheets to run a real security design review. Photocopy these, fill them in, and attach them to the API's catalog entry. A review that produces a filled worksheet is evidence; a review that produces a verbal "looks fine" is not.

C.1 How to run a design review

A good API security design review is a **60–90 minute working session**, not a sign-off meeting. Attendees: the API owner/architect, a platform engineer, and a security reviewer. Optional: SRE for availability concerns.

```
Agenda
 0–10 min   Walk the data-flow diagram; agree on trust boundaries
10–25 min   STRIDE pass per component (worksheet C.3)
25–40 min   Map each threat to an existing control (worksheet C.4)
40–55 min   Identify gaps; file backlog items with owners and dates
55–70 min   Availability review (rate limits, resilience, DDoS)
70–90 min   Compliance and data-classification check; sign worksheet
```

Output: a completed workbook (sections C.3–C.7), a list of backlog items, and a go / no-go / go-with-conditions decision.

C.2 API intake worksheet

Field	Value
API name	
Version	
Owner / team	
Business purpose	

Field	Value
Consumers (mobile / partner / public / internal)	
Data classification (Public / Internal / Confidential / Restricted)	
Contains PII?	Y / N
Contains PCI cardholder data?	Y / N
Contains PHI?	Y / N
Auth model (OIDC / client-credentials / mTLS / API key)	
Hosting (CloudHub / RTF / on-prem / K8s)	
FIPS required?	Y / N
Internet-facing?	Y / N
Expected peak req/s	
Downstream systems	

C.3 STRIDE worksheet (per component)

Fill one block per component on the data-flow diagram (gateway, experience API, process API, system API, data store).

Component: _____

Threat	Applicable?	How it could happen	Control in place	Gap / action
Spoofing				
Tampering				
Repudiation				
Information disclosure				
Denial of service				
Elevation of privilege				

C.4 OWASP API Top 10 control checklist

#	Risk	Control present?	Where enforced	Notes
API1	BOLA	I		
API2	Broken auth	I		

#	Risk	Control present?	Where enforced	Notes
API3	Excessive data exposure	I		
API4	Lack of rate limiting	I		
API5	Broken function auth	I		
API6	Mass assignment	I		
API7	Security misconfiguration	I		
API8	Injection	I		
API9	Improper assets mgmt	I		
API10	Insufficient logging	I		

Any unchecked box is either a deliberate, documented risk acceptance or a blocking gap.

C.5 Authentication & authorization review

```
[ ] Token type and issuer documented
[ ] JWT validation enforces signature, iss, aud, exp, scope
[ ] Algorithm allow-list pinned (RS256/ES256; never none/HS256-by-default)
[ ] Token TTL appropriate to sensitivity (<=15 min high risk)
[ ] Refresh token rotation enabled
[ ] BOLA: object ownership checked in process/system layer
[ ] Function-level: admin operations require admin scope + step-up
[ ] Multi-tenant: tenant_id enforced where applicable
[ ] MFA for workforce and high-risk customer operations
```

C.6 Availability & resilience review

```
[ ] Rate limit per-client set at ~2-3x p99 baseline
[ ] Tighter limit on expensive endpoints (login, search, reports)
[ ] Global ceiling sized to downstream capacity
[ ] Spike control for burst smoothing
[ ] Caching on cacheable public reads
[ ] Circuit breaker + aggressive timeout on every downstream
[ ] Bulkhead isolation per dependency
[ ] Graceful load shedding (fast 503 + Retry-After)
[ ] CDN/DDoS protection in front; origin locked to CDN
[ ] DDoS runbook exists and names owners
```

C.7 Data protection & compliance review

```
[ ] Data classification assigned and tagged in catalog
[ ] TLS 1.2+ inbound; mTLS internal
[ ] Encryption at rest for the data store
[ ] PII fields masked/tokenized in responses where not needed
[ ] PCI: cardholder data tokenized or out of scope
[ ] GDPR: minimization, lawful basis, retention documented
[ ] HIPAA: BAA in place for processors (if PHI)
[ ] Secrets in vault / Secure Properties; none in Git
[ ] Logs contain no tokens, passwords, or PII bodies
[ ] Audit logging to SIEM with correlation IDs
```

C.8 Decision record

Field	Value
Review date	
Reviewers	
Decision	I Go I Go-with-conditions I No-go
Conditions / blocking items	
Risk acceptances (with approver)	
Re-review date (if conditional)	
Signature (security)	
Signature (API owner)	

C.9 Backlog item template

```
Title: [SEC] <component> <threat> mitigation
Severity: Critical / High / Medium / Low
Description: <what's missing, what could happen>
Acceptance criteria:
  - <control implemented and tested>
  - <negative test added to CI>
Owner: <name>
Target date: <date>
Linked review: <workbook reference>
```

C.10 Pre-production go-live security gate

A final gate before an API takes production traffic. All must be green or explicitly risk-accepted.

```
IDENTITY & ACCESS
[ ] Automated JWT policy active and verified on the API [ ]
Negative tests pass (no token -> 401, wrong aud -> 401) [ ]
BOLA negative test passes (other user's object -> 403)

TRAFFIC
[ ] Rate limits configured and tested (over limit -> 429 + Retry-After)
[ ] Caching audited (no private data on shared cache)

CRYPTO
[ ] TLS scan grade A/A+; legacy protocols disabled
[ ] FIPS negative test passes (if FIPS required)

SECRETS
[ ] No secrets in repo (scan clean)
[ ] mule.key / vault access via runtime identity only

OBSERVABILITY
[ ] Audit logs flowing to SIEM with correlation IDs
[ ] Dashboards and alerts configured
[ ] Runbook linked

GOVERNANCE
[ ] API registered in catalog with classification
[ ] Old versions decommissioned or scheduled
[ ] Change ticket approved
```

C.11 Quarterly governance review agenda

```
1. API inventory diff: new APIs, shadow APIs discovered, zombies to retire
2. Client application audit: active credentials, stale ones to revoke
3. Access review: who has org-admin / deploy / policy-edit
4. Policy drift: any prod API missing automated policies?
5. Secret rotation status vs schedule
6. Open security backlog items aging report
7. Incident review: anything since last quarter; lessons applied
8. Upcoming compliance dates (PCI ASV scan, pen test, audit)
```

Appendix D — Worked Case Studies & Incident Walkthroughs

> *“ Six end-to-end scenarios that exercise the patterns from the whole book. Each follows the same shape: the situation, what went wrong (or could), the architecture response, and the lessons. Names and numbers are changed; the mechanics are real.*

D.1 Case study: the credential-stuffing storm

Situation

A retail bank exposed a customer login API: `POST /oauth/token` (resource owner flow, a legacy choice). One evening, the SOC noticed CPU on the auth service climbing toward 100% with only a modest request rate — about 40 requests per second.

What was happening

An attacker was running a credential-stuffing list (leaked username/password pairs from an unrelated breach) against the login endpoint. The volume was *low* by DDoS standards, but each attempt triggered a **bcrypt** password hash — deliberately expensive at ~100 ms.

Forty requests per second meant the thread pool filled with hashing work and legitimate logins started timing out. The platform was effectively down, not from bandwidth, but from **work amplification**.

Architecture response

```
1. Immediate (containment, < 10 min):
   - API Manager: applied a 5/min per-IP rate limit specifically on /oauth/token
   - CDN: enabled managed challenge (JS challenge) for the login path
2. Short term (days):
   - Added breached-password detection at the IdP
   - Migrated off resource-owner password grant to Authorization Code + PKCE
   - Added MFA for the customer segment
3. Structural (weeks):
   - Separate, tighter rate-limit tier for all authentication endpoints
   - Bot management scoring on login, account creation, password reset
```

Lessons

- **Rate-limit expensive endpoints harder than cheap ones.** A global limit sized for read APIs is useless against an attack on a CPU-heavy endpoint.
- **Low request rate can still be a DoS** when each request is expensive (Chapter 2, §2.7; Chapter 3, §3.1).
- **The fix was a config change, not a redeploy** — the payoff of policy-driven front ends (Chapter 3, §3.5).

D.2 Case study: the BOLA that leaked statements

Situation

A pen test against a wealth-management API found that any authenticated customer could read another customer's account statements by changing the account ID in the URL.

What was happening

The endpoint `GET /accounts/{accountId}/statements` validated the JWT (authentication worked) but the data-access code did:

```
account = db.getAccount(pathParam.accountId)
return account.statements
```

It never checked that the *authenticated user* owned `accountId`. Classic **Broken Object Level Authorization** (OWASP API1).

Architecture response

```
account = db.getAccount(pathParam.accountId)
if account.ownerId != token.sub:
    audit("bola_attempt", token.sub, accountId)
    return 403 Forbidden
return account.statements
```

Plus:

- Added a **BOLA negative test** to CI: Alice's token requesting Bob's account must return 403.
- Added a SIEM rule: many 403s on `/accounts/*` from one `sub` = probing alert.

Lessons

- The **gateway can't fix BOLA** — it doesn't know who owns account 100246. Object authorization lives in the process/system layer with business context (Chapter 2, §2.3; Chapter 6, §6.3).
- **Authentication ≠ authorization.** A valid token answers "who," not "allowed."
- **Test the negative case** automatically so a refactor can't silently reintroduce it (Chapter 14, §14.10).

D.3 Case study: the viral spike that wasn't an attack

Situation

A media company's article API fell over every time a story went viral. The pattern looked like a DDoS — tens of thousands of requests per second — but it was just real readers.

What was happening

Each viral article generated a flood of identical `GET /articles/{id}` requests that all hit the origin and the database. No caching. The database connection pool saturated and the whole site degraded.

Architecture response

```
1. Added a 60-second edge cache on the article read endpoint
   (Cache-Control: public, max-age=60)
2. Cache key normalized to ignore unknown query params (anti cache-buster)
3. Database connection pool given a hard cap + query timeout
4. Circuit breaker on the article-fetch downstream so a slow DB
   fails fast instead of exhausting threads
```

The next viral event: the origin saw a handful of requests per minute instead of thousands per second. The cache absorbed the identical traffic.

Lessons

- **Caching is an availability/DDoS control**, not just a speed trick (Chapter 3, §3.2; Chapter 4, §4.4).
- A 60-second cache turned a catastrophic spike into a non-event.
- The same defense works whether the spike is legitimate success or a malicious L7 flood on a cacheable endpoint.

D.4 Case study: the leaked origin IP

Situation

A fintech put Cloudflare in front of its APIs and felt protected. Then a volumetric attack hit and the APIs went down anyway.

What was happening

The attacker had discovered the **origin IP** (from an old DNS A record still in historical records) and aimed the flood directly at the origin, bypassing the CDN entirely. The CDN only protects what traffic actually flows through it.

Architecture response

```
1. Rotated the origin IP
2. Firewalled the origin to accept traffic ONLY from Cloudflare IP ranges
3. Verified from an external host that the origin was unreachable directly
4. Enabled authenticated origin pulls (origin validates CDN client cert)
5. Audited certificate transparency logs and old DNS for further leaks
```

Lessons

- **A CDN only protects what it sits in front of** (Chapter 4, §4.2).
- **Lock the origin to the CDN and verify it from outside** — assumptions aren't verification.
- Origin IPs leak through old DNS, certificate transparency, and email headers; treat the origin address as semi-sensitive.

D.5 Case study: the FIPS deployment that wouldn't start

Situation

A government contractor enabled FIPS 140-2 mode on its Mule workers. Every deployment then failed on startup with a cryptography error.

What was happening

A legacy JMS connector's internal library called `MessageDigest.getInstance("SHA-1")` during initialization. SHA-1 for digital signatures is not FIPS-approved, and the Bouncy Castle FIPS provider rejected it at runtime, aborting startup.

Architecture response

```
1. Reproduced in a lower environment with FIPS enabled (not prod!)
2. Identified the offending connector via the stack trace
3. Upgraded the connector to a version using SHA-256
4. Added a FIPS smoke test to CI: deploy to a FIPS worker and run the
   MD5 negative test on every release
5. Audited all dependencies for non-FIPS algorithm use
```

Lessons

- **FIPS is a runtime substrate, not a checkbox** (Chapter 5, §5.5). It rejects non-approved algorithms anywhere — including inside third-party libraries.
- **Test every application in FIPS mode in a lower environment first.** "We don't use MD5" ignores your dependencies.
- A CI smoke test prevents regression when connectors are upgraded.

D.6 Case study: secrets in a forked repository

Situation

A contractor cloned an internal repo to a personal GitHub account "for backup." Eleven months later, a routine pen test found a valid production database password in `config.yaml` in that public fork.

What was happening

Secrets were committed in plaintext. GitHub secret scanning didn't flag it because a database password isn't a recognizable token format. The repo's "private" status was the only thing protecting it — until it was forked publicly.

Architecture response

```
1. Rotated the exposed database password immediately
2. Created a second DB user, dual-credential window, then dropped the old
3. Migrated all secrets to Vault; config holds only ![ciphertext] or references
4. Added pre-commit hooks (git-secrets, trufflehog) blocking secret patterns
5. Added CI secret scanning as a hard gate
6. Reviewed access logs for use of the exposed credential (none found)
```

Lessons

- **A leaked repo should be a non-event** (Chapter 9). Plaintext secrets in Git make it a breach.
- **Repo privacy is not a security control.** Rotate, vault, and scan.
- **Dual-credential rotation** avoids downtime when you must change a live password (Chapter 9, §9.7).

D.7 Cross-cutting patterns

Across all six, the same themes recur:

Theme	Cases	Chapter
Defense in depth — no single control	All	1
Authentication ≠ authorization	D.2	2, 7
Availability is a security property	D.1, D.3, D.4	3, 4
Config change beats redeploy under pressure	D.1, D.3	3
Test negative cases in CI	D.2, D.5	14
Secrets discipline	D.6	9
Verify, don't assume	D.4, D.5	4, 5

If you take one meta-lesson: **the boring controls, applied consistently and verified, prevent the dramatic incidents.** None of these six required novel security research to fix. They required the discipline this book is about.

Appendix E
Threat & Control Catalog

“ A consolidated reference: the threats covered throughout the book, the control that addresses each, where it's enforced, and the chapter to consult. Use it as a lookup table during reviews and incident triage.

E.1 Master threat-to-control matrix

#	Threat	Primary control	Enforced at	Chapt
1	Eavesdropping on traffic	TLS 1.2+/1.3	Edge, gateway,	8
2	Service impersonation	mTLS / workload identity	Internal mesh	8, 10
3	Stolen bearer token	Short TTL + revocation	IdP, gateway	7
4	Forged JWT (alg:none)	Algorithm allow-list	Gateway	7
5	JWT wrong audience	aud validation	Gateway	7
6	Credential stuffing	Rate limit + MFA + bot mgmt	Gateway, edge, IdP	3, 4, 11
7	BOLA	Object ownership check	Process/system API	2, 6
8	Broken function auth	Scope/role + step-up	Gateway + app	2, 7
9	SQL injection	Parameterized queries	System API	2, 11
10	NoSQL injection	Typed queries + schema	System API	11
11	XXE	Disable external entities	Parser config	11
12	Command injection	No shell with user input	App code	11
13	Mass assignment	additionalProperties:false	Gateway schema	2, 11
14	Excessive data exposure	Output DTOs	Experience/system	2, 11
15	Volumetric DDoS (L3/4)	CDN/scrubbing upstream	Edge	4
16	Protocol DDoS	SYN cookies, timeouts	LB/edge	4
17	Application DDoS (L7)	Rate limit, spike, cache, WAF	Gateway	3, 4
18	Cascading failure	Circuit breaker + timeout	Process API	3
19	Resource exhaustion	Bulkhead isolation	Process API	3
20	Cache data leak	Cache-Control/Vary discipline	Gateway/CDN	3
21	Cache poisoning	Tight cache keys	Gateway/CDN	3, 11

#	Threat	Primary control	Enforced at	Chapter
22	Replay/duplicate txn	Idempotency keys	Gateway/process	3
23	Secrets in repo	Vault + Secure Properties	Build/runtime	9
24	Weak crypto algorithm	FIPS mode / approved list	Runtime	5, 8
25	Expired certificate	Lifecycle monitoring	PKI/ops	8
26	Shadow / zombie API	Catalog + discovery	Governance	2, 13
27	Security misconfiguration	Hardened defaults, automated policies	Platform	2, 5
28	Insufficient logging	Message logging + SIEM	Gateway/ops	12
29	Lateral movement	Microsegmentation + mTLS	Network/mesh	10, 15
30	Supply-chain CVE	SCA + SBOM + pinning	CI/CD	14
31	Container escape / root	Non-root, Pod Security	Kubernetes	15
32	Open redirect (OAuth)	Exact redirect URI match	IdP/client	7
33	CSRF on OAuth callback	state parameter	Client	7
34	CORS misconfiguration	Explicit origin allow-list	Gateway/app	11
35	MIME confusion	Content-Type enforcement	Gateway	11

E.2 Control catalog — quick definitions

Control	One-line definition	Chapter
Rate limiting	Cap requests per window; reject over limit (429)	3
Throttling / spike control	Queue and smooth bursts instead of rejecting	3
HTTP caching	Serve stored responses; shield origin	3
Circuit breaker	Fail fast when a dependency is unhealthy	3
Bulkhead	Isolate resources per dependency	3
Timeout	Bound how long any call can block	3
JWT validation	Verify token signature and claims	5, 7
mTLS	Both parties present and verify certificates	8
Schema validation	Enforce the API contract on input	11
WAF	Signature/behavioral filtering at the edge	4, 11
Bot management	Distinguish humans from automation	11
Secrets vault	Central, audited secret storage	9
Secure Properties	Encrypt Mule config values at rest	5, 9

Control	One-line definition	Chapter
FIPS 140-2 mode	Validated crypto, approved algorithms only	5
Tokenization	Replace sensitive data with a surrogate	8
Encryption at rest	Protect stored data (TDE, field-level)	8
Microsegmentation	Restrict traffic between network zones	10
Automated policies	Apply baseline controls to all APIs	3, 5
Message logging	Structured audit metadata to SIEM	12
Idempotency keys	Safe retries without duplicate effects	3

E.3 Severity guidance for findings

Severity	Criteria	Example	Typical SLA
Critical	Direct data exposure or full auth bypass, internet-facing	BOLA on customer data; auth bypass	Fix < 7 days; consider takedown
High	Exploitable with conditions; significant impact	Missing rate limit on expensive endpoint	< 30 days
Medium	Limited impact or requires chaining	Verbose errors; weak cache	< 90 days
Low	Hardening / best practice	Missing a defense-in-depth header	Backlog

E.4 "What do I reach for?" decision guide

```
Problem: clients overwhelming the API
  -> Rate limiting (per-client) + spike control           [Ch 3] -
 > If identical reads: HTTP caching                      [Ch 3] ->
 If volumetric: CDN/scrubbing + origin lockdown        [Ch 4]

Problem: a downstream is slow/failing and dragging us down
  -> Aggressive timeout + circuit breaker                  [Ch 3]
 -> Bulkhead to isolate it                                [Ch 3]

Problem: need to authenticate API callers
  -> OAuth2/OIDC + JWT validation at gateway               [Ch 7] ->
 M2M: client credentials (+ mTLS)                       [Ch 7, 8]

Problem: users seeing each other's data
 -> Object-level authorization in process/system layer     [Ch 2, 6]

Problem: injection risk
 -> Parameterized queries + schema validation + WAF        [Ch 11]

Problem: secrets sprawl
 -> Vault + Secure Properties; rotate; scan repos          [Ch 9]
```

```
Problem: regulated crypto requirement
 -> FIPS 140-2 mode + approved TLS/keys                    [Ch 5, 8]

Problem: can't see what's happening
  -> Structured logs + correlation IDs + SIEM + dashboards  [Ch 12]
```

E.5 One-line principles (the whole book, distilled)

- Security is a **constraint you design within**, not a feature you add late.
- Know your **trust boundaries**; check every request that crosses one.
- **Authentication is not authorization.** Prove who, then prove allowed.
- **Availability is a security property** — rate limit, cache, isolate.
- **Fail fast** to protect yourself and let dependencies recover.
- Push **coarse controls to the edge**, **fine controls to the data**.
- **No single control is trusted alone** — depth covers the gaps.
- **Reject cheap and early**; don't spend work on doomed requests.
- **Secrets live in vaults**, never in repos; a leaked repo should be a non-event.
- **Verify, don't assume** — test the negative case, scan the origin, prove FIPS.
- **Make controls policies, not prayers** — reusable, central, provable.
- **Maintain security like uptime** — continuously, with owners and evidence.

Bibliography

[1] Allamaraju, S. (2010). RESTful Web Services Cookbook. O'Reilly Media.

[2] Arundel, J. (2022). Practical API security. Manning Publications.

[3] Arundel, J., & Domingus, J. (2019). Cloud native DevOps with Kubernetes. O'Reilly Media.

[4] Bass, L., Weber, I., & Zhu, L. (2015). DevOps: A software architect's perspective. Addison-Wesley.

[5] Berners-Lee, T., Fielding, R. T., & Masinter, L. (2005). Uniform resource identifier (URI): Generic syntax (RFC 3986). Internet Engineering Task Force.

[6] Burns, B. (2018). Designing distributed systems: Patterns and paradigms for scalable, reliable services. O'Reilly Media.

[7] Cloud Security Alliance. (2023). Security guidance for critical areas of cloud computing.

[8] Evans, E. (2004). Domain-driven design: Tackling complexity in the heart of software. Addison-Wesley.

[9] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures (Doctoral dissertation, University of California, Irvine).

[10] Forsgren, N., Humble, J., & Kim, G. (2018). Accelerate: The science of lean software and DevOps. IT Revolution Press.

[11] Hardt, D. (2012). The OAuth 2.0 authorization framework (RFC 6749). Internet Engineering Task Force.

[12] Humble, J., & Farley, D. (2010). Continuous delivery: Reliable software releases through build, test, and deployment automation. Addison-Wesley.

[13] Jones, M., Bradley, J., & Sakimura, N. (2015). JSON web token (JWT) (RFC 7519). Internet Engineering Task Force.

[14] Kim, G., Behr, K., & Spafford, G. (2018). The Phoenix project: A novel about IT, DevOps, and helping your business win (5th anniversary ed.). IT Revolution Press.

[15] Kleppmann, M. (2017). Designing data-intensive applications. O'Reilly Media.

[16] Kohnfelder, L., & Garg, P. (2021). Threat modeling: Designing for security. Wiley.

[17] MuleSoft. (2024). Anypoint platform security guide. Salesforce, Inc.

[18] MuleSoft. (2024). API-led connectivity: Best practices. Salesforce, Inc.

[19] National Institute of Standards and Technology. (2020). Zero trust architecture (Special Publication 800-207). U.S. Department of Commerce.

[20] National Institute of Standards and Technology. (2024). FIPS 140-3: Security requirements for cryptographic modules. U.S. Department of Commerce.

[21] OWASP Foundation. (2023). OWASP API security top 10. OWASP Foundation.

[22] OWASP Foundation. (2024). Application security verification standard (ASVS) version 5.0. OWASP Foundation.

[23] Pautasso, C., Zimmermann, O., & Leymann, F. (2008). RESTful web services vs. big web services: Making the right architectural decision. Proceedings of the 17th International Conference on World Wide Web, 805–814.

[24] Richardson, C. (2018). Microservices patterns. Manning Publications.

[25] Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). Zero trust architecture (NIST Special Publication 800-207). National Institute of Standards and Technology.

[26] Saltzer, J. H., & Schroeder, M. D. (1975). The protection of information in computer systems. Proceedings of the IEEE, 63(9), 1278–1308.

[27] Scarfone, K., & Mell, P. (2007). Guide to intrusion detection and prevention systems (IDPS) (NIST SP 800-94). National Institute of Standards and Technology.

[28] Shackleford, D. (2021). API security for connected applications. SANS Institute.

[29] Stallings, W. (2017). Cryptography and network security: Principles and practice (7th ed.). Pearson.

[30] Weiss, M. (2019). API security in action. Manning Publications.

[31] Yarygina, T., & Bagge, A. H. (2018). Overcoming security challenges in microservice architectures. 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), 11–20.

[32] Zalewski, M. (2012). The tangled web: A guide to securing modern web applications. No Starch Press.

[33] Alshammari, M., & Simpson, A. (2021). Security challenges and solutions in API-driven architectures. Journal of Information Security and Applications, 58, 102760.

[34] Anderson, R. (2020). Security engineering: A guide to building dependable distributed systems (3rd ed.). Wiley.

[35] Barker, E., & Roginsky, A. (2020). Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths (NIST SP 800-131A Rev. 2). National Institute of Standards and Technology.

[36] Brabham, R. (2022). Mastering API architecture. Apress.

[37] Chappell, D. (2020). Enterprise service bus. O'Reilly Media.

[38] Chen, L., Jordan, S., Liu, Y. K., Moody, D., Peralta, R., Perlner, R., & Smith-Tone, D. (2016). Report on post-quantum cryptography. National Institute of Standards and Technology.

[39] Erl, T., Carlyle, B., Pautasso, C., & Balasubramanian, R. (2019). Microservice architecture: Aligning principles, practices, and culture. Pearson.

[40] Ferreira, A., & Antunes, L. (2015). Security concerns in cloud-based API management systems. Future Internet, 7(3), 342–357.

[41] Gartner Research. (2023). API management and security market guide. Gartner Inc.

[42] Greenberg, A. (2019). Sandworm: A new era of cyberwar and the hunt for the Kremlin's most dangerous hackers. Doubleday.

[43] Kavis, M. J. (2014). Architecting the cloud: Design decisions for cloud computing service models. Wiley.

[44] Krebs, B. (2016). Spam nation: The inside story of organized cybercrime. Sourcebooks.

[45] Lewis, J., & Fowler, M. (2014). Microservices: A definition of this new architectural term. ThoughtWorks Technical Insights.

[46] Microsoft Corporation. (2024). Zero Trust deployment guide. Microsoft Security Documentation.

[47] Nygard, M. T. (2018). Release it!: Design and deploy production-ready software (2nd ed.). Pragmatic Bookshelf.

[48] OpenID Foundation. (2023). OpenID Connect Core 1.0 incorporating errata set 2. OpenID Foundation.

[49] Shostack, A. (2014). Threat modeling: Designing for security. Wiley.

[50] Skelton, M., & Pais, M. (2019). Team topologies: Organizing business and technology teams for fast flow. IT Revolution Press.

[51] Stuttard, D., & Pinto, M. (2021). The web application hacker's handbook (2nd ed.). Wiley.
U.S. Department of Homeland Security. (2023). Best practices for mitigating distributed denial-of-service attacks. Cybersecurity and Infrastructure Security Agency (CISA).

SECURING THE ENTERPRISE API

Front-End Patterns · DDoS Defense · FIPS-Grade MuleSoft Architecture

ABOUT THE AUTHOR

Venkata Pavan Kumar Gummadi is a Technical Architect specializing in MuleSoft, API security, and enterprise middleware at a leading financial-services technology company. Over an eighteen-year career across insurance, utilities, financial services, and capital markets, he has designed and delivered API platforms that process billions of dollars in daily transactions — and survived the attacks that come with that territory.

He led the architecture of Wealth InFocus, an award-winning wealth-management platform that earned the Datos Impact Award, built on the same API-led connectivity model this book teaches: experience, process, and system APIs, each with its own security boundary, rate controls, and audit trail. He holds three MuleSoft certifications — Certified Integration Architect, Platform Architect, and Developer.

He is the author of AgentFlow, an open-source framework for AI multi-agent API orchestration, and the creator of ASCEND, a four-layer DevSecOps governance framework. He writes to bridge the gap between integration engineering and security — helping teams build API platforms that are fast, compliant, and genuinely hard to break.

ISBN: 978---93-49-929--
9 789349 929050

WRITTEN BY

VENKATA PAVAN KUMAR GUMMADI

SCIENCE TECH XPLORE

https://sciencetechxplore.org