1ST EDITION

Defending APIs

Uncover advanced defense techniques to craft secure application programming interfaces

COLIN DOMONEY

Foreword by Chris Wysopal, Veracode co-founder and CTO, and Isabelle Mauny, 42Crunch co-founder and CTO

Defending APIs

Uncover advanced defense techniques to craft secure application programming interfaces

Colin Domoney

‹packt›

Defending APIs

Group Product Manager: Pavan Ramchandani

Publishing Product Manager: Prachi Sawant

Book Project Manager: Ashwin Dinesh Kharwa

Senior Editor: Isha Singh

Technical Editor: Nithik Cheruvakodan

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Production Designer: Vijay Kamble

DevRel Marketing Coordinator: Marylou De Mello

First published: February, 2024

Production reference: 112012024

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-80461-712-0

www.packtpub.com

13

Implementing an API Security Strategy

This book's final chapter focuses on applying the knowledge you've gained over the last 12 chapters to create a comprehensive API security strategy for your organization. Your strategy will depend on your current position and your security goals and will largely be influenced by your organizational structure, particularly who owns APIs and their security. In the first section, we will examine typical organizational stakeholders and how their roles and responsibilities must be aligned as part of your strategy. We will then examine the 42Crunch API security maturity model to understand the six domains of API security and what maturation looks like for each domain. After, we'll dive into rolling out your API security strategy, firstly by planning your objectives against the current state and capabilities and, secondly, by running your strategy as part of your daily process. Finally, we'll conclude this chapter – and this book – by looking at your ongoing personal API security journey.

In a nutshell, this chapter is going to cover the following main topics:

- Ownership of API security
- The 42Crunch maturity model
- Planning your program
- Running your program
- Your personal API security journey

Ownership of API security

Your API security strategy cannot exist in isolation from your organization's API business and development strategy. As a security leader, you must understand the other stakeholders responsible for API strategy and delivery to ensure that your security strategy aligns with their objectives.

Ownership of APIs tends to vary from one organization to another, and there is no hard and fast rule regarding how it is assigned. In the *Further reading* section, there is a reference to a blog post from MuleSoft that describes a typical pattern for API ownership; we will use this to frame our discussion. This is shown visually in *Figure 13.1*:

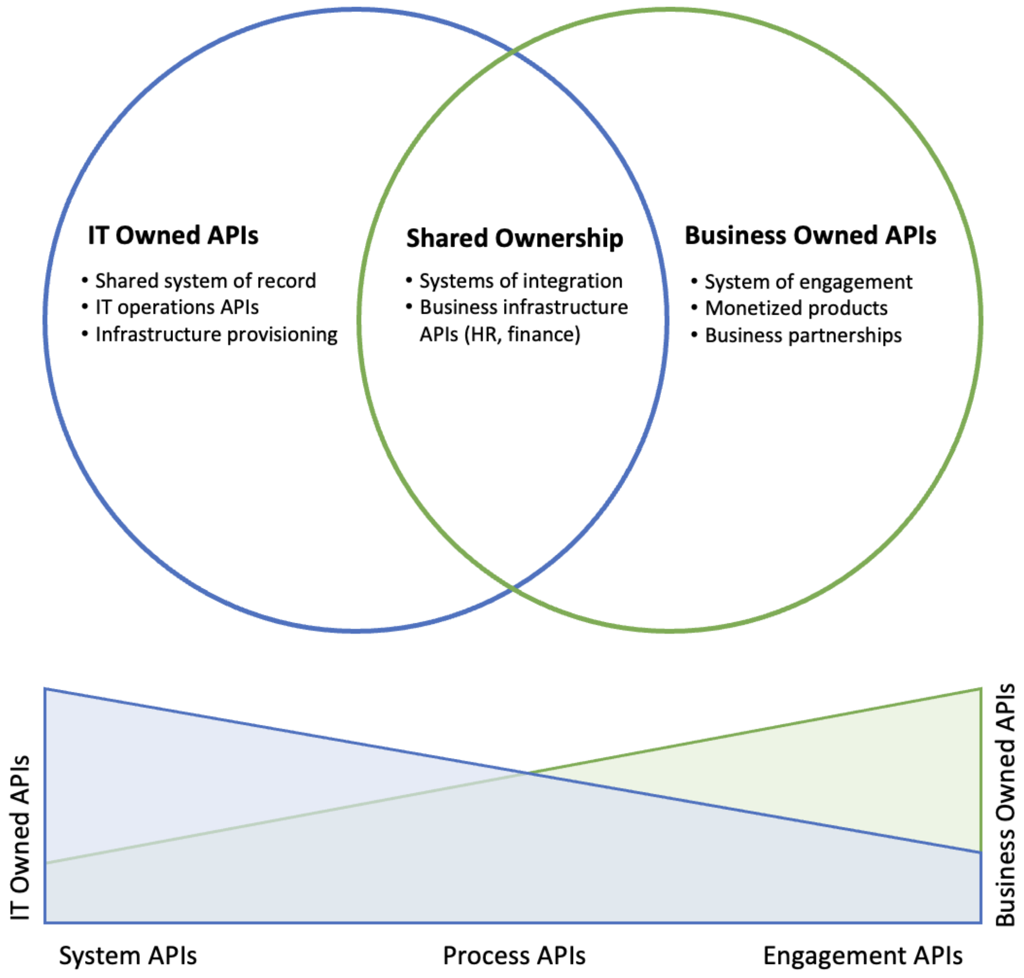

Figure 13.1 – API ownership model

There are three main API owners in this model:

- **IT-owned APIs**: This ownership model aligns most closely with traditional IT systems where the IT department wholly owns the resources. These are core services such as infrastructure provisioning or operations such as ServiceNow or Salesforce platforms. These are often called **system-of-record** APIs, which favor stability and accuracy over features and functionality.

- **Business-owned APIs**: This ownership model reflects the rise of APIs-as-a-product organizations (a typical example is Twilio, where the API is their product) that derive their business value through APIs, which are their **system of engagement**. Here, innovation and agility are key in monetizing existing services to new markets or in delivering new business partnerships. The APIs are owned by the business units, together with their development teams.
- **Shared ownership APIs**: Sitting in the middle ground between these two models is the shared ownership model, which is mostly used to integrate systems such as HR, finance, or inventory systems. The IT department will still exert some control over the operation of these APIs, but their ultimate ownership may reside with other organizational departments.

It is important to understand the ownership of APIs when it comes to kicking off an API security initiative since the ownership will determine the business risk appetite. For **systems of record**, where data integrity is paramount, the owners will likely have a low-risk appetite and engage fully with an API security initiative. They will appreciate the value that such an initiative brings toward producing more dependable APIs that match their objectives. Additionally, many IT security teams may report to the head of IT and have a common senior manager, which will help with the adoption of such an initiative.

The situation is different in the case of the business-owned APIs functioning as a **system of engagement**. Here, the focus is on innovation and agility and an API security initiative will possibly be regarded as an inhibitor to rapid innovation or short release cycles. The risk appetite will likely be higher in this case; in many cases, the business is doing a quick proof of concept that's limited in scope and is less concerned about the consequences of lax security. If this is the case, you will have additional challenges in gaining traction with your initiative. The key recommendation is to form a working relationship with the business unit and reassure them that you are not there to slow them down but rather to aid as an accelerant for innovation while also improving the security posture of their portfolio. These two objectives are not mutually exclusive, given a collaborative working arrangement.

Understanding your stakeholders

Now that we understand the different ownership models, let's look at the various personas across the IT, API, operations, security, and business units to understand their perspectives on API security.

Table 13.1 shows the typical roles in the security domain, along with their key responsibilities:

Role	**Description**
CISO	They are responsible for information security in an organization
Head of AppSec	They are responsible for the AppSec program and activities in the organization
DevSecOps team	This team is responsible for the integration and operation of security tools within the automated SDLC environment

Role	Description
Pentest/red team	This team is responsible for offensive testing of product releases using black box techniques
Risk and compliance team	This team is responsible for managing risk and compliance in the organization based on applicable operating environments

Table 13.1 – Typical roles in the security domain

Table 13.2 shows the typical roles in the business or development domain, along with their key responsibilities:

Role	Description
CIO	They are responsible for the IT operations of a business unit
Product owner	They are responsible for the product management of a business unit's offerings
Technical lead	They are responsible for managing the technical team on a given product
Solution architect	They are responsible for supporting and evangelizing the product to the customer
DevOps team	This team is responsible for the operation of the build and release process through automation

Table 13.2 – Typical roles in the business or development domain

Table 13.3 shows the typical roles in the API product domain, along with their key responsibilities:

Role	Description
API product owner	They are responsible for the product management of a set of APIs offered as a product
API platform owner	They are the owner of the central API platforms (API gateways and management portals) and API PaaS infrastructure
API architect	They are responsible for the overall API strategy (authentication, authorization, and architecture) in the organization

Table 13.3 – Typical roles in the API product domain

The reason for deliberately and explicitly enumerating this array of stakeholders is to illustrate that ownership of API security potentially resides with several organizational units. For example, the API platform owner may be responsible for applying appropriate API gateway policies, the API architect may be responsible for the overall authentication strategy, the head of AppSec may be responsible for SAST/DAST scans, and the CISO is ultimately responsible for the security of the API.

To avoid duplication of responsibilities (or worse still, neglect of responsibility), the roles and responsibilities should be clearly defined.

Roles and responsibilities

In *Chapter 1*, *What Is API Security?*, we saw the typical DevOps cycle in *Figure 1.3*. The cycle has eight SDLC phases: plan, code, build, test, release, deploy, operate, and monitor. In an ideal world, security touchpoints across all eight stages of the SDLC will be encapsulated perfectly by the "shift-left, shield-right" mantra used in this book.

Unfortunately, in reality, many organizations have relatively immature security processes typically characterized by late-stage monitoring of IT systems via an SIEM and SOC. This monitoring typically only identifies attacks or threats as they occur and is a reactive security practice. This monitoring is usually performed by the IT security team, with the CISO being accountable, and is shown in the SDLC in *Figure 13.2*:

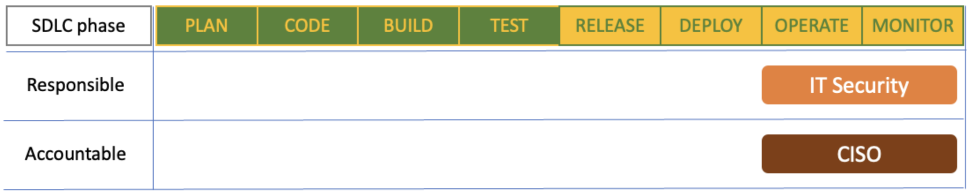

Figure 13.2 – Monitoring in the SDLC

With the growing awareness of application security as a necessary pre-emptive control, more mature organizations have sought to introduce SAST/DAST scanning into the release and deploy cycle. These tools are usually integrated with the help of the DevSecOps team, and the CISO is still accountable for managing the risk identified in this process, as shown in the SDLC in *Figure 13.3*:

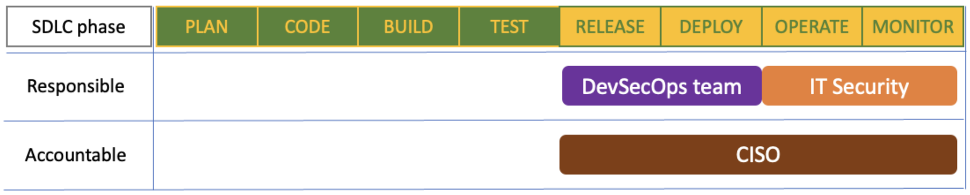

Figure 13.3 – Monitoring and scanning in the SDLC

The benefits of shifting security left, or earlier in the SDLC, have been widely accepted in the industry over the last decade. Typically, this involves co-opting development teams into the security conversation as they take measures to ensure that they are producing secure code by using secure libraries, understanding risks and threats, using secure coding patterns, testing for secure vulnerabilities in the development phases, and more. Furthermore, the accountability for security is shifted from the CISO to the business or product owners, who increasingly regard a secure product as a differentiating feature in the marketplace. An example shift-left SDLC is shown in *Figure 13.4*:

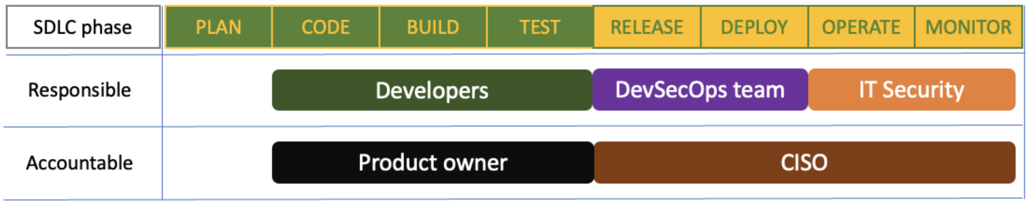

Figure 13.4 – Shift-left in the SDLC

While this shift-left pattern makes great steps toward producing more secure code, it can be improved further by incorporating security right at the design stage to ensure security is considered part of the design before any code is written. As an example, in an API security context, this may relate to decisions about managing identities and roles within an API product. A fully integrated secure SDLC is shown in *Figure 13.5*:

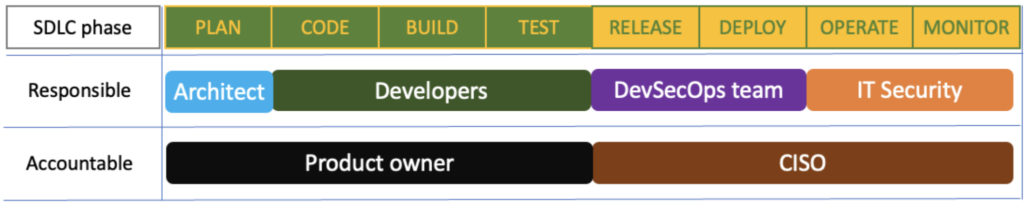

Figure 13.5 – Fully integrated secure SDLC

These examples illustrate that the ownership of both APIs and API security is a complex topic and that often, this ownership will be distributed across two or more organizational units. A distributed ownership model has benefits since it allows domain experts to focus on their specialism: the CISO ensures the APIs are monitored optimally in the SIEM and SOC, and the product owner ensures that the APIs are well-architected and secure by design.

The 42Crunch maturity model

In my time as a technical evangelist at 42Crunch, I formulated a six-domain API security maturity model that has proved to be popular with customs in determining both their current security posture and their roadmap toward a more secure posture.

The maturity model features a set of activities for each domain, which may exist to varying degrees based on maturity. For this discussion, we will bucket the activities as **non-existent**, **emerging**, or **established**.

Inventory

An up-to-date and accurate inventory is key to maintaining visibility into the exposed risk and attack surface.

The adage "*you can't protect what you can't see*" applies perfectly to API security. As APIs grow exponentially, fueled by business demand, it is increasingly difficult for security teams to maintain visibility of what APIs exist and what risks they expose.

Three elements are key:

- How new APIs are introduced and tracked in the organization
- Discovering the API inventory by introspecting the source code repositories to discover hidden API artifacts
- Runtime discovery of APIs (via network traffic inspection, and so on)

At the lowest maturity level, only a basic inventory (usually, APIs deemed critical for the business) is maintained via spreadsheet or manual tracking. There is no management of shadow/zombie APIs.

At the emerging maturity level, an inventory is maintained via API management or a centralized platform. A standard process is used for new API development. For an established level of maturity, the inventory is actively tracked via a centralized platform, and shadow and zombie APIs are deprecated and upgraded.

Design

It is significantly more cost-effective to address security issues at the design phase rather than later in the life cycle – a shift-left approach is key.

A solid API design practice is the foundation of usable, scalable, documented, and secure APIs. Here are some of the key elements of secure API design:

- Authentication methods
- Authorization models and access control

- Data privacy requirements
- Data exposure (explicit/least content/fit for purpose) requirements
- Compliance requirements
- Account reset mechanisms
- Use and abuse cases
- Key and token issue and revocation methods
- Rate limiting and quota enforcement

Additionally, API design teams should perform threat modeling exercises to understand their threat environment and attack surface.

At the lowest maturity level, no formal API design process is in place; instead, a code-first method is used. There is usually no upfront consideration of security concerns, threats, compliance, and data privacy.

At the emerging maturity level, APIs are developed using a design-first approach based on OAS definitions. Security concerns are addressed on an ad hoc basis with no standard process. For an established level of maturity, security is a first-class element of API design that includes standard patterns/practices such as threat modeling.

Development

This vital stage is where the rubber meets the road – developers should ensure they follow security best practices to avoid introducing vulnerabilities into APIs.

A crucial element of secure APIs is the development process, where specifications are implemented in live APIs. Some key considerations here are as follows:

- Choice of languages, libraries, and frameworks
- Correct configuration of frameworks to ensure security best practices are followed
- Defensive coding – do not trust user input and handle all unexpected failures
- Use central points of enforcement of authentication and authorization – avoid "spaghetti code"
- Think like an attacker!

At the lowest maturity level, developers are largely unaware of security concerns in API development or approaches to secure code in general.

At the emerging maturity level, developers are familiar with security considerations and use secure coding practices, albeit sporadically. For an established level of maturity, developers are fully versed in secure code and API security topics and proactively seek to use best practices and defensive coding.

Testing

Without adequate API security testing, an organization runs the risk of deploying insecure APIs – test early, test often, test everywhere.

API security testing is vital to ensure that APIs are verified as secure before deployment. Security testing should be tightly integrated into the CI/CD process and should avoid any manual effort. Tests should be able to "break the build" in the event of failure. The following aspects should be tested:

- Authentication and authorization bypass
- Excessive data or information exposure
- Handling invalid request data correctly
- Verifying response codes for success and failures
- Implementation of rate-limiting and quotas
- Changes in configuration in production environments from their desired target state (so-called configuration drift)

At the lowest maturity level, there is no specific API security testing, with only functional testing in place.

At the emerging maturity level, API security testing largely uses manual testing and lacks automation and CI/CD integration. For an established level of maturity, API security testing is tightly integrated into all stages of the SDLC, and failures can block releases.

Protection

A defense-in-depth approach is the foundation of risk reduction – regardless of how well-designed your APIs are, they will still be attacked by persistent and skilled adversaries.

Despite the best efforts during the preceding phases of the SDLC, APIs will still come under attack and should be protected via dedicated API protection mechanisms.

API protection should include the following:

- JWT validation
- Secure transport options
- Brute-force protection
- Invalid path or operation access
- Rejection of invalid request data
- Filtering of response data

Protection logs should be ingested into standard SIEM/SOC platforms to ensure visibility of API security operations.

At the lowest maturity level, no specific API runtime protection is implemented; standard firewalls or WAFs are the only protection in place.

At the emerging maturity level, some protection is provided, typically using API gateways to provide basic enforcement of rate-limiting, token validation, and more. For an established level of maturity, dedicated API firewalls are implemented to provide localized protection at the API transaction level.

Governance

Trust but verify – a robust governance process is essential to ensure that API development observes organizational methodologies.

The final domain of API security is the overall governance process, which ensures that APIs are designed, developed, tested, and protected according to the organization's process.

Governance covers the following principles:

- APIs are consistent – that is, they use standard patterns for authentication and authorization
- Standard processes, including testing and remediation requirements, are followed to develop new and updated APIs
- Data privacy and compliance requirements are met
- A process is observed for APIs at their end of life to eliminate insecure zombie APIs
- Stakeholders are enabled on API-specific security topics
- Enablement is updated based on emerging threats
- API development is largely ungoverned, with business units each using their own process with no central oversight

At the lowest maturity level, governance addresses only the basic requirements of compliance and regulatory requirements.

At the emerging maturity level, governance is proactive, and APIs are developed to a standard process. For an established level of maturity, deviations are tracked, and discrepancies are addressed.

Planning your program

Now that you have examined the key topics of API and API security ownership and have the foundations of a maturity model, it is time for the rubber to hit the road as you begin to plan your program.

Establishing your objectives

Simon Sinek's seminal TED talk *Start with Why* inspires leaders and organizations to understand their motivation for what they do and the importance of the "why" they do what they do. The same can be said for establishing an API security program – without clear objectives or *raison d'etre*, your program may flounder and fail. You need to understand the compelling reason(s) for implementing a change program of scale. Perhaps you process medical records and cannot risk an API breach disclosing patient data. Or maybe you are a payment processor that is bound by strict regulatory requirements. Or perhaps you are an "API-first" company whose very business succeeds (or fails) on the strength of their APIs and their security.

Find your why, use that to determine your requirements, and plan your program accordingly.

One of the biggest blockers to starting a software security initiative is feeling overwhelmed when establishing the first steps on the journey, particularly if you are responsible for a large estate. With so many applications or APIs, where do we even start? Unfortunately, this hesitancy can undo many initiatives since they fail to even get started, or if they do, the effort is wasted on lesser important assets.

In *Chapter 1*, *What Is API Security?*, I introduced a basic approach to a *risk-based methodology* to prioritize the most important APIs based on your business's security objectives. Use this method to identify your highest priority cohort of APIs, enroll them into your program, and then spread the selection criteria wider to the next cohort and then the next. Avoid the temptation to take on your entire portfolio in one fell swoop as this can lead to early failure when no progress is apparent.

Assessing your current state

You may be working in a greenfield environment regarding API security where nothing is in place, and you are building from ground zero. Or – more likely – you have a brownfield environment where some API security measures are already in place, and you are seeking to improve or mature these. In this case, you must get a good estimate of your current state before making changes.

The first task is to estimate your inventory. This can be done in several ways:

- Scan your Git repositories for identifiers that indicate API code (OAS definitions, route controllers in source code, and more)
- Ingest network traffic and use tooling to identify API-specific traffic
- If you are running an API gateway, extract their inventory of proxies and use this as a starting point

Once you have identified the APIs, use your organizational knowledge to determine ownership of the API, and then use discovery meetings to map their current capabilities to a maturity model (such as the 42Crunch model or more general-purpose models such as **OpenSAMM** or the **NIST SSDF**).

Lastly, decide what controls, processes, and procedures are required to attain your security requirements. Build consensus and awareness with all stakeholders and bake in review, updates, and enablement to ensure longevity and relevance.

Building a landing zone for APIs

In recent years, one of the more useful paradigms that has emerged from the DevSecOps movement is the concept of so-called guardrails or landing zones. The concept is simple: build a zone (comprising elements from across the eight phases of the SDLC) with security tooling baked into the fabric of the zone. The only thing the application development team needs to do is contribute their code; the rest of their environment is already configured optimally for their development process and, of course, for secure development and deployment.

These landing zones can become unwieldy for large, complex, heterogeneous applications requiring many languages, toolkits, SDKs, environments, and so on. However, for APIs, the story is a little simpler. An organization can realistically standardize one or two landing zones and coax development teams to use these secure landing zones to achieve greater API security. It becomes a compelling proposition for security and development teams alike:

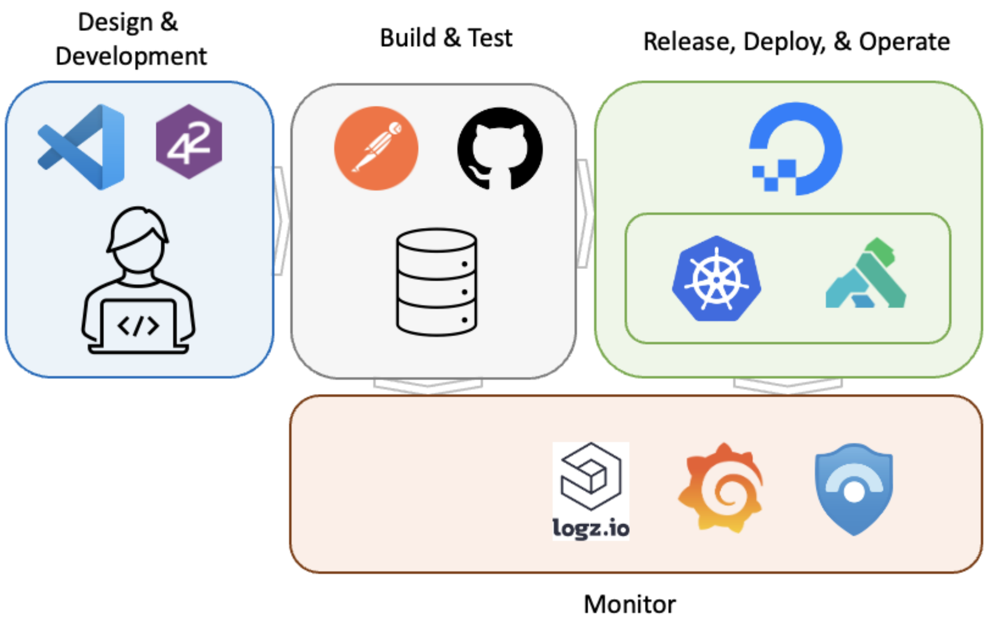

Figure 13.6 – Sample secure API development landing zone

In the sample secure API development landing zone shown in *Figure 13.6*, the following key components are provided:

- A design and development zone based around a VS Code editing experience, with 42Crunch providing OAS definition auditing and instantaneous API testing in the editor
- A build and test zone based on GitHub and its Actions component for automation, and Postman to run automated API testing using a **command-line interface** (**CLI**)
- A release, deploy, and operate zone based on a DigitalOcean tenant (this could be any similar cloud provider) with Kubernetes as a deployment orchestrator using a Kong API gateway as the main ingress from the outside world
- A monitoring zone that uses Azure Sentinel as the SIEM, Grafana as the telemetry and instrumentation dashboard, and `Logz.io` as the logging facility

This is purely an illustrative example; the specific details will vary according to your organization. The key takeaway is that you build out a landing zone with the required elements and then secure that by providing secure defaults and hardened configurations. A developer should be able to go from code to a secured API without having to concern themselves with any intermediate steps.

Running your program

Once you have established your program's goals and identified your stakeholders, you can start running your program. To do that, first and foremost, you need a team composed of the right people for the job. The trick is to find the right people; let's look at some approaches.

Building your teams

First up, you need to build your own team who will work to achieve your objectives. Adam Shostack has written an excellent blog on the topic (see *Further reading*), and his perspectives reflect my reality of having built several large-scale AppSec programs. The key point is the hardest one to grasp: to build an AppSec team, you do not need a team of AppSec specialists. Shostack expresses it perfectly: "*by using exceptional talents doing over-specialization.*" While securing software has an obvious technical element to it, by far, the biggest challenges are human-centric. You will, above all else, require the buy-in and cooperation of your development teams; after all, it is these teams who need to make changes to their processes or fix their code to improve security. What is needed most are diplomats who can lead with empathy and negotiate change. Deep technical specialists are often too inclined to want to solve the problem themselves rather than empowering others to solve their problems. Teams of deep technical specialists will not scale.

Speaking anecdotally from my experience (see the YouTube talk in *Further reading*), I assumed I needed both InfoSec skills (with qualifications such as CISSP and others) and pentest skills when building my team in my first role. Instead, I was assigned a team of generalists with very diverse skills and backgrounds, and we shaped this into a very successful AppSec team.

The other way to approach the scaling issue with AppSec teams is to delegate or outsource much of the day-to-day operation to security-minded members of the application development teams themselves. This is the now popular security **champions approach**, where the champions are responsible for activities such as evangelizing security, developing security standards and policies, running events and activities, doing threat modeling, performing code reviews, and using security testing tools. Think of the security champion as a local extension of your team. Clarity is essential to determine who has ultimate accountability for security decisions. OWASP offers excellent guidance on this topic (see *Further reading*).

Tracking your progress

As your program proceeds, you will need to demonstrate progress. This is best done by selecting a variety of **key performance indicators** (**KPIs**) that reflect the objectives of your program.

Understanding your KPIs

For each metric chosen, several metadata properties indicate the metric's nature:

- **Data source**: This indicates how the metric is measured. Possible values include the 42Crunch platform, platform metrics (GitHub, CI/CD, and so on), SIEM/SOC, the API gateway, and surveys, which are the least accurate as they tend to be subjective.
- **Classification**: This indicates whether the value should increase or decrease to reflect an improvement. Metrics with a decreasing value are good since a value of zero indicates no further improvement is needed, or metrics measured as a percentage with an increasing value are good since 100% indicates no further improvement. For open-ended metrics, a baseline should be established based on the organization's goals and objectives.
- **Unit of measurement**: The units of measurement can be one of the following: a count (a numeric value), percentage, duration (a time or date value), or a Boolean (a yes/no value).
- **Trend**: This specifies whether the metric is increasing or decreasing to indicate improvement. Flaw count is a good example of a decreasing metric, while code coverage is a good example of an increasing metric.
- **Reliability**: This indicates the approximate reliability of the metric. Some metrics that are measured from platforms (42Crunch, GitHub, and so on) can be precise with high reliability, while others can be less reliable, such as detected threats. Others are entirely subjective, such as the cost to remediate.

- **Leading or trailing**: This indicates whether the metric shows the impact of past behaviors (trailing) or that future KPIs should improve based on the measurement based on correlation (leading). Education, enablement, and engagement are examples of leading indicators. Scan results and bug bounty reports are examples of trailing indicators.

Ensure that you understand the nature of your KPIs to determine sensible target values and timescales as these will likely be necessary to achieve them.

Selecting your KPIs

As part of the research in my role as a technical evangelist, I have collated nearly 200 KPIs for API security and made these available in spreadsheet form in this book's GitHub repository.

The KPIs are sorted according to the phase of the SDLC. The typical KPI types are as follows:

- **Plan**: Security definitions metrics, transport security metrics, input validation and parameter security metrics, output and response security metrics, error handling and information disclosure metrics, compliance metrics, and threat modeling metrics
- **Code**: Pull request review metrics, pull request integration metrics, code review metrics, code scanning metrics, and dynamic testing metrics
- **Build**: Dependency and component security, tooling and automation metrics, development, and remediation efficiency metrics
- **Test**: API security testing
- **Operate**: Vulnerability metrics, threat intelligence metrics, performance metrics, and security incident metrics
- **Monitor**: Security monitoring metrics, traffic metrics, and cost metrics

To start with, pick a half dozen or so of the key metrics (and possibly ones you know you can improve), use these as a yardstick for your program, and then expand as your program gathers momentum. From my experience, I would pick metrics based on the measurement of authentication and authorization in the development phase and then coverage metrics for testing.

Integrating with your existing AppSec program

In the *Roles and responsibilities* section, we discussed how most organizations will likely have some form of DevSecOps team performing security testing and scanning activities (see *Figure 13.3*). As you build out an API security initiative, aligning your efforts and activities with those of the DevSecOps team is wise.

Integrate API testing methods

For each of the common security testing types, particular touchpoints applicable to API security are worthy of focus, either by creating specific rules, adjusting the severity categories, or providing remediation advice. For example, here are some suggestions:

- **Static application security testing** (**SAST**): For SAST scans, pay particular attention to issues that have been identified relating to injection vulnerabilities since APIs are prone to injection attacks via external payloads. Also, be aware of findings related to data processing, such as **XML external entity** (**XXE**) based attacks.
- **Dynamic application security testing** (**DAST**): For DAST scans, identify endpoints that are not configured correctly with TLS or fail under high load conditions or with large payloads. These endpoints may be fragile and susceptible to denial-of-service attacks and should be hardened or protected with rate-limiting solutions.
- **Software composition analysis** (**SCA**): For SCA testing, identify packages or components critical to APIs (frameworks such as **Connexion** or **FastAPI** in a Python application, for example) and ensure that the versions that are used are not affected by open vulnerabilities.

Work closely with your AppSec team to leverage their efforts and, in particular, learn from their hard-learned lessons.

Understand your API dependencies

We have already discussed the importance of managing and maintaining your software dependencies to ensure that you are not inheriting risk from vulnerable software. It is also important to understand the provenance of your upstream APIs:

- Do you know if these APIs are built using secure software development methodologies?
- Do you know if these APIs are tested for vulnerabilities?
- In the event of vulnerabilities being discovered, do you have an agreement in place with your supplier to remediate these vulnerabilities?

It is highly advisable to map out a dependency tree of your API infrastructure to identify all contributing elements, including software components or third-party APIs. Use this dependency map to build out a view of your risk profile.

Your personal API security journey

We are now at the end of this book, but that does not mean that your personal API security journey has concluded. I would like to think it has only just started. APIs and API security are rapidly evolving domains, with new technologies (such as GraphQL) posing new risks to organizations. Hopefully, this book has given you a solid foundation in the basics of API security, how to attack APIs, and, most importantly, how to defend them.

To keep up to date on all breaking news relating to API security, including breaches, views and opinions, tools, and techniques, I would recommend the bi-weekly newsletter I curate at APISecurity.io (`https://apisecurity.io/`).

If you prefer a more tactile, hands-on approach to learning, then the good folks at APISecurity University have several online training courses on various API security topics (`https://www.apisecuniversity.com/`).

Happy learning!

Summary

This brief chapter covered the very important topic of building an API security strategy and saw the theory we have learned about API security applied to real-world API development. Understanding who owns your APIs is important in understanding how to drive the messaging around the need for API security. A broad-based approach involving the CISO or IT security organization and their colleagues in the API product development teams is likely to produce the best results since this will include API security touchpoints across all phases of the SDLC.

First, we learned how to plan an API security initiative by understanding our objectives (the "why") and then understanding our current state to form our strategy. We then looked at running a program, focusing on the critical step of building our team and selecting our KPIs to gauge our progress.

Finally, your own continued learning is important for staying on top of emerging threats and changes in technology landscapes.

This chapter brings our journey together to a conclusion. In hindsight, we have covered a lot more material than I might have anticipated when I started with the first chapter nearly 18 months ago. The journey has taken us from the very fundamentals of API requests using HTTP, through to a solid understanding of the core building blocks and then the top vulnerabilities affecting APIs. We then learned how to think like an attacker in understanding how our APIs could be attacked.

The last part of this book honed in on the key topic of defending APIs and took us on a journey of methods and techniques to use throughout the SDLC. A good API security strategy covers touch points from the start of the SDLC (the so-called shift-left approach) to the API runtime (the so-called shield-right approach). Hopefully, at this point, you are inspired and motivated to start securing your organization's APIs.

As mentioned previously, your learning journey is only just beginning, so please do keep in touch, either with me or via this book's dedicated resources, namely the GitHub repository or the YouTube "Code in Action" channel. As I conclude this book, I am happy to report the great success of my first full-day workshop accompanying this book – be sure to keep an eye open for future workshops coming to a venue near you.