Carnegie Mellon University
Software Engineering Institute

On the Design, Development, and Testing of Modern APIs

Alejandro Gomez
Alex Vesey

July 2024

Table of Contents

Definitions

API: An application programming interface (API) is set of functionalities independent of their implementation, allowing the implementation to vary without compromising the users of the component [Bloch 2018].

Authentication: the process of verifying the identity or other attributes claimed by or assumed of an entity [CNSS 2010].

Authorization: access privileges granted to an entity, or process or the act of granting those privileges [CNSS 2010].

Availability: the property of being accessible and useable upon demand by an authorized entity [CNSS 2010].

Chaos Engineering: The discipline of experimenting on a system to build confidence in the system's capability to withstand turbulent conditions in production.

Confidentiality: the ability for a system to disclose information only to authorized entities [CNSS 2010].

Cybersecurity: The Department of Defense defines cybersecurity as, "prevention of damage to, protection of, and restoration of computers, electronic communications systems, electronic communications services, wire communication, and electronic communication, including information contained therein, to ensure its availability, integrity, authentication, confidentiality, and non-repudiation [USG Compendium 2019].

Functional Capability: The ability to perform a task or activity.

Integrity: the property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner. This can apply as well to the integrity of the application [CNSS 2010].

Non-repudiation: assurance that the sender of information is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having received the information [CNSS 2010].

Service Supply Chains: provide services to acquirers, including data processing and hosting, logistical services, and support for administrative functions [Woody 2022a].

Software Assurance: The level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software throughout the lifecycle [DoDI 5200.44].

Stub Function: provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test [Fowler 2007].

System: A collection of components working together towards a common goal [Sudjari 2018].

Preface

Application programming interfaces (APIs) are a fundamental component of modern software applications; thus, all software engineers are designers or consumers of APIs. APIs enable powerful abstractions by making the system's operations available to the end user, limiting the details of how the APIs are implemented.

This white paper covers the following topics:

- What is an API?
- What factors drive API design?
- What qualities do good APIs exhibit?
- What specific socio-technical aspects of DevSecOps apply to the development, security, and operational support of APIs?
- How are APIs tested, from the systems and software security patterns point of view?
- What cybersecurity and other best practices apply to APIs?

An Introduction to APIs

APIs can be found almost everywhere software is used [Myers 2016], and their design, development, and testing are a critical part of any software project. Web services are commonly referred to as *APIs*. The public methods of an API class are sometimes referred to as its *API* because they expose the functionality of the API class to other programs. APIs are also used to specify the interface between applications and the operating system, as the Portable Operating System Interface (POSIX) and Win32 do in providing a common set of APIs to enable portability.

Because the term API is used in many different domains, it is necessary to find a definition general enough to apply in all these cases. According to the National Institute of Standards and Technology (NIST) Glossary, an API is a "system access point or library function that has a well-defined syntax and is accessible from application programs or user code to provide well-defined functionality" [NIST Glossary]. As NIST defines it, an API is a *system access point* or source of access to system functionality that has a *well-defined* syntax (i.e., a syntax that is standardized and agreed upon) and is accessible to the calling applications. The latter functionality is key since it assigns the single responsibility of an API: communicating available operations of a system to the calling applications.

Joshua Bloch provided an alternative definition of API in his presentation "A Brief, Opinionated History of the API," in which he defines an API as "a set of functionalities independent of their implementation, allowing the implementation to vary without compromising the users of the component" [Bloch 2018]. By keeping functionalities separate from implementation, software engineers can use *abstraction,*[1] one of the core principles of software engineering. For example, the Java Cryptography Service Provider [MIT 2004] allows engineers to create a subclass from one of the existing cryptography classes in which they can implement their own encryption while still using the same interface as other encryption methods. The interface is therefore separated from the implementation.

This powerful capability enables another principle: *abstraction layering*,[2] whereby engineers build programs that are based on others without needing knowledge about their internals or how they work (see Figure 1). In this way, engineers can build larger, more complex systems connected via APIs. Ideally, each software module exposes its API as being independently developed and thus capable of being reasoned about separately. However, some software modules must depend on others to operate, in which case API development for one software module depends on the development of the other module. If class B inherits from class A, then B depends on, and must implement functions of, A. Changes in class A will require a change in class B.

[1] According to John Guttag in his book *Introduction to Computation and Programming Using Python,* the essence of abstraction is preserving information that is relevant in a given context and forgetting information that is irrelevant in that context [Guttag 2021].

[2] In his book *Beautiful* Code, Diomidis Spinellis noted that "All problems in computer science can be solved by another level of indirection" [Spinellis 2007].

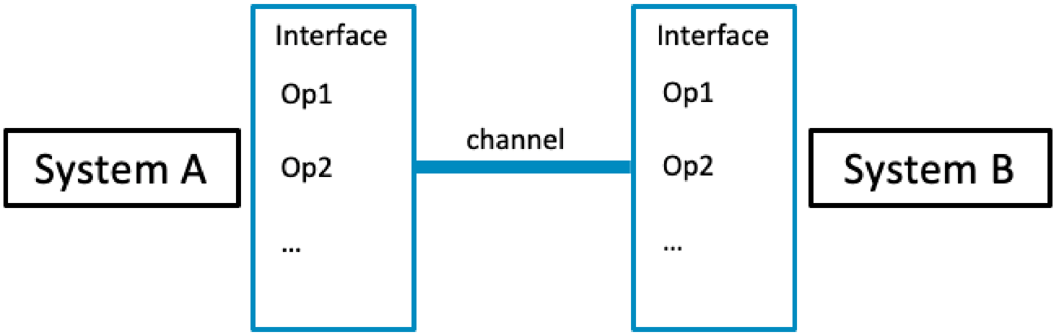


Figure 1: APIs Abstract the System Implementation from the Set of Operations it Exposes Via its Interface. Black boxes are closed-off implementations while interfaces, represented by blue boxes, make a set of operations available for the other system to use across a channel.

APIs are valuable because they allow us to reason about the operations a piece of software should be able to provide while deferring their actual implementation.[3] Further, APIs save their consumers from having to constantly recompile or change their code whenever an API provider changes its implementation. This holds true as long as the syntactic and semantic interface is preserved (unless the implementation changes the behavior of the software).

Having an effective understanding of what an API is and what it's not allows developers and project managers to have a grasp of API's inherent capabilities (i.e. allowing for communication between systems, abstracting interfaces from their implementation and reducing complexity). On the flipside, the definitions provided also clue us in to what APIs cannot do: they do not inherently make systems safer (since they create increased attack surfaces), they do not increase interoperability of systems (since the functions must be understood by both systems first) and they do not make software delivery faster (this is discussed in a later section in detail).

Next, we will look at the effective design of APIs.

[3] In his book *Test-Driven Development by Example*, Kent Beck noted the following: "When we write a test, we imagine the perfect interface for our operation. We are telling ourselves a story about how the operation will look from the outside. Our story won't always come true, but it's better to start from the best-possible application program interface (API) and work backward than to make things complicated, ugly, and 'realistic' from the get-go" [Beck 2014].

Design of APIs

> *Good design is actually a lot harder to notice than poor design, in part because good designs fit our needs so well that the design is invisible* [Norman 1988].
> —Donald A. Norman, *The Design of Everyday Things*

An engineer must consider several issues when designing an API. Software is more complex than ever, and when that software frequently performs life-critical functions the security-related goals of confidentiality, integrity, and availability become essential features. In addition to the functional and nonfunctional features that an API designer seeks to deliver, the API must provide engineers and users a qualitatively good experience and promote sharing; otherwise, no one will want to use it. This section describes key functions and attributes an API designer needs to address and how to test for them.

Qualities of Good APIs

In this paper, we make a distinction between *quality of implementation*[4] versus the *quality in use.*[5] While subtle, the distinction stems from the quality attributes that make a software product functionally correct (that is, it meets functional requirements) and the quality attributes that make a software product usable. When providing an interface to a functional capability, the goal of the design should be to give the user just enough information and control to support their use cases. This goal supports the concept of *least privilege*,[6] which ties directly to functional suitability and its two sub-characteristics, completeness[7] and correctness.[8] “Functional suitability describes the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions” [ISO/IEC 2011]. In addition to fulfilling all use cases, an API exhibiting the characteristics of completeness and correctness implies that it has a high-quality implementation that refines the specification, provides all the functions specified, and does so with the expected results (see the Contract Testing section).[9]

[4] Quality of implementation characterizes how well a software product conforms to its requirements [IEEE 2024].

[5] Quality in use characterizes how well a software product meets the needs of the users [IEEE 2024].

[6] The principle of least privilege stipulates that each entity is granted the minimum system resources and authorizations that the entity needs to perform its function [NIST 2024].

[7] Completeness is the “Degree to which the set of functions covers all the specified tasks and intended users' objectives” [ISO/IEC 2011].

[8] Correctness is the “Degree to which a product or system provides accurate results when used by intended users” [ISO/IEC 2011].

[9] A *contract* in the realm of APIs formally defines a specification of each function or method that the system exposes.

Engineers looking to improve the design of APIs can find a good set of quality attributes in *Application Programming Interface (API) Technical Guidance*, published by the Office of the Executive Director for Systems Engineering and Architecture [DoD 2023c]. These attributes include reusability, reliability, interoperability, discoverability, scalability, security, compliance, and completeness. We will discuss each of these quality attributes more thoroughly throughout this paper.

The user experience of an API should also be considered in its design. *Application Programming Interface (API) Technical Guidance* includes this consideration as a quality called "Easy to use," which falls into the category of quality in use (see API Usability). A good API makes correct use easy and misuse difficult (see Secure Design). While considering user experience, recognizability and accessibility are also characteristics of a good API. If an API presents terms a user is familiar with, and those terms have definitions consistent with the user's view then the API will have a good user experience. The principle of *least astonishment* states that a software component should behave in a way that most users expect. One caveat to this not on using familiar terms is the guidance that APIs should not use overly technical terms or jargon [Fahl 2013]. Also to be considered is Accessibility in terms of APIs could be *discoverability*. A good-quality API enables users to explore what functions the API can perform while reducing the burden of documentation. For example, the API itself can return a list of functions or endpoints automatically.

Engineers can measure an API's quality both statically and dynamically. For instance, engineers can statistically measure quality of implementation using static analyzers that check for code conventions and common errors (see Testing). These static analyzers are discussed more in the Testing section. Conversely, some aspects of an APIs quality can only be evaluated dynamically by engaging real users and gathering feedback (see Testing Skill Sets section). However, the line between what can be tested statically and what can be tested dynamically is not well defined. For example, engineers can assess the scalability of an API by looking at its architecture (is it one server? Many?) but may require load testing for getting accurate metrics on its performance. Likewise, engineers can conduct user acceptance testing on a live running system, or they can conduct a review of endpoint and function names without running any software. Therefore, it is important to measure quality using methods appropriate to each requirement (see Testing). Even if an API is technically sound and performs well, an API that is not user friendly could be considered of low-quality.

API Usability

Often, APIs are difficult to use for users of all experience levels. Difficulty of use can cause faulty programs or open significant security vulnerabilities [Fahl 2013]. The field of usability, sometimes called human-computer-interaction (HCI), investigates how end users use software.[10] Usability research suggests why API design is so difficult: There are many quality attributes an API might be evaluated against, and there are often tradeoffs among them. According to Myers and Stylos,

[10] In this case, the term *end user* refers to the API designer, consumer, or person using an API.

> *Usability includes such attributes as how easy an API is to learn; how productive programmers are using it; how well an API prevents errors; how simple it is; how consistent; and how well it matches its user's mental models. Power includes an API's expressiveness (the kinds of abstractions it provides); its extensibility (how users can extend the API to create convenient user-specific components); its evolvability for the API designers who will update the API and create new versions; its performance (in terms of speed, memory, and other resource consumption); and the robustness and security of the API implementation and the resulting application* [Myers 2016].

Starting from these usability attributes, which fall under *quality in use*, API designers can confidently create features, test whether the final product fulfills these attributes, and gain assurance that the API is usable. Effectively implementing usability attributes in APIs has been shown to have positive outcomes. For example, a redesign of the SAP[11] API using usability principles was shown to improve users' success and time to completion [Myers 2016]. Further studies on the effect of usability on APIs have provided data-backed insights into effective practices that enable the creation of usable APIs. The following is a short list of usability-informed decisions and their outcomes:

1. **Use of Design Patterns**. Some popular software patterns, like the Factory Method pattern, have been shown to make the API less usable and harder to learn [Ellis 2007]. Contrary to best practices, the Factory pattern promotes creating immutable objects, whereas some users prefer to start with a null object and set properties to it as required. The caveat to this finding is that it might be worth trading off some usability in favor of security and correctness in a program. The decision to use a software pattern is less costly to implement earlier in the development lifecycle than after it has been deployed.

2. **Usability**. APIs that demonstrate high usability can improve the security of an API. In a study of thousands of iPhone Operating System (iOS) and Android apps, many were incorrectly using Secure Sockets Layer (SSL), which made them vulnerable to machine-in-the-middle attacks. The causes for this vulnerability were found in the confusing experience that engineers faced in creating self-signed certificates for test environments and code-level customizations of the SSL validation process (e.g., SSL pinning). A cohort of engineers were provided a framework of the SSL library that disallowed insecure settings, and this change made it easier to set up SSL certificates in a secure way. The cohort using this modified framework significantly improved the security of their apps. This research highlights how usability research can improve the security of apps by making it easy for engineers to "bake" security in and make it difficult to apply insecure settings [Fahl 2013].

3. **User Expectations**. Empirical guidelines for evaluating the effectiveness of API designs exist [Piccioni 2013]. These guidelines include identifying aspects of usability in a specific domain, collecting feedback and data on the 11 cognitive dimensions outlined by Steven Clarke [Clarke 2014], and comparing user expectations to provided services along those dimensions.

[11] SAP is a German multinational software company.

4. **Parameter Ordering**. The order of parameters in methods matters for usability. "Inconsistencies in the sequencing of similar parameters in the method signatures can sow doubt in the mind of an engineer at the time of writing code as to whether he/she really remembers the correct method signature for a given method call" [Rama 2013]. Consistent ordering and naming increased usability.

5. **Natural Programming**. Usability studies have shown a positive outcome from gathering requirements using the natural programming technique provided by Myers and Stylos [Meyers 2016].

Just as the developers of front-end applications, such as websites and mobile apps, spend time and resources understanding aspects of the user experience with a graphical user interface, the experience of the user working with an API must also be considered. The usability of an API has wide ranging implications for its security, efficiency, effectiveness, and adoption.

Evolution and Interoperability

APIs change over time. Newer features are added, and existing features are modified, deprecated, or removed. Usage patterns may change. Asking users to change the way they call an API every time a new feature renders the API ineffective. To be effective, an API needs to provide mechanisms to help API engineers incorporate changes, such as improving the user experience, making it easier to modify, and enhancing existing API functionality. An API that exhibits these attributes is said to be evolvable: "An attribute that bears on the ability of a system to accommodate changes in its requirements throughout the system's lifespan with the least possible cost while maintaining architectural integrity" [Brievold 2008].

In addition to the evolvability of the system, engineers must consider the compatibility of the data being exchanged and the ability of the interfaces to accommodate changes over time. Forward and Backward compatibility are attributes of an API that allow for multiple data encodings, and a stable interface over multiple versions [Kleppmann 2017]:

- Backward compatibility: "Newer code can read data that was written by older code."

- Forward compatibility: "Older code can read data that was written by newer code."

Thus, the API is said to be interoperable because accepts data from old versions as well as new ones and can write across different versions with its existing interfaces.

An API engineer must address evolvability throughout the software development lifecycle. Two strategies that address evolvability are versioning and data feature toggles. Interoperability and API versioning are often closely related, but there is no real consensus on this topic.

Versioning

Murphy et. al reviewed 32 different style guides from industry that provide guidance for REST API versioning and found conflicting guidance [Murphy 2017], including guidance on how to identify a version and what different version numbers and formats mean in terms of compatibility. One common thread is the use of semantic versioning [Preston-Warner 2024]. This standard enables users to easily determine the impact a new version has on compatibility if the standard is followed.

Another principle observed in the research and found in early revisions of the TCP standard and in IETF RFC 1122 is Postel's Law, "Be liberal in what you accept, and conservative in what you send." For APIs, this means following the concept of defensive design [NIST 2024]. Two broad approaches exist: either fail gracefully or fail fast. In the former, if an API expects one of four possible enumerations and receives an unspecified fifth type, it should not crash but log the error. This approach enables robustness and is commonly found in web applications. In the latter case, an error detected could be a sign of an intrusion or bad data, which requires immediate handling of the process and may disallow further action from a client. This approach is more common in security-sensitive applications, such as banking or defense. Following these design principles and the guidance in RFC 1122 can lead to more robust systems at the cost of increased error handling and testing.

Data Formats and Encoding

Being conservative in what an API sends is the approach taken by GraphQL, a query language for defining APIs that enables clients to ask for only the data that they specifically need. This contrasts with other APIs, where an endpoint may send back a JSON payload with irrelevant information that is bundled along with the data that is really needed. Any API could suffer from over-fetching of data, but query languages provide granular control over what can be fetched. This enables predictable results from an API, even if new information is added later. Owing to this backward and forward compatibility feature, the GraphQL foundation has issued guidance to avoid versioning of this type of API [GraphQL 2024]. This practice is not limited to only GraphQL APIs but can be enabled through small, targeted API endpoints or functions, that return only the needed data. This practice can also enable security through the principle of least privilege (see Secure Design).

In a similar vain to the GraphQL approach, there are design patterns that can enable backwards and forwards compatibility in an API. One such pattern is the mutation of an API request's data or function call's parameters to the latest API specification. This can also happen in the reverse where an API's return can be modified to match the format a client is expecting. These transformations may be applied sequentially or as specific version X to version Y conversions that enable older clients to continue working [Stripe 2017].

When designing an API, the choice of format for the data can be as important as the previously mentioned consideration of what data to send. There are many open standards for data exchange. Popular web API standards include JSON [ECMA 2017], HTML [WhatWG 2024], and XML [W3C 2008]. ITU and ISO standards include ASN.1 [ITU 2024] as a similar open standard for data exchange born in the telecommunications industry. Also common is the use of Protocol Buffers, a standard for serializing data that is common in lower-level APIs, such as those using gRPC [gRPC 2024]. Each of these standards is language independent and allow APIs and client libraries for the API to be written in different languages, further decoupling the code that uses the API. One important note is that the data format that is used becomes part of the API and, consequently, changes in the data structure can impact the versioning and compatibility of an API.

Feature Toggles

A team evolving an API will generally need to balance delivering new features to users and maintaining current capabilities. Within an API, it can be complex to have multiple code paths for the same function call or endpoint and also manage which one is active. A strategy for dealing with this complexity is *feature toggles*. This pattern enables setting of a variable to enable one code path versus another. Doing so will allow for an easy way to select features in each build of an API. Feature toggles enable canary releasing, in which a small portion of API users are provided the API with a new feature enabled and the remainder are served the current version. This practice extends to A/B testing in which a feature toggle is used, and some measurement is taken regarding the use of the API. Feature toggles enable experimentation because a small subset of users can be opted in, which preserves the user experience of the majority. Feature toggles also enable easier rollback of defects by changing an input or variable rather than deploying or recompiling a new version of an API. This practice can lead to better system stability and uptime when testing new features. To ensure feature toggles are easy to use and roll back, it is important to manage their use and implementation. One common issue encountered is the coupling of the decision point (generally an if statement guarding the execution of one code path versus another) and the toggle (the Boolean value that determines whether the feature is on or off). Centralizing the feature toggle management in a system can be an effective way to manage complexity while harnessing the power of feature toggles to evolve an API [Fowler 2017].

Secure API Design

To design secure APIs, recognize that such activity belongs within the wider context of designing sufficiently secure software. Limiting focus to unit tests, input validation, and authentication does not ensure an API will be secure. APIs are almost always deployed with other software components, which in turn are deployed as part of a system. These deployment environments are fundamentally complex: The interconnectedness between components makes it hard to reason about the system, the codebase becomes cryptic to its own authors over time, and the system often fails unexpectedly. The state of such software systems invites attackers to exploit its inherent vulnerabilities. Furthermore, the security of a system is as strong as its weakest link: Insecure passwords, fake personnel credentials, unencrypted network traffic, and a host of other weaknesses can break the confidentiality, availability, and integrity of a system even when these are used to a very high degree. For example, in 2015 a group of teenagers hacked the email of CIA Director John Brennan by posing as a Verizon employee, tricking Verizon support into revealing the director’s phone and last four digits of his bank account. They later used this bank account to reset Brennan’s AOL password and gain access to his AOL account [Zetter, 2017]. Securing APIs, therefore, requires a wider conversation about how to manage risk and secure software systems.

Creating secure software requires answering questions surrounding trust such as: How can you know the API works as intended? How do you know what to test and when to stop testing? These questions are pertinent to the field of software assurance. Having knowledge of the principles and practices of building “trustworthy” [Saydjari 2018] API systems coupled with a methodology, such as the Software Engineering Risk Analysis (SERA) framework [Alberts 2014], for building a robust threat model should provide API designers the guidance necessary to write Security requirements and the features that implement them.

API Assurance

How can you verify that the API is working as intended, even in the face of adverse conditions?[12] You can check that the implementation matches the requirements (if you have them), but even if the API does meet the specification, the requirements themselves can produce unintended behavior or be ambiguous.

API assurance is achieved by repeated and diverse demonstration of the system’s capabilities. Examples of such demonstrations include proper functioning of the system in both nominal and unexpected conditions, test case reports, penetration testing, and so on. These activities provide increasing assurance to the users of the API that it works as intended and can be trusted, even when things go wrong. Just as in social situations, trust is not a one-time experience but an ongoing relationship, and trust in API systems must be continually evaluated across the software lifecycle. In other words, it is not enough to perform software assurance activities in a single phase, such as the design phase, but rather to do so in the requirements gathering, development, testing, and deployment phases. Activities traditionally thought of as software assurance-focused, such as penetration testing, vulnerability scanning, and unit and integration testing, should also be “shifted left” [IBM 2024]. Extreme Programming and agile development practices are methodologies that encourage this style of work and, consequently, provide higher software assurance.

Breaking the application, either through intentional system failures or through targeted penetration exercises, produces even more confidence in the API because doing so informs its designers of the limits of what it is capable, and not capable, of doing. The following seven principles, which also apply to API development, address the challenges associated with building, deploying, and sustaining software systems while achieving a desired level of confidence [Mead 2016]:

1. *“Risk drives assurance decisions.”* Many times, organizations think of risk when they see data breaches on the news, or cyber-attacks they’ve experienced in the past. This fails to consider the breadth of risk that is spread across the organization and the multitude levels of risks that can be

[12] Adverse conditions refer to conditions that challenge the system to work correctly. These can either be unintentional system failures (e.g., bugs) or intentional ones (e.g., DDoS).

potentially exploited. To create effective assurance, organizations need to be proactive in analyzing where risk resides in their systems and personnel and come up with mechanisms and controls to manage them. The SERA framework described in the next section is a good place to start thinking about risk in the organization.

2. *“Risk concerns shall be aligned across all stakeholders and all interconnected technology elements*.” Security is only as strong as its weakest link, and if security controls are not placed across all facets of an organization, then the systems remain vulnerable to attack. In 2024, United Health was breached because one of its systems did not have multi-factor authentication. This allowed attackers steal patient health data by accessing the system using a stolen password [Siddiqui 2024]. All stakeholders must be aligned in order to consider critical access points, standards, and areas of risk.
3. *“Dependencies shall not be trusted until proven trustworthy*.” With the rise of supply chain attacks, including on open-source software that is increasingly used for mission-critical tasks, these dependencies need to be scanned for vulnerabilities to ensure their integrity. Additionally, these dependencies are not static and must be continually reviewed to mitigate zero-day vulnerabilities and ensure required patches have been applied.
4. *“Attacks shall be expected*.” Organizations are starting to realize that it is not a matter of *if* they’re going to be attacked but *when*. The variety, persistence, and sophistication of attacks means that some compromise of the confidentiality, integrity, and availability of assets is likely. Having processes in place to properly defend and mitigate attacks is part of an API system that provides assurance.
5. *“Assurance requires effective coordination among all technology participants*.” It is not enough to have assurance controls in place if personnel do not know how to implement them. Assurance requires that all members of the technical organization are aligned to the mission of assuring effective service and are knowledgeable of the values, principles, and practices that provide it.
6. “*Assurance shall be well planned and dynamic.*” Change is the only constant. Just as methodologies (such as Agile and XP) have incorporated change into the development process, so too must assurance activities cope with change. Performing “Big Bang”-style exercises with long scripts makes it hard and tedious to change when the threat environment or risk level changes. Organizations that create APIs need to be able to quickly assess and change their assurance posture as needed.
7. “*A means to measure and audit overall assurance should be built in.*” Assurance is an activity that is both qualitative and quantitative: It is perceived by its users as trustworthy and measured in a multitude of ways by its system designers. It is impossible to know what a “normal” operating condition for an API is if it hasn’t been measured. As Meade noted, “Organizations cannot manage what they do not measure, and stakeholders and technology users do not address assurance unless they are held accountable for it” [Mead 2016]. At the same time, it’s important not to be guided by a single metric, as it can become the objective rather than the proxy it stands for.

More information on these principles can be found in the book *Cyber Security Engineering* [Mead 2016]. The principles are not meant to be exhaustive, and they don’t fit neatly in a single software development phase. Starting a new API project with these principles in mind, however, can guide project managers and engineers to consider practices that can be adopted and create work units to develop them.

The DevSecOps model allows for API assurance activities to take place across the development lifecycle. It provides fast delivery of features while maintaining high quality and continual feedback. This feedback can be used to assess whether the work being done is helping, or becoming a hindrance, toward developing a trustworthy API system (see DevSecOps).

Another way to provide API assurance is to build the ability to know when an API is going through adverse conditions and maintain availability to its end users. Modern API systems are constantly failing in some way—some minor, some significant. If you assume that all failure is due to unintentional bugs, you become unable to properly identify an actual *attack*, which can delay response and lead to more damage. Consequently, good monitoring and logging tooling is essential for creating a baseline of "normal" system behavior so that malicious actors can be properly identified. (See the sections Zero Trust Design and Zero Trust Testing for more information on this type of monitoring.) Once an attack is recognized, the API system needs a process to respond and respond or deal with the threat. This capability requires cyber controls to monitor attacks, discover their origin, analyze their impact, and describe potential next steps to mitigate them.

Bridging the gap between the practices described here and the goal of API assurance requires a high-level methodology that can discover the areas of risk to which an API system will be exposed and the threat actors it might face. Because the risks and threats faced by software systems are varied, the input to the system design must be shared among all relevant stakeholders. In the next section, we discuss a framework for thinking about risk and how to design a system that is resilient in the face of risk.

SERA Framework

Traditional security engineering addresses security issues during the operation and maintenance phase of a project. However, the cost of making changes during these later phases of the development lifecycle is significantly higher than doing so earlier. The Software Engineering Risk Assessment (SERA) framework [Alberts 2014] offers a set of activities for major stakeholders and representatives of a business or organization that seeks to design a secure system. The framework was developed to be more effective than traditional security risk analysis methods and has been used across many organizations with restricted environments where security is paramount. We provide a high-level summary of the framework here to showcase its ability to uncover threats in the requirements phase.

The SERA approach is divided into four different activities that seek to uncover where risk resides in an organization and construct security risk scenarios that can be acted on by individuals and the organization. The following list summarizes these four activities:

1. Identify the different components of the system and analyze how data (especially valuable data) moves across it. This step can include multiple views such as the networks, physical facilities,

departments, etc. Close attention must be paid to the boundaries at which data leaves one part of the business and enters another. Security attributes of data (confidentiality, integrity, and availability) are noted.

2. Identify risks for paths over which valuable data travels or systems on which it is stored (i.e., a network or a database). The designer must explore how someone could exploit vulnerabilities in the system to acquire valuable data.

3. Identify enablers that would allow a threat actor to exploit vulnerabilities, such as administrator credentials, master keys, ssh access to servers, etc.

4. Write scenarios in which such risky events are most likely to happen (i.e., a laid-off IT administrator decides to purge the database, a user writes down a PIN or password and loses the note, or an engineer codes a backdoor into the system).

To help participants think of the possible threats their organization or system may face, they can consult the library of Threat Archetypes. Otherwise, they can construct their own custom scenario as detailed in the paper *Using Model-Based Systems Engineering (MBSE) to Assure a DevSecOps Pipeline Is Sufficiently Secure* [Chick 2023]. Appendix B of this paper lays out the structure of [Actor] [Action] [Attack] [Asset] ([Effect] | [Objective]) for each scenario. This structure allows for translation of the threat scenarios to a written statement or to test cases that ensure a threat is mitigated or otherwise properly addressed. A structured risk scenario composed using this approach would look like this: "A disgruntled employee deletes keys to deny access to a database to prevent the system from operating."

A more complete example can be found below:

Part	**Description**
Activity	Create/modify/delete access keys for the database.
Actor	Insider threat
Action	Delete keys from the system.
Attack	Access prevention
Asset	Software keys/certificates
Effect	Prevent access to the database, slow down or prevent organization from performing changes.
Objective	Deny system access to the database.
Statement	A disgruntled employee deletes keys to deny access to a database to prevent the system from operating.

These activities will provide a much more comprehensive picture of the system's areas of risk than a traditional cybersecurity activity, such as brainstorming, could accomplish. In effect, a model of the whole system is made based on employees' different perspectives and experiences on processes. From these views, individuals can determine how likely each risk scenario will occur and the magnitude of its impact.

Spelling out risk scenarios in this way will help prevent the introduction of design flaws pertinent to the design of the system, which can become particularly costly (or impossible) to remove after the system is implemented. Since threats tend to concentrate on trust boundaries where data enters or leaves a component of a system, APIs are an especially valuable target [Shostack 2014]. An API that exposes data or allows a malicious actor to penetrate the defensive perimeter of a system can not only compromise its own functionality but the functionality of the system that controls it. Performing SERA activities on an API will identify the risks and scenarios that will inform requirements in development, which in turn leads to the appropriate testing criteria and stages that will verify the security controls (see Cybersecurity).

AI-Aided Design

Large Language Models (LLM) have been shown in some cases to enhance software requirements elicitation [Ray 2023] and may help to ensure the proper classification [Hey, 2020], completeness [Luitel 2024], co-referencing [Wang 2020], and clarity [Sridhara 2023, Ezzini 2022, Moharil 2023] of requirements. With these advances, artificial intelligence (AI) in general and LLMs specifically represent new tools for designers. However, even specifically trained LLMs are not able to solve all the issues related to code writing [Chen 2021] much less considerations related to architecture [Ahmad 2023]. Despite these limitations, AI can help in API design. Using AI to refine existing work, analyze a natural language description of a requirement, or verify the computer-readable representations are valid use cases. Additionally, some LLMs have the powerful ability to provide a set of different options to the user's query. Designers could use this function with the Alternative Approaches pattern described by [White 2023] to iterate on a design they may need to flesh out. LLMs can also make recommendations of an API call [Wei 2022]. Regardless of the AI use case, designers should verify all information before use. The user of any AI-aided design tools still needs to exercise expertise over the tool. AI models suffer from *hallucinations* [Google 2024], the general term for several types of incorrect responses provided by the models. Hallucinations can include misleading or simply made-up information or, in the case of API design, a recommendation that does not meet the needs of the users.

Because the application of LLMs to software design in general has so far produced mixed results, any approach to using LLMs for API design should be limited in scope. In the design phase, engineers can use LLMs to manage the non-deterministic nature of the output [Fowler 2023]. For example, engineers can prompt the LLM multiple times or, with slight variation in prompt, can explore alternatives the LLM might generate. Engineers could also have the LLM generate multiple options from a single prompting. Doing so, a software architect could use parts and pieces of what the LLM produces as building blocks. Other uses could include having an LLM analyze an existing architectural concept document and providing feedback. This approach may help software architects ideate and catch obscure issues, but any results must be evaluated considering the hallucination issues previously discussed.

System maintainability and extensibility must also be considered when using LLMs and AI models to design and architect APIs. As mentioned, most LLMs are non-deterministic, meaning that the same prompt will not always produce the same output. Likewise, even slightly tweaking a prompt may produce completely different results [Salinas 2024]. Consequently, you should not expect a design to be accurately reproduced from these non-deterministic LLMs. You should also be concerned about feeding proprietary information into LLMs, including not only the information used to train the LLM but also information used in prompts and outputs [MITRE 2024a, OWASP 2024d].

Zero Trust Design Principles

The Cybersecurity and Infrastructure Security Agency (CISA) *Zero Trust Maturity Model* notes that "Zero trust provides a collection of concepts and ideas designed to minimize uncertainty in enforcing accurate, least privilege per-request access decisions in information systems and services in the face of a network viewed as compromised" [CISA 2023]. Zero trust (ZT) concepts have direct applicability to API design. APIs provide engineers and users access to information, compute resources, and more. Consequently, APIs are resources in a ZT architecture (ZTA). When designing an API with ZT in mind, there are four important considerations covered next.

First, ZT assumes that the network is compromised. This means that there is no distinction between an *internal API* and an *external API*. While some concerns, such as authentication, are usually handled for both internally facing and externally facing APIs, it is now also important to consider other security aspects, such as leaking information to authenticated users who do not need to have some aspect of the information.

Second, ZT assumes the management of devices and access points. To manage, monitor, and secure these APIs, a team must know which ones are being exposed. For example, some web applications provide a browser-based interface and an API on a different port of the server. The team may use only the graphical portion of the application, so it is necessary to turn off or otherwise secure the exposed API. Running different versions of APIs provides a second example. Old versions may not have security patches, and new (or beta) versions may have new *zero-day* bugs [OWASP 2023]. Knowing which APIs are deployed on a system is critical to securing them.

Third, ZT assumes fine-grained access control that is context-aware [CISA 2023].[13] The goal is to diversify the information used to grant access to resources, in this case via an API. As we previously mentioned, an API can and, in some cases, should transfer the responsibility of authentication and authorization to other services if the environment already provides those functions. Engineers should

[13] Context-aware systems incorporate not only authentication and authorization but also device compliance and/or other data sources.

take this into account when considering how a specific API will handle requests. The recommended approach for APIs in a ZTA uses short-duration tokens or other authentication to ensure each request is valid only for a given session., However, this approach involves an engineering tradeoff, depending on the system requirements.

Finally, when building a ZTA and providing APIs as a part of the system, avoid proprietary APIs. In Special Publication (SP) 800-207, NIST identified reliance on proprietary APIs as a systematic problem [Rose 2020]. For instance, reliance on APIs provided by only one company causes vendor lock-in. Also, if a vendor changes the behavior of the API, the change can interfere with the interoperability of the systems. In the standard, NIST identified several open-source APIs relevant to ZTA available from groups such as the Internet Engineering Task Force (IETF) and the Cloud Security Alliance (CSA) as viable alternatives to proprietary APIs.

Development of APIs

Securing the Supply Chain

When developing APIs, the risk of supply chain vulnerabilities must be managed [Woody 2022b]. Executive Order 14028 mandates such risk management for federal IT systems [White House 2021] owing to risks stemming from third parties providing a product or service that is malicious, counterfeit, or vulnerable, thereby making the API development system, or deployed product, susceptible to attack. This risk can be viewed from two perspectives: (1) the software supply chain risk and (2) the service supply chain risk [Woody 2022a]. From the software supply chain perspective, there is the risk of third-party dependency in the form of a library that is built with or into the source code of the API. Mitigation strategies for this type of risk include generating and evaluating Software Bills of Material (SBOMs) [Alberts 2023] as well as static code scanning and configuration management of source code [Boyens 2022].

From the service supply chain perspective, the use of third-party APIs to build a system carries risk. NIST SP800-161 notes that external services can reside both inside and outside of authorization boundaries [Boyens 2022]. Unsafe consumption of external APIs is a top-ten risk according to the *OWASP API Security Top Ten – 2023* [OWASP 2023], because engineers tend to conduct less input sanitization or validation on data from APIs. Consequently, when developing an API, one aspect of securing the supply chain is understanding what risks third-party APIs pose to the system.

The use of third-party APIs as a service poses a slightly different risk than third-party code built into the product. What happens if the third-party API goes offline? How is a secure, authenticated connection established? Can the input to the system from the third-party API be tested? Engineers should address these other related questions before integrating a third-party product or service into the system. NIST SP 800-161 provides guidance on establishing a supply-chain risk management practice [Boyens 2022].

DevSecOps

DoDI 5000.87 defines DevSecOps as “an organizational software engineering culture and practice that aims at unifying software development, security, and operations” [DoD 2020e]. This instruction establishes policy for the software acquisition pathways, and it specifically calls out DevSecOps as a recommended development strategy. You should note that DevSecOps is “an entire socio-technical environment that encompasses the people in certain roles, the processes that they are fulfilling, and the technology used to provide a capability that results in a relevant product or service being provided to meet a need” [Chick 2022]. When planning the development of an API, focus on the people and processes over the specific tools and technology backing the API.

APIs were previously defined as a software design pattern. Thus, from a DevSecOps perspective, API development is indistinguishable from other software design patterns, such event-driven architectures or creating factory patterns. APIs do not influence the *Dev* portion of DevSecOps. From taking requirements or user stories from the backlog, to implementing features, to deploying them, APIs do not present anything fundamentally different from other software types or patterns that should be tailored.

Both the versioning and architecture of APIs influence the *Ops* portion of DevSecOps, including how continuous integration (CI) and continuous delivery (CD) (CI/CD) activities are conducted. For a system that is making an API available, the ability to perform continuous integration and delivery will be affected based on the versioning strategy. If the team determines that backward compatibility of any one version with another is not needed, this can hamper continuous delivery because coordination with consumers, or clients, is needed to ensure everyone is aware of a deployment with a recent change. If the clients of the API are not on the same schedule or need to retain an old version, the resulting mismatch can cause issues. Conversely, if the team is using a strategy to ensure its API is always kept backwards compatible, there is no barrier to deploying a new version that is available to all clients as soon as the change passes testing. However, the downside to always maintaining backwards compatibility is that it can restrict the introduction of new functions or innovations. Neither versioning strategy is better, but how an API is versioned and developed affects the deployment strategy. Consequently, the use of APIs in a system affects the DevSecOps pipeline and process.

Security in DevSecOps

In a traditional IT operational model, security assessments and audits are performed as one-time activities according to a regular cadence, usually after software testing has been performed but before it has been released. However, performing changes this late in the process incurs heavy labor cost and schedule delays: The change needs to be made, tests must be re-run, cyber personnel must be involved, etc. Conversely, determining whether a feature or product is vulnerable earlier in the process, at the level of individual changes or commits, saves expensive rework late in the process, which reduces delays and increases velocity. Engineers can make this “shift to the left” by automating cyber assurance evaluations at different steps of the CI pipeline. A series of “checkpoints” corresponding to different stages of a pipeline can be incorporated into the DevSecOps pipeline to provide increasing confidence that the software only functions as intended and is free from known vulnerabilities and weaknesses. A change passing all steps of a pipeline provides confidence that the cybersecurity risk profile of the API has not increased and that the API can be operationally deployed. A change that fails a test provides rapid feedback on what needs to be modified. The following list notes important stages in a pipeline at which an API should be checked for before deployment. It is not exhaustive, but a change that passes all these stages can provide robust assurance that it works as expected and is resilient to adverse conditions—assuming the right things are tested for and implemented correctly.

- **Detecting secrets**: Secrets (passwords, tokens, ssh keys, passphrases, etc.) committed to version control have seriously affected many organizations over the years. Before a change is committed to version control, it should be checked to ensure secrets have not been introduced to the version control history. Engineers can do this by injecting a “pre-commit” hook to perform a series of automated tests. For instance, a pre-commit hook can check for secrets – invertedly left in the code. A tool such as gitleaks provides an automated checker to perform this task. Other examples of security related pre-commit hooks include authentication of the user's commit via GPG commit signing and forbidding vulnerable functions from being used in code (such as `gets()` in C). Pre-commit hooks are meant to be a quick check that provides feedback to an engineer before a change is committed to the codebase and, potentially, a vulnerability in the form of a secret introduced.

- **Static application security testing (SAST)**: SAST involves scanning source code and checking it against a database of known vulnerabilities in the programming language. SAST is a type of white-box testing, which checks for syntax violations, security vulnerabilities, programming errors, undefined or incorrect values that the compiler might miss, and more. The scanner can be paired with a third-party dependency checker to prevent misconfiguration. Engineers typically perform these checks before compilation, which provides them quick feedback before longer-running tests are made.
- **Dynamic application security testing (DAST):** If SAST is a type of white-box testing, DAST is a black-box type in which the internals are not known and, instead, are tested against a multitude of possible inputs and behaviors in an attempt to break functionality or expose a security vulnerability. DAST tools usually operate on interfaces, send or try to break requests and responses, inject scripts and malformed data, etc. For APIs, DAST is especially useful to ensure no malicious inputs are accepted.
- **Test coverage analyzers**: These analyzers scan for code coverage and report the percentage of all code pathways that have been tested. Executing, understanding, and verifying all code paths (there by getting to 100 percent coverage) can be onerous, but doing so can provide assurance that the system will not enter an unknown state. Test coverage analyzers provide a means to understand what percentage of code is executed by test cases, but they do not in and of themselves ensure security. Nor does achieving 100 percent test coverage.
- **Software composition analysis (SCA)**: While SAST checks for vulnerabilities in the source code, SCA checks for vulnerabilities in the supply chain by checking for known dependency vulnerabilities in open-source databases. This helps engineers know if they should use a different version of a dependency they're using. SCA also collects licensing information, which can provide valuable guidance as to proper use of the dependency.
- **Container scanning**: As more code gets packaged and deployed as container images, the risk of using a misconfigured or malicious image becomes a serious concern. Container scanners ensure that containers do not have known vulnerabilities or misconfigurations. This process includes scanning images, such as Dockerfiles, and runtime scanning for containers built from these images. For example, an attacker able to get root privileges in a container's process that runs with root privileges on the host can break out of the container and inflict significant damage on the host server.
- **Infrastructure-as-code testing (IaC)**: This type of test examines IaC tools that can modify the compute, storage, and networking infrastructure of an organization. IaC are highly sensitive

tools: A misconfiguration or malicious actor can cause catastrophic damage to the system. Using IaC should be reserved only for users with the permissions needed to execute such changes and commits should go through a validator or checker. Packages used by the IaC tool, such as community packages or dependencies, should go through a dependency checker to prevent the possibility of a malicious actor taking control of the infrastructure through a supply-chain attack.

Quantifying Security

Manual and automated evaluations of an API can be used to increase your confidence that the software only functions as intended and is free from known vulnerabilities and weaknesses. But what about the question of sufficiency? Can a score, or measure, of assurance be attached to a releasable artifact? It depends on whether the score represents an inherent, objective property of the artifact or a "scorecard" agreed upon by stakeholders. The former claim is impossible to make: You cannot absolutely determine a system's security because you cannot prove the absence of vulnerabilities in an application. You can only test those certain configurations, code sections, and workflows are free of detectable security vulnerabilities via the use of automated code checkers and manual experts. Instead, whether a system is "secure" depends on the threat model the team believes the application is vulnerable to and the level of risk (based on said model) they are willing to assume to release a change to production. For example, if the threat model includes some level of risk in using vulnerable software dependencies, and the release candidate has 0 critical vulnerabilities and 3 "medium" vulnerabilities, the team may accept that level of risk and deploy the release candidate. "Application security is more of a sliding scale where providing additional security layers helps reduce the risk of an incident, hopefully to an acceptable level of risk for the organization" [Scanlon 2018].

The CI pipeline, properly implemented, "encodes" the threat model agreed upon by stakeholder through the stages present in the pipeline. The tests and scanners that run in these stages check the risk level for each change committed and automatically determine whether it passes or fails. So, the measure of assurance becomes a simple "pass" or "fail" question: a snapshot in time of the level of assurance the pipeline provides based on assumed threats and vulnerabilities. Assumptions underlying the pipeline and exploration of new ways to exploit the system should be continuously reevaluated, per the DevSecOps model.

DevSecOps-Enabling Technologies

According to the *SEI DevSecOps Platform Independent Model (PIM),* a requirement of even maturity level 1 (of 4) for a DevSecOps program is to have a source code repository and maintain version control of that source code [Chick 2022]. These are requirements Dev_7 (source code repository) and Dev_7.1 (version control) in the model. The use of version control is essential to enabling a DevSecOps project for several reasons:

- It allows for consistent change management and integration of new changes into the deployed version of the API.

- It is necessary to understand which files, versions, features, and defects are present in any given version of an API for each step in the DevSecOps process [Russell 20221].
- Being able to publish a new version of an API or roll back to a previous version depends on a version control strategy.

Secrets management is particularly important for API producers and consumers. It is common for an API to require authenticated access to call certain endpoints or certain functions. Therefore, API consumers need a secret (e.g., key, token, password) to access these protected resources. Given that many APIs are not directly accessed by human users but, rather, allow automated machine-to-machine access, these secrets must be stored in an accessible way.

When concerned with APIs specifically, a pipeline can be implemented to help answer questions such as the following:

- Am I exposing any vulnerabilities through poor code conventions?
- Do the functions handle the nominal cases and edge cases as specified?
- Am I exposing any secret keys or tokens in the built software?

Addressing these matters is especially important for API development because the API is the consumer's interface to the code, whether the user is a human or non-person entity. The API of a library or system will likely be specified in a contract data requirements list (CDRL), which references a data item description (DID) or another interface document. The DID defines data format and content requirements. A pipeline can ensure that data format and content requirements are being met by the system as implemented. Generally, the API is also the most accessible portion of a codebase to both users and malicious. Implementing quality and security into the code that implements an API is crucial to both the security of the system and its ability to meet its requirements.

API Monitoring and Logging

APIs provide a unique opportunity to measure and monitor a system. This measuring and monitoring helps information gathering during the operations cycle of DevSecOps and enables ZT through user monitoring and logging. Monitoring can also be used to enhance an API's user experience. For example, if users query the API directly, logging and analyzing failed queries can help identify user-interaction patterns that were not obvious in previous design and development cycles. As discussed in the subsequent Testing section, monitoring and rate limiting can help prevent a denial-of-service attack. In a DevSecOps context, APIs can enable operations monitoring, which should be automated according to the *DoD Enterprise DevSecOps Reference Design* architecture [DoD 2019]. The specific metrics to monitor are specific to each system.

When considering APIs for monitoring in a ZTA context, it is necessary to aggregate log data for building patterns of usage for API consumers. According to CISA, an optimal ZTA makes authorization decisions based on "incorporating real-time risk analytics and factors such as behavior or usage patterns” [Google 2024]. Identifying and checking against legitimate usage patterns requires a log of data sufficiently large to prove it is robust for analysis.

Testing

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.
—Gerald Weinberg, former manager of operating systems development for the NASA Mercury Program

APIs, like any software, are likely to have defects; that is, anything that deviates from the expected functioning of the system. Vulnerabilities are a subset of defects, where the system can be abused (intentionally or unintentionally) to create significant risk for the operation of the system, people, and/or organization. What constitutes *risk* depends on the software component, data, and/or code in question; the level of risk the organization is willing to tolerate; and other factors.[14] APIs are *particularly* risky because they serve as system access points. Consequently, an API defect or vulnerability can enable bad actors to enter the system or cause it to behave unexpectedly. Preventing these threats requires continuous testing of the software product to provide evidence that the system works as expected.

Testing is an integral part of the software development lifecycle. It helps to identify issues with the software product and finds potential sources of cybersecurity risks. Categories of testing include not just software but documentation, personnel, system integration, system-of-systems testing, and more [Firesmith 2015]. This section focuses on software testing methods specific to APIs based on available data and common practices. Authorization, authentication, access control, and other security-focused topics are discussed in the Cybersecurity section.

Software Testing

Software testing comes in two basic varieties: *manual* and *automated*. Manual testing is useful for exploring the API and getting qualitative feedback from the perspective of the user. Automated testing is useful for getting fast and repeatable feedback. Using automated tests has been demonstrated to significantly improve software quality and reduce manual labor in the long run, even though there is an upfront cost in creating the tests [Kumar 2016, Williams 2009]. Some of the most common types of automated tests are unit testing, integration testing, and application testing (sometimes called *end-to-end [E2E] testing*). The following list provides a brief description of each type of automated testing and how it is used:

- **Unit testing** is low-level testing performed by engineers on a small part of the software system. Engineers usually write these tests themselves using a testing framework. These tests verify that

[14] For a detailed discussion of risk, refer to *Cyber Security Engineering: A Practical Approach for Systems and Software Assurance* [Mead 2016].

the code implementation provides the intended functionality, and they are usually the highest in number and the most frequently run [Fowler 2014].

- **Integration testing** helps "determine whether independently developed units of software work correctly when they are connected to each other" [Fowler 2018]. For instance, engineers might use integration test to verify that a public library's book reservation service is correctly adjusting the availability in the book page service when a book is checked out. These tests usually stub or mock real services (e.g., databases or web servers) to make them run faster, use fewer resources, and limit persistent side effects. As the system is developed, integration tests verify more representative versions of the evolving system (e.g., hardware instances and target operating environments).
- **Application testing** verifies the developed application in a representative environment meets the stated requirements. This type of testing is sometimes called *end-to-end testing*, and it is the least-run of automated tests. Since application testing uses real-world services and resources, careful planning must be conducted, and safeguards implemented prior to testing. In some cases, application tests can include multiple users and environments to verify not only that system requirements are met but that the system, including people, processes, and support systems, is capable and suitable. This verification is the primary difference between a developmental test[15] and an operational test.[16]

Along with manual testing, these types of tests comprise the testing pyramid model [Vocke 2018], a popular model that visually represents how different forms of software tests are grouped. The highest level represents the most expensive (in terms of computing, effort, time, etc.) and slowest tests, while the lowest level represents the cheapest and fastest tests.

[15] Developmental testing determines whether a system meets requirements.

[16] Operational testing determines whether a system can accomplish the use case(s) in an operational environment.

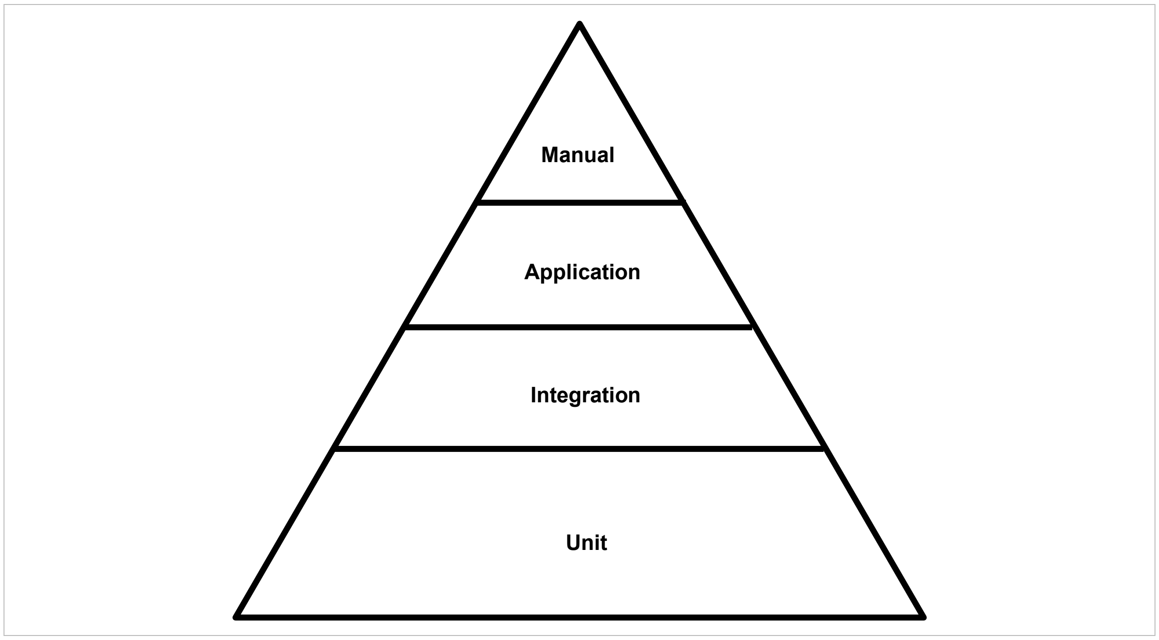


Figure 2: Pyramid Model of Testing

There are other models of software testing (such as the Honeycomb testing model pioneered at Spotify for microservices-based systems [Schaffer 2022]), but the pyramid model remains a classic [Schwering 2024]. It conveys the important notions that (1) different types of testing require varying levels of granularity, and (2) the higher the level of abstraction or *layer* of the application being tested, the fewer tests it should have. In the following sections, we examine how these layers of testing, and a few other types of testing, apply to APIs.

Unit Testing

Unit tests can be written for both the interface and implementation of an API. Examples of interface unit tests include identifying when malformed inputs are passed, ensuring the output of the API conforms with what is expected, and detecting when malicious inputs affect the API. Unit tests can be documented early in the design phase, added during development as engineers discover new data paths they did not think of before, or even (for brownfield projects) added long after implementation to provide regression testing for any kind of refactor. Because it is infeasible to test all the possible combinations of inputs an API can have, engineers can use two main strategies for dividing the input space: deterministic and non-deterministic:

- **Deterministic**. The deterministic strategy uses equivalence partitioning and boundary value analysis [Humble 2010]. Equivalence partitioning divides a set of test conditions into different segments that can be tested. For example, the grades for an exam are A (90-100), B (80-89), C (70-79), D (60-69), and F (0-59). Each grade can be thought of as a different segment to be tested. So, instead of testing the input range from 0-100, we can test a single value from each segment (e.g., 5, 63, 78, 85, and 92). Testing segments this way has the benefit of dramatically reducing the range of inputs to test while keeping the testing values relevant to what we want the software to do.

Boundary value analysis tests the values between the partitions. Using the grades for an exam example, unit tests for the input values 0, 60, 90, and 100 would be used. Combining these techniques provides a way to thoroughly test a variety of values without having to exhaustively test the entire test space.

- **Non-Deterministic**. The non-deterministic strategy uses a fuzz tester [OWASP 2024a], which is an automated, non-deterministic type of test that excels at generating a wide range of testable inputs in a short period. There are different kinds of fuzzers that vary in how structured their inputs are and how they change or mutate over time. Fuzzers can also vary from black-box fuzzers (not considering the internals of the program) to grey-box fuzzers (having some idea of the internal structure and the kinds of inputs it accepts). Fuzzers have successfully discovered bugs in numerous open-source projects, such as OpenSSL, Firefox, Bind, Qt, SQLite, Chromium, and Edge browsers. Combining automated unit tests with a fuzzer increases the coverage of inputs and outputs in an API. There are several open-source fuzzing tools, such as the CERT Basic Fuzzing Framework (BFF) [SEI 2024c], RESTler [Microsoft 2024], and Dredd [Dredd 2024]. Each (except BFF) consumes an OpenAPI specification and automatically generates and executes test cases against an implementation.

Integration Testing

Integration testing verifies that multiple units of code developed separately (such as services, modules, or projects) work correctly when connected to each other. For example, an integration test for an API relying on a database connection will provide a mock (i.e., test double or digital twin) for the database and ensure it functions correctly during its expected (or unexpected) behavior. Integration tests assume stubbed or reduced versions of services and dependencies are accurate and faithful representations of the real services. Some projects written in typed languages store interface classes providing a one-to-one mapping with third-party services to provide compile-time checking of mocks during testing. However, there can be times when a service may change unexpectedly, in which case a *contract test* (see Application Testing) can provide a valuable stopgap for the engineer to ensure mocks map to the real service's interface and behavior.

Engineers tend to trust third-party APIs more than user input [OWASP 2023]. This trust can lead to less rigorous API input sanitization or error handling. Integration testing can help uncover these problems by creating certain error conditions in external services or returning values for the code being tested. Integration tests can also observe API connections or calls to assess the security of the sent data. This type of testing helps mitigate machine-in-the-middle attacks and prevent data leaking from an unencrypted API call or return.

API integration tests should generally cover the issues commonly encountered in distributed architectures and systems (i.e., concurrency, fault tolerance, and failure recovery) [Anderson 2020]. Even when collocated with other software on the same physical system, components interacting through

APIs can fail when multiple users try to use the API simultaneously. Integration tests targeting these cases can be important for systems that need to support atomicity, consistency, isolation, and durability (ACID) properties, such as a relational database system.

Fault tolerance and failure recovery are closely related. Tests for an API should ensure an error-handling mechanism is exercised when the API processes bad data or is in an error state. Failure recovery is then activated to perform the steps to move past the error state and provide service to consumers. In some systems, failure recovery can be entirely manual or infeasible. Even in these cases, ensuring the API sends the correct signals or returns the right error code can be important. Having tests that cover both fault tolerance and failure recovery provide robustness and availability to an API.

Application Testing

Application testing examines the input and output of the API, including the other services it relies on. Such testing is *live* or *operational* testing, and it usually involves using *test data* in a *test database* in a *testing environment*.[17] Consequently, application testing can also be considered *user acceptance testing*. If the API is a user interface (UI), testing will involve an automated UI framework, such as Selenium or Cypress. However, if the API is programmatic, the type of framework used depends on the type of testing environment (e.g., command line interface [CLI] tool, microservice, web application, or desktop app). Ideally, engineers conduct application testing during one of the later stages of the CI/CD pipeline, running a series of automated tests on a testing environment and assessing both the *happy path* (i.e., expected behavior) and a few *bad paths* (i.e., unexpected behavior). Automating this step eliminates a potential manual checkpoint and enables changes to be deployed quickly. Certain industries require a human to review changes before deployment which can be easily performed after application testing as the final step before deployment.

When considering application testing for APIs, it is important to account for adversarial users. Adversarial users can abuse or misuse an API in several ways. In addition to the previously mentioned issues with authentication, OWASP describes “unrestricted resource consumption” and “unrestricted access to sensitive business flows” as two of the top 10 risks for APIs. When an API exposes an expensive operation (i.e., it takes significant resources to execute), an attacker can leverage the operation to conduct a denial-of-service attack that prevents others from accessing the API. An API can mitigate this attack using design patterns, such as rate limiting [Serbout 2023]. Depending on the vector, engineers can test for this vulnerability using an integration test. However, for mock tests the denial issue may not be raised. For instance, if a database call is mocked and runs fast, but the actual database in production is very slow, it may be testable only with a more extensive application test.

[17] See the next section for guidance on testing data and testing environments.

There are times when the application can be misused at runtime, such as by flooding it with requests, modifying incoming packets, creating a stack overflow error, conducting cross-site-scripting attacks, and exploiting the API architecture. These cases are reviewed in the Cybersecurity section.

Other Types of Testing

Container Testing

Containerization[18] is a useful tool for quickly packaging and shipping an API to different environments, and it can be a crucial part of a testing strategy. This is because containers are isolated processes that provide repeatable, sandboxed environments packaged in a standardized format. With containers, engineers don’t have to worry about conflicting library versions or which Linux OS to use: They provide engineers the freedom to develop however they see fit while keeping the application isolated from the host environment. Despite their isolation capabilities, however, containers are not hardware-agnostic, and some applications developed on x86 machines can suddenly and unpredictably fail on ARM-based machines.

There are many different strategies for testing containerized applications. The following are two common approaches:

- **All-in-one approach**. In this scenario, a container contains everything needed to run and test an application, including test scripts, test data, and test dependencies. This approach provides a portable testing environment for the application to run anywhere and is simple to code: Just declare all testing steps on the Dockerfile after the build step. However, this approach inflates the size of the container, requires a rebuild every time the application or testing suite changes, and pollutes the container with additional configuration. These drawbacks affect APIs deployed as containers when deploying, and they can introduce security vulnerabilities through packages not needed in a production environment and an increased attack surface.
- **Multi-stage approach**. In this scenario a container is built in multiple stages [Docker 2024], separating the build step from the test step. A container is first built as normal, then snapshotted as a new base layer; this new layer serves as the base of the test step, which installs testing dependencies and runs test scripts; and, if successful, the output binaries are copied over to a new layer, stripped of development and test dependencies and configurations, creating an ultra-slim container that can be deployed. Each stage is cached, which reduces the time to rebuild if the application or its tests don’t change. This approach trades a reduced size of the final container for increased Dockerfile complexity. The smaller size of the container can help increase speed of deployment and also eliminate extra packages in the APIs container. Because an API is a system access point and will need to interact with other services outside the container, having fewer potentially vulnerable packages and libraries will reduce the attack surface.

[18] According to Google Cloud, “Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services” [Google 2024a].

Accessibility Testing

Applications are frequently used by people with varying levels of physical ability or impairment. In fact, a World Health Organization (WHO) report estimates that 15 percent of the world's population reports that they have a disability [web.dev 2022]. Many of the efforts to improve accessibility have focused on web browsers. Standards and tools exist to enable people to use screen readers, navigate via a keyboard, and even use AI to describe images. Google's Lighthouse tool instantly tests a website's accessibility by verifying whether it conforms to Web Content Accessibility Guidelines (WCAG)—an internationally accepted standard for digital accessibility [web.dev 2023].

Other comprehensive testing tools include the WAVE Accessibility Evaluation Tool by WebAIM and Siteimprove. Some operating systems also provide accessibility options, such as RedHat, Linux, MacOS, and Windows. When developing a web or desktop application, use these guidelines and tools to start testing its accessibility.

Other types of software testing include coverage testing, "chaos-monkey" testing, static application testing (SAST), Red Teams, and load testing. Each of these tests provides an additional layer of verification by testing different parts of an application under varying conditions. Use care when choosing the types of tests to apply to your API: Implementing additional testing involves additional overhead, such as additional requirements, increased up-front cost and slower delivery times (at first, though they mitigate slower delivery times later caused by buggy code without tests). The following sections discuss testing authorization, authentication, access control, and other security-focused topics.

Test Environments

API testing requires a *test environment*, which is the set of systems on which tests are run and the resources needed to create conditions as close to the production environment as possible. A test environment includes hardware configuration of servers, operating system configurations, middleware and services that support the application (such as databases and authentication systems), and the API system(s). According to the *Software Engineering Body of Knowledge (SWEBoK),* a test environment "should facilitate development and control of test cases, as well as logging and recovery of expected results, scripts, and other testing materials" [IEEE 2014]. The objective is to ensure testing can be repeatedly and reliably replicated, and the results provide engineers and product owners confidence that the system is ready for deployment.

Ephemeral Versus Persistent Environments

There are two main forms of testing environments: *ephemeral* and *persistent*:

- An ephemeral environment is stood up to execute tests and removed when testing is complete. This method can save money and resources in a cloud-based or shared computing environment (provided provisioning resources is less expensive than remaining persistent). Because the resources are used only while tests are executing, hourly or per-minute compute costs are minimized [Villalba 2023]. An ephemeral environment also decreases the likelihood of configuration drift, which is the tendency of systems to change over time, resulting in small software differences from the originally deployed system. Examples of ephemeral environments include short-lived containers, serverless components, and environments built through infrastructure-as-code (IaC) that are rebuilt and torn down in each CI build cycle.

- A persistent environment is long lived and is not decommissioned or spun down when testing is complete. This method offers the benefit of pre-allocated resources for testing, potentially leading to quicker test starts. However, since it is a long-lived infrastructure, continued configuration management must be accounted for, including correctly setting the system's initial state and keeping up with system updates. While the setup process can often be the most time-consuming aspect of this approach to testing, having a known initial state ensures tests can be consistently repeated. Ephemeral environments are generally not affected by prior states. Examples of persistent environments are virtual machines or servers specifically provisioned as "test" servers.

Different tests call for different environments, and not all tests can be run in only a single environment. For smaller tests (e.g., unit or limited integration testing of an API), an ephemeral environment may be better suited. However, for larger integration tests or end-to-end testing of an API, it might make more sense to stand up all the services the API might need to connect to in a persistent environment.

When developing a plan for the test environment, the focus should be on enabling engineers and management to understand how well the system is meeting requirements while minimizing the overhead of test environment management. Section 4.3.3 of the *Cybersecurity Test and Evaluation Guidebook* [DoD 2020b] provides helpful guiding questions about the who, what, when, where, and how of test environment planning. The *DevSecOps Fundamentals Playbook* describes 16 different testing activities [DoD 2021c]. Not all of these activities are applicable to all systems, but they do provide a starting point for understanding what types of tests the test environment must support throughout the systems lifecycle. This guidebook recommends tests applicable to APIs, including unit, dynamic application, integration manual security, performance, regression, acceptance, and deployment testing. Other practices, such as auditing, might also be applicable.

In the *DevSecOps Playbook*, the section titled "Play 2: Adopt Infrastructure as Code (IaC)" applies to both persistent and ephemeral test environments [DoD 2021c]. The playbook provides information about the key benefits of IaC. Specifically developed for testing APIs, IaC provides a known state for starting a test. Many APIs are stateful, so testing requires setting up a known good (or bad) state. Automating the setup of ephemeral test environments can speed up engineering velocity.

When managing API testing environments, consider how closely they represent the production environment. APIs typically have requirements regarding scalability and the number of users the API will serve. To meet these requirements, engineers must ensure that the test environment adequately reflects real-world conditions [DoD 2020b, Section 3.8]. Some resources won't be directly available for testing, such as a production database. Therefore, the mock, or substitute, database used in the test environment should contain data, tables, and structured information representative of the deployed system. The degree to which a test environment reflects the production environment has implications for configuration drift. For instance, if the test environment is not updated with real-world data from the production system after it is deployed, the test system will suffer configuration drift.

When testing APIs that require a network to operate, it is essential to make the test network environment as similar as possible to the production network, in so doing, you should implement safeguards that prevent external systems from being affected by testing. Firewalls, virtual switches, and the underlying infrastructure should, to the greatest extent possible, be checked into version control and managed by code.

When planning for testing an API, see the *Cybersecurity Test and Evaluation Guidebook* [DoD 2020b, Section 4.3.2]. It describes other considerations for testing environments. An important example is that some test infrastructures (e.g., appropriate labs or personnel with the correct skills) can have a long lead time to set up. Planning around these setup schedules and fitting them into the DevSecOps lifecycle is important if the program requires this type of testing infrastructure.

Test Data

The U.S. Department of Defense has articulated a *Data Strategy* that provides guidance and goals for managing data. These goals are summarized using the acronym VAULTIS: Visible, Accessible, Understandable, Linked, Trustworthy, Interoperable, and Secure [DoD 2020c]. These goals should be applied to operational APIs, and the data used to test the APIs should be congruent with these goals. As mentioned earlier in this section, it is important to ensure realism in the environment. This is also true of the test data used when building an API. The test data used during development must have the same format (interoperable) and have real or representative values (understandable) that are available for API engineers and consumers to use (accessible) while not leaking any sensitive data, such as personally identifiable information (PII), payment card industry (PCI) data, or other data (secure).

Often, test data may not be readily available. In these cases, it may be necessary to generate simulated test data. Again, this data should be trustworthy, and producers and users of the simulated data should be confident that it represents real-world data [DoD 2020d]. Conversely, for some systems such as machine learning (ML) systems, testing may require so much test data that configuration management of the data becomes a concern. Both the *DoD Data S*trategy and *Data, Analytics, and Artificial Intelligence Adoption Strategy* provide evaluation criteria for data integrity and accessibility [DoD 2023a].

The integrity of test data is a security concern: If tampered with, the test data could be used to hide or insert a backdoor into an API. For instance, in April 2024, the Linux utility xz was infected via corrupted files inserted through the test data housed in the project [Freund 2024]. An API implementation could be vulnerable to a similar attack.

Testing Suite Infrastructure

An automated testing suite of hundreds or thousands of tests that run slowly and operate unreliably can tempt engineers to skip writing additional tests and/or run the suite less frequently, which defeats the purpose of the testing suite. Consequently, it is important for the testing infrastructure to enable a project's test suite to be fast and reliable. What is *fast*? Some practitioners say that tests should take no longer than 10 seconds to run [Seemann 2012] based on research indicating that 10 seconds is the maximum amount of time people will maintain focus on the task at hand. However, 10 seconds may not provide enough time for an appropriate test. If so, team members must agree on how long they are willing to wait without having to switch to another task. Achieving these agreed-on speeds requires configuring the test framework to use parallelization as much as possible.

Some test frameworks allow tests to run continuously as they are written, such as NCrunch. For most projects, test suites should be able to run within the performance requirements of a personal laptop. However, other codebases, especially large monorepos, such as those in large tech companies, opt for cloud-based development environments, where specialized servers are specified to run hundreds of thousands of tests in seconds [Adl-Tabatabai 2022]. The resources needed should depend on the needs of the software project and what is available and economical for the project.

Testing Skill Sets

As mentioned in the Test Environments section, when constructing a system verification or test plan, it is important to ensure that the availability of the appropriate laboratory or range environments. It is equally important to have the correct personnel with the right skills to execute the planned testing. The need to track and manage cyber-skilled engineers is described in DoD Directive 8140.01 [DoD 2020a]. That directive and DoD Instruction 8140.02 [DoD 2021d] provide the basis for the *DoD Cyber Workforce Framework*. This framework offers information about the roles needed for performing cyberspace functions, including knowledge, skills, and abilities. The specific skills needed for performing API testing depend on the specific technology stack and deployed environment the API uses. The current *Department of Defense Cybersecurity Test and Evaluation Guidebook* [DoD 2020b] provides sound recommendations for the roles needed to perform cybersecurity software testing. The DoD Instruction 8140.02 and the *DoD Cyber Workforce Framework* applies to cyber personnel. However, in many systems, there is a need to separate the requirements, development, and testing teams to maintain a degree of independence. Doing so can allow the development and testing teams to operate in parallel, with one developing the product and the other developing the test plans and procedures, and both operating against the same set of requirements. However, this approach can be a problem for systems using a highly iterative lifecycle because there may not be formally stated requirements developed at the unit to support verification and rollup during integration and application verification.

While there are many skill sets needed for testing an API or any IT system, it is important to realize that this testing should be continuous when following a DevSecOps model. Therefore, in a DevSecOps model of development, all team members must be skilled in testing since it is a critical activity. The ISO 2675-2021 standard for DevOps describes the verification process and continuous testing as follows [IEEE 2021]:

> *In DevOps, the Verification process is performed concurrently and continuously with other technical processes. It involves developers, testers, user representatives, and other related stakeholders, including QA. Continuous verification through the DevOps pipeline shall confirm that the*

> *system performs as required. Continuous testing is the practice of interweaving and automating various testing activities into software planning, development, delivery, and deployment processes. While continuous testing encompasses all testing activities in DevOps, automated testing is the primary approach to comprehensive verification of deliverables in a continuous manner.*

Because the pace of development in a DevSecOps project should approach continuous delivery [SEI 2024a], it is necessary to perform testing as often as possible and in sync with the development cycle. Enabling continuous testing using a Lean, Agile, or Scrum framework can conflict with the complete set of skills required. Research has shown that, in general, agile methods work best for smaller teams because they enable easier decision-making and improved face-to-face communication [Keshta 2017].

The current *Cybersecurity Test and Evaluation Guidebook* proposes 26 possible test and evaluation roles as part of a system's cybersecurity working group [DoD 2020b]. For many projects, it might be infeasible to staff this many roles and, in many cases, multiple roles may be filled by the same person. Consequently, each role is applicable to API test and evaluation, so these roles should be accounted for in some capacity. Therefore, a project or program developing an API must plan to engage dedicated or specialized test personnel in addition to conducting routine automated testing. They must be engaged at the appropriate time in the development cycle to (1) inform development team members and (2) regularly test all aspects of cybersecurity represented by the variety of roles described in the *Cybersecurity Test and Evaluation Guidebook*.

Role of the Software Tester

How a team or project organizes their work will change how the role of software tester relates to other roles (such as software engineer) and the allocation of those roles to individuals. Various frameworks define the role of the software tester differently. For instance, in extreme programing quality assurance (QA) could be either the role of an individual or of a separate QA team [Wells, 2013]. In the Scaled Agile Framework (SAFe) it is suggested by Lisa Crispin and Janet Gregory that, while testers should be integrated into the agile team, there is room for an individual to fill only the tester role [Crispin 2009]. Another framework, the *SEI DevSecOps Platform Independent Model* (PIM), does not define a specific software tester role. To understand this, let's first review some goals of a tester from the PIM's perspective:

- Ensure all requirements have been satisfied, presented to authorizing agents, and captured in a repository.
- Ensure the stories (i.e., requirements) are properly implemented.
- Ensure that testing is comprehensive and effective.
- Ensure changes don't break previous functionality.
- Ensure coding standards and organizational process guidelines are followed.
- Ensure service level agreements are satisfied.

These goals are not exhaustive, but they show that what could be considered part of test is split in the PIM across 3 roles (software engineer, site reliability engineer, and release engineer). Other goals of the tester role are to verify that the requirements are met for security testing, that vulnerabilities are remediated, that the scanner tools used in the pipeline are correct, and that no vulnerabilities are being passed through.

Traditionally, testers would have accomplished these goals by writing test cases (at the unit, integration, and end-to-end levels). Testers may also have set up and maintained resources, such as a dedicated testing server, on which to run the software to accomplish manual testing. Regardless of the type of test being run, once it was complete the tester would inform the engineer if the code passed the tests.

This is not how test personal should operate in a DevSecOps model. The origin of DevSecOps traces to the foundations of agile software development and the extreme programming and test-driven development models that proceeded it [Rodriguez 2013]. In both DevSecOps and agile software development, the engineer is responsible for writing test cases. In this way, the role of a tester in writing test code has transitioned into mentoring, guiding, and evaluating the test that is written and used by the engineers. Within agile software development and DevSecOps, a self-sufficient team possessing all the skills needed to deliver a feature, while automating as much of the process as possible, precludes a tester role separate from the development team. The Ops portion of DevSecOps encompasses any test infrastructure. Therefore, the infrastructure should be handled by the team. Automated pipelines preclude the need to manually kick off testing.

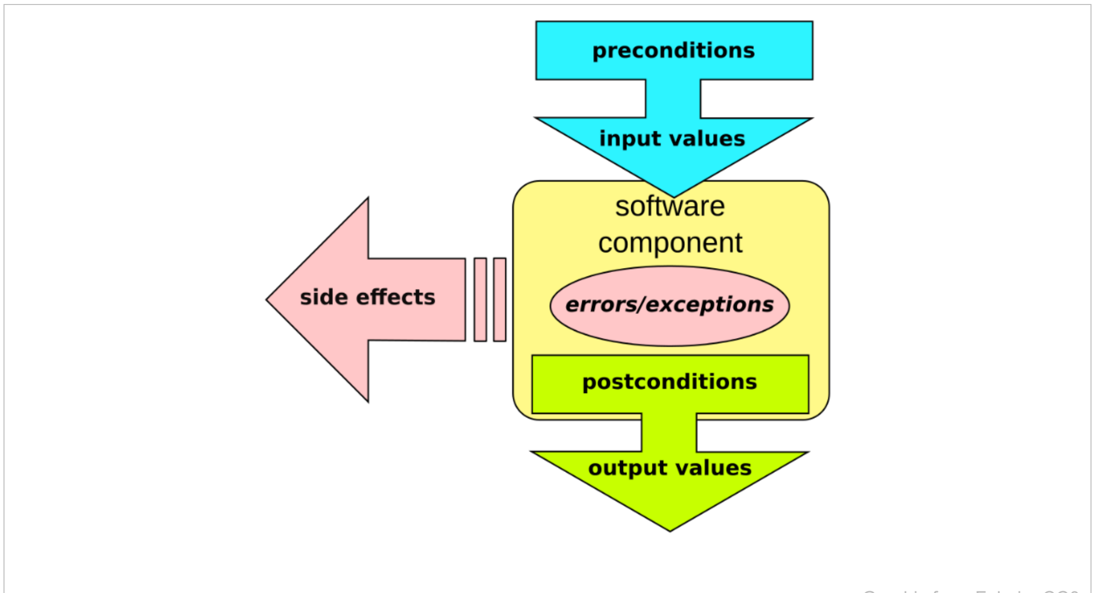
While the tools and approaches have changed, the goals of the tester role have not. As noted, most programs cannot support 26 independent test and evaluation roles, and some DevSecOps teams may not have a single person dedicated to the role of a tester. Parts of this role permeate the development team. From defining and implementing unit tests, to adding stages to the CD pipeline, to preforming release gate checks, the role of "tester" is distributed throughout the development process, including to both engineers and the CD pipeline.

While much of testing can be automated, there is still some case to be made for manual testing, particularly in areas such as usability and interface aesthetics. While some aspects of usability can be automatically tested (for instance, whether an API end point uses a verb or a noun), other aspects are harder to capture in code. Additionally, testers can provide input on how to construct quality unit tests or how to coordinate integration testing. Also, by providing a different perspective, testers can enhance software test cases by providing off-nominal or unique cases. In the case of penetration testing or red team assessment, testers still play a role in contributing the advanced and specialized knowledge required to break software. While manual testing is important, it generally should not be done as continuously as automated testing, and it should not stop work for engineers or delay deployments when an automated tool will do the same job.

Another skill set that needs to be considered in planning for test and evaluation roles are users that can properly perform user acceptance testing. When conducting this type of testing, it is important to have user representation on the development team or to test using actual end users. Complications arise when engineers stand in for these users. For instance, engineers may not understand the intricacies of workflows versus policies or procedures handed down as requirements. API users' domain knowledge is therefore a required skill set when testing APIs.

Contract Testing

In a general sense, an API is a contract between a software application's provider and consumer that specifies what the system will do. Bertrand Meyer coined the idea of *programming by contract* or *design by contract* [Meyer 1997]. In this paradigm, a software engineer formally defines a specification of each function or method that the system exposes. These specifications include preconditions, postconditions, and side effects,[19] as show in Figure 3 [KU 2024]. Preconditions are the criteria required before a function is executed. These are things that the service or API provider expects to be true before a function is called. For an API, a precondition for accessing a protected endpoint would be a requirement for the caller to provide a valid authentication token. Postconditions are the state or set of criteria that must be true after a function is executed. Postconditions for an API might be the return of specified data and an HTTP 200 status code. Invariants are the data or states that, regardless of the operation or transformation applied by the function, will not change.



Graphic from Fabuio. CC0.
File: Design by contract.svg; created 20 October 2013.
https://en.wikipedia.org/wiki/Design_by_contract#/media/File:Design_by_contract.svg

Figure 3: Design by Contract Scheme

[19] Conversely, one can define invariants. However, for a complex system with an API, the number of invariants could be vast and irrelevant to the API. Knowing what side effects exist can be critical.

The benefits of the formal specification of a contract when testing an API are twofold:

- Before any implementation of an API is provided, a well-defined API contract makes it possible for the development team to write example client code that uses the functions or endpoints. This API user testing eliminates any time spent implementing irrelevant methods or API calls that are hard to use from the client perspective.
- During integration testing of different software modules, some systems can be hard to simulate in the test environment, the target system can be expensive to operate on (e.g., quantum computing hardware), or the target system may not even exist yet. In both cases, having a contract that accurately represents a software module or library can aid the integration testing done by both producers and consumers of the software. Defining pre and post conditions encourages a higher fidelity stub to be created for use in integration testing for unavailable modules.

How contracts are defined in a particular language varies. In Java, engineers commonly use Javadoc comments to document the parameters, return values, exceptions, and purpose of a function. Doing so also serves as a common way to informally document a contract. A variety of tools offer varying levels of formality for defining contracts that can facilitate verification of API usage.

OpenAPI is an especially relevant specification for API contract development given the prevalence of HTTP-based representational state transfer (REST) APIs [Swagger 2024]. It is a tool and format used for specifying REST APIs, including endpoints, inputs, and outputs for each operation; authentication methods; and other information. The use of the OpenAPI specification to define an API aligns well with the design-by-contract paradigm of specifying preconditions, such as the domain of inputs and a description of the endpoint. OpenAPI also allows an engineer to specify the return from an endpoint, including the return code, a description of what that return code means, the schema of any returned data, and examples of the data. In the realm of API security, OpenAPI also allows engineers to specify the authentication and authorization requirements for each endpoint. OpenAPI documentation refers to this specification as a *security scheme*. In version 3.0 of the documentation, OpenAPI includes HTTP authentication, API Keys, Oauth2, and OpenID Connect. OpenAPI potentially falls short in its ability to specify invariants of a function. For instance, in a REST API, GET requests should be idempotent, or unchanged when applied to itself. However, there is no way to document what an endpoint may or may not change in terms of states outside of a text description.

There are several open source tools, such as RESTler [Microsoft 2024] and Dredd [Dredd 2024], that will consume an OpenAPI specification and automatically generate and execute test cases against an implementation. These fuzzers can infer the state of API calls that require sequencing [Atlidakis 2019]. Similarly, with Java and the Java Modeling Language (JML) there are applications that can transform Javadoc comments into runtime assertions (e.g., the JML4C compiler) [Sarcar 2009].

AI-Enabled Testing

There are several ways in which AI, machine learning (ML), and natural language processing (NLP) have been applied to testing APIs. Currently, AI resides at the peak of the Gartner Hype Cycle [Perri

2023]. Consequently, there are many proposed AI applications with limited tools. The following sections discuss experimental techniques for AI-enabled testing.

Test Generation

One proposed technique is to use NLP to enhance API specifications, such as OpenAPI, with natural language descriptions [Kim 2023]. Doing so makes sense because natural language descriptions are commonly part of API specifications. These plain-text descriptions often provide examples, parameters, and input domain information, and some might infer new rules that have not been documented. Researchers extracted these descriptions via an NLP model and parsed them for additional API rules that were not previously written. These new rules are validated against the document and injected as new rules. This novel method could be used as part of a validation stage in a CI pipeline to ensure every API change provides standardized, accurate specifications.

Another approach for generating tests is analyzing recordings from user interactions [Loadmill 2024]. From these interactions, the AI model can learn patterns of behavior and iterate on them to generate test cases. This approach could be applied to stateful APIs, where the sequence or ordering of calls is important and the interaction is complex. Similar tools can apply ML without the need for user recordings [Martin-Lopez 2020].

Researchers at Meta wanted to determine whether an LLM could generate valid tests for their Android codebase. Specifically, they wanted to know if they could generate new test cases from existing ones that could cover previously missed corners, thereby improving test coverage [Alshahwan 2024a]. Their model was able to create correct test cases 75 percent of the time, increased coverage by 25 percent, and the model's testing recommendations were approved for production deployment by Meta engineers 73 percent of the time. Scaling the use of LLMs remains challenging due to the cost of inference. Furthermore, LLMs are *fashion followers*, which means they tend to generate code based on previously seen styles in its dataset, which may or may not be helpful. However, the success of this research could provide an opportunity for smaller codebases to improve their coverage by leveraging smaller LLMs trained for this specific task.

Some research has explored automating test generation altogether by having an AI generate test cases via a configuration file that enhances existing API specifications [Martin-Lopez 2020]. This configuration file includes information such as parameters, dependencies, and authorization data—any kind of data relevant to the running of the API—and has AI-generated test cases developed from this information. The resulting test cases pass through a validation step before being published. This level of automation has not yet been tried in real-world practice.

In another case, researchers used an LLM to generate test data for an API successfully from an OpenAPI specification. They thoughtfully used prompt engineering to generate useful response datasets. The researchers explained, "We design[ed] a prompt specific to the testing endpoint and status code being evaluated. This prompt includes relevant information about the scenario, such as the API endpoint, the expected status code, and any additional instructions for generating suitable test data or verifying the response" [Nguyen 2023]. Applying an LLM in this way can be helpful for finding corner cases where existing tools, such as fuzzers, may fail to find.

Verifying Model Correctness

With AI-enabled testing, there is a need to verify the correctness of the output of the models. Research has shown that foundational[20] models can fail to identify well-understood errors in code [Sherman 2024]. Additionally, as with any test method, there is a need to integrate the use of the tool or method into the existing test pipeline. Cornell University researchers have proposed a process for LLMs called *Assured LLM-Based Software Engineering* [Alshahwan 2024b]. This strategy puts some guardrails on the generation of code by models that safeguard against regression and ensure a measurable enhancement in code coverage. This is especially important for API testing because regressions in API capabilities are the most visible to the product's end user. With this set of guarantees in place, commercial companies have improved the deployment of their systems using these types of tests [Alshahwan 2024a]. However, they are still in the early experimental phases of using this strategy, and they are targeting improvement of their test cases, not replacing the foundational work of writing tests.

As we discussed earlier, applying AI to APIs is a nascent field with significant research results that point to the use of AI as an additional verification stage in the CD pipeline. Research on the use of AI in APIs has been mostly confined to using APIs and generating test cases or documentation. However, additional use cases can include development (in the case of Github Copilot), verification, and security. Additional research in these areas will provide answers to questions about these uses.

Zero Trust Testing

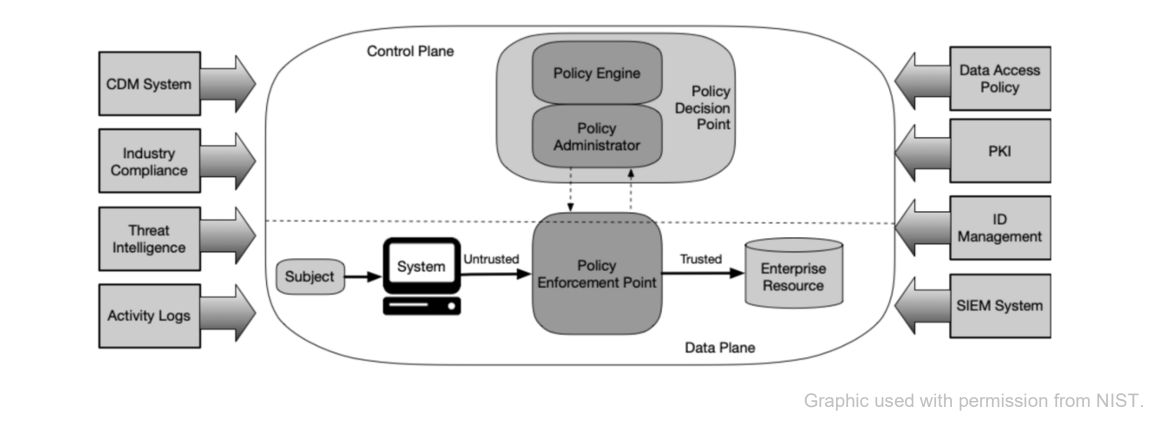
API Compliance with Zero Trust

The nexus of ZT and API testing can be evaluated through the following seven tenets of ZT, as defined in NIST SP 800-207 [Rose 2020]:

1. *All data sources and computing services are considered resources.*
2. *All communication is secured regardless of the network location.*
3. *Access to individual enterprise resources is granted on a per-session basis.*
4. *Access to resources is determined by dynamic policy—including the observable state of client identity, application/service, and the requesting asset—and may include other behavioral and environmental attributes.*

[20] Foundational models are LLMs trained on a diverse dataset that are not fine tuned to a specific domain.

5. *The enterprise monitors and measures the integrity and security posture of all owned and associated assets.*
6. *All resource authentication and authorization are dynamic and strictly enforced before access is allowed.*
7. *The enterprise collects as much information as possible about the current state of assets, network infrastructure and communications and uses it to improve its security posture.*



Graphic used with permission from NIST.

Figure 4: Core ZT Logical Components

The following sections discusses APIs through the lens of the ZT tenets.

APIs are resources and, based on their function, could control significant computing resources.

Starting at the most basic level, all parts of an API's configuration must be tested. These tests can include several levels of configuration, such as the operating system, API, and network. Using infrastructure-as-code (IaC), engineers can significantly help with configuration testing by providing discoverability, repeatability, and the ability to manage configurations with version control. The architecture itself, in which the API is deployed, must be tested. Several tools exist for this type of testing [Bell 2022, Ansible 2024, Puppet 2024]. As a system access point, APIs are a natural target for attackers, so proper controls for authentication and authorization of the entity calling the API must be considered.

All APIs, whether for internal or external use, need to provide the same level of authentication and authorization guarantees. The security of the network does not provide any guarantees for the security of the API.

Continuing to move up the stack from configuration and infrastructure, a team testing an API should look for common broken authentication issues within the API. Broken authentication is a condition in which a malicious user can bypass authentication mechanisms, such providing default credentials, using default or weak passwords, or exposing authentication details to users. Broken authentication is the second highest security risks to APIs [OWASP 2023]. This type of testing should occur in unit, integration, and application tests, depending on how authentication is integrated with the API. Tests should focus on finding flaws in an API's implementation that would allow a user to assume the identity of someone else; this is where testing for *non-repudiation* is useful. NIST SP 800-18 defines non-repudiation as "assurance that the sender of information is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having processed the information" [Swanson 2006]. Testing of non-repudiation could involve checking that log files record the correct user ID or that there is no way to anonymously make an API call.

Broken authorization is another dimension that needs to be tested in ZTAs. Broken authorization is a condition in which a malicious user can modify a request to gain access to resources through an API that they should not have permission to request. Compared to authentication, which verifies that a user is who they claim to be, authorization verifies that the user has the right permissions to use a resource. With ZT, users should be given the minimum permissions to accomplish the intended function and no more (i.e., the principle of least privilege). When testing an API for broken authorization, testers should try to find flaws in what a user is able to request or modify through the API. For example, a tester can determine whether an attacker can modify the id of an object that is sent in a request, which indicates a broken object-level authorization (BOLA) [OWASP 2023]. Modifying ids can lead to data disclosure and even full-account takeover. Other types of broken authorization include object property level (which allows an authenticated user to access sensitive user object properties) and function-level authorizations (which allows a user that sends an authenticated call to retrieve information they are not authorized to retrieve). These types of tests can generally be handled at the unit level, but they might require integration testing if the API offloads the authentication and authorization to a central service, which is a recommended best practice [OWASP 2023].

APIs should use session-based authentication; providing long-term access keys or permanent tokens for an API are discouraged.

Replay attacks, in which a malicious actor reuses an expired credential, is similar to broken authentication. Testing against this type of vulnerability ensures that any tokens or keys used are short-lived and valid only for connecting to the API in the current session. While a session may be long lived, the rotation of shorter term access keys with a refresh token, as in OAuth 2.0 or by similar means, should be preferred [Hardt 2012].

Requests and responses from an API should be secured. (See Security Testing.) Passing information such as query parameters, URIs, or other business-sensitive information in cleartext should be avoided.

Securing API requests and responses is necessary for confidentiality but not for integrity or availability. Testing that the data exchanges between an API and its users are secured (e.g., through encryption) is specialized testing that requires a particular skill set. As with authentication and authorization, the accepted best practice is to use libraries and functions provided for this purpose and not implement them from scratch. A common approach to encryption is through Transport Layer Security (TLS) in HTTP-supported networks and asymmetric encryption as a general procedure [SSL 2023]. To verify that an API is appropriately implementing a secured connection falls into the integration testing category. It may require capturing network traffic between the user and the API to ensure that a secure handshake was made, and the correct protocol options were used. Other similar testing can ensure that the API does not accept self-signed certificates [Geogiev 2012]. This type of security testing, while necessary for ensuring compliance with ZT, is admittedly specialized and may require dedicated personnel trained in this area.

ZT in API Testing and Development Environments

As mentioned in the Test Environments section, engineers must have an infrastructure in place to run tests and pipelines to enable the testing process. These development and testing environments should conform to a ZT model. Therefore, testing, preproduction, and other environments must be accounted for and interoperate with the tools that enforce ZT [Sanders 2021]. The attack on SolarWinds' Orion software provides a case study in ZT security. Malicious actors were able to run processes within the DevSecOps pipeline that SolarWinds used to build its software [MSRC 2020]. That does not mean that ZT would have prevented the attack entirely, but monitoring and enforcing least privilege as part of ZT could have minimized the number of intrusions and lessened the effects of the attack.

APIs require continuous monitoring. Information about the use of the API, including its availability, security, and integrity, should be shared with security and testing teams.

APIs provide a unique opportunity to enable the monitoring of deployed systems. When considering a ZTA, it is important that, once the system is in place, it is continuously evaluated for security-policy violations. In ZT there is no single set of rules that is defined once: ZT security policy is context aware and adaptable [Rose 2020]. Consequently, it may be necessary to pull different types of information from different parts of an information system to determine whether security has been compromised. APIs can enable this determination by allowing access to the information about API access, including who accessed what and when. This ability to log user activity should be considered in the design phase, but it should also be exercised in the testing phase. Subsequently, in the operations phase, API logs could be used in an after-action forensics report.

Cybersecurity

Someone cracked my password. Now I need to rename my puppy.
—Unknown

As we described earlier, cybersecurity complements software testing by verifying and evaluating the mechanisms in place to defend the system from attack; that is, to prevent, detect, contain, mitigate or eliminate, and recover from attacks. These attacks can either be unintentional, such as the accidental exposure of credentials in plain text, or conducted by a dedicated and intelligent agent, such as a hacking syndicate. In the Secure Design section, we discussed including security in the design of an API early in its development. Thus, in this section, we assume an API designer has already identified areas of risk and boundaries of trust for both data and applications, and we will focus on how to provide effective cybersecurity mechanisms to an API to ensure the organization's security policies are fulfilled.

Cybersecurity's goals are to identify threats and vulnerabilities in an application, identify where risk is involved, and ensure the right mechanisms are in place to handle that risk [Saydjari 2018]. These goals can be broken down into the attributes of confidentiality, integrity, availability, authorization, authentication, and non-repudiation. All of these (with the possible exception of vulnerabilities in the implementation) are part of the system definition and analysis, and they are performed early and often as the system is redefined or evolved. The following list provides a brief definition, per the National Information Assurance Glossary, for each attribute:

- Confidentiality: the ability for a system to disclose information only to authorized entities.
- Integrity: the property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner. This can apply as well to the integrity of the application.
- Availability: the property of being accessible and useable upon demand by an authorized entity.
- Authentication: the process of verifying the identity or other attributes claimed by or assumed of an entity.
- Authorization: access privileges granted to an entity, or process or the act of granting those privileges.
- Non-repudiation: assurance that the sender of information is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having received the information [CNSS 2010].

Below are some of the mechanisms an API system needs to maintain each of these attributes and the processes for testing them.[21] By maintaining these attributes, the system can be effective at lowering risks and keeping vulnerabilities at bay, becoming trustworthy for its designers and users.

Confidentiality

There are many mechanisms that can be used or combined to improve API confidentiality, such as

- Encryption. Encrypting communication between senders and receivers helps improve confidentiality. Encryption can be performed in the network, transport, or application layer, depending on the requirements. Some protocols combine encryption across multiple layers; for example, WebRTC, a popular protocol for secure, real-time communication, uses Datagram Transport Layer Security (DTLS) to establish a secure channel via a handshake. It then encrypts subsequent traffic with Secure Real-Time Transport Protocol (SRTP). It optionally supports Stream Control Transmission Protocol (SCTP) for reliable, guaranteed message delivery [WebRTC 2024]. Which layer of the stack encryption occurs in ultimately depends on the API's requirements as well as its interoperability with existing services. Many of these encryption protocols rely on asymmetric encryption, which requires secure handling of keys and key rotation on a regular basis.

  The decision on where to encrypt communication in a system depends on the API's trust boundary: the boundary in which the API's level of "trust" changes [MITRE 2024]. For example, a web API will have one side facing the public Internet and another facing internal systems. To take care of adding and stripping TLS/SSL from HTTP, a proxy or load balancer can be placed in front of the server; either way, the effect of this design is that the encryption and/or decryption of network traffic occurs outside of the API's trust boundary, making data being passed into the API implicitly trusted. A demilitarized zone (DMZ) or perimeter network is an example of an architecture that separates trust boundaries between external and internal systems. Software that fails to keep data confidential by sending and/or receiving it across trust boundaries falls under the CWE-501: Trust Boundary Violation [MITRE 2024], which some static code analyzers can detect.

  A relatively recent breakthrough in encryption, homomorphic encryption has the potential to enhance confidentiality in APIs by allowing systems to process encrypted data without deciphering it, thereby preserving its privacy. The ratio of how much encrypted data gets decrypted varies from "partially homomorphic" to "fully homomorphic." The disadvantage of these algorithms is that performing computations on data that has been encrypted with such scheme can be up to 100

[21] Risk transference is a concept common throughout the testing of all the attributes discussed in this section; specifically, risk transference to other services outside the API. Risk transference aligns with the concept of separation of concern, in which each software component addresses a particular concern. This paper focuses on the API component; therefore, most of the guidance provided advocates for using established services and standards to provide much of the cybersecurity for components around an API that the API itself should not be concerned with.

trillion times slower than performing those same operations on the corresponding plaintexts. Consequently, these encryption algorithms are still in the research phase [Yackel 2023].

- Masking. APIs can mask or scramble sensitive information from loggers and monitoring services, such as personally identifiable information (PII). Some industries require strict conformance to these rules, such as HIPAA in healthcare and Regulation P in Finance. Testing that the API encrypts and decrypts data per requirements likely falls under unit testing, but sometimes (if the encryption service is external to the API system) it may fall under integration testing. Some automated tools exist to help protect privacy, such as automated PII detectors. These instruments scan a codebase or mask ingested logs to keep certain information that conforms to well-known formats out of the database. Similarly, a class of tools called secrets detectors use the same approach for finding passwords and credentials in the codebase. Many static code analyzers use a secret detector as part of their code analyzer suite. Note, however, that these tools themselves can be exploited. This possibility necessitates recursive cyber security or, in other words, securing the tools that confirm the security of the system.
- Differential privacy. Another aspect of API confidentiality lies in the data that the API exposes. Many APIs are used to access large datasets. Differential privacy stipulates that the addition or deletion of any one record from a database does not affect the analysis of data [Dwork 2008]. This concept is often extended to include the idea that any one person's information in a dataset should not be revealed. Many APIs intentionally provide the ability to recall specific records. However, if an API is only intended to provide aggregate or anonymized data, this would need to be established as a requirement of the API and tested for.
- Multi-level security (MLS). This concept applies to cases in which confidentiality of data must be maintained across multiple security levels. MLS is a well-known area of study involving computer systems that can process information with incompatible classifications, permit access to users with different security clearances, and prevent unauthorized access. Well-known implementations of operating systems that integrate MLS include SELinux and FreeBSD. We are not aware of an API currently in service using an MLS system.

How to Test Confidentiality

The best way to secure sensitive information is not to store it at all. When it must be stored, it should be encrypted at rest and in transit at the application, database, filesystem, or hardware levels, depending on the application. APIs should not define their own way of encrypting data but, instead, rely on proven encryption methods, modules, and ciphers [OWASP 2024e]. Since encryption software such as GPG or TLS are thoroughly tested and proven to work, testing the encryption functions directly is wasted effort. Instead, testing for confidentiality focuses on the infrastructure that supports it: key management, SSL termination, and data analysis tools that ingest customer data and anonymize it, encrypt it, and/or decrypt it for compliance. More often than not, these capabilities are provided as services, including the following:

- AWS KMS, HashiCorp Vault, and LastPass are key managers, which create, rotate, revoke, and expire keys and passwords.

- Nginx and HAProxy provide SSL termination through their own configuration language.
- AWS Macie and Azure AI PII detection scan large datasets looking for data that should be private and selected for processing.

The aforementioned cases present other services as a solution to provide confidentiality in a system. The vendor should be vetted to ensure it is using standard cryptographic implementations and has security measures in place to protect its systems. If the vendor has a poor history regarding application security, is not transparent in how the client's data is handled, or has custom implementations of cryptographic algorithms, then stakeholders must evaluate and take actions to mitigate risks associated with that provider.

To assure these services work as intended, scanning and validation of their configuration language is required. Static analyzers can perform this function in a CI pipeline. NIST Special Publication 800-57 Part 1, Revision 5 includes specific guidelines on

- which secure cryptographic algorithms to use
- how to store keys
- how often to rotate keys
- processes for dealing with compromised keys
- how to securely distribute keys in a team and more

Additionally, NIST Special Publication 800-122 provides guidance on how to identify PII data and distinguish it among multiple levels of confidentiality risk impact (low, medium, high). Teams should create tests that verify the requirements that apply to their applications as listed in these documents.

Integrity

Integrity is sometimes referred to as the bedrock of information security, because if you can't guarantee that your data hasn't been tampered with then there's nothing of value to protect [Saydijari 2024]. APIs, in particular, must assure the integrity of data coming in and out of the systems they interface. This means using communication protocols that support data integrity as it travels between software components; these protocols will employ verification schemes such as checksums, message authentication codes (MACs), or digital signatures, to ensure data packets haven't been tampered with in transit. The following list provides brief descriptions of these methods and applications:

- Checksums create sparse, fixed-width hashes of the contents of a data payload. Usually, the sender calculates the checksum, or cryptographic hash, of the data and sends it along with the data; the receiver then recalculates the checksum, or hash, and compares it with the received value to verify data integrity. So long as the hashing function doesn't create collisions (such as MD5, SHA-1, and SHA-2 [Plover 2006]), then the hash is safe for a receiver to use to verify that it has the same message. Checksums are a computationally cheap and common method to check for data integrity in many software package distributions. However, they cannot provide authentication guarantees and are vulnerable to man-in-the-middle attacks [Forouzan 2012].

- Message authentication codes work using checksums in addition to a secret key to encrypt data. Consequently, they provide both integrity and authentication. Because the sender and receiver require a secret key, MACs authenticate the identity of the sender of the message (assuming the key hasn't been compromised). Different MACs can use either symmetrical or asymmetrical encryption. They are used in many protocols, such as Bluetooth [Padgette 2022], Zigbee [Kulkarni 2015] , SRTP [Baugher 1970], and others. Thus, they provide a flexible way to ensure the integrity of data for APIs in many different applications.
- Digital signatures use asymmetric encryption to verify the integrity of data by having the sender compute a cryptographic hash of the data, then sign it with its private key. The receiver computes the hash by using the same hashing function, then decrypts it using the sender's public key: If the decrypted hash values match, the integrity of the message is verified. Digital signatures provide integrity, confidentiality, and authentication. They are used in TLS/SSL and many other Internet-based protocols. Compared to MACs, they can be computationally more expensive for APIs to use, and they require an effective public key infrastructure (PKI) to manage keys securely [Forouzan 2012].

Once the integrity of the message is verified, the message must still be checked for potentially malicious code before processing. This is necessary because multiple attacks [MITRE 2006] exist that perform remote code execution (such as Shellcode), even if the serialized data passes all API parameter type checks. For this reason, testing of serialized and deserialized data needs to be verified through additional unit tests for this specific vulnerability. The choice of serializer can be programming language specific, such Python's pickle library, in which case it constrains the system's ability to interoperate with other systems with different programming languages, or it can be language independent, such as JSON or Protocol Buffers, which can be interoperable but require an agreed-to standard encoding scheme. Additionally, APIs should never accept executable code as a parameter to an API (which would, for example, leave the system open to SQL injection).

How to Test Integrity

When testing for the integrity of APIs, it is important to focus on the following elements:

- Packages and artifacts. Software packages usually have checksums listed on their online registries. When the package is downloaded, take its checksum and compare it with the one listed on the registry. If they are not the same, then the downloaded package has either been corrupted or tampered with and should be removed. This process can be performed through simple scripts. Artifacts created by a CI pipeline should themselves be signed and their checksum saved before distribution.
- Codebase. Code signing assures the provenance of the codebase and guarantees its integrity. Use tools such as sigstore or signserver to sign your code and provide verification to others with whom you're sharing it.

- Version control/configuration management. Sign commits with a key to authenticate the author of the commit and assure its integrity. GPG commit signing is a popular method to accomplish this task.
- Messages. Messages sent via a network should have their contents signed by a key and verified on arrival. Though the process can become complex, many transport protocols (including TCP) have this capability, making this assurance transparent to the receiver.

Availability

Keeping an API available means the service is healthy, reachable, and operating as expected, even during adverse conditions. This section reviews some of the important documents that guide availability requirements for APIs; methods for protecting availability are covered in the Architecture Cybersecurity section.

The service level objective (SLO) is a document that targets a specific level of reliability for a service (usually measured in percent of uptime in a time range) [Beyer 2018]. It is commonly used to define the availability requirement of a given system. The level of reliability in an SLO is measured by metrics that stakeholders agree on. The SLO also forms the basis for a service level indicator (SLI), a document that outlines what system metrics, or indicators, will be tracked to measure for availability. When the system is provided as a service to another organization, as is very often the case with APIs, then a service level agreement (SLA) is also produced, which outlines what the provider of the service guarantees of its system and what penalties it incurs if those guarantees are not met. Third-party services for APIs can also enter into an SLA contract to ensure the APIs' SLOs do not exceed its dependencies' availability requirements.

Site reliability engineering (SRE) has a culture of extensive testing in its systems, and it generally assumes peak loads much higher than average to maintain its high availability guarantees [Beyer 2018].

How to Test Availability

Testing for availability means pushing the performance limits expected of the system in normal and abnormal conditions and observing its behavior. Availability testing can take place in various CI stages, including load, stress, and failover testing. Organizations that are confident in their system's resilience perform chaos engineering on their live applications, which provides them a wealth of information on the availability of their systems, though at the risk of disrupting operational services. There are many automated software libraries that can perform these tests in a CI pipeline. Availability tests are usually performed at the tail end of the pipeline due to the time and resources required to run them. To verify the availability of the software system, it is essential monitor and log of the elements that make up the SLO so that deviations in the parameters are recorded and teams alerted.

Authentication

Broken authentication was one of the OWASP Top 10 API vulnerabilities for 2023 and continues to be a top concern in API security [OWASP 2023b]. There are many ways authentication can be broken including not encrypting credentials, using weak passwords or encryption algorithms, not validating token expiration dates, or having weak credential changing procedures. A key issue in this area is the continued use of usernames and passwords for authentication. Passwords have been proven to be vulnerable to brute force attacks, and they suffer from low entropy. Their improper use in authentication flows makes them vulnerable to spoofing and credential stealing. OWASP and others recommend the of use multi-factor authentication (MFA), which commonly consist of passwords, or something you know, along with some other kind of authenticating mechanism, usually something you have. Research has shown using MFA reduces the likelihood of compromise by 99.9 percent (as of 2019) [Weinert 2024]. The systems that provide authentication vary in their architectures but usually involve a centralized authority that an API authorizes itself to. The traditional web API authentication mechanism uses cookies: unique sessions created on a server and stored on clients. Future requests from the client uses this session id, or cookie, to authenticate requests. This approach creates a stateful session between the client and server. However, this authentication mechanism is vulnerable to cross site request forgery (XSRF) attacks in which, for instance, a bad actor listens to network traffic and then uses the cookie to access an API as a different user. Also, cookies must be kept in a database to keep track of user sessions. This requirement can become a bottleneck to scaling services, since most cloud applications are scaled horizontally. Thus, cookies should not be used for APIs.

In a similar manner, token authentication works by requiring users to submit their credentials to a server, which generates a unique token (i.e., JSON Web Token or JWT) signed with its private key, which is then sent back to the client where it is kept in local storage. Future requests append the token to the header where the server validates the signature without having to store each individual token in a database, thereby enabling more scalable architectures. Tokens, however, can become a security concern if their lifespan is not managed; that is, if tokens are not invalidated after a defined period. OpenID Connect, which is built on top of OAuth, provides an example of a technology that uses JWTs to manage authentication for users and can be integrated as a SaaS product.

Within many security-conscious industries, authentication is generally handled via public key infrastructure (PKI). PKI manages the issuance and revocation of digital certificates to authenticate users. These certificates may simply be files on a computer system or, in a more secure system, protected by a hardware security module that is used to securely store the digital certificates. These hardware modules come in many forms, the most common of which is a “smart card.” Within the US government, personnel are issued a Common Access Card (CAC), or Personal Identity Verification (PIV) card, containing different types of certificates tied to that user. These certificates constitute the first step in setting up a more secure session using a technology like OpenID Connect. In some cases, a user may be able to access some system resources or API endpoints via a software certificate but may have to re-authenticate with a hardware-based certificate to access sensitive resources or API endpoints.

Sometimes, two sides of a communication channel may need to verify each other’s identity (instead of a single side verifying the other), in which case a mutual authentication system, such as mutual transport layer security (mTLS), can be used. When used for API security, mTLS ensures that API requests being sent to a server cannot be used by imposters to steal data from the currently established session. Although this prevents a number of attacks, it requires additional computing power to exchange information and additional work to set up credentials for all users.

How to Test Authentication

API systems should use established, rather than bespoke, authentication frameworks and protocols. For example, the U.S. Department of Defense's Common Access Card (CAC) infrastructure. The CAC, which can be used by personnel to log into approved DoD systems, provides both a physical card and unique digital signature to all personnel. (See DOD 8520.02 for the issuance that establishes the policy and procedures for establishing the DoD PKI.)

In most cases, using a SAST scanner will help uncover potential vulnerabilities in the authentication configuration of a software system. Some of these checks include the following:

- ensuring credentials are not being sent unencrypted over a network
- ensuring API keys or tokens are not being stored in the application as plain-text
- finding vulnerable configurations that provide authentication in proxies, servers, and programming languages (Mozilla has a helpful tool that can generate secure TLS configurations at https://ssl-config.mozilla.org.)
- ensuring libraries are kept up to date to avoid known authentication vulnerabilities such as Heartbleed[22]

Note that scanners cannot catch all vulnerabilities. Operating systems and software packages come with default passwords that should be changed immediately on startup. Using a password alone for authentication is insecure, and passwords should be coupled with an additional authentication mechanism, such as testing something a user has (such as a YubiKey or Phone number) or a user's biological feature (such as a fingerprint).

Authorization

In the context of APIs, authorization is the process of granting permission or authority to a client to use an API to perform some action. For instance, a database user might be authorized to create, read, update, or delete a record. Once an API verifies that a client is who they say they are, it next needs to know what operations the client can perform. Unfortunately, broken object-level authorization (BOLA), was the top OWASP vulnerability for APIs in 2023. This vulnerability enables a user to change or modify an object for which the user is not authorized through an API. Failing to protect against this vulnerability can lead to "unauthorized information disclosure, modification, or

[22] https://heartbleed.com

destruction of all data" [OWASP 2023a]. There are several schemes for implementing authorization that are technology specific and some that are policy or architecture-focused. This section starts with the latter.

Least privilege is the first authorization policy to be enforced (that is, giving the user or client only the permissions they need to do the job or function intended). The idea of separation of concerns, where individual roles exist for providing different functions, is similar to least privilege. When these policies are not enforced, an account might be able to perform backups of a system while updating fields as part of a different maintenance task. A proper implementation would have a role permitted to conduct backups and a different role permitted to update the configuration.

The practice of assigning roles to users to perform actions is referred to as role-based access control (RBAC). Within Unix systems, RBAC can be achieved by adding users to groups that have the right permissions to perform certain actions or access certain directories. An alternative method derived from RBAC is attribute-based access control (ABAC), which requires additional attributes such as a context, action, resource, or a specific user to determine access (see NIST SP 800-162 for more detail) [NIST 2014].

Deny-by-default is another concept that can improve API authorization security. When a client makes a request to an API, it can either be approved or denied; in other words, if the authorization system cannot match an authentication request to its policy rules, the default behavior is to deny the request. This includes instances in which the API or authorization system is in an error state.

Finally, permissions must be validated on every API request. If a client has previously authenticated to an API with one method or another, that client should still present the authorization token (or other mechanism) to the API. The API must then perform the authorization process again.

Some tools for performing API authorization include mTLS, OAuth 2.0, and JWT. With mTLS, as opposed to TLS, the client must provide a digital signature to the API to authenticate. This can enable the API to then check the client against rules defined for their session (usually by a policy engine, such as used in ZT systems) and perform acceptance or denial based on that mutual assuredness. OAuth 2.0 is used to grant authorization generally to a third-party service without providing credentials to that service. This is common in websites where a user can log in with their email or social media profile. This form of authorization allows the third party to access API resources on behalf of the user. Finally, JWTs are generated by a server using a secret key that only the server knows, and they generally include permissions associated with the user who requested the token. When a request is made by the client, the JWT is included and used to check permissions for the request.

How to Test Authorization

As with most features and functions of any software system, the underlying architecture will play a key part in the ability to test. One recommendation made in the API design section is offloading of authorization to a dedicated part of the system. This transfers the risk and duty to test the actual authorization mechanisms to the dedicated service. The API can then rely on the service to check permissions for a given user and operation or endpoint. Moving the code doing the comparisons of rules and requests should help simplify the code that implements the handling of a particular request to the API.

Given that BOLA is the top OWASP API vulnerability, and some variety of broken authorization makes up 3 of the top 10 vulnerabilities, there has been much focus on testing to ensure this key operation of an API performs correctly. OWASP provides resources for identifying common weaknesses in authorization for an API through example scenarios that can be tailored to fit a given API or workflow. For example,

> *An automobile manufacturer has enabled remote control of its vehicles via a mobile API for communication with the driver's mobile phone. The API enables the driver to remotely start and stop the engine and lock and unlock the doors. As part of this flow, the user sends the Vehicle Identification Number (VIN) to the API. The API fails to validate that the VIN represents a vehicle that belongs to the logged in user, which leads to a BOLA vulnerability. An attacker can access vehicles that don't belong to him.* [OWASP 2023]

These types of scenarios can help identify test cases, such as a case in which a user tries to access resources that are not theirs. These tests for authentication, if they follow the architecture previously discussed, will likely fall into the realm of integration tests. The tests would instrument a user with some identity and set of privileges. The tests would then try to access endpoints or functions of the API with that user's identity. Ensuring that the return code or values from the API match expectations for all endpoints for that given set of privileges should provide sufficient coverage. Using a design of experiments may help to reduce the number of combinations for a large API with many endpoints and fine-grained permissions. OWASP also provides guidance on how to prevent such authorization issues and specifically highlights writing tests that can act as a regression case to prevent deployment of broken authentication mechanisms.

Non-Repudiation

As previously mentioned, our working definition of non-repudiation is the one articulated in NIST SP 800-18. To rephrase this definition to apply specifically to APIs, we could say that an API provides non-repudiation if it guarantees that the entity performing a request has a proven identity and is informed of delivery of the request. Non-repudiation in APIs can be checked with a digital signature: a public and private key pair that is generated and used to sign a message. A consumer of the API can use its public key to verify the authenticity of the message sent by the API. Hardware-based key pairs are sometimes used to provide non-repudiation through digital signatures. Within the US DoD, the CAC smartcard holds keys used to sign data and authenticate to systems. For APIs, the Internet Engineering Task Force (IETF) has defined multiple methods for enabling non-repudiation under the JSON Object Signing and Encryption (JOSE) standards. These standards include JWT in RFC 7519, which builds upon JSON Web Signatures (JWS) in RFC 7515, JSON Web Keys in RFC 7517, and JSON Web Encryption in RFC 7516 [IETF 2024]. JWS and JWT are both standardized and widely available forms of non-repudiation [Okta 2024].

How to Test Non-Repudiation

The JOSE framework in RFC 7520 provides examples of how to protect the various parts of a request, such as the header or payload using JWS [IETF 2016]. These examples provide a good foundation for building tests. For instance, in section 4.5 the document details the creation of a signature with detached content. This process generates a JWS object without a JWS payload field. The example provides the required data, the functions to perform the signing operation, and the JWS-protected header, both as plain text and base64 encoded. The section then provides the example value of the signing inputs and the final JWS signature when the signing algorithm is applied.

When testing an API, these input values could be used to ensure test cases are operating on standards-conforming data and producing standards-conforming output. When testing for non-repudiation, it is necessary to test at the integration level. Because the purpose of non-repudiation is to confirm that a particular API request has been made by the actual requestor, it is necessary to instrument a mock user with an asymmetric cryptographic key pair to perform the signing of an API request. This type of signature verification (and logging of the request-signature combination) may also happen outside of the API at the API gateway. For example, the Amazon API Gateway product can be configured to validate claims (signatures) in a provided JWT, including the issuer, signing key id, audience, expiration, and other fields [Amazon, 2024b]. In this case, the tests for non-repudiation become more extensive, pushing into the realm of an end-to-end test. The parts of the signature verification, however, could be distributed into unit level tests that ensure properties such as the signature and request are logged, the system correctly rejects an invalid signature, or the system rejects requests without signatures.

Architecture Cybersecurity

Since APIs are a system access point, they are especially vulnerable to attackers and improper usage. The architecture of the system in which the API is deployed in will affect the kinds of attacks it will face, the vulnerabilities it is susceptible to, and the ways it can be exploited. This section looks at architectural components that are usually deployed alongside APIs that can become vectors of risk, specifically rate limiters, load balancers, firewalls, content delivery networks (CDNs), and proxies.

APIs are susceptible to resource consumption attacks, in which malicious actors attempt to overwhelm the system by sending many requests, leading to performance degradation or complete service disruption. Rate limiting is commonly used to mitigate such attacks by imposing limits on the number of requests a client can make within a specific time frame. There are various algorithms that implement rate limiting, which can be fine-tuned according to the use case. Improper configuration of the rate limiter can create unintended bottlenecks in the number of requests it can accept [Malik 2023].

To evaluate the effectiveness of rate limiting mechanisms in an API, engineers can perform various tests, including

- Stress testing. This test sends many legitimate requests within a short period to verify that the rate limiting mechanism is functioning correctly and enforcing the specified limits.

- Limit evaluation. This test assesses the appropriateness of the rate limiting thresholds by simulating different usage patterns and analyzing the impact on system performance and availability.
- Token management. In this test, the API uses token-based rate limiting, test the token generation, distribution, and revocation processes for potential vulnerabilities.
- Bypass attempts. This test attempts to bypass the rate limiting mechanism by exploiting potential vulnerabilities, such as modifying request headers, IP address spoofing, or using distributed attack vectors. This is a form of Dynamic Application Security Testing.

Distributed denial-of-service (DDoS) attacks can thwart some rate limiting schemes by making many requests to an API from many consumers, thereby overloading the system while trying to stay under a rate limit that is imposed on a per-IP-address or per-session basis. DDoS attacks can come from threat actors, but they also may result from an under-resourced system that is not architected to handle the load of legitimate users. Load balancers distribute incoming requests across multiple servers of an API to enhance reliability and responsiveness and reduce strain on the servers [NGINX 2024a]. Load balancers are closely tied to API gateways because they provide a layer of indirection for incoming API calls to be routed to potentially different servers [Amazon 2024].

To further defend APIs against DDoS attacks, engineers can use firewalls and CDNs. Firewalls inspect incoming traffic and block malicious requests based on predefined rules or ML models, acting as an additional layer of defense against various attack vectors, including DDoS. CDNs, with their globally distributed network of servers, can absorb and mitigate DDoS attacks by filtering and redirecting traffic, reducing the load the API.

Firewalls, CDNs, and gateways are important components in securing APIs, but they can also become potential attack vectors if not properly configured and maintained. Firewalls, for instance, can be bypassed through techniques like IP spoofing, in which an attacker forges the source IP address to masquerade as a trusted entity. Misconfigured firewall rules or outdated firmware can also introduce vulnerabilities that can be exploited to gain unauthorized access or conduct DDoS attacks [Constatntin 2024]. Documentation of the firewall software should provide examples of proper configuration.

CDNs, while providing distributed caching and DDoS mitigation, can be targeted through cache poisoning attacks [Kettle 2018]. In such attacks, malicious content is injected into the CDN’s cache, which is then served to legitimate clients, potentially leading to data breaches, disrupted service, or the distribution of malware. Additionally, if the CDN’s infrastructure is compromised, an attacker could potentially gain access to the cached content, including sensitive data.

Proxies, which act as intermediaries between clients and servers, can be exploited through attacks like HTTP request smuggling [PortSwigger 2024]. In this attack, the attacker crafts an ambiguous HTTP request that is interpreted differently by the proxy and the back-end server, potentially leading to cache poisoning, request hijacking, or other unintended behaviors. Vulnerabilities in proxy software, such as outdated libraries or misconfigured security settings, can also be leveraged by attackers to gain unauthorized access or conduct man-in-the-middle attacks. Documentation of proxy software should provide examples of proper configuration.

API gateways and load balancers can work in tandem to enhance the security and scalability of APIs. API gateways act as a single entry point for all client requests, providing a centralized layer for enforcing security policies, rate limiting, authentication, and other cross-cutting concerns. They can also offload computationally intensive tasks, such as data transformation, caching, and protocol translation, from the back-end services. Load balancers, on the other hand, distribute incoming requests across multiple instances of the back-end services, ensuring that no single instance is overwhelmed, and that the overall system can handle a high volume of traffic. However, predictable patterns in traffic distribution across back-end servers could allow attackers to target specific servers, potentially overloading or exploiting them more effectively [Burke 2024].

In summary, while architectural components like rate limiters, load balancers, firewalls, and CDNs play crucial roles in securing APIs and mitigating threats such as DDoS attacks, they can also inadvertently introduce new attack vectors if not properly configured and maintained. Some SAST scanners may be able to find configuration vulnerabilities of these components in a CI pipeline.

Conclusion

From assembly instruction labels that provide reusable code to the powerful web-based APIs of today, there has always been a need to abstract the functionality of a program from its implementation. APIs expose complicated functionality from large codebases worked on by dozens if not hundreds of people, often rotating in and out of projects while simultaneously dealing with changing requirements in an increasingly adversarial environment. Under these conditions, an API must continue to behave as expected, otherwise calling applications inherit the unintended behavior the API system provides. As systems grow in complexity and size, the need for clear, concise, and usable APIs will remain.

An API's ability to evolve and interoperate impacts its lifespan and greatly determines how widely it is used and how well it adapts to changes. Security-focused design elements, including the design principles required for it to be zero-trust-compliant, must be implemented to ensure the safety and security of systems and data made accessible by the API. As more devices become interconnected, the network can no longer be considered a safe perimeter in which to exchange unencrypted messages. AI-aided design and its emerging practices hold promise in resolving issues with creating requirements, documentation, creating test cases and increasing the security of APIs, yet shortcomings remain. We expect much to change in the AI landscape in the coming future as it pertains to APIs.

The testing and cybersecurity of APIs provide assurance that it is performing as expected. This confidence is born out of continuous running of extensive tests across multiple levels of the system – from the functions and conditional statements to the databases and whole architectural components supporting the API. Viewing an API as a contract can help engineers keep a mental model of the ins and outs of the system, especially in relation to its consumers which expect a uniform interface. To maintain the qualities of confidentiality, integrity, availability and non-repudiation in APIs, requires considerable testing for the supporting services which provide these, even under changing conditions; we believe a DevSecOps development model is the best methodology to build APIs with such attributes.

In summary, we prepared this paper to provide a holistic view of the current state of API design, development, and testing. We've included a comprehensive bibliography for readers interested in exploring these topics further, with a view to the changing landscapes of APIs in the coming years.

Appendix A: API Architecture Patterns

We define *API architecture* as the design decisions related to the overall system structure and behavior that exposes a well-defined API. APIs may exchange data with other software systems using various formats and encodings, such as raw bits, packets, or objects. They may have different time constraints on their communication strategies, such as sending and receiving messages synchronously or asynchronously. They can also have a varying number of consumers and producers, ranging from one-to-one, one-to-many, and many-to-many. Understanding these and many other design decisions requires a thorough understanding of the best architecture to use for the given problem. "Best" is a misnomer here, however, since there is rarely a single solution to the given problem. The architectures described should not be taken as straightforward solutions for a given API: Each architecture has its advantages and disadvantages depending on the environment, skill level, resources available, and requirements. The same application, (for instance, an email client), can end up having very different architectures based on small differences in requirements for the various systems in which they are deployed.

In this appendix, we will briefly discuss some popular API architectures and the operations they expose, including representational state transfer (REST), remote procedure calls (RPC), query languages, event-driven architectures, and API gateways. An exhaustive analysis of all API architectures is beyond the scope of this paper and may help most readers. Instead, the information presented here provides an overview of each API architecture, their tradeoffs, and their impact on cybersecurity testing.

REST

REST is an architecture style for building web-scale, distributed hypermedia23 systems. Roy Fielding's dissertation describes REST as a set of constraints upon a client-server model using a uniform transport protocol [Fielding 2000]. Most web services today claim to follow REST. Yet this architecture is heavily misunderstood, as Fielding relates: "I am getting frustrated by the number of people calling any HTTP-based interface a REST API" [Fielding 2008]. REST follows a client-server architecture, where the server stores and manipulates information and serves it efficiently to the client, while the client displays this data to the user or uses it to take subsequent action. This separation of concerns allows the client and server to evolve independently, because it only requires the interface to stay the same.

[23] An extension to hypertext that provides multimedia facilities, such as those handling sound and video (Oxford). Hypertext itself is the ability to browse through text on a screen via links.

REST is not tied to the HTTP protocol (although it takes advantage of many of its features, such as caching and uniform access verbs). It is, in fact, not dependent on any protocol, so long as it uses unique uniform resource locators (URLs) or any other unique id for a resource for the sake of identification.

According to Fielding, “A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types” [Fielding 2008]. In other words, hypertext, not metadata, should be the driver of interaction for the API consumer.

Suppose we have an API for managing a library system. When a client requests information about a book, the API returns a response or a representation of the book resource in some kind of media type (usually JSON). Suppose the response only includes data about the requested book, without any hyperlinks or navigation guidance. It would look something like the following:

```
{
  "book": {
    "title": "The Great Gatsby",
    "author": "F. Scott Fitzgerald"
  }
}
```

In this scenario, the API fails to use hypertext to drive interaction as REST requires, so clients must rely on external knowledge (such as documentation or metadata) to drive the next state. This situation violates the REST principle that interactions should not be fully discoverable or self-descriptive [99s 2024].

The following representation includes hyperlinks to related resources, such as the author of the book or similar books in the library, and is REST-compliant. These hyperlinks provide clients with information on how to navigate the API’s resources further, which enables the client to proceed to the next state. The end user of this response need not be a human: The response is standardized (with the JSON HAL standard) which makes it machine-readable.

```
{
  "book": {
    "title": "The Great Gatsby",
    "author": {
      "name": "F. Scott Fitzgerald",
      "links": [
        {
          "rel": "self",
          "href": "/authors/1"
        }
      ]
    },
    "links": [
      {
        "rel": "self",
        "href": "/books/123"
      },
      {
        "rel": "borrow",
        "href": "/borrow",
        "method": "POST",
        "title": "Borrow this book"
      },
      {
        "rel": "similar",
        "href": "/books?similar_to=123",
        "title": "Similar books"
      }
    ]
  }
}
```

Figure 5: Example API Response for the Book The Great Gatsby

Thus, the API uses hypertext to convey not just data about the book but also the available actions (i.e., getting information about the author or borrowing the book) through hyperlinks. The client can discover and interact with different resources based on the hypertext provided in the response, which promotes discoverability.

REST requests must be stateless, which implies that no client state can be kept on the server. This prevents the complexity that comes from managing state in a distributed system as well as dealing with request routing in a horizontally scaling system. How to authenticate to a REST API if no “cookies” are allowed? According to Fielding, a client would have to authenticate itself on every request with the server, which ties in with zero trust principles (see the Zero Trust Design Principles section for more on this topic).

To improve performance, REST requires caching on the client-side and on any intermediary components (for example, local storage, client-delivery networks (CDNs), and server-side caching). REST is a layered system, which means that individual components do not have knowledge of layers beyond the one they are communicating with. So, a client connecting to a proxy, for example, does not have knowledge of what is beyond it. This characteristic provides composability and modularity to the architecture.

As an architecture, REST has some benefits and downsides. Because REST is stateless, which allows for scaling, tests that use the API now must maintain their state. This requirement can complicate tests when a long sequence of calls in a workflow need to be executed. Additionally, for a reasonably complex system, the combination of all HTTP verbs and parameters need to be tested. Consequently, automation in test generation is needed. Finally, as with most APIs, separating the testing of the API itself and the implementation can be difficult. REST, however, has ample tooling to support this separation, such as the OpenAPI spec, which is mainly targeted at REST APIs [OpenAPI 2024].

REST is a set of constraints that, taken together, provide a powerful architecture for client-server systems communicating at web scale. These constraints include a uniform interface, stateless requests, a client server architecture, cacheable responses and ability to layer the API system. Not being able to store the state of clients on the REST server, means that a server that loses connection to a client could lose track of where it was in a process. However, REST does not impose a restriction on storing state on the client, which can direct the server back to its previous step in the workflow. REST also does not provide a model to the client of how it can communicate to the server, instead relying on unique resource identifier paths that have little to no descriptive power on the shape of responses it provides. One way to avoid this is to structure paths by requiring standardized URI structures, such as using resource names as base paths and verbs as sub-paths that point to different endpoints, which enables discoverability.

RPC

According to Martin Kleppmann, a remote procedure call (RPC) “tries to make a request to a remote network service look the same as calling a function or method in your programming language within the same process” [Kleppmann 2017]. A function call is one of the most common forms of APIs that engineers routinely code, going back to FORTRAN in 1958 [Nally 2018]. It is natural, therefore, to extend this paradigm to a distributed environment. But there are perils in doing this. In the seminal paper *A Note on Distributed Computing* [Waldo 1994], the authors discuss how “local invocations and remote invocations have very different characteristics with respect to latency, memory access, concurrency, and partial failure.” Based on these observations, Kleppmann concludes that RPC is a fundamentally flawed way to make an API citing the following reasons:

- “A local function is predictable and it either succeeds or fails based on the parameters you control. A network request is unpredictable … network problems are common, so you have to anticipate them by retrying a failed request.”

- “A local function call can either return a result or throw an exception or never return. A network request has another possible outcome: it may never return a result due to a timeout.”
- “If you retry a failed network request, it could happen that the requests are getting through and only the responses are getting dropped. In that case, retrying will cause the action to be performed multiple times, unless you build a mechanism for deduplication (idempotence) into the protocol. Local function calls don’t have this problem.” [Kleppmann 2010]

However, there are counterarguments in favor of using RPC. For instance, newer RPC libraries try to make it semantically clear that the RPC’s functions are asynchronous in nature, such as by wrapping them in a future to provide error handling or chaining multiple actions via a stream to improve performance [Geewax 2021]. Proponents also say RPCs provide a more direct way of defining how to get remote resources from a server compared to the method of wrangling query parameters in URL paths along with HTTP verbs, as RESTful services do. While RPC may be easier to understand and use, it may require services in the back end of the host that allow for network communications with the server and client, which creates maintenance overhead.

Testing RPC, as with REST, may involve a lot of overhead to spin up the RPC service. In this architecture pattern, therefore, it is best to plan for testability as a function of the system, specifically by decoupling the RPC interface from the implementation. This approach will have an interface definition, an implementation of that definition that used the RPC, and a separate function containing the business logic that is insulated from reliance on RPC code. While such decoupling is a good programming practice in general, with common RPC implementations and frameworks, like gRPC, the architecture can lead to higher coupling [Amazon 2024a]. gRPC also provides code generation, which may complicate testing if the generated code is not testable.

Query Languages

Whereas REST relies on using URIs to describe resources to fetch, and an RPC stubs functions on a remote machine, query languages (QL) rely on a single endpoint to declaratively fetch data: The data consumer or client writes the data it requires and the QL implementation performs internal optimizations to fetch data in the most efficient way possible. There are numerous kinds of QLs, from text search (i.e., Elasticsearch), to structured (i.e., SQL), to logical (i.e., Datalog). Perhaps the most popular QL today is SQL (and its variants) for managing databases. SQL will not be discussed in this report, as numerous books already treat this subject at length. Furthermore, SQL is limited to database management systems rather than applications or processes. Because GraphQL has grown in popularity in the past few years to become the most common data query language used in web APIs, and because it easily introduces the concept of query languages, we will provide a brief overview.

Developed at Facebook (now Meta) in 2012 and then released as open-source, GraphQL was created to solve the problem of fetching Facebook Newsfeed resources in the low-bandwidth mobile environment. These resources were viewed as graph objects rather than stand-alone entities (i.e., blog posts or comments), since Facebook is a social network (with multiple interconnected objects). Further, Meta needed a way to have potentially thousands of engineers fetch the required data over mobile and web clients in an intuitive way. These conditions led to the development of the GraphQL specification [Byron 2018]. GraphQL queries mirror the response, which makes it easy for engineers to predict the shape of data returned. It is also strongly typed, which means each data item must conform to a specific type, and engineers can define their own types. This helps to catch potential errors at compile time. Another interesting feature of GraphQL is its ability to do introspection; that is, it allows for the client to explore the type signatures of fields and objects, providing powerful documentation information for engineers to explore and quickly learn about a new API.

GraphQL is ideal for use cases in which multiple microservices serving data need to be aggregated into a single response, with performance being a consideration. GraphQL also serves well as a unified platform for multiple kinds of clients, such as web, mobile, and edge devices, and where engineers require a short learning curve. On the other hand, there are a number of use cases for which GraphQL is not, in our opinion, ideal: GraphQL requires clients to know the schema of the data objects they request, which can be problematic if the back-end and front-end teams develop independently of each other. Caching in REST is straightforward, since every URL can be cached via local HTTP caching. However, since GraphQL has a single endpoint, caching responses requires handling them at the application level, which increases complexity. Malicious attackers can also use introspection to their advantage by gaining information about the API, potentially exposing private data related to the underlying API model.

Many other kinds of QLs exist, such as Datalog, SPARQL, CODASYL, and Cypher. Their particular use as an API will depend on some of the factors previously discussed. Because some of these QLs are more esoteric than others, the potential for a quick learning curve or widespread use may be limited. However, they offer varied advantages and may solve a particular use case extremely well.

For query languages, testing relies heavily on the data in the system on which the query is operating. Therefore, a common method for testing is to have a database loaded with test data to operate on [Binnig 2008, Chandra 2015]. However, this approach has several limitations. First, the data must be representative in all aspects of the deployed or production system (see Test Data Management). Second, the cost or overhead of maintaining a separate instance of a database on which to operate tests can be expensive.

Event-Driven

Event-driven APIs correspond to the event-driven paradigm, in which the program responds to changes as they are either dispatched or received. The main parts of this architecture are an *event producer* (commonly called a publisher), *event broker* (or event router), *event consumers* (or subscribers) and a *derived message*. The event producer produces messages and sends them along a channel (i.e., data path). An event broker may store these messages and perform some light processing of messages. The event broker will route messages to one or more event consumers, which will receive the messages and process them. Afterwards, the system may or may not immediately produce a derived message, which is broadcasted to the system as a result of successfully receiving and consuming the message. We note three advantages of the event-driven architecture:

- Consumers can be configured to subscribe to publishers either synchronously or asynchronously, depending on the requirements. Asynchronous subscribes can buffer data as it comes, which can reduce congestion.
- Events can be persisted and replayed, which helps with issues of messages getting lost in transit and providing a trail of audit and compliance.
- Subscribers and consumers can be added or removed independently of each other, allowing each side to evolve and scale according to each's need.

Event-driven architectures provide a higher degree of fault tolerance than many other architectures. For instance, an event broker can persist messages and resend to the right subscribers if they disconnect or become reconnected. This architecture is also suitable for event flows that require multiple actions per event trigger. Furthermore, a team may not have knowledge of all the different actions an event might trigger: A team working in an architecture based on request/responses will need to know the proper order of actions in which a message must be processed, whereas an event-based one does not. This lends itself to highly scalable architectures and personnel organization, since teams do not require a global view of the entire system to affect changes to it.

Event-driven architectures may be difficult to test. Due to their asynchronous capabilities, various scenarios need to be simulated, which can incur time and testing infrastructure overhead. Access control for this architecture can be complex since each subscriber needs to have the appropriate permissions to consume the events it is interested in. Improper access control can lead to unauthorized access to sensitive data or the ability for subscribers to perform malicious actions by subscribing to events they should not have access to. Robust identity management should be performed to ensure each subscriber is given the minimum necessary permissions. Additionally, the event broker should enforce strict access control policies to prevent unauthorized access to the event stream. Event brokers can further the security of the system by ensuring that messages coming in are authentic and their integrity has not been tampered with before propagating to subscribers. Using TLS or digital signatures can help provide message authentication and integrity.

Natural Language Text

Large language models have recently enabled natural language, specifically free text (i.e., text without structure, usually inputted in forms), to serve as the API for obtaining a useful response from the AI model. Research has shown that providing certain words or phrases, such as "Pretend you are X" or "Let's think step by step," can significantly improve the quality or correctness of the response [Brown 2020]. In this way, providing certain "key words" becomes part of the informal API that LLMs accept as input. In fact, ChatGPT4 allows users to enter known key words before a prompt, known as a "system prompt," to steer responses in a certain manner [OpenAI 2024]. In the same manner these key words can provide a helpful response, they can also provide a bad response: A bad actor can inject a different prompt to poison the response, or a crafty user can try to get around the AI safety rules or divulge information that it shouldn't. Some people in the software community, however, have said that free text presents a "blank canvas" challenge, since there is no exploration path or suggestive element that allows a user to use the system as intended. Analyst Benedict Evans observed, “Using an LLM to do anything specific is a series of questions and answers, and trial and error, not a process” [Evans 2023]. So, alternative interfaces to LLMs that are more restrictive and guided may emerge in the coming years.

API Gateway

“An API gateway is a component of the app-delivery infrastructure that sits between clients and services and provides centralized handling of API communication between them” [NGINX 2024]. This pattern allows for an intermediary between clients and the API implementations to monitor and re-direct traffic as needed. It also provides discoverability, authentication, and authorization capabilities. The “interface” of an API gateway is the aggregation of all endpoints a system exposes for public consumption. The implementation of these endpoints can be services that are part of a monolith or independently deployable services. While there are several benefits of API gateways, their use for any given system should be considered in the context of the problem, the number of consumers, and API complexity. Additionally, many API gateway products are targeted at web and, specifically, REST-style APIs, which may not be congruent with a given system or particular API. Gateways offer several benefits, which we briefly describe in this section.

API gateways allow for the logging and monitoring of all API use, which enables usability analysis and security monitoring. Logging and Monitoring can also ease certain operations, such as deprecating an unused endpoint or spinning up a new endpoint to certain users. Gateways provide a centralized way to route traffic and respond to cyber threats for a wide number of services. In the case of a denial-of-service attack (or even an unintentional system overload), an API gateway can be configured to drop traffic from some clients. They can perform protocol translation between clients and back-end services, allowing them to communicate using different protocols. For example, a client might send a request using HTTP, which the gateway translates into a gRPC or message queue-based request before forwarding it to the target API.

An API gateway can act as a centralized point in the architecture for implementing authentication and authorization mechanisms. It can integrate with identity providers (e.g., OAuth, OpenID Connect, or JWT) to validate incoming requests and ensure clients have permissions to access the requested resources. To protect services from excessive load or abuse, the gateway can enforce rate limits on incoming requests.

Gateways can implement resiliency patterns, such as circuit breakers, retries, and timeouts to handle failures in the back-end services. When a service becomes unresponsive or fails, the gateway can automatically redirect requests to other available instances, ensuring high availability and fault tolerance.

If the capabilities described above sound like they can be filled by a dedicated load balancer and proxy, it’s because they can: An API gateway unites many different components, such as load balancers, proxies, and caches into a single component that can be more easily reasoned about in system architectures.

The ability and responsibility of testing API gateways largely depends on the specific solution used. As discussed, API gateways perform data plane tasks such as request routing and rate limiting. However, API gateways often come as products bundled with other API management functions, such as API versioning, policy management, and other control plane functions. API gateways are also commonly provided as a software as a service (SaaS) model, because most gateways are an open systems interconnection (OSI) layer 7 application. Consequently, if an SaaS solution is used, it may come with a service agreement that precludes it from needing to be tested explicitly [Amazon 2022]. In other cases, such as when the API gateway is under control of, or internal to, the system hosting the API, testing may be easier.

Bibliography

[Adl-Tabatabai 2022]

Adl-Tabatabai, Ali-Reza, et al. Devpod: Improving Developer Productivity at Uber with Remote Development. *Uber Blog*. December 13, 2022. www.uber.com/blog/devpod-improving-developer-productivity-at-uber/

[Ahmad 2023]

Ahmad, Aakash, et al. Towards Human-Bot Collaborative Software Architecting with ChatGPT. Pages 279-285. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. June 2023. https://doi.org/10.1145/3593434.3593468

[Alberts 2014]

Alberts, Christopher; Woody, Carol; & Dorofee, Audrey. *Introduction to the Security Engineering Risk Analysis (SERA) Framework*. CMU/SEI-2014-TN-025. Software Engineering Institute, Carnegie Mellon University. 2014. https://insights.sei.cmu.edu/library/introduction-to-the-security-engineering-risk-analysis-sera-framework/

[Alberts 2014]

Alberts, Christopher; Woody, Carol. *Security Engineering Risk Analysis (SERA) Threat Archetypes*. Software Engineering Institute, Carnegie Mellon University. 2020. https://insights.sei.cmu.edu/library/security-engineering-risk-analysis-sera-threat-archetypes/

[Alberts 2023]

Alberts, Christopher J.; Bandor, Michael S.; Wallen, Charles M.; & Woody, Carol. The SEI SBOM Framework: Informing Third-Party Software Management in Your Supply Chain. *SEI Blog*. November 6, 2023. https://insights.sei.cmu.edu/blog/the-sei-sbom-framework-informing-third-party-software-management-in-your-supply-chain/

[Alshahwan 2024a]

Alshahwan, Nadia, et al. Automated Unit Test Improvement Using Large Language Models at Meta. *Cornell University*. February 14, 2024. https://arxiv.org/abs/2402.09171

[Alshahwan 2024b]

Alshahwan, Nadia; Harman, Mark; Harper, Inna; Marginean, Alexandru; Sengupta, Shubho; & Wang, Eddy. Assured LLM-Based Software Engineering. *Cornell University*. February 6, 2024. https://doi.org/10.48550/arXiv.2402.04380

[Amazon 2022]

Amazon Web Services. Amazon API Gateway Service Level Agreement. *Amazon Website*. May 5, 2022. https://aws.amazon.com/api-gateway/sla/

[Amazon 2024]
Amazon. Features of an API Gateway. *Amazon Website.* July 12, 2024 [accessed]. https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html#api-gateway-overview-features

[Amazon 2024a]
Amazon Web Services. What’s the Difference Between gRPC and REST? *Amazon Website.* July 12, 2024 [accessed]. https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/

[Amazon 2024b]
Amazon. Authorizing API requests with a JWT authorizer. *Amazon Website.* July 12, 2024 [accessed]. https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-jwt-authorizer.html

[Anderson 2020]
Anderson, Ross. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons. 2020. ISBN: 9781119644682. https://onlinelibrary.wiley.com/doi/book/10.1002/9781119644682

[Ansible 2024]
Ansible. About Ansible Molecule. *Ansible Website*. April 15, 2024 [accessed]. https://ansible.readthedocs.io/projects/molecule/

[Atlidakis 2019]
Atlidakis, Vaggelis; Godefroid, Patrice; & Polishchuk, Marina. RESTler: Stateful REST API Fuzzing. *GitHub*. 2019. https://patricegodefroid.github.io/public_psfiles/icse2019.pdf

[Baugher 1970]
Baugher, M., et al. “The Secure Real-Time Transport Protocol (SRTP).” *RFC Editor*. March 1, 1970. www.rfc-editor.org/rfc/rfc3711

[Beck 2001]
Beck, Kent, et al. Principles Behind the Agile Manifesto. *Agile Manifesto Website*. 2001. https://agilemanifesto.org/principles.html

[Beck 2014]
Beck, Kent. *Test-Driven Development by Example*. Addison-Wesley Professional. 2014. ISBN-13: 9780137585236. https://www.pearson.com/en-us/subject-catalog/p/test-driven-development-by-example/P200000009421/9780137585236

[Bell 2022]
Bell, Sam. PsRule: Introduction to Infrastructure as Code (IAC) Testing [blog post]. *Microsoft ITOps Blog*. August 10, 2022. https://techcommunity.microsoft.com/t5/itops-talk-blog/psrule-introduction-to-infrastructure-as-code-iac-testing/ba-p/3580746

[Beyer 2018]

Beyer, Betsy, et al. *The Site Reliability Workbook: Practical Ways to Implement SRE*. O'Reilly Media. ISBN-13: 978-1492029502 2018.

[Binnig 2008]

C. Binnig; Kossmann, D.; Lo E. & Saenz-Badillos, A. Automatic Result Verification for the Functional Testing of a Query Language. Pages1534-1536. In *2008 IEEE 24th International Conference on Data Engineering*. Cancun, Mexico. 2008. DOI: 10.1109/ICDE.2008.4497614.

[Bloch 2018]

Bloch, Joshua. Video. A Brief, Opinionated History of the API. *YouTube*. 2018. https://www.youtube.com/watch?v=LzMp6uQbmns

[Boyens 2022]

Boyens, Jon; Smith, Angela; Bartol, Nadya; Winkler, Kris; Holbrook, Alex; & Fallon, Matthew. *Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations*. National Institute of Standards and Technology (NIST). NIST SP 800-161r1. May 2022. https://doi.org/10.6028/NIST.SP.800-161r1

[Breivold 2008]

Breivold, Hongyu Pei; Crnkovic, Ivica; & Eriksson, Peter J. Analyzing Software Evolvability. Pages 327–330. In *2008 32nd Annual IEEE International Computer Software and Applications Conference.* July 28-August 1, 2008. https://ieeexplore.ieee.org/document/4591576

[Brown 2020]

Brown, Tom, et al. Language Models Are Few-Shot Learners. *Cornell University*. July 22, 2020. https://arxiv.org/abs/2005.14165

[Burke 2024]

Burke, Q.; McDaniel, P.; La Porta, T.; et al. Misreporting Attacks Against Load Balancers in Software-Defined Networking. *Mobile Netw Appl*. 2024. https://doi.org/10.1007/s11036-023-02156-0

[Byron 2018]

Byron, Lee. GraphQL: A Data Query Language. *Engineering at Meta Blog*. June 26, 2018. https://engineer ing.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/

[Chandra 2015]

Chandra, B.; Chawda, B.; Kar, B.; et al. Data generation for testing and grading SQL queries. *The VLDB Journal.* Volume 24. Pages 731–755. 2015. https://doi.org/10.1007/s00778-015-0395-0

[Chen 2021]

Chen, Mark, et al. Evaluating Large Language Models Trained on Code. *Cornell University*. July 14, 2021. https://arxiv.org/abs/2107.03374

[Chick 2022]
Chick, Timothy, et al. *DevSecOps Platform Independent Model (PIM).* Software Engineering Institute, Carnegie Mellon University. May 1, 2022. https://insights.sei.cmu.edu/library/devsecops-platform-independent-model-pim/

[Chick 2023]
Chick, Timothy A.; Pavetti, Scott; & Shevchenko, Nataliya. *Using Model-Based Systems Engineering (MBSE) to Assure a DevSecOps Pipeline is Sufficiently Secure*. CMU/SEI-2023-TR-001. Software Engineering Institute. 2023. https://doi.org/10.1184/R1/22592884

[CISA 2023]
Cybersecurity and Infrastructure Security Agency (CISA). *Zero Trust Maturity Model, Version 2.0*. CISA. April 2023. www.cisa.gov/sites/default/files/2023-04/zero_trust_maturity_model_v2_508.pdf

[Clarke 2004]
Steven Clarke. "Measuring API Usability." *Dr. Dobb's*. May 1, 2004. www.drdobbs.com/windows/measuring-api-usability/184405654.

[Cloudflare 2024]
Cloudflare. What Is Mutual Authentication? | Two-Way Authentication. *Cloudflare Website*. April 15, 2024 [accessed]. https://www.cloudflare.com/learning/access-management/what-is-mutual-authentication/

[CNSS 2010]
Committee on National Security Systems (CNSS). *National Information Assurance (IA) Glossary.* CNSS Instruction No. 4009. CNSS. April 26, 2010. www.dni.gov/files/NCSC/documents/nittf/CNSSI-4009_National_Information_Assurance.pdf

[Constantin 2024]
Constantin, Lucian. Over 178,000 Sonicwall Firewalls Still Vulnerable to Old Flaws. *CSO Online*. January 17, 2024. www.csoonline.com/article/1291729/over-178000-sonicwall-firewalls-still-vulnerable-old-flaws.html.

[Crispin 2009]
Crispin, Lisa & Gregory, Janet. *Agile Testing: A Practical guide for Testers and Agile Teams*. Addison-Wesley. ISBN-13: 978-0-321-53446-0

[Docker 2024]
Docker. Multi-stage builds. *Docker Docs Website*. https://docs.docker.com/build/building/multi-stage/

[DoD 2019]
Department of Defense (DoD). *DoD Enterprise DevSecOps Reference Design, Version 1.* Department of Defense. August 12, 2019. https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf

[DoD 2020e]
Department of Defense (DoD). *Operation of the Software Acquisition Pathway*. DoD Instruction 5000.87. Department of Defense. October 2, 2020. https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/500087p.PDF

[DoD 2020a]
Office of the Department of Defense Chief Information Officer. *DoD Directive 8140.01: Cyberspace Workforce Management*. Department of Defense. October 5, 2020. www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodd/814001p.pdf

[DoD 2020b]
Department of Defense (DoD). *Department of Defense Cybersecurity Test and Evaluation Guidebook*. Case # 20-S-0618. Department of Defense. 2020. https://www.dau.edu/sites/default/files/2023-09/Cybersecurity-Test-and-Evaluation-Guidebook-Version2-change-1.pdf

[DoD 2020c]
Department of Defense (DoD). *Executive Summary: DoD Data Strategy Unleashing Data to Advance the National Defense Strategy*. Department of Defense. 2020. https://media.defense.gov/2020/Oct/08/2002514180/-1/-1/0/DOD-DATA-STRATEGY.PDF

[DoD 2020d]
Department of Defense (DoD). *DOD Issues New Data Strategy*. Department of Defense. October 8, 2020. https://www.defense.gov/News/Releases/Release/Article/2376629/dod-issues-new-data-strategy/

[DoD 2021a]
Department of Defense (DoD). *DevSecOps Fundamentals Playbook*. Department of Defense. March 2021. https://dl.dod.cyber.mil/wp-content/uploads/devsecops/pdf/DoD-Enterprise-DevSecOps-2.0-Playbook.pdf

[DoD 2021b]
Department of Defense (DoD). *DevSecOps Fundamentals Guidebook: DevSecOps Tools & Activities, Version 2.0*. Department of Defense. March 2021. https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOpsTools-ActivitiesGuidebook.pdf

[DoD 2021c]
Department of Defense (DoD). *DevSecOps Playbook*, *Version 2.1*. Department of Defense. September 2021. https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOps%20Playbook_DoD-CIO_20211019.pdf

[DoD 2021d]
Department of Defense (DoD). *Identification, Tracking, and Reporting of Cyberspace Workforce Requirements*. DoD Instruction 8520.02. Department of Defense. December 21, 2021. https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/814002p.PDF

[DoD 2023a]
Department of Defense (DoD). *Data, Analytics, and Artificial Intelligence Adoption Strategy: Accelerating Decision Advantage*. Department of Defense. June 27, 2023. https://media.defense.gov/2023/Nov/02/2003333300/-1/-1/1/DOD_DATA_ANALYTICS_AI_ADOPTION_STRATEGY.PDF

[DoD 2023b]
Department of Defense (DoD). *Identity Authentication for Information Systems*. DoD Instruction 8520.03. Department of Defense. May 19, 2023. https://www.esd.whs.mil/Portals/54/Documents/DD/issuances/dodi/852003p.pdf

[DoD 2023c]
Department of Defense (DoD). *Application Programming Interface (API) Technical Guidance.* Department of Defense. October 2023. https://www.cto.mil/wp-content/uploads/2023/11/API-Guide-2023.pdf

[Dredd 2024]
Dredd. Dredd—HTTP API Testing Framework. *Dredd Website.* April 15, 2024 [accessed]. https://dredd.org/en/latest/

[Dwork 2008]
Dwork, Cynthia. Differential Privacy: A Survey of Results. *Microsoft Website*. July 12, 2024 [accessed]. https://www.microsoft.com/en-us/research/wp-content/uploads/2008/04/dwork_tamc.pdf

[ECMA 2017]
European Computer Manufacturers Association (ECMA). ECMA-404 The JSON data interchange syntax. *ECMA Website*. December 2017. https://ecma-international.org/publications-and-standards/standards/ecma-404/

[Ellis 2007]
Ellis, Brian; Stylos, Jeffrey; & Myers, Brad. The Factory Pattern in API Design: A Usability Evaluation. In *Proceedings 29th International Conference on Software Engineering (ICSE'07).* May 20-26, 2007. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0162cd54b5e0ddbc793545fc0b15176e85ebb358

[Evans 2023]
Evans, Benedict. Unbundling AI. *Benedict Evans Website.* October 5, 2023. https://www.ben-evans.com/benedictevans/2023/10/5/unbundling-ai

[Ezzini 2022]
Ezzini, Saad; Abualhaija, Sallam; Arora, Chetan; & Sabetzadeh, Mehrdad. Automated handling of anaphoric ambiguity in requirements: a multi-solution study. Pages 187-199. In *Proceedings of the 44th International Conference on Software Engineering*. May 21-29, 2022.

[Fahl 2013]
Fahl, Sascha; Harbach, Marian; Perl, Henning; Koetter, Markus; & Smith, Matthew. Rethinking SSL Development in an Appified World. Pages 49–60. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security.* November 2013. https://doi.org/10.1145/2508859.2516655

[Fielding 2000]
Fielding, Roy Thomas. *Architectural Styles and the Design of Network-Based Software Architectures* [Doctoral Dissertation]. University of California, Irvine. 2000. https://ics.uci.edu/~fielding/pubs/dissertation/top.htm

[Fielding 2008]
Fielding, Roy T. REST APIs Must Be Hypertext-Driven. *Untangled Website*. October 20, 2008. https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[Firesmith 2015]
Firesmith, Donald. A Taxonomy of Testing: What-Based and When-Based Testing Types. *SEI Blog.* September 21, 2015. https://insights.sei.cmu.edu/blog/a-taxonomy-of-testing-what-based-and-when-based-testing-types/

[Forouzan 2012]
Forouzan, Behrouz A. & Mosharraf, Firouz. *Computer Networks: A Top-down Approach*. McGraw-Hill. 2012.

[Fowler 2007]
Fowler, Martin. Mocks Aren't Stubs. *martinFowler Website.* January 2, 2007. https://martinfowler.com/articles/mocksArentStubs.html

[Fowler 2014]
Fowler, Martin. UnitTest. *martinFowler Website.* May 5, 2014. https://martinfowler.com/bliki/UnitTest.html

[Fowler 2017]
Fowler, Martin. Feature Toggles (aka Feature Flags). *martinFowler Website*. October 9, 2017. https://martinfowler.com/articles/feature-toggles.html

[Fowler 2018]
Fowler, Martin. Integration Test. *martinFowler Website.* January 16, 2018. https://martinfowler.com/bliki/IntegrationTest.html

[Fowler 2023]
Fowler, Martin. An example of LLM Prompting for programming. *martinFowler Website.* April 13, 2023. https://martinfowler.com/articles/2023-chatgpt-xu-hao.html

[Freund 2024]
Freund, Andres. [oss-security] backdoor in upstream xz/liblzma leading to ssh server compromise. *LWN Website*. March 29, 2024. https://lwn.net/ml/oss-security/20240329155126.kjjfduxw2yrlxgzm@awork3.anarazel.de/

[Geewax 2021]
Geewax, J. J. *API Design Patterns*. Manning. 2021. ISBN 9781617295850. https://www.manning.com/books/api-design-patterns

[Geogiev 2012]
Georgiev, Martin, et al. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. Pages 38-49. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*. October 2012. http://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf

[Google 2024]
Google LLC. What Are AI Hallucinations? *Google Cloud Website.* April 15, 2024 [accessed]. https://cloud.google.com/discover/what-are-ai-hallucinations

[Google 2024a]
Google LLC. What are Containers? *Google Cloud Website*. July 12, 2024 [accessed]. https://cloud.google.com/learn/what-are-containers

[GraphQL 2024]
GraphQL. Best Practices. *GraphQL Website*. July 12, 2024 [accessed]. https://graphql.org/learn/best-practices/

[gRPC 2024]
gRPC. *gRPC Website*. https://grpc.io/

[Guttag 2021]
Guttag, John. *Introduction to Computation and Programming Using Python*. The MIT Press. 2021. https://mitpress.mit.edu/9780262542364/introduction-to-computation-and-programming-using-python

[Hardt 2012]
Hardt, D. The OAuth 2.0 Authorization Framework. *Internet Engineering Taskforce Data Tracker*. October 2012. https://datatracker.ietf.org/doc/html/rfc6749#section-1.5

[Hey 2020]
Hey, Tobias; Keim, Jan; Koziolek, Anne; & Tichy, Walter F. Norbert: Transfer learning for requirements classification. Pages 169-179. In *2020 IEEE 28th International Requirements Engineering Conference*. August 31-September 4, 2020.

[Hornblower 2012]
Hornblower, Simon; Spawforth, Antony; & Eidinow, Esther. *The Oxford Classical Dictionary, Fourth Edition*. Oxford University Press. 2012. ISBN: 978-0198606413. https://global.oup.com/academic/product/the-oxford-classical-dictionary-9780199545568

[Humble 2010]
Humble, Jez & Farley, David. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, First Edition*. Pearson Education. 2010. ISBN: 9780321670229. https://www.pearson.com/en-us/subject-catalog/p/continuous-delivery-reliable-software-releases-through-build-test-and-deployment-automation/P200000009113/9780321670229

[IBM 2024]
International Business Machines. What is shift-left testing? *IBM Website*. July 12, 2024 [accessed]. https://www.ibm.com/topics/shift-left-testing

[IEEE 2014]
Institute of Electrical and Electronics Engineers (IEEE). Edited by Pierre Bourque and Richard E. (Dick) Fairley. *SWEBOK V3.0: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society. 2014. https://cs.fit.edu/~kgallagher/Schtick/Serious/SWEBOKv3.pdf

[IEEE 2021]
Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard for DevOps: Building Reliable and Secure Systems Including Application Build, Package, and Deployment*. ISO/IEC/IEEE Std 32675:2022. February 9, 2021. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9415476

[IEEE 2024]
Institute of Electrical and Electronics Engineers (IEEE) Computer Society. *What is Software Quality? IEEE Website*. July 12, 2024 [accessed]. https://www.computer.org/resources/what-is-software-quality

[IETF 2016]
Miller, Matthew A. *Examples of protecting content using JSON Object Signing and Encryption*. Internet Engineering Taskforce. May 2016. https://datatracker.ietf.org/doc/rfc7520/

[IETF 2023]
Thomson, M. & Schinazi, D. *Maintaining Robust Protocols.* Internet Engineering Taskforce. June 2023. https://datatracker.ietf.org/doc/html/rfc9413

[IETF 2024]
Bradley, John; Mattsson, John; O'Donoghue, Karen; & Cooley, Deb. Javascript Object Signing and Encryption (jose). Internet Engineering Taskforce. April 2024. https://datatracker.ietf.org/wg/jose/about/

[INCOSE/IEEE/SERC 2023]

The International Council on Systems Engineering; Institute of Electrical and Electronics Engineers (IEEE) Systems Council; & Systems Engineering Research Center (SERC). *Guide to the Systems Engineering Body of Knowledge (SEBoK), Version 2.9*. SEBoK Wiki. November 20, 2023. https://sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK)

[ISO/IEC 2011]

International Standards Organization. *Systems and software Quality Requirements and Evaluation (SQuaRE).* International Standards Organization. March 1, 2011. https://www.iso.org/standard/35733.html

[ITU 2024]

International Telecommunication Union (ITU). Introduction to ASN.1. *ITU Website*. July 12, 2024 [accessed]. https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx

[Keshta 2017]

Keshta, N., & Morgan, Y. Comparison Between Traditional Plan-Based and Agile Software Processes According to Team Size Project Domain (a Systematic Literature Review). Pages 567-575. In *IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON).* October 2017. https://doi.org/10.1109/IEMCON.2017.8117128

[Kettle 2018]

Kettle, James. Practical Web Cache Poisoning. *PortSwigger Website*. August 9, 2018. https://portswigger.net/research/practical-web-cache-poisoning

[Kim 2023]

Kim, Myeong-Soo; Corradini, Davide; Sinha, Saurabh; Orso, Alessandro; Pasqua, Michele; Tzoref-Brill, Rachel; & Ceccato, Mariano. Enhancing REST API Testing with NLP Techniques. Pages 1232-1243. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis.* July 2023. https://doi.org/10.1145/3597926.3598131

[Kleppmann 2017]

Kleppmann, Martin. *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*. O'Reilly. 2017. ISBN: 9781491903100. https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

[KU 2024]

Kutztown University (KU). Invariants, Preconditions, and Postconditions. *Kutztown University Website*. April 15, 2024 [accessed]. www.kutztown.edu/Departments-Offices/A-F/ComputerScienceInformationTechnology/Documents/Student%20Resources/DocumentationInvariants.pdf

[Kulkarni 2015]

Kulkarni, S.; Ghosh, U.; & Pasupuleti, H. Considering security for ZigBee protocol using message authentication code. Pages 1-6. In *2015 Annual IEEE India Conference (INDICON)*. December 2015. DOI: 10.1109/INDICON.2015.7443625.

[Kumar 2016]

Kumar, Divya; & Mishra, K. K. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science*. Volume 79. 2016. Pages 8–15. https://doi.org/10.1016/j.procs.2016.03.003

[Leach 2017]

Leach, Brandur. APIs as infrastructure: future-proofing Stripe with versioning. *Stripe Blog.* August 5, 2017. https://stripe.com/blog/api-versioning

[Lindsey 1976]

Lindsey, C. H. Proposal for a Modules Facility in ALGOL 68. *ALGOL Bulletin*. Volume 68. Issue 39. February 1976. Pages 20-29. https://dl.acm.org/doi/10.5555/1061669.1061675

[Loadmill 2024]

Loadmill. Loadmill—AI-Powered Solution. *Loadmill Website.* April 15, 2024 [accessed]. https://docs.loadmill.com/

[Luitel 2024]

Luitel, Dipeeka; Hassani, Shabnam; & Sabetzadeh, Mehrdad. Improving Requirements Completeness: Automated Assistance through Large Language Models. *Cornell University*. February 14, 2024. https://doi.org/10.48550/arXiv.2308.03784

[Malik 2023]

A. E. Malki & Zdun, U. Combining API Patterns in Microservice Architectures: Performance and Reliability Analysis. Pages 246-257. *2023 IEEE International Conference on Web Services (ICWS)*. July 2023. https://doi.org/10.1109/ICWS60048.2023.00044

[Martin-Lopez 2020]

Martin-Lopez, Alberto. AI-Driven Web API Testing. Pages 202–205. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. June 2020. https://doi.org/10.1145/3377812.3381388

[Mead 2016]

Mead, Nancy R. & Woody, Carol. *Cyber Security Engineering: A Practical Approach for Systems and Software Assurance*. Addison-Wesley. 2017. ISBN: 978-0134189802. https://insights.sei.cmu.edu/library/cyber-security-engineering-a-practical-approach-for-systems-and-software-assurance/

[Meng 2017]
Meng, Michael; Steinhardt, Stephanie; & Schubert, Andreas. Application Programming Interface Documentation: What Do Software Developers Want? *Journal of Technical Writing and Communication.* Volume 48. Issue 3. July 2017. Pages 295–330. https://doi.org/10.1177/0047281617721853

[Meyer 1997]
Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall. 1997. ISBN: 978-0-13-629155-8. https://dl.acm.org/doi/10.5555/261119

[Meyers 2016]
Myers, Brad A. & Stylos, Jeffrey. Improving API Usability. *Communications of the ACM*. Volume 59. Issue 6. May 2016. Pages 62–69. https://www.cs.cmu.edu/afs/cs/Web/People/NatProg/papers/API_Usability_Article_submitted.pdf

[Microsoft 2024]
Microsoft Corporation. restler-fuzzer. *GitHub.* November 29, 2023. https://github.com/microsoft/restler-fuzzer

[MIT 2004]
Massachusetts Institute of Technology (MIT). How to Implement a Provider for the Java^{TM} Cryptography Architecture. *MIT CERT Website*. July 25, 2004. https://web.mit.edu/java_v1.5.0_22/distrib/share/docs/guide/security/HowToImplAProvider.html#Step%201

[MITRE 2006]
Common Weakness Enumeration. CWE-502: Deserialization of Untrusted Data *MITRE CWE Website*. July 19, 2006. https://cwe.mitre.org/data/definitions/502

[MITRE 2024]
Common Weakness Enumeration. CWE-501: Trust Boundary Violation. *MITRE CWE Website*. February 29, 2024. https://cwe.mitre.org/data/definitions/501

[MITRE 2024a]
LLM Data Leakage. MITRE ATLAS Website. October 25, 2023. https://atlas.mitre.org/techniques/AML.T0057/

[Moharil 2023]
Moharil, Ambarish & Sharma, Arpit. TABASCO: A Transformer Based Contextualization Toolkit*.* *Science of Computer Programming*. Volume 230. Issue C. August 1, 2023. https://doi.org/10.1016/j.scico.2023.102994

[MSRC 2020]
Microsoft Security Response Center (MSRC). Customer Guidance on Recent Nation-State Cyber Attacks. *MSRC Blog*. December 14, 2020. https://msrc.microsoft.com/blog/2020/12/customer-guidance-on-recent-nation-state-cyber-attacks/

[Murphy 2017]

Murphy, Lauren; Alliyu, Tosin; Macvean, Andrew; Kery, Mary Beth; & Myers, Brad A. Preliminary Analysis of REST API Style Guidelines. Plateau 2017. October 2017. https://www.cs.cmu.edu/~NatProg/papers/API-Usability-Styleguides-PLATEAU2017.pdf

[Nally 2018]

Nally, Martin. REST vs. RPC: What Problems Are You Trying to Solve with Your APIs? *Google Cloud Blog.* October 15, 2018. https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-your-apis

[Nally 2020]

Nally, Martin. Understanding GRPC, OpenAPI and REST and When to Use Them. *Google Cloud Blog.* April 10, 2020. https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them

[NGINX, 2024]

What Is an API Gateway? A Quick Learn Guide: Nginx Learning. *NGINX*. March 28, 2023. https://www.nginx.com/learn/api-gateway/#:~:text=An%20API%20gateway%20is%20a,%2Dcloud%2C%20and%20hybrid%20environments.

[NGINX 2024a]

What Is Load Balancing?. *NGINX*.1 June 2023 [accessed].https://www.f5.com/glossary/load-balancer.

[Nguyen 2023]

Nguyen, Cuong; Bui, Huy; Nguyen, Vu; & Nguyen, Tien. *An Approach to Generating API Test Scripts Using GPT*. Pages 501–509. In *Proceedings* of the 12th International Symposium on Information and Communication Technology. December 2023. https://doi.org/10.1145/3628797.3628947

[Nine Nines 2024]

Nine Nines. Cowboy: REST Principles. *Nine Nines Website.* April 15, 2024 [accessed]. https://nine-nines.eu/docs/en/cowboy/2.8/guide/rest_principles/

[NIST 2024]

National Institute of Standards and Technology (NIST). Glossary. *NIST Website*. February 26, 2024. https://csrc.nist.gov/glossary

[NIST 2014]

National Institute of Standards and Technology (NIST). *NIST Special Publication 800-162 Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. NIST. 2014. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf

[NIST CSRC 2024]
National Institute of Standards and Technology (NIST) Computer Security Resource Center (CSRC). Glossary. *NIST Website*. April 15, 2024 [accessed]. https://csrc.nist.gov/glossary/term/API

[NIST Glossary]
National Institute of Standards and Technology (NIST). Glossary. *NIST Website*. May 14, 2024 [accessed]. https://csrc.nist.gov/glossary/term/application_programming_interface

[Norman 1988]
Norman, Don. *The Design of Everyday Things*. Basic Books. 1988. https://www.hachettebookgroup.com/titles/don-norman/the-design-of-everyday-things/9780465050659/?lens=basic-books

[Okta 2024]
Okta. JSON Web Tokens. *JWT.io Website*. https://jwt.io/

[OpenAI 2024]
OpenAI Developer Platform. Prompt Engineering. *OpenAI Developer Platform.* April 15, 2024 [accessed]. https://platform.openai.com/docs/guides/prompt-engineering

[OpenAPI 2024]
Open API Initiative. *Open API Website*. April 26, 2024 [accessed]. https://www.openapis.org/

[Oracle 2024]
Oracle Corporation. How to Implement a Provider for the Java Cryptography Architecture. *Oracle Website.* February 23, 2024 [accessed]. https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/HowToImplAProvider.html

[OWASP 2023]
Open Web Application Security Project (OWASP). OWASP Top 10 API Security Risks – 2023. *OWASP Website*. 2023. https://owasp.org/API-Security/editions/2023/en/0x11-t10/

[OWASP 2023a]
Open Web Application Security Project (OWASP). API1:2023 Broken Authorization. OWASP Website. 2023. https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/

[OWASP 2023b]
Open Web Application Security Project (OWASP). API2:2023 Broken Authentication. OWASP Website. 2023. https://owasp.org/API-Security/editions/2023/en/0xa2-broken-authentication/

[OWASP 2024a]
Open Web Application Security Project (OWASP). Fuzzing | OWASP. *OWASP Website.* April 15, 2024 [accessed]. https://owasp.org/www-community/Fuzzing

[OWASP 2024b]
Open Web Application Security Project (OWASP). Deserialization of Untrusted Data. *OWASP Website.* April 15, 2024 [accessed]. https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data

[OWASP 2024c]
Open Web Application Security Project (OWASP). Vulnerability Scanning Tools. *OWASP Website*. April 15, 2024 [accessed]. https://owasp.org/www-community/Vulnerability_Scanning_Tools

[OWASP 2024d]
Open Web Application Security Project (OWASP). LLM02:2023 – Data Leakage. *OWASP Website*. 2023. https://owasp.org/www-project-top-10-for-large-language-model-applications/Archive/0_1_vulns/Data_Leakage.html

[OWASP 2024e]
Open Web Application Security Project (OWASP). Cryptographic Storage Cheat Sheet. *OWASP Website*. July 25, 2024 [accessed]. https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

[Padgette 2022]
Padgette, John; Bahr, John; Batra, Mayank; Holtmann, Marcel; Smithbey, Rhonda; Chen, Lily; & Scarfone, Karen. *NIST Special Publication 800-121 Guide to Bluetooth Security*. NIST. 2022. https://doi.org/10.6028/nist.sp.800-121r2-upd1

[Perri 2023]
Perri, Lori. What's New in Artificial Intelligence from the 2023 Gartner Hype Cycle. *Gartner Website.* August 17, 2023. https://www.gartner.com/en/articles/what-s-new-in-artificial-intelligence-from-the-2023-gartner-hype-cycle

[Piccioni 2013]
Piccioni, Marco; Furia, Carlo A; & Meyer; Bertrand., An empirical study of API usability. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. Pages 5-14. October 2013. https://doi.org/10.1109/esem.2013.14

[Plover 2006]
Plover. Common Weakness Enumeration. CWE-328: Use of Weak Hash. *MITRE CWE Website*. July 19, 2006. https://cwe.mitre.org/data/definitions/328.html

[PortSwigger 2024]
HTTP Request Smuggling. *Web Security Academy*. March 22, 2024 [accessed]. https://portswigger.net/web-security/request-smuggling

[Preston-Warner 2024]

Preston-Warner, Tom. Semantic Versioning 2.0.0. *Semantic Versioning Website*. July 10, 2024 [accessed]. https://semver.org

[Puppet 2024]

Validating and Testing Modules. *Puppet Website.* March 5, 2024 [accessed]. www.puppet.com/docs/pdk/3.x/pdk_testing.html

[Rama 2013]

Rama, Girish Maskeri & Kak, Avinash. Some structural measures of API usability. *Software: Practice and Experience*. Volume 45. Issue 1. September 3, 2013. Pages 75–110. https://doi.org/10.1002/spe.2215

[Ray 2023]

Tikayat Ray, Archana.; Cole, Bjorn. F.; Pinon Fischer, Olivia. J.; Bhat, Anirudh. P.; White, Ryan. T.; & Mavris, Dimitri. N. Agile Methodology for the Standardization of Engineering Requirements Using Large Language Models. *Systems.* Volume 11. Issue 352. May 14, 2023. https://doi.org/10.3390/systems11070352

[Rodriguez 2013]

Rodríguez, Pilar. (2013). *COMBINING LEAN THINKING AND AGILE SOFTWARE DEVELOPMENT How do software-intensive companies use them in practice?* [Doctoral Dissertation]. University of Oulu, Finland. 2013. https://www.researchgate.net/publication/275654650_COMBINING_LEAN_THINKING_AND_AGILE_SOFTWARE_DEVELOPMENT_How_do_software-intensive_companies_use_them_in_practice/

[Rose 2020]

Rose, Scott; Borchert, Oliver; Mitchell, Stu; & Connelly, Sean. *NIST Special Publication 800-207 Zero Trust Architecture*. NIST. 2020. https://doi.org/10.6028/nist.sp.800-207

[Russell 2021]

Russell, Stuart & Norvig, Peter. *Artificial Intelligence: A Modern Approach, Fourth Edition*. Pearson. 2021. ISBN: 9780137505135. https://www.pearson.com/en-us/subject-catalog/p/artificial-intelligence-a-modern-approach/P200000003500/9780137505135

[Salinas 2024]

Salinas, Abel & Morstatter, Fred. The Butterfly Effect of Altering Prompts: How Small Changes and Jailbreaks Affect Large Language Model Performance. *ArXiv preprint.* April 1, 2024. https://arxiv.org/abs/2401.03729

[Sanders 2021]

Sanders, Geoffrey; Morrow, Timothy; Richmond, Nathaniel; & Woody, Carol. *Integrating Zero Trust and DevSecOps*. Software Engineering Institute, Carnegie Mellon University. 2021. https://insights.sei.cmu.edu/library/integrating-zero-trust-and-devsecops/

[Sarcar 2009]

Sarcar, Amritam. *Runtime Assertion Checking for JML on the Eclipse Platform Using AST Merging* [Dissertation]. University of Texas, El Paso. January 2009. https://wiki-int.sei.cmu.edu/confluence/pages/viewpage.action?pageId=118817253

[Saydjari 2018]

Saydjari, O. Sami. *Engineering Trustworthy Systems.* McGraw Hill. 2018. ISBN: 978-1260118179. www.engineeringtrustworthysystems.com/

[Scanlon 2018]

Scanlon, Tom. 10 Types of Application Security Testing Tools: When and How to Use Them [blog post]. *SEI Blog*. July 9, 2018. https://insights.sei.cmu.edu/blog/10-types-of-application-security-testing-tools-when-and-how-to-use-them/

[Schwering 2024]

Schwering, Ramona. Pyramid or Crab? Find a Testing Strategy That Fits. *Web.dev Website.* February 23, 2024 [accessed]. https://web.dev/articles/ta-strategies

[Schaffer 2022]

Schaffer, André. Testing of Microservices [blog post]. *Spotify Engineering Blog*., January 11, 2018. https://engineering.atspotify.com/2018/01/testing-of-microservices/

[Searls 2020]

@searls (Searls, Justin). *People love debating what percentage of which type of tests to write, but it's a distraction. Nearly zero teams write expressive tests that establish clear boundaries, run quickly & reliably, and only fail for useful reasons. Focus on that instead.* X (formerly Twitter). *9:58 PM May 14, 2021*. https://twitter.com/searls/status/1393385209089990659?lang=en

[Seemann 2012]

Seemann, Mark. TDD Test Suites Should Run in 10 Seconds or Less. *Ploeh Blog.* May 24, 2012. https://blog.ploeh.dk/2012/05/24/TDDtestsuitesshouldrunin10secondsorless/

[SEI 2016]

Software Engineering Institute (SEI). *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*. SEI, Carnegie Mellon University. June 30, 2016. https://insights.sei.cmu.edu/library/sei-cert-c-coding-standard-rules-for-developing-safe-reliable-and-secure-systems-2016-edition/

[SEI 2024a]

Software Engineering Institute (SEI). *Continuous Delivery*. SEI. April 16, 2024 [accessed]. https://cmu-sei.github.io/DevSecOps-Model/?_gl=1*i7p6w5*_ga*MTM1NjUzMjk1My4xNzA5MTI4MDM5*_ga_87WECW6HCS*MTcwOTE0NDg1MS4xLjAuMTcwOTE0NDg1MS42MC4wLjA.#Diagrams__7b8cab04-0f58-4b71-88b6-8eef5374734e

[SEI 2024b]
SEI CERT Oracle Coding Standard for Java. *Software Engineering Institute Wiki*. April 15, 2024 [accessed]. https://wiki.sei.cmu.edu/confluence/display/java

[SEI 2024c]
SEI CERT Basic Fuzzing Framework. *Software Engineering Institute*. Github. https://github.com/CERTCC/certfuzz/wiki

[Serbout 2023]
Serbout, Souhaila; El Malki, Amine; Pautasso, Cesare; & Zdun, Uwe. API Rate Limit Adoption—A Pattern Collection. Pages 1–20. In *Proceedings of the 28th European Conference on Pattern Languages of Programs*. July 2023. https://doi.org/10.1145/3628034.3628039

[Sherman 2024]
Sherman, Mark. Using ChatGPT to Analyze Your Code? Not So Fast. *SEI Blog.* February 12, 2024. https://insights.sei.cmu.edu/blog/using-chatgpt-to-analyze-your-code-not-so-fast/

[Shostack 2014]
Shostack, Adam. *Threat Modeling: Designing for Security*. John Wiley & Sons, Inc. Page 139. ISBN: 978-1-118-80999-0. https://shostack.org/books/threat-modeling-book

[Siddiqui 2024]
Siddiqui, Zeba. UnitedHealth hackers used stolen login credentials to break in, CEO says. *Reuters*. April 30, 2024. https://www.reuters.com/technology/cybersecurity/unitedhealth-hackers-took-advantage-citrix-vulnerabilty-break-ceo-says-2024-04-29/

[Spinellis 2007]
Spinellis, Diomidis. Another level of indirection. In *Beautiful Code: Leading Programmers Explain How They Think*. Oram, Andy & Wilson, Greg [editors]. 2007. O’Reilly and Associates. Pages 279-291. https://www2.dmst.aueb.gr/dds/pubs/inbook/beautiful_code/html/Spi07g.html

[Sridhara 2023]
Giriprasad Sridhara, Sourav Mazumdar, et al. 2023. ChatGPT: A Study on its Utility for Ubiquitous Software Engineering Tasks. *ArXiv preprint*. 2023.. https://arxiv.org/abs/2305.16837

[SSL 2023]
What Is SSL/TLS: An In-Depth Guide. *SSL Website.* April 15, 2024 [accessed]. www.ssl.com/article/what-is-ssl-tls-an-in-depth-guide/

[Sukhbaattar 2015]
Sukhbaatar, Sainbayar; Szlam, Arthur; Weston, Jason; & Fergus, Rob. End-To-End Memory Networks. Cornell University. *ArXiv preprint*. March 2015. https://arxiv.org/abs/1503.08895

[Swagger 2024]
OpenAPI Specification. *Swagger Website*. April 15, 2024 [accessed]. https://swagger.io/specification/

[Swanson 2006]
Swanson, Marianne; Hash, Joan; & Bowen, Pauline. *NIST Special Publication 800-18 Rev. 1. Guide for Developing Security Plans for Federal Information Systems*. NIST. February 2006. https://csrc.nist.gov/pubs/sp/800/18/r1/final

[Testcontainers 2024]
Testcontainers. Unit tests with real dependencies. *Testcontainers Website*. July 10, 2024 [accessed]. https://testcontainers.com/

[Thurgood 2018]
Thurgood, Steven & Ferguson, David.. Implementing SLOs. In *Google SRE Book*. Beyer, Betsy & Hidalgo, Alex [editors]. Google. 2018. https://sre.google/workbook/implementing-slos/

[USG Compendium 2019]
United States Government Compendium of Interagency and Associated Terms. USG Compendium Website. November 2019. https://www.jcs.mil/Portals/36/Documents/Doctrine/dictionary/repository/usg_compendium.pdf?ver=2019-11-04-174229-423

[Villalba 2023]
Villalba, Marcia. Previewing Environments Using Containerized AWS Lambda Functions. *AWS Compute Blog*. February 6, 2023. https://aws.amazon.com/blogs/compute/previewing-environments-using-containerized-aws-lambda-functions/

[Vocke 2018]
Vocke, Ham. The Practical Test Pyramid. *Martin Fowler Website.* February 26, 2018. https://martinfowler.com/articles/practical-test-pyramid.html

[Vossen 2009]
Vossen, Gottfrid; Long, Darrell D.E.; Yu, Jeffry Xu [ed.]. *Web Information Systems Engineering—WISE 2009*. Springer. 2009. https://link.springer.com/book/10.1007/978-3-642-04409-0

[Waldo 1994]
Waldo, Jim et. Al. *A Note on Distributed Computing.* Sun Microsystems Laboratories, Inc. 1994. https://scholar.harvard.edu/files/waldo/files/waldo-94.pdf

[Wang 2020]
Wang, Yawen; Shi, Lin; Li, Mingyang; Wang, Qing; & Yang,Yang. A Deep Context-wise Method for Coreference Detection in Natural Language Requirements. Pages 180-191. In *2020 IEEE 28th International Requirements Engineering Conference*. https://ieeexplore.ieee.org/document/9218208/authors

[web.dev 2022]
What Is Digital Accessibility, and Why Does It Matter? *Web.dev Website.* September 30, 2022. https://web.dev/learn/accessibility/why

[web.dev 2023]
Automated Accessibility Testing. *eb.dev Website.* January 12, 2023. https://web.dev/learn/accessibility/test-automated

[WebRTC 2024]
WebRTC For The Curious. March 9, 2024 [accessed]. https://webrtcforthecurious.com

[Wei 2022]
Wei, Moshi; Harzevili, Nima Shiri; Huang Yuchao; Wang, Junjie; & Wang, Song. Clear: Contrastive Learning for API Recommendation. Pages 276-387. In *Proceedings of the 44th International Conference on Software Engineering*. May 2022. https://ieeexplore.ieee.org/document/9793955

[Weinert 2024]
Weinert, Alex. Your Pa$$word Doesn't Matter [blog post]. *Microsoft Tech Community*. February 7, 2024. https://techcommunity.microsoft.com/t5/microsoft-entra-blog/your-pa-word-doesn-t-matter/ba-p/731984

[Wells 2013]
Wells, Don. Acceptance Tests. *Extreme Programming.* July 18, 2024. http://www.extremeprogramming.org/rules/functionaltests.html

[WhatWG 2024]
Web Hypertext Application Technology Working Group (WHATWG). HTML Living Standard. *WHATWG website*. April 24, 2024. https://html.spec.whatwg.org/

[White 2023]
White, Jules; Fu, Quchen; Hays, Sam; Sandborn, Michael; Olea, Carlos; Gilbert, Henry; Elnashar, Ashraf; Spencer-Smith, Jesse; & Schmidt, Douglas C. A Prompt Pattern Catalog to Enhance Prompt Engineering With ChatGPT. *ArXiv preprint*. 2023. https://arxiv.org/abs/2302.11382

[White House 2021]
The United States White House. *Executive Order on Improving the Nation's Cybersecurity*. The White House. May 12, 2021. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[Williams 2009]
Williams, Laurie; Kudrjavets, Gunnar; & Nagappan, Nachiappan. On the Effectiveness of Unit Test Automation at Microsoft. Pages 81–89. In *2009 20th International Symposium on Software Reliability Engineering.* November 2009. *https://ieeexplore.ieee.org/document/5362086*

[Woody 2016]
Woody, Carol. Security Engineering Risk Analysis (SERA). Pages 23–24. In *Proceedings of the 3rd International Workshop on Software Engineering Research and Industrial Practice.* May 2016. https://doi.org/10.1145/2897022.2897024

[Woody 2022a]
Woody, Carol. An Acquisition Security Framework for Supply Chain Risk Management. *SEI Blog*. October 17, 2022. https://insights.sei.cmu.edu/blog/a-cybersecurity-engineering-strategy-for-devsecops-that-integrates-with-the-software-supply-chain/

[Woody 2022b]
Woody, Carol. A Cybersecurity Engineering Strategy for DevSecOps that Integrates with the Software Supply Chain. *SEI Blog*. January 31, 2022. https://insights.sei.cmu.edu/blog/a-cybersecurity-engineering-strategy-for-devsecops-that-integrates-with-the-software-supply-chain/

[W3C 2008]
World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Fifth Edition). *W3C Website*. November 26, 2008. https://www.w3.org/TR/xml/

[Xavier 2017]
Xavier, L; Hora, A.; & Valente, T. Why Do We Break APIs? First Answers from Developers. Pages 392–396. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. February 2017. https://ieeexplore.ieee.org/abstract/document/7884640

[Yackel 2023]
Yackel, Ryan. What Is Homomorphic Encryption, and Why Isn't It Mainstream? *Keyfactor website*. July 6, 2021.https://www.keyfactor.com/blog/what-is-homomorphic-encryption/

[Zetter 2015]
Zetter, Kim. Teen Who Hacked CIA Director's Emails Tells How He Did It. *WIRED*. October 19, 2015. https://www.wired.com/2015/10/hacker-who-broke-into-cia-director-john-brennan-email-tells-how-he-did-it/

[99s 2024]
REST Principles. *99s. March 20, 2024 [accessed]. https://ninenines.eu/docs/en/cowboy/2.8/guide/rest_principles*

Legal Markings

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific entity, product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute nor of Carnegie Mellon University - Software Engineering Institute by any such named or represented entity.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

DM24-0454

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu