Zero Trust for the Microsoft identity platform developer

Background

Application developers play a key role in organizational security. Gone are the days when an application and its developers can assume that the network perimeter is secure. With work originating from anywhere on any device, applications need to be developed in a way that incorporates Zero Trust principles throughout the development cycle. A compromised application today can have an impact on the entire organization. Developing apps that incorporate the Zero Trust framework will increase security, reduce the blast radius of a security incident and help recover swiftly.

This whitepaper first covers the Zero Trust model and how it impacts the work developers do. Then, it provides Zero Trust best practices that developers can follow when developing with the Microsoft identity platform.

The Zero Trust model

The Zero Trust model is a reevaluation of security practices and processes from a perspective where a **culture of implicit trust must give way to that of explicit verification**. The key principles of Zero Trust are explained below, along with how they apply to the work developers do:

Verify Explicitly

Always authenticate and authorize based on all available data points, including user, application and device identity, location, device health, service or workload, data classification, and anomalies.

Use least privileged access

Limit user, application, and device access with granular permissions, role assignments, just-in-time and just-enough-access (JIT/JEA), risk-based adaptive polices, and data protection to help secure both data and productivity.

Assume Breach

Help minimize the blast radius for breaches and prevent lateral movement. Practice good credential hygiene that enables swift replacement and rotation of credentials. Keep sessions, network access, and data encrypted end to end. Build and test durable and repeatable backup and restore processes. Generate telemetry and analytics to quickly get visibility, drive threat detection, and improve defenses. Avoid insecure legacy authentication and authorization protocols.

Developers have an important role in Zero Trust

In the past, code security was all about your own app – if you got it wrong, your own app was at risk. Cloud security is about your company's infrastructure. The infrastructure is only as secure as the weakest link. A single app can potentially be the weakest link that malicious actors can hijack to gain access to additional business-critical data and operations.

Zero Trust is not a replacement for security fundamentals. Security still must be kept in mind throughout the development lifecycle. But adding a Zero Trust perspective is helpful for two reasons. First, we've seen the level of sophistication continue to rise for attacks. Second, the "work from anywhere"

workforce has redefined the security perimeter. Data is being accessed outside the corporate network and shared with external collaborators such as partners and vendors. Corporate applications and data are moving from on-prem to hybrid and cloud environments. Traditional network controls can no longer be relied on for security. Controls need to move to where the data is: on devices and inside apps.

Developers and IT pros must work together

IT organization are increasingly blocking apps with vulnerabilities. As IT departments embrace a Zero Trust approach, developers who do not create their applications to abide by Zero Trust principles risk not having their apps adopted. Following Zero Trust principles can help you ensure your application is eligible for adoption in a Zero Trust world.

App developers will usually implement, evaluate, and validate aspects of Zero Trust before working with the organization's IT pros to achieve full compliance and adherence. Developers are responsible for building and integrating apps so IT pros can use tools at their disposal to carry out steps to further secure the applications. Partnering with IT Pros can help to

- minimize the probability of, if not prevent, compromises from happening at all.
- quickly respond to attacks and compromises to reduce damage to their business.

Customers are adopting Zero Trust principles, starting with identity

Zero Trust implementation is still evolving. While each organization's journey is unique, the logical place to start for most is user and application identity. The following are some of the policies and controls we see organizations prioritizing as they roll out Zero Trust:

1) **Credential hygiene and rotation policies**. Secrets such as certificates or passwords are one of the most important assets to secure because they allow an attacker to move deeper within the system.
2) **Strong authentication**. IT administrators expect to be able to set polices requiring multi-factor authentication.
3) **Restricting consent to apps with low-risk permissions that are publisher verified**. While access to data in APIs like Microsoft Graph allows for building rich applications, your organization or your customer will evaluate permissions your app is requesting and the trustworthiness of your app.
4) **Blocking legacy protocols and APIs**. This includes blocking older authentication protocols such as "Basic authentication" and requiring modern protocols like Open ID Connect and OAuth2. Organizations are ensuring applications they depend on are prepared.

Azure AD B2C

All of the guidance provided in this document applies to applications that use Azure AD B2C for their Identity and Access needs as well. In addition, applications using Azure AD B2C and those subscribed to Microsoft Dynamics 365 Fraud Protection with Azure Active Directory B2C can use it to assess if attempts to create new accounts and attempts to login to client's ecosystem are fraudulent.

Zero Trust best practices with the Microsoft identity platform

The Microsoft identity platform is designed to simplify authentication and authorization for developers. It is continually updated with the latest security best practices and abstracts much of the work away from the code a developer must write.

However, there are actions developers can take in using the platform that will affect app security. This section will help developers as they create apps to ensure they take advantage of security features and best practices.

Note: Occasionally, pieces of guidance only apply to a subset of apps (for example, web apps). If so, the guidance will call this out explicitly. The definitions of app types is included in the references section at the end of this document.

Authentication

Authentication is about verifying the identity of the person or service accessing a resource. The best practices here often are in support of the Zero Trust principle of **verify explicitly**.

Use modern protocols, and consider enhanced security extensions

Use modern protocols, especially OpenID Connect and OAuth2 for single sign-on in your applications. Modern protocols remove the need for applications to ever directly handle a user's credentials. This vastly improves the security for both users and applications as the identity provider becomes the security perimeter. Also, these protocols continuously evolve to address new paradigms, opportunities, and challenges in identity security.

The Open ID Connect protocol's constructs allows identity providers like Azure AD to build extensions to enhance security. In Azure AD, some of the most used extensions include the following.

- Conditional Access (CA) is the tool used by Azure Active Directory to bring signals together, make access decisions, and enforce organizational policies. These policies ensure that a user meets specific criteria to access an application. Criteria can include things like requiring a managed device, accessing from a specific location (or blocking a specific location), and specific attributes like group membership. Conditional Access can redirect the user back to the identity provider for multi-factor authentication, or to meet requirements such as a password change.
- Conditional Access authentication context allows apps to apply granular policies to protect sensitive data and actions, instead of just at the app level.
- Continuous Access Evaluation (CAE) enables Azure AD applications to subscribe to critical events that can then be evaluated and enforced. This includes evaluation of the events like the user account being disabled or deleted, password changes, token revocations, or a user detected at being risk.

Applications that use enhanced security features like CAE and Conditional Access authentication context must be coded to handle claims challenges. Open protocols enable you to use the claims challenges and claims requests to invoke additional client capabilities. This might be to indicate to apps that they need to re-interact with Azure AD, like in case of an anomaly or if the user no longer satisfy the conditions under which they had authenticated earlier. Developers can code for these extensions without disturbing their primary code flows for authentication.

Use industry-standard SDKs and libraries

Use industry-standard SDKs to implement authentication in your applications instead of programming the protocol yourself. The Microsoft Authentication Libraries (MSAL) is a set of Microsoft's authentication libraries for developers. MSAL uses the OpenID Connect and OAuth2 protocols. You can also use a Certified OpenID Connect Implementations to authenticate your apps with the Microsoft identity platform.

With MSAL, developers can authenticate users and applications, as well as acquire tokens to access corporate resources using just a few lines of code. In addition, it provides additional security, resiliency, performance, and better end user experiences. Most importantly, MSAL is regularly updated to address newer features like passwordless, Individual Claims Requests and more.

Use the correct authentication flow by app type for authentication

Different application types can use different authentication flows.

Web applications should always aim to use the confidential client flows. The flows for web applications that can hold a secret (confidential clients) are considered more secure than Public clients (desktop, console and daemons. You should avoid implicit grant flow and use PKCE instead. Don't use the Resource Owner Password Credentials (username/password flow). This flow requires a very high degree of trust in the application and carries risks which are not present in other flows. You should only use this flow when other more secure flows can't be used.

Mobile applications should use brokers or the system browser for authentication. Use brokers to conduct authentication when available on the device. Using MSAL will automatically use the Microsoft Authenticator app, if present. If using the MSAL library is not an option, use the system web browser for a secure SSO experience that support app protection policies. Build secure and manageable mobile app experiences by following the guidance in Support single sign-on and app protection policies in mobile apps .

Make your authentication implementation resilient.

Rarely, a call to retrieve a token could fail due to an issue like network or infrastructure failure or authentication service outage. We have published guidance for making your authentication experience resilient, outlining steps a developer can take to increase resilience in their applications if a token acquisition failure occurs. The best practices include:

- Locally cache and secure tokens with encryption.
- Do not pass security artifacts like tokens around on non-secure channels.
- Understand and act on exceptions and service responses from the identity provider.

Authorization

Authorization involves granting permissions (scopes) and verifying that a caller has been granted proper permission before allowing access. You can enforce least privileged access in your apps by enabling fine-grained permissions that allow you to grant the smallest amount of access necessary.

General best practices

Validate tokens using an established standard token validation library. Don't implement the validation yourself as this is prone to errors and will need to be maintained and updated. Microsoft recommends the Microsoft.Identity.Web and Microsoft.IdentityModel.Tokens libraries for token validation.

Handle signing key rotations by identity providers. Applications using OpenID Connect and OAuth2 must be designed to handle this automatically. Standard token validation libraries will enable this for you.

Limit access to your application with application settings in the tenant. Use the "User assignment Required" flag to ensure that only a set of users are able to sign-in to the application. Without this flag, all users in a tenant are eligible to access the application. By default, guest users in a tenant can also sign-in to your app. If you want to block guest users, opt-in to the optional claim "acct". If it is `1`, that means the user is classified as a guest. Block tokens with acct==1 if you want to block guests.

Publish app roles in your apps and drive authorization using role assignments, especially for administrative and high-privilege functions. Consider Azure AD's Privileged Identity Management (PIM) with app roles, which provides users just-in-time and time-bound access to sensitive roles like administrators, thus reducing the chance of a malicious actor getting that access, or an unauthorized user inadvertently impacting a sensitive resource. Azure AD also supports sending security groups that a user is assigned as claims in a token. This can be used to drive authorization in certain line of business (LoB) apps. For flexibility and control, we recommend apps publish app roles and assign security groups to app roles instead.

Web APIs: Use OAuth2 Access Tokens for authorization. Web APIs validate bearer tokens to authorize callers. Do not accept ID tokens as a proof of permission.

Designing your own validation logic

If you implement token validation logic, ensure that it includes the following:

- Check the token is for your application by matching the audience (**aud**). Accept only tokens for your API client Id or app ID URI. See aud claim.
- Do not accept tokens issued to popular APIs like *https://graph.microsoft.com* as proof of *authentication*.
- Authorize the callers (users, apps) based on the claims scopes (scp) and roles (roles) or both.
    - Scopes are assigned to users. These are also called Delegated permissions.
    - Roles are assigned to applications. These are also known as Application permissions.
- Optionally extend token validation to:
    - Use the tid claim to restrict the tenants in which a token can be obtained by your API.
    - Use the azp/appid claims to filter apps that are allowed to call your API.
    - Use the oid claim to further narrow down access to individual users.

Application management

The Microsoft identity platform app registration portal is the primary entry point for applications intending to use the platform for their authentication and associated needs. When registering and

configuring your apps, several choices you make will drive and affect how well your application can satisfy the principles of Zero Trust, especially when **assuming breach** and enabling **least privileged access**.

Best practices

Use Integration assistant to ensure your application follows the best practices.

Properly define your redirect URLS. Read the Redirect URI (reply URL) restrictions and limitations to avoid compatibility or security issues.

Check redirect URIs used in your app registration for ownership and to avoid domain takeovers. Redirect URLs should be of domains you know and own. Review and remove unnecessary and unused URIs on a regular basis. Do not use non https URIs in production apps.

Ensure app and service principal owners are always defined and maintained for your apps registered in your tenant. Avoid orphaned apps, i.e. apps and service principals that have no owners assigned. Locating app owners should not be a time-consuming process during an emergency. Similarly, keep the number of app owners small and make it harder for one compromised user account to affect multiple applications.

Avoid using the same app registration for multiple apps.

- Unless tightly coupled, apps that sign-in users and those that expose data and operations via API should have separate app registrations. This allows permissions, like those for Graph and any credentials (like secrets and certificates) for a higher privileged API at a distance from apps signing-in users and interacting with them. This helps with enabling both least privileged access and reducing impact during a breach.
- Prefer to use separate app registrations for web apps and APIs. This also helps ensure that if the web API has a higher set of permissions, then the client app does not inherit those.

Do not create your application as a multi-tenant app unless you really intended to. Multi-tenant apps can be provisioned in tenants outside yours, and thus require additional management overhead to filter unwanted access. Unless you intended to develop your app as a multi-tenant app, start with a **SignInAudience** value of **AzureADMyOrg**. If you need to integrate your application as a multi-tenant app, then you'd also additionally follow the steps provided in the section below.

Permissions and Consent

This section is about obtaining permissions to access protected APIs, whether those are popular APIs like Microsoft Graph or APIs owned by your own organization. The focus is largely on supplying **least privileged access** to ensure access is only granted when necessary.

Requesting permissions

Evaluate the permissions you request to ensure that you seek the absolute least privileged set to get the job done. Often, developers request higher privilege permissions to avoid the effort that is required to carefully work through the large number of permissions that APIs like Microsoft Graph provide. We

recommend developers spend time to locate and use just the right number of permissions to get their needs addressed. APIs often provide access to an organizations data stores and thus attract the attention of attackers who seek to get access to an organization's data. Thus, keeping apps that read and write data via APIs to the least privileged (i.e. smallest scoped) set of permissions cannot be emphasized enough.

Designing API permissions

APIs should provide granular permissions to allow least-privileged access. Start with dividing the functionality and data access into sections that can be controlled via scopes and App roles. It's usually a bad idea to publish just one app role (app permission) and one scope (delegated permission) by an API for its clients.

Do not create "catch-all" permissions with access to the entire API surface. If your API exposes a lot of operations, consider providing granular set of scopes to avoid exposing all operations to each client. Many API developers define just a single scope (for example *user_impersonation* or *access_as_user*) which is insufficient if your API surface is large.

Offer read-only permissions. "Write access" includes privileges for create, update, and delete operations. A client should never require write access to simply read data.

Offer both Delegated and Application permissions. Skipping application permissions can create hard requirement for your clients to achieve common scenarios like automation, microservices and more.

Create an individual permission for each core scenario. Refer to Microsoft Graph permissions reference for a good understanding of this.

Permission strings should clearly inform users and admins of the consequences of consent, while being written as concisely as possible. Assume that the individual reading your description strings has no familiarity with your APIs or product.

Do not add your APIs to existing permissions in a way that changes the semantics of the permission. Overloading existing permissions dilutes the reasoning upon which consents were granted for clients earlier.

Consider "standard" and "full" access permissions if working with PII or sensitive data. Restrict the sensitive properties so that they cannot be accessed using a standard e.g. *Resource.Read* permission. And then implement a "Full" access permission, e.g. Resource.ReadFull, that returns all available properties including sensitive information.

Additional resources

Microsoft Graph permissions reference is an excellent reference of how to declare permissions for a web API.

Best practices for least privileged access for applications lays out the how the various roles in an organization approach least privilege, including developers.

Follow the guidelines provided in Securing Azure environments with Azure Active Directory as how to create separate development resources in a non-production tenant. This is useful if the development team needs access to high-privileged permissions during development. Without a separate development

resource, a developer's compromised account might be used by attackers to access confidential data in the tenant.

Credential management

Credential hygiene is amongst the first steps to ensure that the application can quickly recover from a potential breach. These steps guide the application developers to integrate and deploy their applications in a manner where detection and remediation can be carried out while avoiding a downtime or affecting legitimate users. The steps support the Zero Trust principle of assume breach by preparing you to respond to a security incident.

Remove all secrets from code and configuration. Place them in Key vault and access them via a Managed Identity. This makes the code resilient to handle secret rotations if a compromise occurs. Also, IT admins can remove/rotate secrets and certificates without taking down the application or affecting legitimate users. Read up on Managed Identities for Azure resources. For developers using the Azure platform, Managed Identities should be the first choice wherever supported and applicable.

Use certificates instead of client secrets, unless a secure process is in place to manage secrets. Client secrets tend to be handled less securely, and attackers know this. Certificates are usually better managed and can be revoked if compromised. If secrets are to be used, build or use a secure no-touch deployment and rollover process for them. Use secrets with a set expiry time period (1 year, 2 years) and avoid *never expires*. A leaked secret usage is hard to track.

Roll over the certificates and secrets regularly. This builds resiliency in your application and avoids any outage due to an emergency rollover.

Telemetry & Logging

Telemetry and logging best practices support the Zero Trust principle of assume breach by making it easier to detect the occurrence of security incidents.

Log detailed telemetry from the authentication and authorization layers of your app. This information is very valuable in analyzing, detecting, and controlling comprised access or data loss in case of a breach. Tokens issued to users and applications will have claims like *oid*, *upn*, *tid*, *sub*, *preferred_username*, *aud*, *idp*/*iss*, *appid*/*azp* which should be logged into your logging and telemetry. For SaaS multi-tenant applications, it also provides a detailed view to the distribution of the multi-tenant apps across Azure AD tenants.

Additionally, also log actions specific to the app, device info (if available), environmental conditions (see optional claims like IP address (*ipaddr*))

SaaS applications

These best practices are useful if you are building an app that will be used by customers in their own tenants.

Consider integrating using the multi-tenant pattern if you are developing a SaaS application that will be provisioned into a number of customer tenants. Credentials remain with the SaaS developer (ISV), thus removing scope of abuse in a tenant. In addition, it provides unique identification of an app for a

Conditional Access (CA) policy, reporting, and establish Quality of Service (QoS). Finally, the app can be identified and isolated during attacks much more quickly.

Develop and implement the means to explicitly verify the tenants that the users of your app are coming from. You can further make this process granular by being able to authorize or deny access for individual user accounts. Multi-tenant apps can be provisioned in any Azure AD tenant and thus an absence of logic to filter unwanted users can potentially be abused by an attacker. Extending the default Token validation to manually validate an additional set of claims will enable granular authorization.

The **best practices** are an absolute must for multi-tenant applications. Multi-tenant apps get no visibility of their configuration in tenants other than where they are homed in and logging tenant and user identifiers from tokens presented is the way to keep track of all who are accessing the application on a regular basis.

Publish the Graph permissions that the customer is expected to consent to in detail if your SaaS application will read/write a customer data via an API like Microsoft Graph. This becomes even more important if your application will require consenting to high-privileged permissions.

Support custom logging. We encourage SaaS application developers to look into supporting Azure Monitor and Azure Sentinel compatible activity logging so that customers can utilize their existing dashboards and alerts for the SaaS applications as well.

Get the SaaS offerings publisher verified. When an application is marked as publisher verified, it means that the publisher has verified their identity using a Microsoft Partner Network account that has completed an established verification process. Publisher verification helps admins and end users understand the authenticity of applications. If you are in the business of building and selling applications to Microsoft's customers, it improves the trust and security posture of your applications for customers.

Check out the code samples dedicated to multi-tenant app development at the Microsoft identity platform code samples page.

Session management

Applications will usually create a session for a user once the authentication completes successfully with Azure AD. User session management drives how frequently a user will be sent back to Azure AD for re-authentication. Thus, its role in keeping an **explicitly verified** user in front of the app with the right privilege and for the right amount of time cannot be understated.

Do not aim for very long or very short session lifetimes. Let the granted token's lifetime drive this decision. Keeping an app's sessions active beyond a token validity will violate the rules, policies and concerns that drove an IT admin to set a token validity duration and can be abused for unauthorized access. Also note that very short sessions degrade user experience and do not necessarily increase the security posture.

Several popular frameworks like ASP.NET allow you to set session and cookie timeouts from Azure AD's ID token's expiry time. Following the identity provider's (for example Azure AD's) token expiry time will ensure that your user's sessions are never longer than what policies on the IdPs dictate.

Data management

Data management is a large topic and thus this guide would only touch the aspects of data management that is relevant to stay true to Zero Trust principles and identity best practices. Data management processes should ensure data storage and access is carried out in a secure, and controlled fashion and quick restoration and retrieval of data is possible during a breach.

Encrypt sensitive, personally identifiable data. Control access to data in production with just in time (JIT) access and practice data classification and labeling.

Practice credential hygiene for private keys used for access and encryption. Execute practice runs on how credential and keys rotation in case of a breach can keep the application running and keep the data safe, swiftly recoverable and available.

Backup and restore the app and its data regularly. Establish and test backup and restore strategy. Ensure data can be reverted to an earlier stage successfully.

Data kept behind APIs should be segregated to enable least privileged access. For example, segregate data access for users and applications like Microsoft Graph does. This helps to keep the data loss limited to the compromised user.

Isolate development resources from production. Non-production Azure resources and applications that may affect other resources. For example, a new beta version of an application may need to be isolated from the production instance of the application and production user objects. Follow the guidelines provided in Securing Azure environments with Azure Active Directory as how to separate development resources in a non-production tenant.

References and additional resources

Application type definitions

The Microsoft identity platform supports the following application types.

Web applications – These are usually websites that users sign in to and interact with using a browser. They run exclusively inside the browser's boundary and do not have access to the operating system's resources

Web APIs – APIs that expose operations and data and do not directly interact with users. They are almost exclusively called by other applications.

Desktop applications – Applications that are used on a desktop such as a Windows laptop. Examples are those built with frameworks like WPF and WinForms. They also sign-in and interact with users and also have access to the local resources like disk and other system components.

Mobile applications – Applications that are developed for mobile devices, like for iOS and Android. They also sign-in the user (usually the device owner) and have access to a limited set of the device's resources.

Daemon & Services - Apps that have long-running processes, bots or that operate without interaction with a user also need a way to access secured resources, such as web APIs.
NOTE: To discern the right application type when developing embedded apps, tabs, or plugins, evaluate the characteristics such as the ability to interact with a user and ability to access a device's resources to select the right application type.

Additional Resources

The Microsoft identity platform
Zero Trust Resource Center
Zero Trust Security Model and Framework | Microsoft Security
Microsoft identity platform best practices and recommendations
Azure Active Directory Data Security Considerations
Securing Azure environments with Azure Active Directory
Microsoft identity platform authentication libraries
Microsoft identity platform code samples
Developer guidance for Azure Active Directory Conditional Access
Developers' guide to Conditional Access authentication context
Increase resilience of authentication and authorization applications you develop