Process Documentation: Integrating Unity with a Post-Quantum Secure XR SDK to Build Your First XR Experience

Overview

This guide is designed to lead you **step by step through the entire process** of combining Unity with a **Post-Quantum Secure XR SDK.** The user will get acquainted with how to:

- Use and set up the SDK in Unity.
- Use **cross-platform APIs** for immersive XR.
- Use **post-quantum encryption** to protect multiplayer XR sessions.
- Simply deploy your XR build on **Meta Quest, HoloLens** as well as **iOS/Android.**

Extremely **secure XR environment of advanced development** and **enterprise security** will be your final output.

Prerequisites

Before starting, ensure you have the following:

- **Unity Hub** (2022.3 LTS or higher)
- **XR SDK (Meta Post-Quantum XR SDK v2.0)**
- **OpenXR Plugin** installed from Unity Package Manager
- **Visual Studio 2022 / JetBrains Rider** for C# scripting
- **Platform SDKs**:
    - Android SDK + NDK (for Meta Quest / ARCore)
    - Windows Mixed Reality SDK (for HoloLens)
    - iOS SDK (for ARKit)

- **GPU with Vulkan / DirectX12 support**
- Basic knowledge of:
    - Unity Scene Hierarchy
    - C# asynchronous programming (async/await)
    - Cryptography fundamentals

Warning: This guide assumes you are working with **sensitive enterprise or user data**. Post-quantum cryptography is CPU-intensive—optimize carefully to avoid frame drops in VR/AR.

Step 1: Setting up the Unity Project

1. Open **Unity Hub** → Create a new project → Choose **3D (URP)** for lightweight rendering.
2. Name the project:
3. XR_PQC_Secure_Demo
4. Configure **Player Settings**:
    - Under **XR Plug-in Management** → Enable **OpenXR**.
    - Add **Meta Quest Support**, **Windows Mixed Reality**, and **ARKit** in the build targets.

Step 2: Import the Post-Quantum XR SDK

1. Download the latest SDK from the vendor's portal:
2. pqc-xr-sdk-2.0.unitypackage
3. In Unity → **Assets → Import Package → Custom Package**.
4. Select the SDK and import all dependencies.

This SDK includes:

- **Quantum-Resistant Networking API (QNet)**
- **Secure Asset Pipeline (SAP)** for encrypted asset streaming

- **XR Hand Tracking Extensions** with secure biometric binding

---

Step 3: Configure Secure Networking

To enable **Post-Quantum TLS (PQTLS)** for multiplayer XR:

```
using PQCXR.Networking;
using UnityEngine;


public class SecureConnection : MonoBehaviour
{
   async void Start()
   {
      var client = new QNetClient();
      await client.ConnectAsync("xr.metaverse.secure", port: 443);
      Debug.Log("□ Connected with Post-Quantum TLS 1.3+");
   }
}
```

This establishes a **hybrid key exchange** using **Kyber-1024 + AES-GCM**, resistant to both **classical** and **quantum** attacks.

Tip: Benchmark latency with client.GetHandshakeLatency()—aim for <50ms in XR multiplayer environments.

Step 4: Building Your First XR Scene

1. Create a new Unity Scene → Name it SecureLobby.
2. Add:
    - **XR Rig** (from XR Interaction Toolkit).

- A **Secure Portal Object** (prefab provided by SDK).
- An **Encrypted Avatar Controller** with voice chat secured by **Dilithium signatures**.

3. Attach script for **Secure Hand Tracking**:

```
using PQCXR.HandTracking;


public class HandAuth : MonoBehaviour
{
   void OnEnable()
   {
      HandTracker.OnHandGesture += gesture =>
      {
         Debug.Log($"Secure gesture: {gesture}");
      };
   }
}
```

Step 5: Cross-Platform Deployment

Meta Quest (Android-based VR)

- Switch build target → **Android**
- Enable **ARM64** → Vulkan backend
- Build & Run

HoloLens (Mixed Reality)

- Switch build target → **UWP**
- Enable **DX12**

- Deploy via Visual Studio

iOS (ARKit-based XR)

- Switch build target → **iOS**
- Configure **Metal Rendering**
- Archive via Xcode → TestFlight

Performance Optimization

- Use **Job System + Burst Compiler** for PQC-heavy computations.
- Offload **cryptographic handshakes** to background threads via Task.Run().
- Reduce **polygon count & texture size**—encryption adds network overhead.

Tip: Use Unity Profiler → CryptoJob marker to identify PQC bottlenecks.

Security Checklist

- Post-Quantum TLS enabled
- Encrypted voice & avatar streams
- Secure biometric hand-tracking
- Signed assets (Dilithium + SHA3)
- No fallback to classical RSA/ECC

Warning: Never hardcode private keys in Unity scripts. Store credentials in **Secure Enclave (iOS)** or **KeyStore (Android)**.

Troubleshooting

Issue	Cause	Solution
High latency (>150ms)	PQTLS handshake overhead	Enable session resumption in QNetClient.
App crashes on Quest	Out of memory due to cryptographic threads	Reduce handshake concurrency with client.MaxThreads = 2;.
Build fails on iOS	Missing Metal shader	Reimport shaders under **Secure Asset Pipeline**.

Conclusion

You have now successfully:

- Integrated Unity with a **Post-Quantum Secure XR SDK**.
- Built a **cross-platform XR scene**.
- Ensured **quantum-resistant security** for multiplayer environments.

This workflow not only demonstrates **technical fluency with Unity + XR** but also showcases knowledge of **rare, forward-looking security standards (PQC, hybrid cryptography, biometric binding)**—making it a **cutting-edge enterprise-ready XR solution**.