TRACEABLE

REFERENCE ARCHITECTURE

API Security Reference Architecture for a Zero Trust World

Contents

Executive Summary

Zero Trust is a cybersecurity framework that challenges the traditional approach of assuming trust within a network and instead operates on the principle of "never trust, always verify." It establishes that all users, devices, and applications, irrespective of their location, are potentially untrusted entities and access to resources is granted based on continuous verification of identity, context, and security posture. The core goal of Zero Trust is to enhance security by minimizing the potential attack surface, reducing the risk of data breaches, and mitigating the impact of cyber threats. While network-centric measures have been effective in safeguarding critical assets, the application layer has yet to receive comparable attention from those adopting Zero Trust.

Traceable's reference architecture addresses securing the application layer with Zero trust principles.

APIs are the **Universal Attack Vector.**

The type of attacks that used to occur at the browser, the application layer, at the network layer, at the infrastructure layer have started to all converge at the API layer. You can execute a DDoS attack, a bot attack, an injection attack, a remote code execution, all via an API.

Every type of attack that developed in the lower tiers of virtualization (OS, server, network, device, database, etc) can now be done at Layer 7. DDoS attacks couldn't be leveraged to attack a database - one being a network attack surface and the other a data layer attack surface. But, APIs can be used as a conduit for DDOS-type attacks, data exfiltration, account takeovers, fraudulent account creation, token theft attacks, and more.

APIs will continue to be an attractive target because of their frequent association with customer-facing applications where access to financial transactions makes a breach more profitable for threat actors because they can attack at scale.

Without API security, you can not achieve Zero Trust security.

Context

1. **ZT works**: Zero Trust has helped many organizations, including the US Government, to secure their digital assets.
2. **ZT not focused on apps**: However, Zero Trust architectures and most adoption efforts are more network-focused than application layer-focused, with no information about APIs.
3. **Apps have changed:** The application layer has evolved since Zero Trust was first defined (microservices, cloud-native, APIs, IdMs, etc)
4. **Apps = APIs:** Modern applications are built using APIs as the conduit between all components and all users. APIs are becoming the next layer of the network (eg. communications, data movement)
5. **ZT for APIs undefined:** It is undefined how to secure APIs using Zero Trust concepts. Although APIs are part of the application layer they have different security requirements and need separate definitions.

Pillars of Zero Trust for APIs

1. **Data/Resources:** what you are protecting
2. **Conduits (APIs):** the pathways to what you are protecting
3. **Identity:** who you are protecting it from
4. **Entity:** used by who you are protecting it from
5. **Visibility & Analytics:** Context is key to success
6. **Automation & Orchestration:** Fast decisions and actions required
7. **Governance:** All actions auditable and reportable

Requirements of Zero Trust for APIs

Pillar	Requirements
Data / Resources	• **Data identification and classification:** Automatic identification of sensitive data and data sets
Conduits (APIs)	• **Keep an updated inventory of all APIs:** Enables understanding and management of attack surface. • **Detect & block high-risk APIs:** The security and integrity of the APIs themselves is important to keep them managed. • **Flexible PEP + DCP distribution:** A protection mesh of flexibly distributed PEP's and data collection points.
Identity	• **In-depth user authN/authZ tracking:** Tracks users across transactions, IP changes, token changes, and across time.
Entity	• **Entity risk assessment:** Evaluate risk attributes of source and/or target entities.
Visibility & Analytics	• **Complete transaction visibility:** Ability to see North/South, East/West, and inside encrypted TLS transactions. • **Gather all API transactional data for full context:** Monitoring and recording all API transactions.
Automation & Orchestration	• **Data & resource access policy framework:** Granular data access policies. • **Security posture contributor and consumer:** Contribute to and consume from XDR, SIEM, and SOAR.
Governance	• **All actions on API requests are auditable:** Validation of proper system functioning and proper protection.

Zero Trust Introduction

Zero Trust is a cybersecurity framework that challenges the traditional approach of assuming trust within a network and instead operates on the principle of "never trust, always verify." It establishes that all users, devices, and applications, irrespective of their location, are potentially untrusted entities, and access to resources is granted based on continuous verification of identity, context, and security posture. The core goal of Zero Trust is to enhance security by minimizing the potential attack surface, reducing the risk of data breaches, and mitigating the impact of cyber threats.

While Zero Trust has made significant progress in enhancing security, its focus and execution have been primarily on network-level and identity access controls. The framework emphasizes continuous monitoring of network traffic, user behavior, and application activity to detect anomalies and potential security incidents. Granular access controls, least privilege principles, and micro-segmentation are employed to limit lateral movement within the network and contain potential breaches.

While these network-centric measures have been effective in safeguarding critical assets, the application layer has yet to receive comparable attention from those adopting Zero Trust.

APIs: The New Network

Modern applications now heavily rely on APIs. They have become the conduits of the modern-day application landscape and serve as the fundamental mechanism through which applications communicate, exchange data, and integrate with external services.

Just like plumbing pipes or electrical lines that route and transport resources, APIs facilitate the flow of information, functionality, and services between applications, enabling the development of interconnected and interoperable software ecosystems. With their ability to connect and extend the capabilities of applications, APIs have become the backbone of modern-day applications.

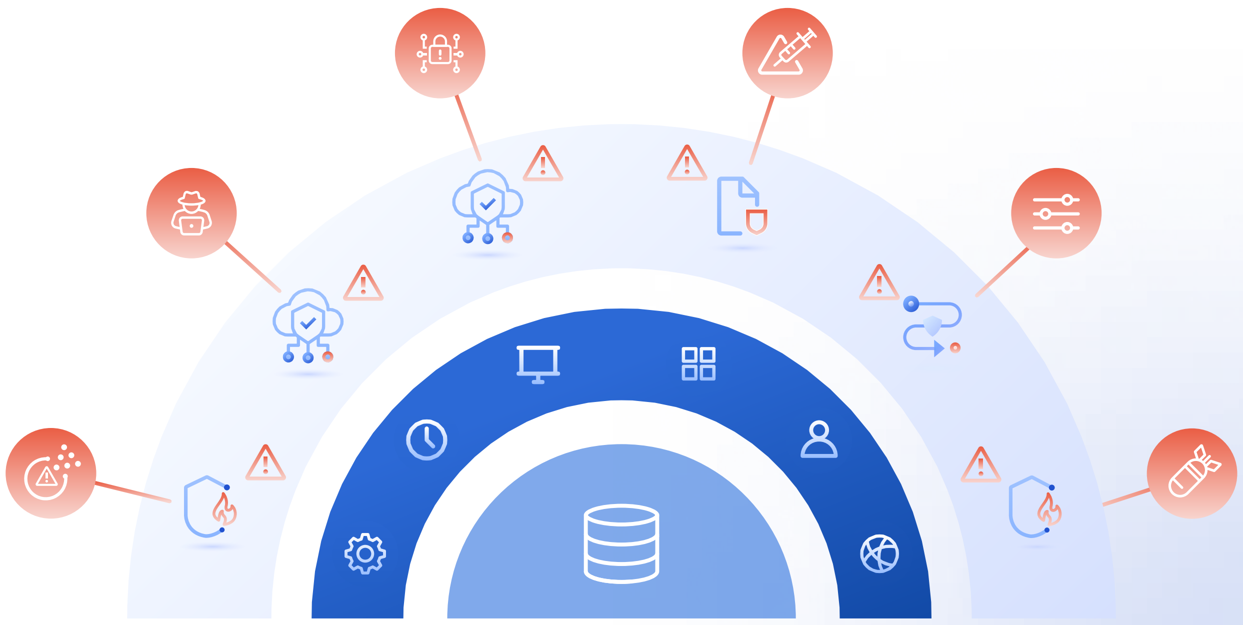
With the reach of APIs across the application stack, a new, larger, and more pervasive attack surface has emerged. While securing the network layer and below is necessary to get better security from Zero Trust, it is no longer sufficient.

As technology marches on, each layer of the OSI model becomes more solved and commoditized and the remaining layers take them for granted. At this point, this is true up to the application layer. Modern applications assume presentation, session, transport, network, etc will work as expected, and they focus on routing communications and handling security at their own layer, as the OSI stack was designed.

The rich opportunities for growth and exploitation are now flourishing at the application layer, as the layers below more or less dutifully deliver. This doesn't mean that we no longer need to secure the other layers (we absolutely do), but there is a new frontier of battle where security practices have not yet matured, and it needs our attention. That is the APIs of the application layer. The application layer itself has had security's focus for a while now (eg. WAF's), but as already discussed, it is now driven by APIs.

Before Widespread API Use

Before widespread API use, an attacker would have to learn to attack each layer they were trying to get through. They would have to learn different attacks for different attack vectors at each layer of the stack. And also learning how to get around each of the different security technologies that typically protected each attack vector. There was no single pipe that would shuttle them to their soft gooey target.

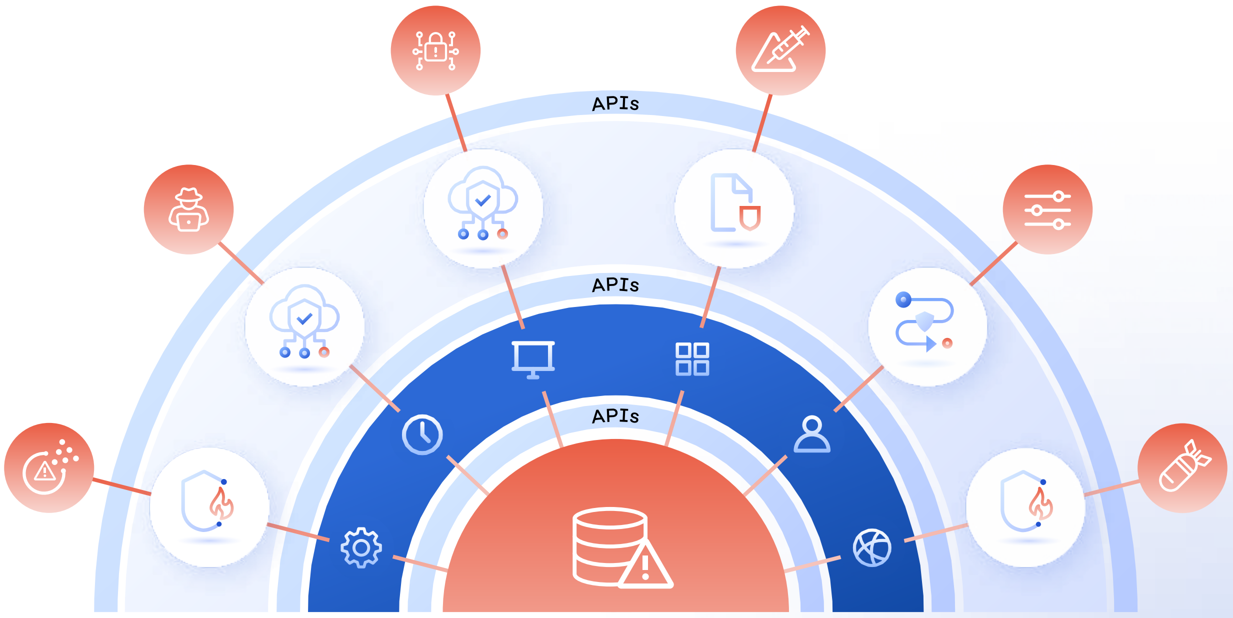

Universal Attack Vector before API use

The Universal Attack Vector

APIs have created a Universal Attack Vector.

The type of attacks that used to occur at the browser, the application layer, at the network layer, at the infrastructure layer have started to all converge at the API layer. You can do a DDoS attack, a bot attack, an injection attack, a remote code execution, all via an API.

Every type of attack that developed in the lower tiers of virtualization (OS, server, network, device, database, etc) can now be done at Layer 7. DDOS attacks couldn't be leveraged to attack a database - one being a network attack surface and the other a data layer attack surface. But, APIs can be used as a conduit for DDOS-type attacks, data exfiltration, account takeovers, fraudulent account creation, token theft attacks, and more.



API's are now the UniversalAttack Vector.
Every Attack Vector and Method is in one place

Leveraging the NIST Zero Trust Architecture

Why NIST Framework?

For the creation of the first Zero Trust reference architecture for APIs, we looked at many different architectures/frameworks for Zero Trust. While some of those ideas are synthesized into our ZT API Access architecture, the majority of concepts and alignment is to the NIST Zero Trust Architecture (documented in NIST Special Publication 800-207)

We have leveraged mostly off of NIST ZT Architecture because it is the reference architecture that is vendor-neutral, publicly available, and in widespread adoption by Government entities such as CISA, DOD, DISA, NSA, and GSA, NCCoE, as well as leading cybersecurity vendors.

NIST ZT Conceptual Model

NIST ZT Conceptual Model

Figure 2. NIST Conceptual Design of Zero Trust Environment

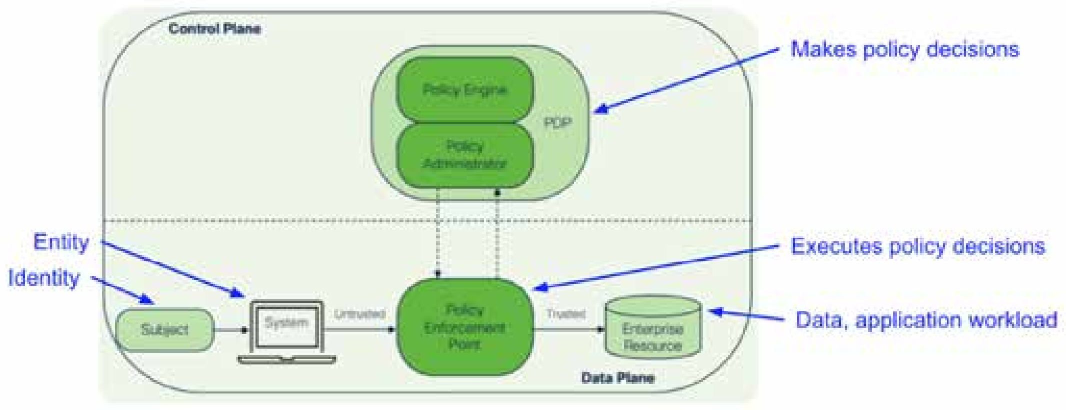


Figure credit: Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly, "Zero Trust Architecture," NIST Special Publication 800-207, August 2020.

Key tenets of NIST Zero Trust translated for APIs

NIST identifies that for Zero Trust, it is focused on Enterprise access between Enterprise controlled assets, and not anonymous, cloud-based access, as those can not necessarily be locked down or verified. However, NIST does also acknowledge the ability to potentially apply these tenets to non-Enterprise assets assuming that there is a relationship that allows implementation of ZT policies on the non-Enterprise owned assets.

"These tenets apply to work done within an organization or in collaboration with one or more partner organizations and not to the anonymous public or consumer-facing business processes. An organization cannot impose internal policies on external actors (e.g., customers or general internet users) but may be able to implement some ZT-based policies on nonenterprise users who have a special relationship with the organization (e.g. registered customers, employee dependents, etc.)."
Following are the key Zero Trust tenets proposed by NIST and how they apply when looking at ZT for APIs.

Key Tenet	NIST ZT Definition	Meaning in ZT for APIs
1. Consider every data source and computing device as a resource.	"All data sources and computing services should be considered as valuable resources" and therefore protected as such.	The majority of communications between clients, services, storage, and 3rd parties, whether enterprise owned or not, are using APIs and therefore should have all communications secured via ZT principals. Relevant for all connections including North/South connections, East/West (internal) connections, 3rd party, and how they all connect to each other.
2. Keep all communication secured regardless of network location	All communication should be secured regardless of network location or perceived safety of the location. This applies to whether the communication is external or internal.	For APIs this means that internal and external APIs should all be treated as external APIs (ie. it doesn't matter where on the network they are coming from or going to). There should not be a concept of an "internal" API that doesn't need to be secured as diligently. All APIs should be authenticated, authorized, and encrypted.
3. Grant resource access on a per-session basis	"Trust in the requester is evaluated before the access is granted. . . . authentication and authorization to one resource will not automatically grant access to a different resource." (NIST means network session here)	API and application sessions are better aligned with the business needs and user behavior than transport/network sessions. Thus, for APIs, this means that every session should require and persist authN and authZ. The AuthN could be through session cookies, tokens such as JWT, etc.

4. Moderate access with a dynamic policy	"includes the observable state of client identity, application/ service, and the requesting asset—and may include other behavioral and environmental attributes." (what NIST refers to here as "requesting asset" we'll refer to as an entity.)	Continuous visibility in real-time is required to secure any communications channel and therefore visibility to API behavior and dynamic API policy controls are required. For relevance with APIs for each potential criterion that goes into the dynamic policy, see the "Dynamic Policy Inputs for API Protection" section below.
5. No asset is inherently trusted	"The enterprise evaluates the security posture of the asset when evaluating a resource request. . . Assets that are discovered to be subverted, have known vulnerabilities, and/or are not managed by the enterprise may be treated differently" (what NIST refers to here as "requesting asset" we'll refer to as an entity.)	For APIs, a policy decision for whether a request should go through should also include dynamic information about the requesting system (asset / entity). Such information can help make a better determination if the request is malicious or not, by considering such inputs as the overall security position of the requesting entity (is it known to be compromised or full of vulnerabilities?), the type of system it is (is it a bot or known anonymous proxy?), the IP reputation (is it using an IP known to be used for malicious activity?), etc.
6. Rigorously enforce authentication and authorization	"All resource authentication and authorization are dynamic and are strictly enforced before access is allowed. This should be done in a **constant cycle of** obtaining access, scanning and assessing threats, adapting, and continually **re-evaluating trust in ongoing communication**."	For APIs, this means that all API requests should be properly authenticated and authorized before an API call is processed. This also means checking every API endpoint and call for authentication, encryption, authorization, and exploits of vulnerabilities, including reevaluating access rights within a user session.
7. Gather data for improved security	"Collect as much information as possible about the current state of assets, network infrastructure, and communications and use it to improve security posture."	The majority of communications between clients, services, storage, and 3rd parties, whether enterprise owned or not, are using APIs and therefore should have all communications secured via ZT principals. Relevant for all connections including North/South connections, East/West (internal) connections, 3rd party, and how they all connect to each other.

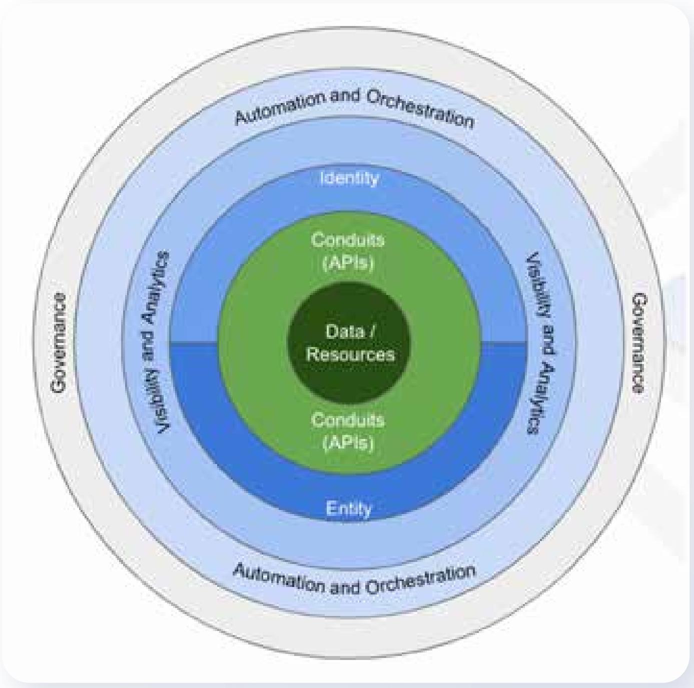
Pillars of Zero Trust for APIs

Zero Trust API Access pillars are defined in alignment with Zero Trust architectures derived from the NIST ZT framework, such as CISA, GSA, and DOD. However, there are differences based on adaptation to the unique properties and requirements of APIs.

Data / Resources

Data and resources are a primary focus of protection for any Zero Trust efforts. With APIs, protecting these is even more important. APIs punch through other layers of technologies in the stack and their associated protections, and land the user right at the data or resource they want access to. But not all data and resources are equal. It is important to know the sensitivity level of different data sets, as well as the value, cost and impact of resources, should they be compromised.

Make sure to inventory your data and identify which is sensitive, and to what level. Make sure that if you have resources that are costly or have some other high impact on your company that your business logic flows around them including safety nets such as intelligent rate limiting. Make sure that all transactions that lead to your sensitive data or costly resources are properly authenticated, authorized, and encrypted.



Pillars of Zero Trust for APIs Diagram

Conduits (APIs)

APIs are communication critical infrastructure for applications. When Zero Trust says “secure the application”, APIs are an often overlooked yet critical piece of doing that. But they are NOT the network itself, nor subject to the same restrictions, barriers, or protection as the network. These APIs are like the telephone or cable lines shuttling the Internet to you, hanging above, or buried under, the roadways (ie. networks). A roadblock wouldn’t stop the Internet from getting to you. Conduit integrity needs to be separately analyzed, understood and managed. This also means being able to collect data and take action in the locations where the conduits run.

Make sure you know where all of your exposed APIs are. Make sure you can continually track and validate the security posture of all your APIs. Make sure you have a way to block the traffic of known bad or dangerous APIs. Test APIs before they get exposed to production so vulnerabilities can be proactively dealt with.

Identity

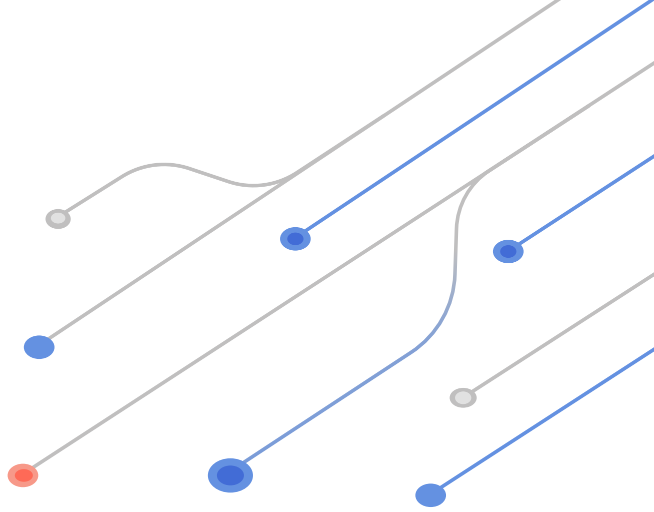
Securing API transactions requires knowing who is requesting resources. Identity (of either a human user or an NPE (non-person entity) typically comes from looking at authentication details. However, with APIs, subsequent requests don't necessarily include the same identity information, but instead a proxy for the authenticated identity, such as a bearer token. It is important to be able to track the identity of an API requestor across multiple API requests/responses to get a full understanding of if they are properly authenticated and what they are doing.

Make sure that your Zero Trust tools are able to authenticate and authorize identities, but also able to track those identities even if they aren't part of the authentication and authorization process.

Entity

An entity is a system, device, or server that a request is originating from (the source) or in the case of 3rd party APIs, that the request is going to (the target). NIST calls these "enterprise-owned assets" but for Zero Trust at the API level, we need to broaden the scope. This is because many API calls come in and go out to non-enterprise-controlled systems, such as 3rd party APIs. Regardless, knowing about the security posture of those calling or called systems is still important.

With entities, it is important to understand information about them such as where they are coming from, what type of client they are, what their overall security posture is, etc because this is context that provides more information than just a user ID for the policy engine to make a proper determination with. For example, a known user requesting banking information, but from an entity that registers as a bot with an IP address that has a critical risk rating, should probably not be sent the requested data.



Make sure that any entities you control are secure (OSs updated, apps patched, configurations checked, etc), and to the extent possible, that the entities you call also have good security posture. Set your ZTAA policies to check for unwanted client attributes in addition to just a userID or email address.

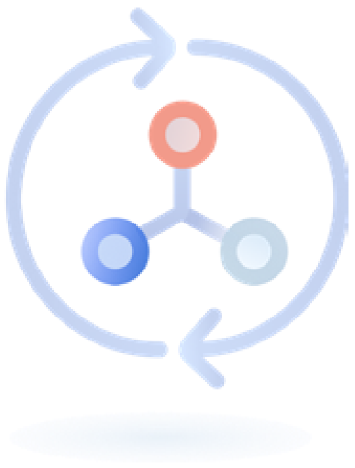
Visibility & Analytics

Visibility and analytics provide context, which is critical for successful security. This is especially true for APIs. For example, an API call to "checkout" might look normal without context, but not if I can see that it was called 100 times in a row without the user ever adding anything to their shopping cart." Better context comes from being able to see all the North/South AND East/West traffic so that an end-to-end picture of transactions can be analyzed. Along with this is being able to see within encrypted traffic to analyze potential malicious payloads, not just seeing the meta-data about the traffic.

Another example of the importance of visibility and analytics is being able to capture and analyze all traffic, good and bad because you can't track a user as thoroughly if you only see their activity occasionally.

This would be like trying to solve a crime but only being able to see the crime scene as quick flashes in 5-minute intervals. The context that visibility and analysis provide connects the dots.

Make sure the data collected for your policy decisions aren't just from one traffic ingress/egress point (unless it's a small app and that's truly the only place all APIs go through). Make sure that you are able to see API traffic between your services too. The deeper the data you can see, and the wider the window of time you can analyze the data, the more intelligent decisions can be made by the policy engine.



Automation & Orchestration

Automation and orchestration are important for any security tool given the speed and sophistication with which attacks can happen over large numbers of disparate systems. This is especially true for APIs which are typically used to glue together services at scale across the network. Manual and uncoordinated responses can no longer keep up. Automation and orchestration should happen within your API security systems, and also to external systems, such as SIEM and SOAR for more widely coordinated responses.

Automation enables action to be efficiently and quickly taken on unacceptable API calls. One example of this is the policies that enable the Zero Trust components (PDP, PEP) to make proper decisions and take proper action, given an undesirable API transaction has been requested or is happening. Some systems have built-in policies that tell them how and when to take action. Others enable the users to configure policies to match their automation and orchestration needs. Either way, an effective Zero Trust API Access implementation should be able to automate its responses to unacceptable API traffic.

Most application security tools used today do not have adequate API data collection or intelligence built into them. This is where orchestration can add value. By sharing API-centric data with SIEM and SOAR systems the whole security system can become more aware of issues found at the API level. Additionally, by integrating and orchestrating with control points (PEPs) that are not API aware (like WAFs), API intelligence can effectively be added to those control points.

Use automation & orchestration to execute your manual security process to make sure that policy-based API security actions are executed across your disparate application components with speed and at scale.

ZT Requirements for APIs

Data / Resources (In Transit, Governance)	1. **Data identification and classification:** Automatic identification and classification of sensitive data and data sets (such as HIPAA, GDPR, and PCI-DSS). Identifies sensitivity level of data types and sets. Customizable. Enables API risk assessments and sensitive data awareness in policies.
Conduits (APIs) (API Integrity)	2. **Keep an updated inventory of all APIs:** You can not protect what you don't know about. Since APIs are invisible, yet easy to add, API sprawl happens easily. An unknown API endpoint is a risky API endpoint. It is critical to keep a constant update on what APIs you have exposed, and how much risk each API endpoint presents. Enables understanding and management of attack surfaces. 3. **Detect & block high-risk APIs:** APIs are application-critical infrastructures that manipulate the data and make requests for resources. Therefore the security and integrity of the APIs themselves are important to keep managed. A ZTAA solution should be able to find and track all APIs, scan them for vulnerabilities that affect proper authentication and authorization, and block any attacks on such vulnerabilities. 4. **Flexible PEP + DCP (data collection point) distribution:** A protection mesh - Cloud-native and API-based apps are often highly distributed with varied architectures and varied organizational friction points. This requires flexibly distributed PEP's and data collection points to support many different deployment scenarios (eg. agent-gateway, enclave-based, resource portal, sandboxing).management of attack surfaces.
Identity (User, Service)	5. **In-depth user authN/authZ tracking:** Auto recognizes all authentication and authorization types and tracks users across transactions, IP changes, token/session changes, and across time. Identifies token issues. Critical for using identities to validate access rights.
Entity (Source, Target)	6. **Entity risk assessment:** Evaluate risk attributes of source and/or target entities in policy decision-making. Examples of relevant attributes include IP Type, IP/IP Reputation, geo-location, ASN, connection type, user agent, region, and security posture (eg. EDR). Enables multi-dimensional and contextual decision-making.

Visibility and Analytics (Coverage, Capture, Correlate)	7. **Complete transaction visibility (N/S, E/W, TLS):** North/South, East/West API call sequences AND how they all connect. This includes being able to see inside encrypted TLS transactions, preferably without requiring an added decryption termination point, which adds friction to implementation. Enables full coverage and visibility of all communication channels. 8. **Gather all API transactional data for full context:** Monitoring and recording all API transactions no matter where they connect from or to, whether thought to be good or bad, and storing them, over time, bringing full context for every API endpoint. Enables more context and more intelligent decision-making by the PDP when processing policies.
Automation & Orchestration (Coverage, Capture, Correlate)	9. **Data & resource access policy framework:** Granular data access policies based on sensitive data classification, granular source-based attributes, and requesting identities or domains. Enables precise & manageable policy definitions to protect data and resources. 10. **Security posture contributor and consumer:** Share API-level security posture assessments to external XDR, SIEM, SOAR systems. Enables the addition of API-specific signals to higher level Zero Trust visibility points for better automation and orchestration. Consume external posture signals in policy decision-making. Enables better API-level policy decisions.

ZT for APIs Core Components

NIST ZT Conceptual Model

Figure 2. NIST Conceptual Design of Zero Trust Environment

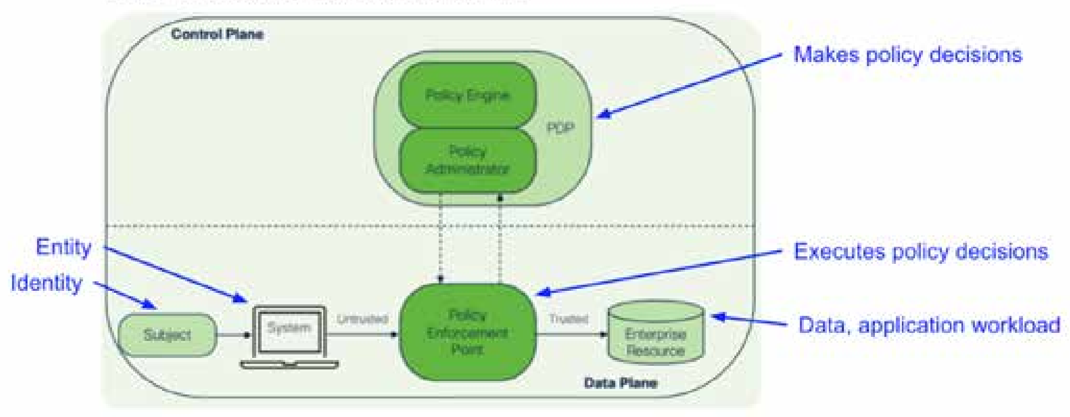


Figure credit: Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly, "Zero Trust Architecture," NIST Special Publication 800-207, August 2020.

Policy Decision Point (PDP)

Core Components	Meaning at API Layer
Policy Decision Point (PDP) Consists of the Policy engine (PE), responsible for making and logging the decision to grant access to a resource for a given subject, and the Policy administrator (PA), responsible for establishing and/or shutting down the communication path between a subject and a resource, via communicating to the PEP. PDP is in the control plane.	For APIs, the PDP uses dynamic policy data to determine if an API call should be allowed to go through and/or if a response is allowed to happen. Because APIs are consumed and can be abused in far more complex ways than traditional non-API communications, the policy should use dynamic data which includes a deep understanding of each application's business logic and its typical behavior. **Key to maintaining ZT at the API layer is the policies that are set and how rich the data is with which they determine decisions.**

Policy Enforcement Point (PEP)

Core Components	Meaning at API Layer
Policy enforcement point (PEP) This system is responsible for enabling, monitoring, and eventually terminating connections between a subject and an enterprise resource. . . This is a single logical component in ZTA which can be a single portal component that acts as a gatekeeper for communication paths. Beyond the PEP is the trust zone.	For APIs, any in-line system (of the communication path in the data plane) which handles the API traffic, and which can alter the transmission of that traffic (eg. API gateway, WAF, proxy, load balancer, mesh, an in-language agent, etc). **For APIs, in a resource-based deployment, the Trust Zone is as small as possible, allowing only the running of the approved API call. This is the ideal smallest practical trust zone you can have.**

API Context

When we say context or context-sensitive when talking about APIs and ZT-associated policy decisions, we are talking about the metadata and data of each call. The more data a system has about each API call, the better it will be at identifying patterns and indicators of malicious activity. This data might include information such as changes, ownership, risk assessments, behaviors, attributes, integrity, metrics, and classifications.

For example, knowing that a particular API endpoint, such as checkout, is not usually called without the user first going to their cart and adding something, and that when the checkout API is called, it's usually done only once. The context-sensitive policy and Trust Algorithm that is processing it can use this baseline to identify an untrustworthy event, such as a sudden barrage of individual "checkout" calls.

API
External & Internal
Mirror, Edge, In-app, Serverless
3rd Party & Partner

Trust Algorithm and Context

The trust algorithm in ZT is the process by which policies and policy inputs are calculated into a pass/block decision. The output of which is passed by the PDP to the PEP for enforcement. The Trust Algorithm (TA) therefore plays a critical role in determining the effectiveness of your ZT implementation, followed only by your policies and the inputs provided to them.

“[In a] ZTA deployment, the policy engine [(PE, part of the PDP)] can be thought of as the brain and the PE’s trust algorithm as its primary thought process. The trust algorithm (TA) is the process used by the policy engine to ultimately grant or deny access to a resource.“ (NIST Special Publication 800-207 Zero Trust Architecture - Rose, Borchert, Mitchell, Connelly - 2020 - Section 3.3, pg 17)

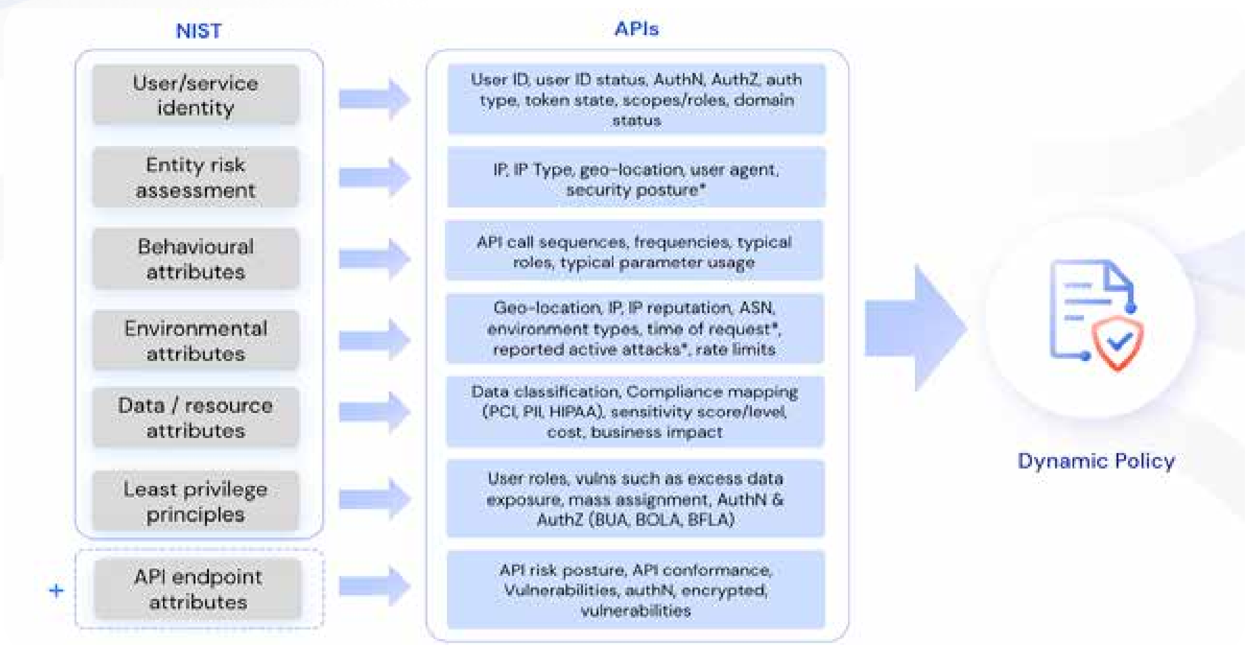
A more sophisticated TA will incorporate context into its processing, as described by this passage from one of the key books on ZT, “Zero Trust Security”:

“**Singular versus contextual**: A singular TA treats each request individually and does not take the subject’s history into consideration when making its evaluation. This can allow faster evaluations, but there is a risk that an attack can go undetected if it remains within a subject’s allowed role. A **contextual** TA takes the subject or network agent’s recent history into consideration when evaluating access requests. This means the **PE must maintain some state information on all subjects and applications** but may be more likely to detect an attacker using subverted credentials to access information in a pattern that is atypical of what the PE sees for the given subject. This also means that the PE **must be informed of user behavior by the PA (and PEPs) that subjects interact with when communicating**. Analysis of subject behavior can be used to provide a model of acceptable use, and deviations from this behavior could trigger additional authentication checks or resource request denials.” (NIST Special Publication 800-207 Zero Trust Architecture - Rose, Borchert, Mitchell, Connelly - 2020 - Section 3.3.1, pg 20)

“Ideally, a ZTA trust algorithm should be contextual” (NIST Special Publication 800-207 Zero Trust Architecture - Rose, Borchert, Mitchell, Connelly - 2020 - Section 3.3.1, pg 19)

Because APIs in combinations make up the business logic, out in the open, they invite manipulation and therefore need to be secured differently than singular network or web calls. This means that the context, of previous calls, of sequences, of the user, is a key input to making proper security policy decisions for APIs.

Dynamic Policy Inputs for API Protection


NIST
APIs
User/service identity
Entity risk assessment
Behavioural attributes
Environmental attributes
Data / resource attributes
Least privilege principles
+
API endpoint attributes
User ID, user ID status, AuthN, AuthZ, auth type, token state, scopes/roles, domain status
IP, IP Type, geo-location, user agent, security posture*
API call sequences, frequencies, typical roles, typical parameter usage
Geo-location, IP, IP reputation, ASN, environment types, time of request*, reported active attacks*, rate limits
Data classification, Compliance mapping (PCI, PII, HIPAA), sensitivity score/level, cost, business impact
User roles, vulns such as excess data exposure, mass assignment, AuthN & AuthZ (BUA, BOLA, BFLA)
API risk posture, API conformance, Vulnerabilities, authN, encrypted, vulnerabilities
Dynamic Policy

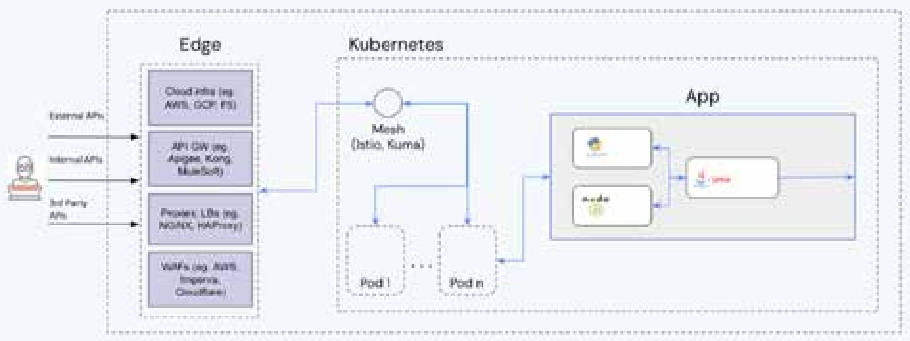
Policy Input	NIST	APIs
User/Service Identity	NIST's intent Includes the user account (or service identity) and any associated attributes assigned by the enterprise to that account or artifacts to authenticate automated tasks.	For APIs, user identity comes from the authenticated user of the API session, or their IP address (as a fallback) if they are not authenticated. Additionally, the policy should consider the status of a particular ID such as whether it is marked as stolen, or from a compromised domain. Authentication type should be looked at for known weak authentication, which might affect the level of access given. This extends to the token state if a token-based authentication/authorization is used, such as if it is expired or from a different source. Additionally, the scope/roles assigned to the ID should be taken into consideration looking to make sure that they match defined parameters. For APIs, the ability to detect and stop business logic attacks, in addition to authentication and authorization attacks (such as Broken User Authentication, Broken Object Level Authorization, and Broken Function Level Authorization) relies on the ability to track user activity over time and transactions so this input is even more important for ZT at the API level.
Asset Risk Assessment	NIST's intent here is for understanding the state of the system/client initiating the connection. This includes device characteristics such as software versions installed, network location, time/date of request, previously observed behavior, and installed credentials.	For APIs, the notion of asset state is about the system they are making the request from such as the reputation of the software they are using to make the API calls (eg. user agent and browser version), where the asset is making the request from, such as known sanctioned countries, and that the security posture of the requesting asset is, for example, what an endpoint protection tool such as SentinelOne or Eset might determine.

Behavioral Attributes	NIST's intent here includes but is not limited to, automated subject analytics, device analytics, and measured deviations from observed usage patterns.	For APIs, behavioral attributes should focus less on the device and more on the user and API usage patterns. Example patterns to consider include typical API call sequence patterns, frequency of API endpoint usage, typical roles of the users who use the endpoint, and typical parameter usage patterns.
Environmental/ Network Attributes	NIST's intent includes factors such as requestor network location, time, and reported active attacks.	For APIs, environmental and network attributes that should be taken into consideration include information such as the geo-location of the calling asset, its IP address, and its IP reputation, and even the ASN it's coming from. It's also valuable to look at what environment the request is going to (eg. an API request to a production environment might be considered higher risks than to development environments), and the time of the requests (eg. an application might only expect traffic during an event). Another environmental factor that can be valuable to include in ZT policies is whether there are active attacks in the environment being called, or doing the calling. And finally, but not least, it is recommended that implicit rate limits be configured on the called APIs.
Data Attributes	NIST's intent is that for highly sensitive data more restricted access to that data might be imposed than for non-sensitive data.	For APIs, in order to make secure policy decisions about an API call that accesses data, it is important to know what type of data it is (eg. is it a name, address, credit card number, social security number, etc), whether the data maps to any data sets that have compliance requirements (such as PCI, HIPAA, GDPR, etc) and what the sensitivity score/level of the data is.
Least Privilege Principles	NIST's intent is that only the minimal required permissions are given to a user such that they are restricted to only necessary visibility and accessibility.	For APIs, this means that once a requestor is authenticated AND authorized, they can only access, provide, or update the data which they requested and that they are authorized to get/change. Ideally, the ZT for API PDP/PEP combo can detect AND block the vulnerabilities that lead to violation of least privilege principles, such as mass assignment, excessive data exposure, basic user authentication errors, broken object-level authorization, and broken function-level authorization.
API Endpoint Attributes	NIST does NOT identify this as a policy consideration for ZT. Traditional ZT touches on applications (layer 7) but does not dive deeper into how to implement ZT **within** the applications.	In ZT for APIs, the APIs are a combination of the application workload and the access pipes themselves. For the PDP to make smart zero trust policy decisions for APIs, it is important to understand the risk posture of those "pipes" (APIs) such as if they are unencrypted, external, and not authenticated. It's also important to understand if the API and its usage conforms with documented expectations (eg. that the called API is known and called correctly). Finally, and importantly, just as for ZTNA it is important to understand the patch state of the firmware on your network gear, it's important to know what vulnerabilities the called API has.

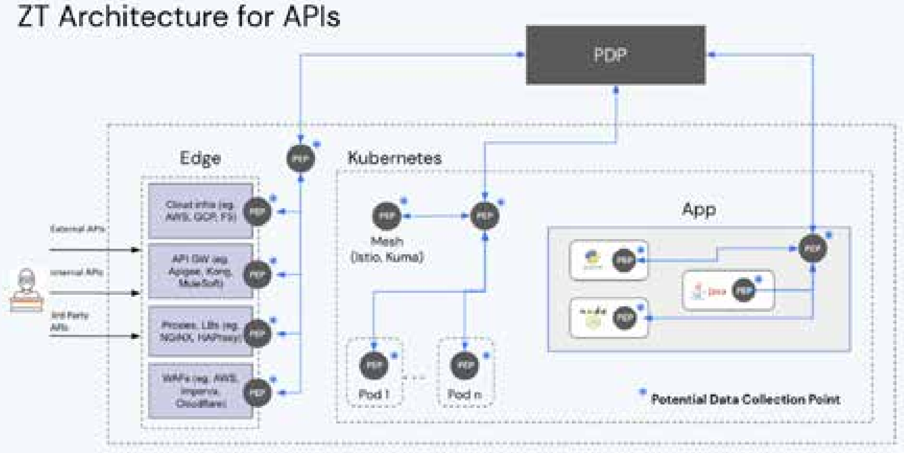
ZT for API Deployment Models

Cloud-native application architectures are by nature distributed, with services oftentimes spread across multiple clouds and potential control points. This requires not only a centralized PDP, which Zero Trust architecture calls for, but more importantly widely distributed PEPs and data collection points.

Cloud-Native Architecture


Edge
Kubernetes
App
External APIs
Internal APIs
3rd Party APIs
Cloud infra (eg. AWS, GCP, F5)
API GW (eg. Apigee, Kong, MuleSoft)
Proxies, LBs (eg. NGINX, HAProxy)
WAFs (eg. AWS, Imperva, Cloudflare)
Mesh (Istio, Kuma)
Pod 1
Pod n
java

ZT Architecture for APIs


PDP
Edge
Kubernetes
App
PEP
External APIs
Internal APIs
3rd Party APIs
Cloud infra (eg. AWS, GCP, F5)
API GW (eg. Apigee, Kong, MuleSoft)
Proxies, LBs (eg. NGINX, HAProxy)
WAFs (eg. AWS, Imperva, Cloudflare)
Mesh (Istio, Kuma)
Pod 1
Pod n
java
Potential Data Collection Point

TRACEABLE

About Traceable

Traceable is the industry's leading API Security company that helps organizations achieve API protection in a cloud-first, API-driven world. With an API Data Lake at the core of the platform, Traceable is the only intelligent and context-aware solution that powers complete API security – security posture management, threat protection and threat management across the entire Software Development Lifecycle – enabling organizations to minimize risk and maximize the value that APIs bring to their customers. To learn more about how API security can help your business, book a demo with a security expert.