Federal Office
for Information Security

Work Package 4: Telemetry

Version: 1.0

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Phone: +49 22899 9582-0
E-Mail: bsi@bsi.bund.de
Internet: https://www.bsi.bund.de
© Federal Office for Information Security 2018

Table of Contents

1 Introduction 5
1.1 Zusammenfassung 5
1.2 Executive Summary 9
1.3 Event Tracing for Windows 11
1.3.1 Concepts and Terms 11
1.3.2 Initialization 14
2 Technical Analysis of Functionalities 17
2.1 Telemetry: Architecture 17
2.2 Telemetry Data: Sources 18
2.2.1 Telemetry Data: Collection and Processing 26
2.3 Telemetry: Network Interface 30
2.4 Telemetry: Monitoring Activities 38
3 Configuration and Logging Capabilities 40
3.1 Configuration Capabilities 40
3.2 Logging Capabilities 42
Appendix 44
Tools 44
API Monitor: Main Module 45
API Monitor: Extension Definition 46
API Monitor: Extension EnableTraceEx2 46
API Monitor: Extension AddDataToRingBufferA 47
Yara Rule 47
RBS Decompression Script 48
External Executables 48
Server Authentication Certificates: Leaf 50
Server Authentication Certificates: Intermediate 53
Server Authentication Certificates: Root 57
Windows Performance Recorder Profile 60
Reference Documentation 62
Keywords and Abbreviations 63

Figures

Figure 1: The _GUID structure 12
Figure 2: The relationship between ETW providers, sessions and consumers 13
Figure 3: The initialization process of Windows 10 as depicted by Windows Performance Analyzer 14
Figure 4: Function stack: Kernel initialization 14
Figure 5: Function stack: Phase 1 15
Figure 6: Pseudo-code of EtwInitializeAutoLogger 16
Figure 7: The DiagTrack service 17
Figure 8: DiagTrack operating within a service host process 18
Figure 9: The process for initializing Autologger-Diagtrack-Listener 19
Figure 10: The kernel reading session initialization data from the system's registry 20

Figure 11: AutoLogger-Diagtrack-Listener: Session initialization data......20
Figure 12: AutoLogger-Diagtrack-Listener: Session operational information......21
Figure 13: The name and GUID of Diagtrack-Listener......22
Figure 14: Diagtrack-Listener delivering logged data to utc_myhost.exe......23
Figure 15: Pseudo-code of execution of an external function......25
Figure 16: Data for configuring the execution of icacls.exe......26
Figure 17: Flow of telemetry data (the ‘Diagtrack-Listener’ ETW session)......27
Figure 18: Telemetry data delivered by Diagtrack-Listener......27
Figure 19: Telemetry data processed in Microsoft::Diagnostics::CRingBufferEventStore::AddData......28
Figure 20*: Compressed telemetry data*......28
Figure 21: A snippet of the content of Events_Realtime.rbs......28
Figure 22*: Telemetry data being sent to the Microsoft’s back-end infrastructure*......29
Figure 23: *Telemetry data stored in Events_NormalCritical.rbs*......29
Figure 24: Decompressed telemetry data stored in Events_NormalCritical.rbs......29
Figure 25: Properties of ‘Diagtrack-Listener’......30
Figure 26: A snippet of the output of Microsoft Message Analyzer......30
Figure 27: Communication frequency and intensity (host: 40.77.226.250)......31
Figure 28: Communication frequency and intensity (host: 40.77.226.249)......32
Figure 29: Cached MSA token......32
Figure 30: A portion of a ‘client hello’ message......33
Figure 31: A portion of a ‘server hello’ message......33
Figure 32: TLS secure session negotiate with 40.77.226.250......34
Figure 33: A POST request......34
Figure 34: The exchange between DiagTrack and 40.77.226.249......35
Figure 35: Part of the data frame stored in the HTTP2 packet......35
Figure 36: Establishment of a TLS connection (WinHTTP)......36
Figure 37: The CheckCertForMicrosoftRoot function......36
Figure 38: Logged events related to certificate operations......37
Figure 39: Detailed information about the event with ID 30......37
Figure 40: Activity monitoring architecture......38
Figure 41: The DiagTrack service reading configuration data stored in the registry......41
Figure 42: The DiagTrack service reading configuration files......41
Figure 43: A portion of the utc.app.json file......41
Figure 44: A portion of a Windows Performance Recorder custom profile......42
Figure 45: Event categories......43
Figure 46: Detailed event information logged by Microsoft-Windows-Diagtrack......43

Tables

Tabelle 1: Telemetrie-Level und Anzahl der assoziierter ETW-Anbieter mit Autologger-Diagtrack-Listener......7
Tabelle 2: Telemetrie-Level und Anzahl der assoziierter ETW-Anbieter mit Diagtrack-Listener......7
Table 3: Telemetry levels and number of ETW providers associated with Autologger-Diagtrack-Listener......21
Table 4: Telemetry levels and number of ETW providers associated with Diagtrack-Listener......24
Table 5: Information on hosts hardcoded in diagtrack.dll......31
Table 6: Values of Telemetry levels......40

1 Introduction

1.1 Zusammenfassung

Dieses Kapitel stellt das Ergebnis von Arbeitspaket 4 des Projekts „SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10“ dar. Das Projekt wird durch die Firma ERNW GmbH im Auftrag des Bundesamts für Sicherheit in der Informationstechnik (BSI) durchgeführt.

Ziel dieses Arbeitspaketes ist die Analyse der Funktionalitäten und Eigenschaften der Microsoft Telemetrie-Komponenten (im weiteren Verlauf auch als *Telemetry* bezeichnet), wie sie in Windows 10 implementiert sind. Ausgehend von den Anforderungen des Bundesamts für Sicherheit in der Informationstechnik liegt der Fokus auf Windows 10, Version 1607, 64 Bit, deutsche Sprache aus dem *Long-Term Servicing Branch* (LTSB). Die Telemetrie-Komponente sammelt Systemabsturz- und Nutzungsdaten (sogenannte Telemetriedaten) und lädt diese Daten auf von Microsoft betriebene Remote-Server hoch (d. h. die Back-End-Infrastruktur von Microsoft; siehe [ERNW WP2], Abschnitt 2.4.2).

In einem dedizierten Dokument ist beschrieben, wie Telemetrie-Aktivität effektiv deaktiviert werden kann (siehe [ERNW WP4.1]).

Die wesentlichen Inhalte dieser Arbeit sind:

- Ein Überblick über die Systemprotokollierungsfunktionalitäten zur Erfassung von Telemetriedaten: Telemetry nutzt *Event Tracing for Windows* (ETW) zur Erfassung von Telemetriedaten (siehe [ERNW WP2], Abschnitt 2.4.2). ETW ist eine Kernprotokollierungsfunktion von Windows 10. Wir definieren zunächst grundlegende Konzepte und Begriffe im Zusammenhang mit ETW und beschreiben dann die Funktionalität der ETW-Protokollierung via ETW-Anbieter (*ETW providers)* (d. h. Entitäten, die Daten protokollieren), ETW-Konsumenten (*ETW consumers*; d. h. Entitäten, die Daten von ETW-Anbietern verarbeiten) und ETW-Sitzungen (*ETW sessions*; d. h. Entitäten, die Daten, die von assoziierten ETW-Anbietern protokolliert werden, an ETW-Konsumenten liefern). Darüber hinaus geben wir einen Überblick darüber, wie ETW initialisiert und für den Einsatz in Verbindung mit Telemetry vorbereitet wird.
- Eine Analyse der Erfassung und Verarbeitung von Telemetriedaten: Zunächst identifizieren und beschreiben wir die Quellen, die Telemetriedaten an Telemetry liefern. Besonders relevant ist die Beziehung zwischen ETW und Telemetry sowie die Art und Weise, wie Telemetry ETW nutzt, um Telemetriedaten zu erhalten. Darüber hinaus identifizieren wir weitere Quellen (d. h. externe Applikationen oder ausführbare Dateien), die zusätzlich zu ETW Daten an Telemetry liefern. Wir beschreiben auch das Format der Telemetrie-Daten und die Art und Weise, wie die Daten von Telemetry verarbeitet werden, bevor sie an die Back-End-Infrastruktur von Microsoft gesendet werden.
- Eine Analyse der relevanten Netzwerkschnittstellen: Wir beschreiben die Übertragungsmechanismen über die Telemetriedaten an die Back-End-Infrastruktur von Microsoft gesendet werden. Im Fokus unserer Analyse steht die Netzwerkschnittstelle zwischen der Telemetrie-Komponente und dieser Infrastruktur sowie deren Schutz.
- Ein Ansatz zur Erkennung und Beobachtung von Telemetrie-Aktivitäten: Wir stellen einen Ansatz zur Erkennung und Beobachtung von Aktivitäten der Telemetrie-Komponente auf einer bestimmten Plattform vor. Dieses Vorgehen berücksichtigt die relevanten Eigenschaften und Funktionen der Telemetrie-Komponente. Es umfasst die Kommunikation mit spezifischen Servern, die Teil der Back-End-Infrastruktur von Microsoft sind, Registrierung und Aktivierung von ETW-Anbietern, die Telemetriedaten erzeugen, sowie weitere Dinge.

Die Arbeit sowie die entsprechenden Ergebnisse stellt sich wie folgt dar:

- Abschnitt 1.3 liefert Informationen über ETW: Abschnitt 1.3.1 führt Konzepte und Begriffe zu ETW ein, welche zum besseren Verständnis des präsentierten Inhalts beitragen. Das sind zum Beispiel *ETW sessions*, *ETW providers* und *ETW consumers*.

- Abschnitt 1.3.2 beschreibt die Initialisierung von ETW. Der Windows 10-Initialisierungsprozess ist ein verketteter Vorgang, der aus aufeinanderfolgenden Schritten besteht. In der Analyse und Beschreibung konzentrieren wir uns auf die Phasen Pre-Session- (*pre-session initialization*) und Session-Initialisierung (*session initialization*), da ETW als Teil davon initialisiert wird:
  - *Pre-session initialization*: In diesem Schritt wird der Windows-Kernel in zwei Phasen initialisiert: Phase 0 und Phase 1. In Phase 0 wird unter anderem der System-Kernel-Thread angelegt. Hierbei ruft `KiInitializeKernel` auch die `Phase1Initialization` Funktion auf und leitet Phase 1 ein. In dieser Phase initialisiert der Kernel unter anderem die zentrale *Input/Output* I/O-Infrastruktur des Windows-Systems. Im Rahmen dessen wird auch ETW initialisiert. Sobald die ETW-Umgebung initialisiert ist, initialisiert der Kernel die ersten ETW-Sitzungen vom Subtyp „global logger“ und „autologger“.
  - *Session initialization*: Die Session-Initialisierung beginnt, wenn die `Phase1Initialization` Funktion abgeschlossen ist. Als Teil dieses Schrittes initialisiert der Session-Manager unter anderem die Registry des Systems, erstellt den Windows-Subsystem-Prozess (ausführbare Datei: `csrss.exe`), lädt die Subsystem dynamic-link library (DLL) und ruft schließlich Winlogon auf. Das Laden der Subsystem-DLLs durch den Session-Manager ermöglicht die Nutzung der ETW- *Application Programming Interface* (API). Sobald die ETW-API zur Verfügung steht, ist die ETW-Protokollierungsfunktion voll funktionsfähig und kann verwaltet werden, z. B. können ETW-Sitzungen und ETW-Anbieter registriert und aktiviert werden.
- Abschnitt 2 liefert die Ergebnisse der technischen Analyse von Telemetrie-Funktionalitäten:
  - Abschnitt 2.1 liefert eine Übersicht über die Architektur der Telemetrie-Komponente:

    Der Connected User Experiences and Telemetry Service, auch DiagTrack genannt, ist der zentrale Baustein der Windows Telemetrie-Komponente und ist in der Datei `%SystemRoot%\System32\diagtrack.dll` implementiert. Der DiagTrack-Dienst bezieht Daten von zwei ETW-Sitzungen vom Typ „other user“: eine mit dem Namen `Diagtrack-Listener` und eine mit dem Namen `Autologger-Diagtrack-Listener`. Autologger-Diagtrack-Listener ist während der Systeminitialisierung aktiv und speichert die protokollierten Daten in einer binären Datei unter `%ProgramData%\Microsoft\Diagnosis\ETLLogs\AutoLogger\AutoLogger-Diagtrack-Listener.etl`. Diese Sitzung wird deaktiviert, wenn DiagTrack initialisiert wird. Der Diagtrack-Listener wird beim Start des DiagTrack-Dienstes initialisiert. Er liefert aufgezeichnete Daten in Form eines Echtzeit-Protokoll-Feeds (*real time log feed*) an DiagTrack.
  - Abschnitt 2.2 identifiziert und beschreibt die Quellen, die Telemetrie-Daten an Telemetry liefern:

    Im Mittelpunkt steht der Initialisierungsprozess von Autologger-Diagtrack-Listener und Diagtrack-Listener sowie die ETW-Anbieter, die mit diesen Sitzungen verbunden sind:
    - **Autologger-Diagtrack-Listener** Diese ETW-Sitzung wird im Pre-Session-Initialisierungschritt initialisiert. Der Kernel initialisiert diese Sitzung auf der Grundlage von Daten, die in der Registry des Systems unter `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\AutoLogger\AutoLogger-Diagtrack-Listener` gespeichert sind. Der Kernel verknüpft dann ETW-Anbieter mit dem Autologger-Diagtrack-Listener basierend auf den *Globally Unique Identifier* (GUIDs) dieser Anbieter, die im Registrierungsschlüssel `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ControlSet\WMI\AutoLogger\AutoLogger-AutoLogger-Diagtrack-Listener\{ProviderGUID}` gespeichert sind. Danach wird die Autologger-Diagtrack-Listener ETW-Sitzung aktiviert. Einmal aktiviert, speichert Autologger-Diagtrack-Listener die protokollierten Daten in der Datei `%ProgramData%\Microsoft\Diagnosis\ETLLogs\AutoLogger\AutoLogger\AutoLogger-Diagtrack-Listener.etl`. Der Inhalt dieser Datei wird später durch den DiagTrack-Dienst verarbeitet. Je nach konfiguriertem Telemetrie-Level gibt es eine unterschiedliche Anzahl ETW-Anbieter, die mit Autologger-Diagtrack-Listener assoziiert sind:

Telemetrie-Level	ETW-Anbieter
`Security`	9
`Basic`	93
`Enhanced`	105
`Full`	112

Tabelle 1: Telemetrie-Level und Anzahl der assoziierter ETW-Anbieter mit Autologger-Diagtrack-Listener

- **Diagtrack-Listener** Die Initialisierung der Diagtrack-Listener ETW-Sitzung erfolgt durch den Dienst DiagTrack, der im Session-Initialisierungsschritt gestartet wird. Dieser Dienst wird automatisch durch den Systemunterstützungsprozess Service Control Manager gestartet. Die Diagtrack-Listener-Sitzung liefert aufgezeichnete Daten in Form eines Echtzeit-Protokoll-Feeds an den DiagTrack-Dienst. ETW-Anbieter sind über GUIDs mit dem Diagtrack-Listener verbunden. Einige der GUIDs sind in der Bibliothek `diagtrack.dll` fest hinterlegt. Die überwiegende Anzahl von GUIDs wird jedoch innerhalb der `%ProgramData%\Microsoft\Diagnosis\DownloadedSettings\utc.app.json` Datei gespeichert. Diese Datei wird von Microsoft kontrolliert und in regelmäßigen abständen erneuert. Das bedeutet, dass sich die Definition eines konfigurierten Telemetrie-Levels (d.h. die Anzahl der ETW-Anbieter, die abhängig vom konfigurierten Telemetrie-Level einer ETW-Sitzungen zugeordnet sind) sich durch Updates ändern kann. Darüber hinaus ist es technisch möglich, dass beliebige Prozesse (die über administrative Rechte oder höher verfügen) ETW-Anbieter registrieren und mit den DiagTrack ETW-Sitzungen assoziieren können. Abhängig vom konfigurierten Telemetrie-Level gibt es einen signifikanten Unterschied in der Anzahl der ETW-Anbieter, die dem Diagtrack-Listener zugeordnet sind (siehe nächste Tabelle). Es muss betont werden, dass die Anzahl der ETW-Anbieter, die abhängig vom konfigurierten Telemetrie-Level dem Diagtrack-Listener zugeordnet sind, dynamisch ist (d.h. die Anzahl der ETW-Anbieter ist auch innerhalb eines konfigurierten Telemetrie-Level nicht konstant). Die Anzahl unterscheidet sich basierend auf der eingesetzten Betriebssystemversion, sowie dem zugrunde liegenden Systemzustand. Systemzustand umfasst z.B. laufende Prozesse, installierte Software, Betriebssystemkonfiguration und so weiter. Dies gilt auch für den Autologger-Diagtrack-Listener.

Telemetrie-Level	ETW-Anbieter
`Security`	4
`Basic`	410
`Enhanced`	418
`Full`	422

Tabelle 2: Telemetrie-Level und Anzahl der assoziierter ETW-Anbieter mit Diagtrack-Listener

- Telemetry bietet weiterhin die Funktionalität zusätzliche Programme oder auch einzelne Funktionen in Bibliotheken aufzurufen, um weitergehende Informationen wie bspw. Speicher-Dumps zur Ergänzung einer Fehlermeldung zu erheben. Die aufgerufenen Programme und Funktionen sind beschrieben und untersucht (eine Liste dieser Programme findet sich im Anhang); in der untersuchten Version war es über diese Funktionalität nicht möglich, entfernt die Ausführung beliebigen Programmcodes zu veranlassen (wie es beispielsweise möglich wäre, wenn Telemetry `PowerShell.exe` ausführen könnte). Eine Möglichkeit der De-Installation des Telemetrie-Frameworks würde dennoch die Angriffsoberfläche verringern, da so ausführbare Dateien entfernt würden, die mit höheren Rechten ausgeführt werden (und anfällig gegen Angriffe zur Privilegienerhöhung sein könnten, wie bspw. via *Binary Planting*) oder von Angreifern auf dem System zur Erhebung weiterer Informationen genutzt werden könnten (beispielsweise deutet die Datei `disksnapshot.exe` auf Funktionalität zur Erhebung von Daten hin).

- Abschnitt 2.2.1 analysiert die Erhebung und Verarbeitung von Telemetriedaten durch Telemetry:

1. Die Autologger-Diagtrack-Listener ETW-Sitzung speichert die protokollierten Daten in den Dateien `%ProgramData%\Microsoft\Diagnosis\ETLLogs\AutoLogger\AutoLogger\AutoLogger-Diagtrack-Listener.etl` und `%ProgramData%\Microsoft\Diagnosis\ETLLogs\ShutdownLogger\AutoLogger-Diagtrack-Listener.etl`. Der Inhalt dieser Dateien ist in einem binären Format gespeichert. Im Gegensatz zum Autologger-Diagtrack-Listener speichert die Diagtrack-Listener ETW-Sitzung keine protokollierten Telemetriedaten in Dateien, sondern liefert die Daten in Form eines Echtzeit-Protokoll-Feeds an den DiagTrack-Dienst. Die Daten werden in den von der Diagtrack-Listener ETW-Sitzung verwalteten Kernelpuffern (*kernel buffers*) zwischengespeichert. Die in den Kernelpuffern zwischengespeicherten Telemetriedaten werden periodisch (vergleiche Kapitel 2.2.1 und 2.3, unter normalen Umständen alle 15 Minteun) zum DiagTrack-Dienst weitergegeben, der einen ETW-Verbraucher repräsentiert. Nachdem der DiagTrack-Dienst die Telemetriedaten in seinen Speicherpuffern gespeichert hat, konvertiert er die Daten in das *JavaScript Object Notation* (JSON)-Format, d. h. der Dienst bereitet die Daten für den Versand an die Microsoft Back-End-Infrastruktur vor. Zusätzlich werden die Telemetriedaten mit dem `DEFLATE` Algorithmus komprimiert.

- Abschnitt 2.3 erörtert die Netzwerkschnittstelle zwischen Telemetry und der Back-End-Infrastruktur von Microsoft zum Austausch von Telemetriedaten:

  DiagTrack und die Back-End-Infrastruktur von Microsoft kommunizieren über eine durch *Transport Layer Security* (TLS) geschützte Netzwerkschnittstelle (d. h. sie tauschen verschlüsselte Daten aus). Das zur Server-Authentifizierung notwendige Authentifizierungsmerkmal ist in der `crypt32.dll` Datei fest einprogrammiert. Bei dem Aufbau jeder TLS-Verbindung vergleicht DiagTrack das von dem Servern bereitgestellte Zertifikat mit dem fest einprogrammierten Authentifizierungsmerkmal. Wenn sie nicht übereinstimmen, wird die Verbindung vom DiagTrack-Dienst beendet. Der DiagTrack-Dienst verhindert die Verarbeitung von Daten von unbekannten Servern und die Offenlegung von Telemetriedaten bei Man-in-the-Middle-Angriffen erfolgreich. Das Serverzertifikat-Pinning, das DiagTrack durchführt, schützt jedoch nicht gegen die Offenlegung von Daten die aus der Microsoft Back-End-Infrastruktur stammen.

- Abschnitt 2.4 bietet einen Ansatz zur Erkennung und Beobachtung von Aktivitäten der Telemetrie-Komponente auf einer spezifischen Instanz des Betriebssystems Windows 10:

  Das vorgeschlagene Konzept kann in Form eines Frameworks umgesetzt werden. Das Framework führt die Überwachung von Telemetrie-Aktivitäten auf einem bestimmten System durch, basierend auf externen Quellen der Protokolle solcher Aktivitäten.

  - In diesem Kontext wurde ebenfalls festgestellt, dass einzelne Windows Komponenten (in diesem Fall konkret: Microsoft Internet Explorer) auch direkt, d.h. ohne Nutzung des Telemetrie-Dienstes, Verbindungen zum Telemetrie-Backend aufbauen. Die genauen Hintergründe werden gesondert betrachtet.

In den technischen Diskussionen stellen wir Function Call Stacks und Pseudocodes dar. Diese Call Stacks enthalten zum einfacheren Verständnis nur Funktionen, die relevant für die Ergebnisse sind; es werden nicht notwendigerweise alle Funktionen dargestellt, die Bestandteil der in Windows 10 implementierten Call Stacks sind. Dargestellter Pseudocode ist zudem eine hohe Abstraktion von realem Code und folgt nicht konsequent der Syntax einer bestimmten Programmiersprache. Wir stellen den Pseudocode in einer Form dar, die für ein besseres Verständnis der besprochenen Materie als optimal erachtet wird. Der Pseudocode, der in dieser Arbeit dargestellt wird, folgt lose einer C- und C++-ähnlichen Programmiersprachensyntax.

- Abschnitt 3 bietet einen Überblick über die Konfigurations- und Protokollierungsfunktionen von Windows 10 zur Verwaltung der Telemetrie-Komponente und zur Protokollierung relevanter Ereignisse:

  - Abschnitt 3.1 beschreibt die Konfigurationspunkte, die Windows 10 Benutzern zur Verfügung stellt, um Telemetrie zu konfigurieren:

Wie jeder Dienst, der in Windows 10 bereitgestellt wird, kann der DiagTrack-Dienst konfiguriert werden (z. B. gestartet oder gestoppt), indem das Dienstprogramm *Services* (ausführbare Datei: *services.msc*) genutzt wird oder die Registry-Werte unter `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\DiagTrack` geändert werden. Zusätzlich werden Telemetrie-Level verwendet, um festzulegen, welche und wie viele ETW-Anbieter aktiv für die Erfassung von Telemetriedaten genutzt werden. Der DiagTrack-Dienst bezieht Telemetrie-Daten aus den ETW-Sitzungen Diagtrack-Listener und Autologger-Diagtrack-Listener. ETW-Sitzungen können mit dem Dienstprogramm *Performance Monitor* konfiguriert werden.

- Abschnit 3.2 beschreibt den Einsatz von ETW zur Protokollierung von Telemetrie-Ereignissen;

  Windows 10 verwendet ETW für die Protokollierung von Telemetrieereignissen. Im Rahmen der durchgeführten Analysen analysierten wir die in *diagtrack. dll* implementierten Protokollierungsfunktionalitäten der Bibliotheksdatei, die den DiagTrack-Dienst implementiert. Es wurden vier ETW-Anbieter identifiziert, die bei der Initialisierung des DiagTrack-Dienstes registriert werden.

Der Appendix gliedert sich wie folgt: Im Abschnitt „Tools“ sind die verwendeten Werkzeuge inklusive Quelle aufgelistet. Die Abschnitte „API Monitor: Main Module“, „API Monitor: Extension EnableTraceEx2’“, und „API Monitor: Extension AddData“ enthalten Skripte zur Identifizierung von ETW-Anbietern, die Telemetriedaten erzeugen, sowie zur Anzeige dieser Daten, die für die Diskussionen in den Abschnitten 2.2 und Abschnitt 2.2.1 relevant sind; der „Yara Rule“ Abschnitt stellt Skript-Code zur Identifizierung von ETW-Anbietern bereit, die Telemetrie-Daten erzeugen, die für die Diskussionen im Abschnitt 2.2 relevant sind; der „RBS decompression script“ Abschnitt stellt Skript-Code zur Dekomprimierung von Telemetrie-Daten bereit, der für die Diskussion im Abschnitt 2.2.1 relevant ist; der „External Executables“ Abschnitt enthält eine Liste von Namen und Beschreibungen von ausführbaren Dateien, die zusätzlich zu ETW-Anbietern Telemetrie-Daten an den DiagTrack-Dienst liefern können; die „Server Authentication Certificates: Leaf“, „Server Authentication Certificates: Intermediate“, und „Server Authentication Certificates: Root“ Abschnitte enthalten die Zertifikate zur Server Authentifizierung, die für die Diskussionen im Abschnitt 2.3 relevant sind; der „Windows Performance Recorder Profile“ Abschnitt stellt eine Profildefinition im Extensible Markup Language(XML)-Format bereit, die für die Diskussionen im Abschnitt 3.2 relevant ist. Die im Appendix enthaltenen Methoden und Werkzeuge stellen dabei sicher, dass die Ergebnisse dieses Dokumentes mit vertretbarem Aufwand auf anderen Windows-Versionen nachgestellt werden können.

Fazit: Windows 10 verwendet ETW für die Protokollierung von Telemetrie-Daten. Die Entitäten, die Daten protokollieren (d.h. ETW-Anbieter) werden vom konfigurierten Telemetrie-Level definiert. Die Anzahl der ETW-Anbieter die mit den DiagTrack ETW-Sitzungen verbunden sind schwankt signifikant zwischen den einzelnen Telemetrie-Leveln. Aufgrund der durchgeführten Analyse lässt sich jedoch keine Verbindung zwischen Anzahl an ETW-Anbietern und Telemetry-Datenmenge, sowie deren Qualität ableiten. Ein Vorschlag zur Erkennung, Beobachtung, und Analyse der gesammelten Telemetry-Daten sowie der Aktivitäten der Telemetrie-Komponenten sind beschrieben.

Die Definition eines konfigurierten Telemetrie-Levels (d.h. die Anzahl der ETW-Anbieter, die abhängig vom konfigurierten Telemetrie-Level einer ETW-Sitzungen zugeordnet sind) wird durch Microsoft kontrolliert. Die notwendigen Metainformationen (d.h. GUIDs der designierten ETW-Anbieter) sind innerhalb der `%ProgramData%\Microsoft\Diagnosis\DownloadedSettings\utc.app.json` Datei gespeichert. Diese Datei wird von Microsoft kontrolliert und in regelmäßigen abständen erneuert, was den Eigenschaften der gesamten Konfigurations- und Protokollierungsfunktion eine Dynamik verleiht. Die Modifikation von Konfigurationsparametern kann auch im laufenden Systembetrieb durch geführt werden, da sie aus administrativer Sicht vollständig transparent ablaufen und ohne Zutun eines Anwenders erfolgen.

1.2 Executive Summary

This chapter implements the work plan outlined in Work Package 4 of the project “SiSyPHuS Win10: Studie zu Systemaufbau, Protokollierung, Härtung und Sicherheitsfunktionen in Windows 10“ (orig., ger.),

contracted by the German Federal Office for Information Security (orig., ger., Bundesamt für Sicherheit in der Informationstechnik (BSI)). The work planned as part of Work Package 4 has been conducted by ERNW GmbH in the time period between July and September 2017, in accordance with the time plan agreed upon by ERNW GmbH and the German Federal Office for Information Security.

The objective of this work package is the analysis of the functionalities and properties of the Telemetry component of Windows 10. As required by the German Federal Office for Information Security, the exact release of the Windows 10 system in focus is build 1607, 64-bit, long-term servicing branch (LTSB), German language. The Telemetry component of this system collects system crash and usage data (referred to as telemetry data) and uploads this data to remote servers operated by Microsoft (i.e., the Microsoft's back-end infrastructure, see [ERNW WP2], Section 2.4.2).

A dedicated document describes how to effectively disable telemetry activity (see [ERNW WP4.1]).

The core contributions of this work are:

- An overview of system logging functionalities for collecting telemetry data: Telemetry leverages Event Tracing for Windows (ETW) for collecting telemetry data (see [ERNW WP2], Section 2.4.2). ETW is a core logging facility of Windows 10. We first define essential concepts and terms related to ETW, and then discuss ETW logging functionalities. This includes discussions on: ETW providers – entities logging data; ETW consumers – entities consuming data logged by ETW providers; ETW sessions – entities delivering data, logged by associated ETW providers, to ETW consumers; and so on. In addition, we provide an overview of how ETW is initialized and made ready to be used by Telemetry;
- An analysis of the collection and procession of telemetry data: We first identify and discuss the sources delivering telemetry data to Telemetry. We primarily analyze the relationship between ETW and Telemetry – we discuss the ways in which Telemetry leverages ETW for obtaining telemetry data. Among other things, we identify and discuss the ETW sessions delivering data to Telemetry, as well as ETW providers associated with these sessions. In addition, we identify other sources that, in addition to ETW, deliver data to Telemetry. We also discuss the format of telemetry data and the way in which this data is processed by Telemetry, before it is sent to the Microsoft's back-end infrastructure;
- An analysis of relevant network interfaces: We discuss the way in which telemetry data is sent to Microsoft. We focus our discussions on the network interface between the Telemetry component and this infrastructure. We also discuss the way in which data exchanges between Telemetry and the Microsoft's back-end infrastructure are protected;
- An approach for monitoring Telemetry activities: We provide an approach for monitoring activities of the Telemetry component on a given platform. This approach takes into account relevant properties and features of the Telemetry component. This includes communication with specific servers that are part of the Microsoft's back-end infrastructure. It also includes registration and activation of ETW providers that produce telemetry data.
  - In this context, the direct communication of Windows components (here: Microsoft Internet Explorer) without using the telemetry service with the telemetry back-end was observed. The detailed analysis will be carried out separately.

This work is structured as follows:

- Section 1.3 provides information on ETW: Section 1.3.1 introduces concepts and terms related to ETW, which are relevant for better understanding the contents we present; Section 1.3.2 discusses the procedure implemented as part of Windows 10 for initializing ETW;
- Section 2 provides technical information on functionalities, whose analysis is in the scope of this work package:
  - Section 2.1 provides an overview of the architecture of the Telemetry component;
  - Section 2.2 identifies and discusses the sources delivering telemetry data to Telemetry;
  - Section 2.2.1 analyzes the collection and procession of telemetry data by Telemetry;

  - Section 2.3 discusses the network interface established between Telemetry and the Microsoft's back-end infrastructure for the purpose of exchanging telemetry data;
  - Section 2.4 provides an approach for monitoring activities of the Telemetry component on a given instance of the Windows 10 operating system.

  Throughout the technical discussions, we depict function call stacks and pseudo-code. We emphasize that these call stacks, for the sake of comprehension, present only functions that we consider relevant to the discussion. We do not necessarily depict all functions that are part of the call stacks as implemented in Windows 10. In addition, depicted pseudo-code is a high abstraction of real code and does not consistently follow the syntax of a particular programming language. We depict pseudo-code in a form considered optimal for better understanding of the discussed matter. The pseudo-code depicted in this work loosely follows a C and C++-like programming language syntax.

- Section 3 provides an overview of the configuration and logging capabilities of Windows 10 for managing the Telemetry component and logging relevant events: Section 3.1 discusses the configuration points that Windows 10 provides for users to configure Telemetry; Section 3.2 discusses the use of ETW for logging Telemetry events;
- In the Appendix: the 'Tools' section lists used tools and where they can be found: the 'API Monitor: Main Module', 'API Monitor: Extension Definition', 'API Monitor: Extension EnableTraceEx2', and 'API Monitor: Extension AddData' sections provide scripts for identifying ETW providers that produce telemetry data and viewing this data, relevant to the discussions in Section 2.2 and Section 2.2.1; the 'Yara Rule' section provides script code for identifying ETW providers that produce telemetry data, relevant to the discussions in Section 2.2; the 'RBS decompression script' section provides a script for decompressing telemetry data, relevant to the discussions in Section 2.2.1; the 'External Executables' section provides a list of names, and descriptions, of executables that deliver telemetry data to Telemetry; the 'Server Authentication Certificates: Leaf/Intermediate/Root' secions provide certificates for verifying server identity, relevant to the discussions in Section 2.3; the 'Windows Performance Recorder Profile' section provides a script for viewing internal Telemetry events, relevant to the discussions in Section 3.2.

1.3 Event Tracing for Windows

Telemetry collects system crash and usage data. This data is mainly produced by the core logging mechanism of this operating system – ETW. We introduced ETW in [ERNW WP2], Section 2.5.1. In this section, we provide a brief overview of relevant concepts and terms related to ETW for better understanding the content of this work (Section 1.3.1). We also discuss the process for initializing ETW, implemented in Windows 10 (Section 1.3.2) .

1.3.1 Concepts and Terms

The architecture of ETW consists of the following essential components: session, provider, controller, and consumer (see [ERNW WP2], Section 2.5.1). Next, we focus on each of these components.

ETW sessions ETW writes logged events in designated kernel buffers. System threads deliver logged events to entities consuming them (see Figure in [ERNW WP2], Section 2.5.1). Delivered logged events may be in the form of files or real-time log feeds. Multiple kernel buffers are managed by a single kernel entity named ETW session, also known as logger. An ETW session can be in an activated or a deactivated state. By default, ETW supports 64 sessions operating simultaneously.[1] An ETW session can be one of the following types: 'NT kernel logger' or 'other user' ([Soulami 2012], Chapter 12). A session of type 'NT kernel logger' obtains logging data delivered from predefined ETW providers. This includes, for example, providers delivering disk input/output I/O or page fault events. A session of type 'other user' obtains data from all other ETW providers. We introduced the concept of ETW providers in [ERNW WP2], Section 2.5.1. In paragraph 'ETW providers' in this section, we discuss ETW providers in more detail.

1 https://msdn.microsoft.com/en-us/library/windows/desktop/aa364117(v=vs.85).aspx [Retrieved: 26/10/2017]

In addition to the previously mentioned types of ETW sessions, an ETW session be one of the following sub-types: 'global logger'[2] and 'autologger'[3]. These sessions obtain logging events occurring early in the system boot process. Resources can use these sessions to obtain logged data before the actual user environment is initialized. The sub-type 'autologger' differs from the sub-type 'global logger' in the following ways:

- There can be multiple sessions of sub-type 'autologger', whereas there can be only one session of sub-type 'global logger';
- A session of sub-type 'autologger' cannot be of type 'NT kernel logger' – only a session of sub-type 'global logger' can be a 'NT kernel logger' session.

Relevant information about the session of sub-type 'global logger' is located in the system's registry at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\GlobalLogger`. Relevant information about a session of sub-type 'autologger' is located in the system's registry at the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\AutoLogger`. For example, `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\AutoLogger\LoggerSession\{ProviderGUID}`contain the global unique identifier (GUIDs) of the ETW providers delivering data to the session of type 'autologger', named `LoggerSession`. We discuss on ETW providers and their GUIDs in the following paragraph.

ETW providers Logged data is produced and delivered to sessions by ETW providers associated with these sessions. ETW providers are entities implemented and declared using the ETW application programming interface (API) (see [ERNW WP2], Section 2.5.1), as part of code-instrumented user- or kernel-land resources. The entity declaring a given ETW provider registers the provider and generates a handle to it. This handle is used for referencing the provider when actual data is logged. Each ETW provider is used for logging a specific category of events. For example, the Transmission Control Protocol(TCP)/Internet Protocol(IP) driver implements its own provider logging network events. For ETW providers to actively deliver data, they have to be registered and enabled. This notifies the operating system of the provider's existence.

```
typedef struct _GUID {
  DWORD Data1;
  WORD  Data2;
  WORD  Data3;
  BYTE  Data4[8];
} GUID;
```

Figure 1: The _GUID structure

ETW providers are registered based on GUIDs, which are declared as instances of a structure named `_GUID`. Figure 1 depicts the implementation of the _GUID structure.[4] A GUID is a 128-bit value consisting of: a 32-bit integer value (i.e., data type `DWORD`, *Data1* in Figure 1) in the form of 8 hexadecimal digits; two 16-bit integer values (i.e., data type `WORD`, *Data2* and *Data3* in Figure 1) in the form of 4 hexadecimal digits; and an array containing 8 bytes in the form of 16 hexadecimal digits (data type BYTE, *Data4[8]* in Figure 1). An example of a GUID is `00112233-4455-6677-8899-AABBCCDDEEFF`. ETW providers can be structured into ETW provider groups, referenced by provider group GUIDs. These GUIDs have the same format as ETW provider GUIDs.

In terms of how ETW providers are implemented in code, there are the following types of providers:[5] Managed Object Format (MOF) providers, Windows software trace preprocessor (WPP) providers, Manifest-based providers, and TraceLogging providers.[6] Among other things, providers of these types differ in terms

2 https://msdn.microsoft.com/en-us/library/windows/desktop/aa363690(v=vs.85).aspx [Retrieved: 26/10/2017]
3 https://msdn.microsoft.com/en-us/library/windows/desktop/aa363687(v=vs.85).aspx [Retrieved: 26/10/2017]
4 https://msdn.microsoft.com/en-us/library/windows/desktop/aa373931(v=vs.85).aspx [Retrieved: 9/8/2017]
5 https://msdn.microsoft.com/en-us/library/windows/desktop/aa364161(v=vs.85).aspx [Retrieved: 9/1/2017]
6 https://msdn.microsoft.com/en-us/library/windows/desktop/aa363668(v=vs.85).aspx#providers [Retrieved: 9/1/2017]

of what functions of the ETW API are used for provider registering, and what information is required for a provider to be registered.

ETW controllers The management of ETW sessions and association of ETW providers to ETW sessions is done by special-purpose applications, referred to as ETW controllers. An example ETW controller (outside of the core operating system, which is directly using the ETW-API) is the `xperf` utility, distributed as part of the Windows Performance Toolkit.[7]

ETW consumers Logged data managed by sessions, and produced by ETW providers, are consumed by ETW consumers. ETW consumers are entities viewing and parsing logged data, such as the `xperf` or the `Windows Event Viewer` utility. Logged data may be delivered to ETW consumers in real-time, or may be periodically flushed to files.

Figure 2 depicts the relationship between ETW providers, sessions, and consumers (generalized as *ETW entities* in Figure 2). Multiple ETW providers can deliver logged data to one ETW session, and some providers can deliver data to multiple sessions ([Soulami 2012], Chapter 12). A session delivering logged data to ETW consumers in the form of a real-time log feed can deliver data to only one consumer.[8] A session delivering logged data to ETW consumers in the form of files can deliver events to consumers.

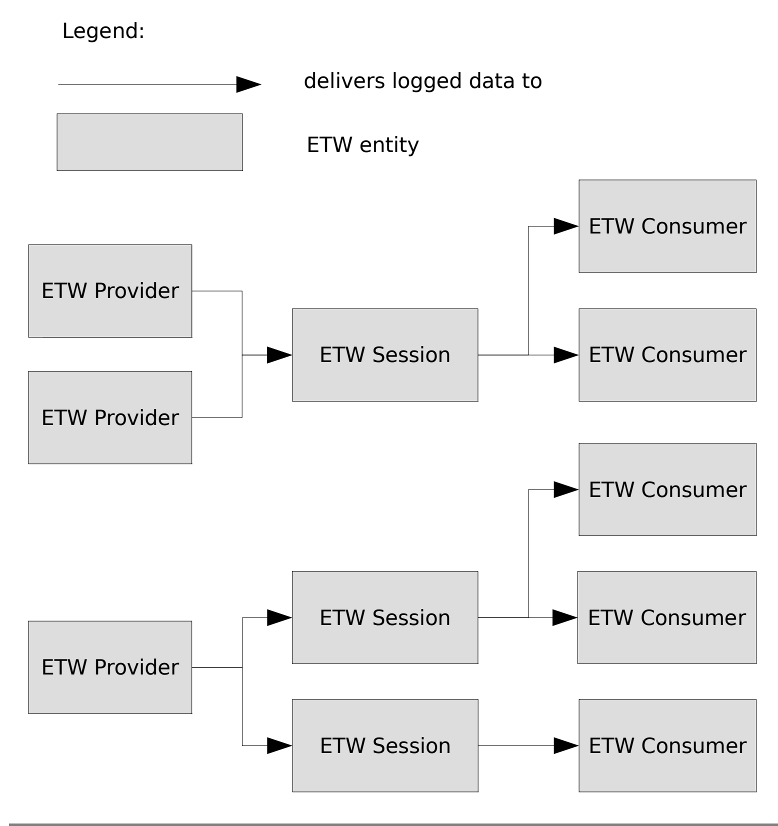


Figure 2: The relationship between ETW providers, sessions and consumers

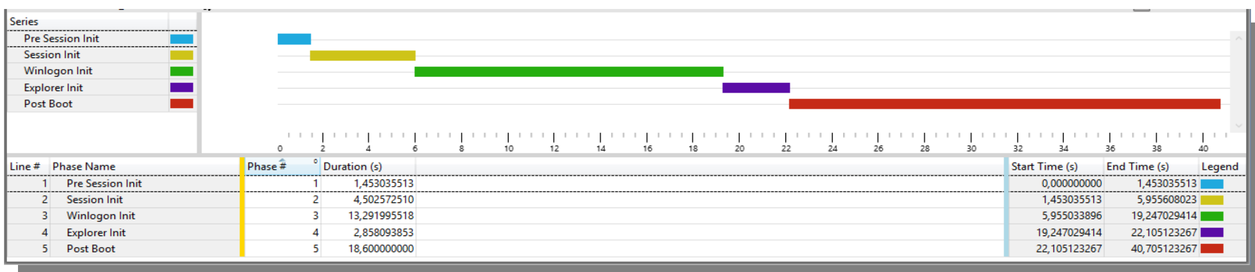
In summary, ETW enables the logging of relevant data. This data is produced by ETW providers and stored in designated kernel buffers. ETW providers deliver logged data to associated ETW sessions. An ETW session is a kernel entity that manages the designated memory buffers storing the logged data, and delivers the logged data to ETW consumers. An ETW controller is an entity responsible for associating ETW providers with ETW

7 https://docs.microsoft.com/en-us/windows-hardware/test/wpt/ [Retrieved: 7/5/2017]
8 https://msdn.microsoft.com/de-de/library/windows/desktop/aa364089%28v=vs.85%29.aspx [Retrieved: 9/7/2017]

sessions and managing these sessions. This may involve, for example, specifying the way in which an ETW session delivers data to ETW consumers and the size of the designated kernel buffers.

1.3.2 Initialization

In this section, we discuss the process for initializing ETW implemented in Windows 10. We focus our discussion on activities performed as part of this process by the operating system. Therefore, we structure the discussion in this section by taking into account the initialization of the operating system itself. Figure 3 depicts the initialization process of Windows 10 as depicted by the `Windows Performance Analyzer` utility. This is a chain process that consists of the following sequential steps: pre-session initialization, session initialization, Winlogon initialization, and Explorer initialization.[9] These steps are conducted by the Windows kernel (see [ERNW WP2], Section 2.3.1). Some of the activities performed as part of these steps are kernel initialization, initialization of system support processes, and initialization of system services (see [ERNW WP2], Section 2.3.1). Next, we focus on the pre-session and session initialization steps. This is because ETW is initialized as part of them.



Line #	Phase Name	Phase #	Duration (s)	Start Time (s)	End Time (s)	Legend
1	Pre Session Init	1	1,453035513	0,000000000	1,453035513	
2	Session Init	2	4,502572510	1,453035513	5,955608023	
3	Winlogon Init	3	13,291995518	5,955033896	19,247029414	
4	Explorer Init	4	2,858093853	19,247029414	22,105123267	
5	Post Boot	5	18,600000000	22,105123267	40,705123267	

Figure 3: The initialization process of Windows 10 as depicted by Windows Performance Analyzer

Pre-session initialization In this step, the Windows kernel is initialized. The kernel is initialized in two phases: Phase 0 and Phase 1 ([Russinovich 2012], Chapter 13], see Figure 4 and [ERNW WP5], Section 2.3.1). The `KiSystemStartup` function is the first function, which is invoked after the kernel is loaded. `KiSystemStartup` invokes `KiInitializeKernel`, which in turn calls `InitBootProcessor`. `InitBootProcessor` is responsible for conducting relevant tasks, for example, defining process and thread objects (*Phase 0 executive routines* in Figure 4). Among other things, the System kernel thread is created (see [ERNW WP2], Section 2.3.1).

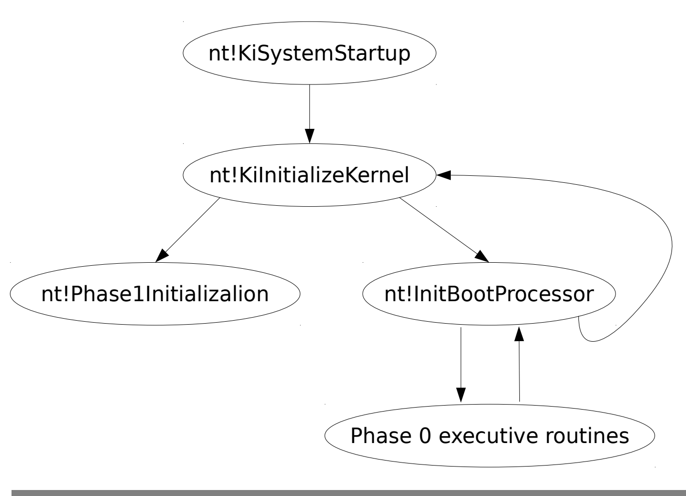


Figure 4: Function stack: Kernel initialization

9 https://social.technet.microsoft.com/wiki/contents/articles/11341.the-windows-7-boot-process-sbsl.aspx#Sub_phase_2_SMSSInit_Session_Initialization [Retrieved: 26/10/2017]

`KiInitializeKernel` also invokes the `Phase1Initialization` function, initiating Phase 1. In this phase, among other things, the kernel initializes the core I/O infrastructure of the Windows system (*IoInitSystem* in Figure 5). This involves initializing ETW (*EtwInitialize* → *EtwpInitialize* → *EtwInitializeSiloStat* in Figure 5). Once the ETW environment is initialized, the kernel initializes ETW sessions. These may be of the types and sub-types we discussed in Section 1.3.1 (*EtwInitializeAutoLogger* → *EtwStartAutoLogger* in Figure 5).

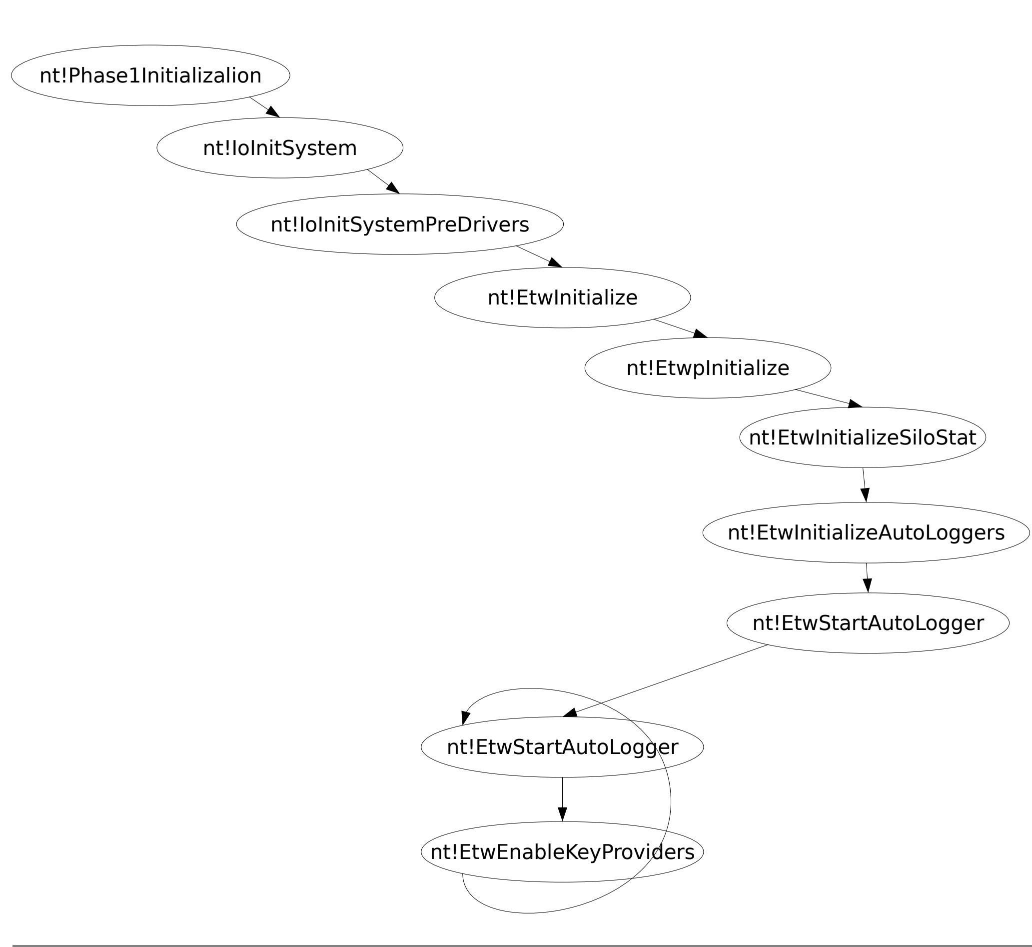


Figure 5: Function stack: Phase 1

Figure 6 depicts the implementation of the `EtwInitializeAutoLogger` function in the form of pseudo-code. This function first invokes `EtwStartAutoLogger`, a function responsible for initializing the session of the 'global logger' sub-type. Relevant information about this session is obtained from the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\GlobalLogger` (*L"WMI\\GlobalLogger"* in Figure 6, see Section 1.3.1). This information is then passed to `EtwStartAutoLogger`. `EtwStartAutoLogger` activates the session. The initialization of sessions of the 'autologger' sub-type is performed in a conceptually identical manner. Each session of this sub-type is initialized in a cycle, where relevant information from the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\AutoLogger` is obtained by invoking the `ZwOpenKey`[10] and `ZwEnumerateKey`[11] functions (see the *RtlInitUnicodeString*[12] function and *ObjectAttributes.ObjectName*[13] = *&DestinationString* in

10 https://msdn.microsoft.com/en-us/library/windows/hardware/ff567014(v=vs.85).aspx [Retrieved: 26/10/2017]
11 https://msdn.microsoft.com/en-us/library/windows/hardware/ff566447(v=vs.85).aspx [Retrieved: 26/10/2017]
12 https://msdn.microsoft.com/en-us/library/windows/hardware/ff561934(v=vs.85).aspx [Retrieved: 26/10/2017]
13 https://msdn.microsoft.com/en-us/library/windows/hardware/ff557749(v=vs.85).aspx [Retrieved: 26/10/2017]

Figure 6) . In addition, the `EtwEnableKeyProviders` function enables the ETW providers that deliver logged data to each initialized session in the cycle (see Figure 6).

```
void EtwpInitializeAutoLoggers()
{
    [...]
    // GlobalSourceString = L"GlobalLogger"
    EtwStartAutoLogger(GlobalSourceString, L"WMI\\GlobalLogger", [...]);
    [...]
        RtlInitUnicodeString(
            &DestinationString,
            L"\\Registry\\Machine\\System\\CurrentControlSet\\Control\\WMI\\AutoLogger");
        [...]
        ObjectAttributes.ObjectName = &DestinationString;
        [...]
        if ( ZwOpenKey(&KeyHandle, [...], &ObjectAttributes) >= 0 )
        {
            Index = 0;
            do
            {
                NTStatus = ZwEnumerateKey(KeyHandle, Index, 0, &KeyInformation, Length, &ResultLength);
                [...]
                // AutoSourceString = AutoLogger source string e.g. L"AppModel"
                // FullAutoSourceString = Full AutoLogger source string e.g. L"WMI\\AutoLogger\\AppModel"
                if ( [...] && (signed int)EtwStartAutoLogger(AutoSourceString, FullAutoSourceString, [...]) >= 0 )
                {
                    [...]
                    EtwpEnableKeyProviders([...], FullAutoSourceString, (unsigned int *)&KeyInformation);
                    [...]
                }
                [...]
                ++Index;
            }
            while ( NTStatus >= 0 );
            ZwClose(KeyHandle);
        }
    [...]
}
```

Figure 6: Pseudo-code of EtwInitializeAutoLogger

Before the pre-session initialization step is complete, `Phase1Initialization` initializes the system support process named session manager (executable: `smss.exe,` see Figure 1).

Session Initialization The session initialization step begins when the `Phase1Initialization` function completes. The session manager, among other things, initializes the system's registry (as in registry functionality, pre-session initialization, the registry is directly accessed on the file level), creates the Windows subsystem process (executable: `csrss.exe`), loads the subsystem DLLs, and ultimately invokes Winlogon (see [ERNW WP2], Section 2.3.1).

The loading of the subsystem DLLs by the session manager makes the use of the ETW API possible. Once the ETW API is available for use, the ETW logging facility is fully operational (see Section 1.3.1).

2 Technical Analysis of Functionalities

In this section, we discuss the architecture and operational principles of the Telemetry component. We base our discussion on an analysis of the Telemetry component deployed in Windows 10 (see Section 1.2). We performed the analysis using both static and dynamic analysis methods. To remind, Telemetry collects system crash and usage data, which we refer to as telemetry data.[14] Telemetry then uploads this data to remote servers operated by Microsoft over transport layer security (TLS) -secured network channel (see [ERNW WP2], Section 2.4.2). These servers are part of the back-end infrastructure for processing and storage of telemetry data, named Microsoft Data Management Service.[15]

This section is structured as follows: In Section 2.1, we focus on the architecture of Telemetry, summarizing the content presented in [ERNW WP2], Section 2.4.2 for reference purposes; in Section 2.2, we discuss sources of telemetry data (e.g. ETW providers, see Section 1.3); in Section 2.2.1, we focus on the collection and procession of telemetry data; in Section 2.3, we present on the TLS-secured network channel between Telemetry and the Microsoft's back-end infrastructure; and in Section 2.4, we provide an approach for detecting and observing activities of the Telemetry component on a given instance of the Windows 10 operating system that is in the focus of this work.

2.1 Telemetry: Architecture

As we mentioned in ([ERNW WP2], Section 2.4.2), the Connected User Experiences and Telemetry service, also referred to as DiagTrack, is the core building block of the Telemetry component (see Figure 11). This service is implemented in the `%SystemRoot%\System32\diagtrack.dll` library file (Figure 7 for a snippet of the output of the `Process Explorer` utility). DiagTrack starts automatically at system startup and operates within a service host process (`"%SystemRoot%\System32\svchost.exe -k utcsvc"`, see Figure 8 for a snippet of the output of the `Process Explorer` utility).[16] The service executes in the security context of the `LocalSystem` account, a predefined system account typically used for service operation. This account has extensive privileges.[17]

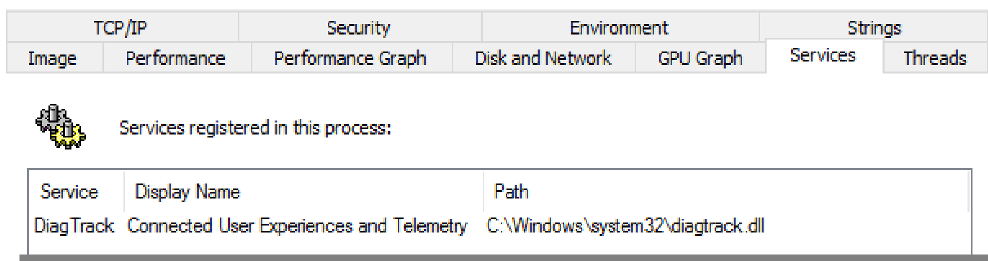


Figure 7: The DiagTrack service

The DiagTrack service collects telemetry data and uploads this data to the Microsoft's back-end infrastructure. For collecting this data, DiagTrack primarily relies on ETW (see Section 1.3). The DiagTrack service obtains data from two ETW sessions of type 'other user': one session named 'Diagtrack-Listener', and one named 'Autologger-Diagtrack-Listener'. 'Autologger-Diagtrack-Listener' is active during system initialization (see Section 1.3.2) and stores logged data in a file located at `%ProgramData%\Microsoft\Diagnosis\ETLLogs\AutoLogger\AutoLogger-Diagtrack-Listener.etl`. This session is

14 https://docs.microsoft.com/en-us/windows/configuration/configure-windows-telemetry-in-your-organization [Retrieved: 26/10/2017]
15 https://docs.microsoft.com/en-us/windows/configuration/configure-windows-telemetry-in-your-organization [Retrieved: 26/10/2017]
16 For more detail see the following link: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681957(v=vs.85).aspx [Retrieved: 9/8/2017]
17 For more detail see the following link: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686005(v=vs.85).aspx [Retrieved: 9/8/2017]

deactivated when DiagTrack is initialized. 'Diagtrack-Listener' is initialized when the DiagTrack service is initialized. It delivers logged data to DiagTrack in the form of a real-time log feed (see Section 1.3.1).

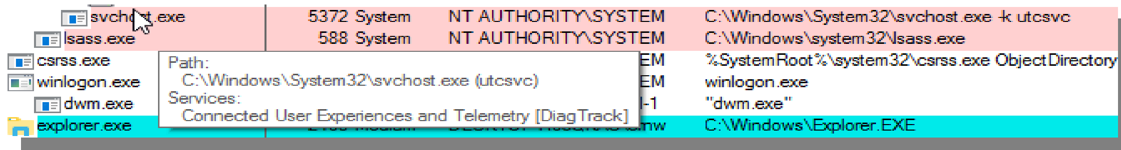


Figure 8: DiagTrack operating within a service host process

The 'Autologger-Diagtrack-Listener' and 'Diagtrack-Listener' ETW sessions obtain data from the ETW providers associated with them (see Section 1.3.1). The number of these providers depends on the configured Telemetry level. This level can be set to `Security`, `Basic`, `Enhanced`, or `Full`, which we list in an increasing order with respect to the number of ETW providers assigned to 'Autologger-Diagtrack-Listener' and 'Diagtrack-Listener'.[18] We discuss more on the Telemetry levels in Section 2.2 and Section 3.1.

2.2 Telemetry Data: Sources

In this section, we discuss the 'Autologger-Diagtrack-Listener' (paragraph 'Autologger-Diagtrack-Listener') and 'Diagtrack-Listener' ETW sessions (paragraph 'Diagtrack-Listener'). These sessions deliver telemetry data to the DiagTrack service (see Section 2.1). We focus our discussion on the initialization process of 'Autologger-Diagtrack-Listener' and 'Diagtrack-Listener'. This includes discussions on the ETW providers associated with these sessions.

As part of the analyses we conducted for this work, we observed that, in addition to the ETW sessions mentioned above, the DiagTrack service obtains telemetry data from other sources. Therefore, in this section, we identify and discuss these sources (paragraph ‚Additional sources').

Autologger-Diagtrack-Listener Figure 9 provides a compact overview of the process for initializing 'Autologger-Diagtrack-Listener'. This ETW session is initialized in the pre-session initialization step (see Section 1.3.2). This involves creation of the kernel buffers that the session manages (*Buffer n* in Figure 9). The kernel initializes this session based on relevant data stored in the system's registry at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\AutoLogger\AutoLogger-Diagtrack-Listener`. This includes, for example, a value indicating whether the session should be activated immediately after it is initialized. The kernel then associates ETW providers with 'Autologger-Diagtrack-Listener' (*Provider n* in Figure 9) based on the GUIDs of these providers stored at the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\WMI\AutoLogger\AutoLogger-Diagtrack-Listener\{ProviderGUID}` (*Registry* in Figure 9). `{ProviderGUID}` denotes the GUID of an ETW provider. The 'Autologger-Diagtrack-Listener' ETW session is then activated.

18 https://docs.microsoft.com/en-us/windows/configuration/configure-windows-telemetry-in-your-organization [Retrieved: 26/10/2017]

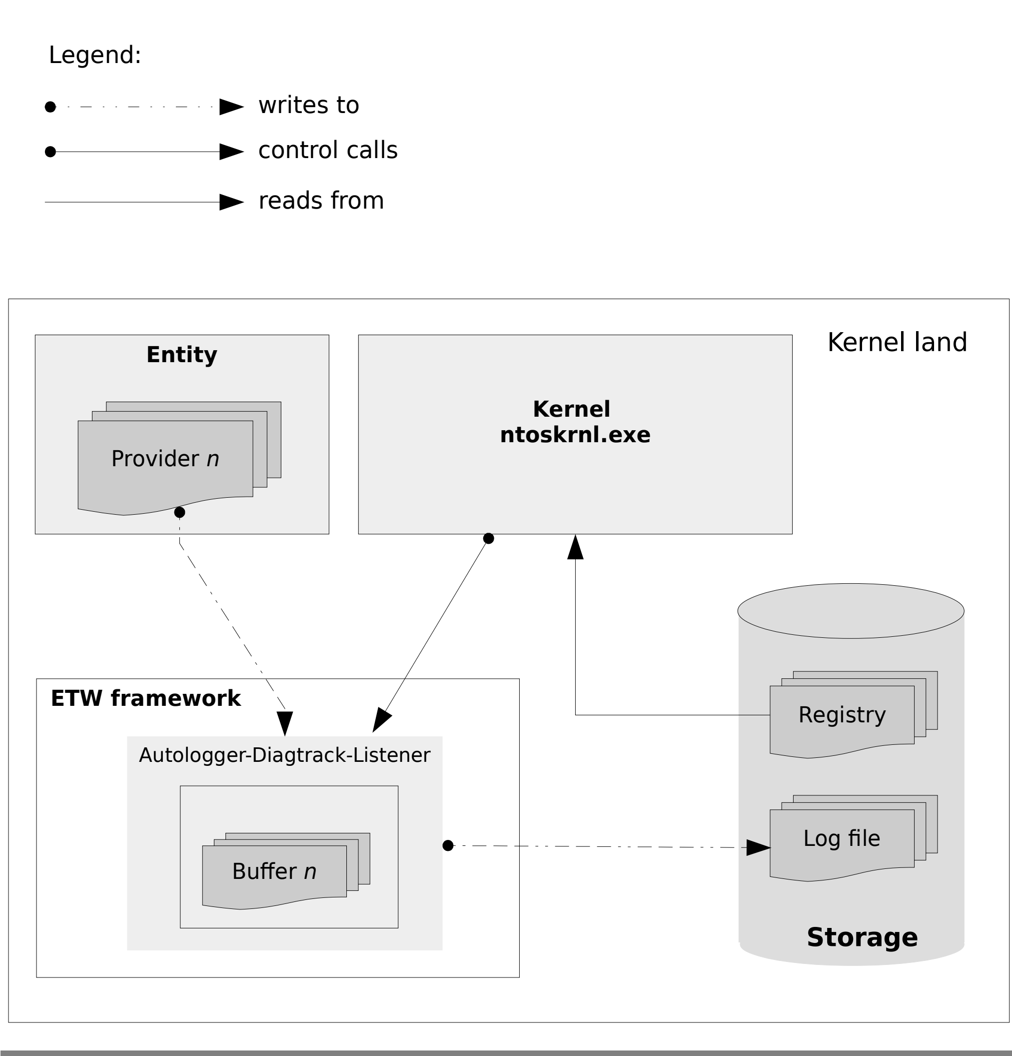


Figure 9: The process for initializing Autologger-Diagtrack-Listener

Once activated, 'Autologger-Diagtrack-Listener' stores logged data in the `%ProgramData%\Microsoft\Diagnosis\ETLLogs\AutoLogger\AutoLogger-Diagtrack-Listener.etl` file (*Log file* in Figure 9). This contents of this file are later consumed by the DiagTrack service (see Section 2.1). 'Autologger-Diagtrack-Listener' logs data until it is deactivated, which takes place when the DiagTrack service is initialized. When at some point the DiagTrack service is terminated, the 'Autologger-Diagtrack-Listener' session is activated and stores logged data in `%\Microsoft\Diagnosis\ETLLogs\ShutdownLogger\AutoLogger-Diagtrack-Listener.etl` (*Log file* in Figure 9).As we previously mentioned, the kernel initializes 'Autologger-Diagtrack-Listener' based on relevant data stored in the registry key `\Registry\Machine\System\CurrentControlSet\Control\WMI\AutoLogger\AutoLogger-Diagtrack-Listener`. Figure 10 depicts the kernel accessing the data stored at this key by invoking the `ZwOpenKey` function.[19] This key stores data, such as the name of the file in which the 'Autologger-Diagtrack-Listener' logs data. Figure 11 depicts some of the data stored at `\Registry\Machine\System\CurrentControlSet\Control\WMI\AutoLogger\AutoLogger-Diagtrack-Listener`. The initialized 'Autologger-Diagtrack-Listener' session can be verified by viewing session operational information with the `windbg` debugger, as depicted in Figure 12.[20]

19 https://msdn.microsoft.com/en-us/library/windows/hardware/ff567014(v=vs.85).aspx [Retrieved: 26/10/2017]
20 https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-wmitrace-logger [Retrieved: 9/7/2017]

```
kd> r
rax=ffff9a0047f533d8 rbx=0000000000000000 rcx=ffff9a0047f53390
rdx=000000000002001f rsi=00000000000000c0 rdi=ffffc7033adcf330
rip=fffff8006dfe0250 rsp=ffff9a0047f53348 rbp=ffff9a0047f53450
 r8=ffff9a0047f53430  r9=0000000000000000 r10=000000007ffffd2
r11=fffffffffff5730a r12=0000000000000000 r13=0000000000000001
r14=ffffc7033ad266a0 r15=ffff9a0047f53990
iopl=0         nv up ei pl nz ac po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000216

nt!ZwOpenKey:
fffff800`6dfe0250 488bc4          mov     rax,rsp
```

```
kd> dt nt!_OBJECT_ATTRIBUTES -r @r8
   +0x000 Length           : 0x30
   +0x008 RootDirectory    : (null)
   +0x010 ObjectName       : 0xffff9a00`47f533d8 "\Registry\Machine\System\CurrentControlSet\
                                            Control\WMI\AutoLogger\AutoLogger-Diagtrack-Listener"
      +0x000 Length           : 0xbe
      +0x002 MaximumLength    : 0xc0
      +0x008 Buffer           : 0xffffc703`3adcf330  "\Registry\Machine\System\CurrentControlSet\
                                            Control\WMI\AutoLogger\AutoLogger-Diagtrack-Listener"
   +0x018 Attributes       : 0x240
   +0x020 SecurityDescriptor : (null)
   +0x028 SecurityQualityOfService : (null)
```

Figure 10: The kernel reading session initialization data from the system's registry

```
KeyPath    \REGISTRY\MACHINE\SYSTEM\ControlSet001\Control\WMI\Autologger\AutoLogger-Diagtrack-Listener

[ValueType]    [ValueName]    [ValueData]
REG_DWORD      BufferSize     40
REG_SZ         FileName       C:\ProgramData\Microsoft\Diagnosis\ETLLogs\AutoLogger\AutoLogger-Diagtrack-Listener.etl
REG_DWORD      FlushTimer     12c
REG_SZ         Guid           {11D8A17B-F2D8-4733-B41B-6F4959ACD701}
REG_DWORD      LogFileMode    8000001
REG_DWORD      MaxFileSize    20
REG_DWORD      Start          1
REG_DWORD      Status         0
```

Figure 11: AutoLogger-Diagtrack-Listener: Session initialization data

We observed that, depending on the configured Telemetry level (see Section 2.1), there is a difference in the number of ETW providers associated with 'Autologger-Diagtrack-Listener'. Table 3 presents this number (column '*ETW Providers*') for each of the Telemetry levels `Security`, `Basic`, `Enhanced` and `Full` (column '*Telemetry Level*'). For each Telemetry level, we issued the PowerShell command `Get-EtwTraceProvider | where {$_.SessionName -match "Autologger-Diagtrack-Listener"} | measure | select Count` in order to observe the number of ETW providers associated with 'Autologger-Diagtrack-Listener'. In addition, the GUIDs of these providers can be obtained by issuing the PowerShell command `Get-EtwTraceProvider | where {$_.SessionName -match "Autologger-Diagtrack-Listener"}`.[21] We emphasize that the number of ETW providers that are active when a given Telemetry level is configured is dynamic. This number may differ between different versions of Windows 10 and different states of the system. A system state involves running processes, installed software, operating system configuration, and so on. We discuss the data logged by the providers presented in Table 3 and its richness at the end of paragraph 'Diagtrack-Listener'.

21 https://docs.microsoft.com/en-us/powershell/scripting/getting-started/cookbooks/working-with-registry-keys?view=powershell-5.1 [Retrieved: 26/10/2017]

```
kd> !wmitrace.logger 0x06
(WmiTrace) LogDump for Logger Id 0x06
    Logger Id 0x06 @ 0xFFFF86074B7FB4C0 Named 'AutoLogger-Diagtrack-Listener'
      CollectionOn        = 1
      LoggerMode          = 0x08800001 ( seq ind )
      HybridShutdown      = persist
      BufferSize          = 64 KB
      BuffersAvailable    = 1
      MinimumBuffers      = 2
      NumberOfBuffers     = 2
      MaximumBuffers      = 24
      EventsLost          = 0
      LogBuffersLost      = 0
      RealTimeBuffersLost = 0
      LastFlushedBuffer   = 0
      MaximumFileSize     = 32
      FlushTimer          = 300 sec
      LoggerThread        = 0xffff86074b830040 (2 context switches)
      PoolType            = NonPaged
      SequenceNumber      = 2
      ClockType           = PerfCounter
      EventsLogged        = 0
      LogFileName         = 'C:\ProgramData\Microsoft\Diagnosis\ETLLogs\AutoLogger\AutoLogger-Diagtrack-Listener.etl'

      Buffer      Address           Cpu RefCnt State
      ===========================================================================================
      Buffer   1: ffff86074b810000 ,  0:   15    General Logging    , Offset:     14960 ,  22% Used
      Buffer   2: ffff86074b820000 ,  0:    0    Free List          , Offset:        72 ,   0% Used
```

Figure 12: AutoLogger-Diagtrack-Listener: Session operational information

Telemetry Level	ETW Providers
`Security`	9
`Basic`	93
`Enhanced`	105
`Full`	112

Table 3: Telemetry levels and number of ETW providers associated with Autologger-Diagtrack-Listener

Diagtrack-Listener The 'Diagtrack-Listener' ETW session is initialized by the DiagTrack service. The DiagTrack service is started in the session initialization step (see Section 1.3.2). Therefore, in order to analyze the initialization of 'Diagtrack-Listener', we analyzed the operation of the DiagTrack service when it is being initialized. This service is started automatically by the system support process named Service Control Manager (see [ERNW WP2], Section 2.3.1).[22] For us to be able to conduct the analysis of DiagTrack, we first stopped the service and then performed the following:

- We created the registry value `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ServicesPipeTimeout` and assigned the value of `86400000 to it`. This value specifies the time period (in milliseconds) over which a service will wait for a debugger to be attached to it. The debugger is used for analyzing the operation of the DiagTrack service in a dynamic manner;
- We changed the value assigned to the registry value `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\DiagTrack\ImagePath` from `svchost.exe` to `utc_myhost.exe`. This is to isolate the DiagTrack service in a separate service host process named `utc_myhost.exe`. We also created a copy of `svchost.exe` and named it `utc_myhost.exe;`
- We configured the DiagTrack service to run in a separate service host process by issuing the `sc config DiagTrack type=own` command. This makes the dynamic analysis focusing on the DiagTrack service possible;
- We created the registry key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\utc_myhost.exe`. We then created the

22 https://msdn.microsoft.com/en-us/library/windows/desktop/ms681957(v=vs.85).aspx [Retrieved: 9/7/2017]

value `Debugger` and assigned a `REG_SZ` value[23] of type `"C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\ntsd.exe" -server npipe:pipe=service_name_debug` to it. This specifies the file-system path to the Microsoft NT Symbolic Debugger debugger. The debugger has to be attached to service host process named `utc_myhost.exe` in the configured time period of `86400000` miliseconds (see above).

```
0:010> r
rax=0000000080000000 rbx=0000000000010000 rcx=000001911ed3db88
rdx=000001911ed368f0 rsi=000001911ed3db50 rdi=000001911ed3db60
rip=00007ffb732e8120 rsp=0000003c4387e408 rbp=0000000000000000
 r8=0000003c4387e680  r9=000000000000001b r10=00007ffb755c15c0
r11=0000003c4387e2c0 r12=0000000000000107 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz ac po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000216

sechost!StartTraceW:
00007ffb`732e8120 4889542410      mov     qword ptr [rsp+10h],rdx ss:0000003c`4387e418=0000000000001078

0:010> du @rdx
00000191`1ed368f0  "Diagtrack-Listener"

0:010> dt nt!_guid @r8+18
ntdll!_GUID
 {4c04b0b4-8f26-11e7-b8f7-080027d84a38}
   +0x000 Data1            : 0x4c04b0b4
   +0x004 Data2            : 0x8f26
   +0x006 Data3            : 0x11e7
   +0x008 Data4            : [8]  "???"
```

Figure 13: The name and GUID of Diagtrack-Listener

After the steps above were completed, we started the DiagTrack service and we attached the Microsoft NT Symbolic Debugger to it. We observed the automatic initialization of 'Diagtrack-Listener' by setting a breakpoint at the `sechost!StartTraceW` function, implemented as part of the DiagTrack service.[24] This function initializes the 'Diagtrack-Listener' ETW session, whose name and GUID are stored in its second and third parameter. Figure 13 depicts the name of the initialized session (i.e., 'Diagtrack-Listener') and the GUID of the session (see the values of the registers *rdx* and *r8* in Figure 13).

After it is initiated, 'Diagtrack-Listener' is activated and delivers logged data to `utc_myhost.exe`. We verified this as follows. We first obtained the address of the kernel structure of type `nt!_WMI_LOGGER_CONTEXT` implementing the 'Diagtrack-Listener' session (*0xffffa700edadd2c0* in Figure 14) by issuing the `!wmitrace.strdump` debugger command. We then printed out the contents of this structure by issuing the `dt nt!_WMI_LOGGER_CONTEXT` debugger command. We observed that the structure field `Consumers` points to the ETW consumer that consumes logged data from the 'Diagtrack-Listener' session. `Consumers` points to a kernel structure of type `nt!_ETW_REALTIME_CONSUMER`. This structure contains a pointer to a kernel structure of type `nt!_EPROCESS`. This structure defines the process implementing the actual consumer, named `ProcessObject`. We then displayed information about this process by issuing the `!process 0xffffa700`ee3e9280 0` debugger command, where `0xffffa700`ee3e9280` is an address at which an instance of a `nt!_EPROCESS` structure is stored. This information reveals the name of the process consuming data delivered from 'Diagtrack-Listener' – `utc_myhost.exe`. 'Diagtrack-Listener' session delivers logged data `utc_myhost.exe` in the form of real-time log feed (see Section 2.1).

23 https://msdn.microsoft.com/en-us/library/windows/desktop/ms724884(v=vs.85).aspx [Retrieved: 26/10/2017]
24 https://msdn.microsoft.com/en-us/library/windows/desktop/aa364117(v=vs.85).aspx [Retrieved: 26/10/2017]

```
LoggerContext Array @ 0xFFFFA700ED441390 [64 Elements]
   [...]
   Logger Id 0x06 @ 0xFFFFA700EDADD2C0 Named 'Diagtrack-Listener'
   [...]

lkd> dt nt!_WMI_LOGGER_CONTEXT 0xFFFFA700EDADD2C0
   +0x000 LoggerId         : 6
   +0x004 BufferSize       : 0x10000
   +0x008 MaximumEventSize : 0xffb8
   [...]
   +0x088 LoggerName       : _UNICODE_STRING "Diagtrack-Listener"
   [...]
   +0x114 InstanceGuid     : _GUID {69112195-9d2e-11e7-b906-080027d84a38}
   [...]
   +0x148 Consumers        : _LIST_ENTRY [ 0xffffa700`edaeed50 - 0xffffa700`edaeed50 ]
   +0x158 NumConsumers     : 1
   [...]
   +0x2c0 ClientSecurityContext : _SECURITY_CLIENT_CONTEXT
   +0x308 TokenAccessInformation : 0xffffe508`af1fe010 _TOKEN_ACCESS_INFORMATION
   +0x310 SecurityDescriptor : _EX_FAST_REF
   +0x318 StartTime        : _LARGE_INTEGER 0x01d3315b`67b6688f
   [...]

lkd> dt nt!_ETW_REALTIME_CONSUMER 0xffffa700edaeed50
   +0x000 Links            : _LIST_ENTRY [ 0xffffa700`edadd408 - 0xffffa700`edadd408 ]
   +0x010 ProcessHandle    : 0xffffffff`80000e0c Void
   +0x018 ProcessObject    : 0xffffa700`ee3e9280 _EPROCESS
   [...]

lkd> !process 0xffffa700`ee3e9280 0
PROCESS ffffa700ee3e9280
    SessionId: 0  Cid: 0f84    Peb: 12366bf000  ParentCid: 0f48
FreezeCount 1
    DirBase: 661a1000  ObjectTable: ffffe508acae0f00  HandleCount: <Data Not Accessible>
    Image: utc_myhost.exe
```

Figure 14: Diagtrack-Listener delivering logged data to utc_myhost.exe

We now identify the ETW providers associated with the 'Diagtrack-Listener' ETW session. In addition to the activities described above, the association of ETW providers to 'Diagtrack-Listener' is performed as part of the initialization process of this session. In order to analyze this activity, we set a breakpoint at the `sechost!EnableTraceEx2` function. This function is used for associating an ETW provider with 'Diagtrack-Listener'. Its second parameter contains the GUID of the ETW provider.[25] In order to automate the extraction of the GUIDs of ETW providers from the second parameter of `EnableTraceEx2`, we developed a script for the `windbg` debugger. The script is configurable and extensible such that it extracts parameter values passed to functions specified by users. Therefore, this script plays the role of an API monitor. The main module of the API monitor is placed in the Appendix, section 'API Monitor: Main Module'. The extension of the API monitor enabling the extraction of GUIDs from the second parameter of `EnableTraceEx2` is placed in the Appendix, section 'API Monitor: Extension EnableTraceEx2'. The execution of an extension requires the specification of a configuration file. An example of such a file is placed in the Appendix, section 'API Monitor: Extension Definition'.

We emphasize that some of the GUIDs that are passed to `EnableTraceEx2` are stored in the `%ProgramData%\Microsoft\Diagnosis\DownloadedSettings\utc.app.json` file. We verified this by observing that the DiagTrack service reads GUIDs of ETW providers stored in this file. As mentioned above, this function initializes the ETW providers uniquely identified by the GUIDs.

We observed that some GUIDs passed to `EnableTraceEx2` are not stored in the `utc.app.json` file. When Telemetry is configured at the Telemetry level `Full` (see Section 2.1), the following ETW provider GUIDs, among others, were passed to `EnableTraceEx2`: `22fb2cd6-0e7b-422b-a0c7-`

25 https://msdn.microsoft.com/en-us/library/windows/desktop/dd392305(v=vs.85).aspx [Retrieved: 26/10/2017]

2fad1fd0e716, 331c3b3a-2005-44c2-ac5e-77220c37d6b4, 96f4a050-7e31-453c-88be-9634f4e02139, dbe9b383-7cf3-4331-91cc-a3cb16a3b538, 2839ff94-8f12-4e1b-82e3-af7af77a450f, and a19fdc69-a626-5289-be4d-1f508a8c9a3b. These GUIDs are hardcoded in the diagtrack.dll library file (see Section 2.1). We identified the GUIDs in diagtrack.dll by searching the file for patterns of GUID strings (see Section 1.3.1) using the yara tool. The expression we used for the search operation is placed in the Appendix, section 'Yara Rule'. The expression uses the byte array representations of GUID strings (see Section 1.3.1). Same as for the 'Diagtrack-Listener' session, the GUIDs of the ETW providers assigned to the 'Autologger-Diagtrack-Listener' session (see paragraph 'Autologger-Diagtrack-Listener'), are stored in the utc.app.json file and hardcoded in the diagtrack.dll library file.

We observed that, depending on the configured Telemetry level (see Section 2.1), there is a difference in the number of ETW providers associated with 'Diagtrack-Listener'. Table 4 presents this number (column '*ETW Providers*') for each of the Telemetry levels Security, Basic, Enhanced, and Full (column '*Telemetry Level*'). For each Telemetry level, we issued the PowerShell command `Get-EtwTraceProvider | where {$_.SessionName -match "Diagtrack-Listener"} | measure | select Count` in order to observe the number of ETW providers associated with 'Diagtrack-Listener'. In addition, the GUIDs of these providers can be obtained by issuing the PowerShell command `Get-EtwTraceProvider | where {$_.SessionName -match "Diagtrack-Listener"}`.[26] We emphasize that the number of ETW providers that are active when a given Telemetry level is configured is dynamic (see paragraph 'Autologger-Diagtrack-Listener').

Telemetry Level	ETW Providers
Security	4
Basic	410
Enhanced	418
Full	422

Table 4: Telemetry levels and number of ETW providers associated with Diagtrack-Listener

The ETW providers presented in Table 3 and Table 4 are associated with the ETW sessions Autologger-Diagtrack-Listener and Diagtrack-Listener by the kernel and the Diagtrack service, respectively (see Section 1.3.1). As we mentioned earlier, the GUIDs of these providers are stored in the %ProgramData%\Microsoft\Diagnosis\DownloadedSettings\utc.app.json file. This file is sent by Microsoft and is periodically renewed (see Section 2.3). This indicates that what ETW providers are associated with the Autologger-Diagtrack-Listener and Diagtrack-Listener ETW sessions may change over time. We emphasize that it is technically feasible for entities other than the kernel and the Diagtrack service to register ETW providers and associate them to the Autologger-Diagtrack-Listener and Diagtrack-Listener ETW sessions.

Observing the data logged by the ETW providers presented in Table 3 and Table 4 and evaluating the richness of this data can be performed in several ways. Utilities, such as wevtutil (see [ERNW WP2], Section 2.5.3), display data logged by specific ETW providers under specific conditions. For example, wevtutil displays data only from ETW providers that have names assigned to them. Alternatively, the logging performed by the ETW providers presented in Table 3 and Table 4 can be reverse-engineered from the implementation of the entities registering the providers (see Section 1.3.1). However, this may be a time-consuming and practically challenging process. Therefore, an external framework for monitoring the activites of the Telemetry component is needed. This includes observing the data logged by the ETW providers presented in Table 3 and Table 4. We propose such a framework in Section 2.4.

26 https://docs.microsoft.com/en-us/powershell/module/eventtracmancmdlets/get-etwtraceprovider?view=win10-ps [Retrieved: 26/10/2017]

Additional sources We analyzed the implementation of the DiagTrack service and observed the existence of functions for executing external executable files. We observed that these executables are executed by DiagTrack for the purpose of obtaining telemetry data. This data is in addition to the data provided by the 'Autologger-Diagtrack-Listener' and 'Diagtrack-Listener' ETW sessions discussed above. An example is the function `Microsoft::Diagnostics::CrunExeWithArgsAction::CrunExeWithArgsAction`. This function is implemented as part of the `Microsoft::Diagnostics::CrunExeWithArgsAction` data structure.[27] We then searched for executable names and file extensions among the strings hardcoded in `diagtrack.dll`. We identified a number of executables that may be executed by DiagTrack. We refer to these executables as external executables. In the Appendix, section 'External Executables', for each identified external executable, we present in tabular form its name (column '*Executable*') and a brief description (column '*Description*'). We provide description information only when such information is available from official Microsoft sources.

In addition to the external executables, we observed that the DiagTrack service executes functions for the purpose of obtaining Telemetry data. These functions are implemented in library DLL files, which we refer to as external functions. An example external function is `MiniDumpWriteDump`. It is implemented in the library DLL file `dbghelp.dll`.[28] Figure 15 depicts, in the form of pseudo-code, the execution of this function as implemented in `diagtrack.dll`.

```
// Loads the specified module into the address space
HMODULE *__fastcall Microsoft::Diagnostics::Utils::LoadSystemLibrary(HMODULE *HandleToLoadedModule_1)
{
  HMODULE *HandleToLoadedModule; // rbx@1

  HandleToLoadedModule = HandleToLoadedModule_1;
  *HandleToLoadedModule = LoadLibraryExW(L"dbghelp.dll", [...]);
  return HandleToLoadedModule;
}
[...]

// Retrieves the address of an exported function
Microsoft::Diagnostics::Utils::LoadSystemLibrary((HMODULE *)ExitCode);
[...]
    addressFunction = GetProcAddress(*(HMODULE *)ExitCode, "MiniDumpWriteDump");
    [...]
```

Figure 15: Pseudo-code of execution of an external function

We emphasize that the analysis of how, when, and what, external executables and functions are executed is out of the scope of this work. We note that we could not observe actual executions of these executables and functions. External functions and functions implementing the execution of external executables were not invoked over a considerable time interval.

We observed that the execution of the external executables and functions is triggered by the Microsoft's back-end infrastructure. This takes place over a network interface established between this infrastructure and the DiagTrack service (see [ERNW WP2], Section 2.4.2). We discuss this interface in more detail in Section 2.3. When an external executable or function needs to be executed, a configuration file is sent to the DiagTrack service. This file is stored at `%\Microsoft\Diagnosis\DownloadedScenarios\WINDOWS.DIAGNOSTICS.xml`. The file contains, for example, filenames of executables and parameters. Figure 16 depicts data stored in `WINDOWS.DIAGNOSTICS.xml` for configuring the execution of the `icacls.exe` external executable.

The external executables and functions are executed only if specific conditions are fulfilled. These conditions are defined by a logic implemented as part of the Microsoft's back-end infrastructure. This logic is unknown to us. Therefore, we can only assume that the conditions for executing the external executables and functions take into account telemetry data.

27 In this work, we use the scope operator `::` when referring to functions declared as part of data structures.
28 https://msdn.microsoft.com/en-us/library/windows/desktop/ms680360(v=vs.85).aspx [Retrieved: 26/10/2017]

```
[...]
<action actionname="H_WpSystem_ACLs" ignorefailure="1">
    <runexewithargsaction>
        <exename>%windir%\system32\icacls.exe</exename>
        <commandline>H:\WpSystem</commandline>
        <maximumruntimems>60000</maximumruntimems>
    </runexewithargsaction>
</action>
[...]
```

Figure 16: Data for configuring the execution of icacls.exe

The process for executing the external executables and functions remains to be analyzed using dynamic analysis methods. For us to conduct such an analysis, we have to be able to trigger the execution of these executables and functions. We emphasize that this is possible only if we can generate network traffic specifically crafted for this purpose. The traffic has to be the same as traffic originating from the Microsoft's back-end infrastructure. This would require knowledge on the content and format of this traffic. Obtaining such knowledge has proved to be challenging, since we were unable to observe such traffic. We note the possibility of additional challenges. This involves, for example, handling the scenario where DiagTrack processes only network traffic originating from the Microsoft's back-end infrastructure, which it verifies cryptographically (see Section 2.3).

We observed that specific external executables (controlled by regular expressions) and functions may be executed with no parameters, with parameters set to hard-coded values, or with parameters set to dynamically generated values. We analyzed the functionalities of the external executables and functions with the goal of identifying possibilities for arbitrary code execution achieved by code injection in parameter values. We focused on executables and functions whose functionalities and possible parameter values are documented. Based on our analysis, we observed that all callable external executables and functions do not allow for obvious arbitrary code execution. The other executables and functions may, in theory, allow for arbitrary code execution. However, this remains to be verified by analyzing the implementation and operation of these executables and functions using dynamic and static analysis methods. Any external executable or function may be potentially used as part of a chain of malicious activities resulting in a security breach. An example is the executable `icacls.exe`, which may be used to display access control lists (see the Appendix, section 'External Executables').

2.2.1 Telemetry Data: Collection and Processing

In this section, we discuss the actual telemetry data logged by the 'Autologger-Diagtrack-Listener' and 'Diagtrack-Listener' ETW sessions (see Section 2.2). We focus on the way in which this data is collected and processed by the DiagTrack service (see Section 2).

As we mentioned in Section 2.2, the 'Autologger-Diagtrack-Listener' ETW session stores logged data in the files `%ProgramData%\Microsoft\Diagnosis\ETLLogs\AutoLogger\AutoLogger-Diagtrack-Listener.etl and %ProgramData%\Microsoft\Diagnosis\ETLLogs\ShutdownLogger\AutoLogger-Diagtrack-Listener.etl`. The content of these files is in a binary format and can be viewed using the `Windows Performance Analyzer` tool.[29] The log files named `AutoLogger-Diagtrack-Listener.etl` can be inspected only if the DiagTrack service is prevented from starting at system start-up after once being started and operational. This is for the files to be present on the file-system. We observed that at the time it is started, the DiagTrack service consumes the data stored in the `AutoLogger-Diagtrack-Listener.etl` files and deletes them. In the registry key `\Registry\Machine\System\CurrentControlSet\Control\WMI\AutoLogger\AutoLogger-Diagtrack-Listener`, the values `MaxFileSize` and `LogFileMode` indicate the maximum size of these files and how the system manages them. The default value of `MaxFileSize` is 0x20 – the maximum

29 https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/trace-log [Retrieved: 26/10/2017]

file size is 32 Megabyte (MB). The default value of `LogFileMode` is 0x08000001 – this indicates that logging is stopped once the file has reached the maximum size.[30]

Figure 17 depicts the flow of telemetry data delivered by the 'Diagtrack-Listener' ETW session. In contrast to 'Autologger-Diagtrack-Listener', the 'Diagtrack-Listener' session does not store logged telemetry data in files, but delivers data to the DiagTrack service in the form of a real-time log feed (see Section 2.2). In order to view this data, we issued the `windbg` debugger command `!wmitrace.logdump`.[31] Figure 18 depicts the output of this command. We emphasize that the data viewed in this manner is only the data logged at the time point when the `windbg` command is issued. The data is temporarily stored in the kernel buffers managed by the 'Diagtrack-Listener' ETW session (see Section 1.3.1). In order to view all of the data that 'Diagtrack-Listener' delivers to the DiagTrack service, the 'Diagtrack-Listener' session has to be reconfigured to store logged data on the file-system. This can be achieved by using the `Performance Monitor` utility (see [ERNW WP2], Section 2.4.2, Figure 14, tab '*Trace Session*').

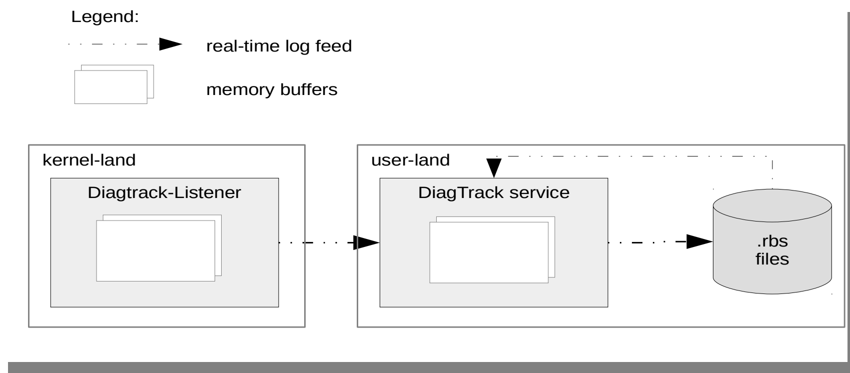


*Figure 17: Flow of telemetry data (*the 'Diagtrack-Listener' ETW session*)*

```
lkd> !wmitrace.logdump 0x6
(WmiTrace) LogDump for Logger Id 0x06
Found Buffers: 4 Messages: 17, sorting entries
[0]16AC.0D48::  131497929216632582 [Microsoft-Windows-Store/StoreLaunching/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "correlationVectorRoot": "VEG25Q9DRkmp/Qc/"}
[0]16AC.11BC::  131497929217518987 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "PartB_OutgoingServiceRequest": {"operationNam
[1]16AC.15BC::  131497929217696344 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "PartB_OutgoingServiceRequest": {"operationNam
[0]16AC.09D4::  131497929217884177 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "PartB_OutgoingServiceRequest": {"operationNam
[1]16AC.1388::  131497929217888986 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "PartB_OutgoingServiceRequest": {"operationNam
[1]16AC.0E18::  131497929218038399 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "PartB_OutgoingServiceRequest": {"operationNam
[1]16AC.0E18::  131497929218059892 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "PartB_OutgoingServiceRequest": {"operationNam
[1]16AC.0E18::  131497929218080393 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.5", "PartB_OutgoingServiceRequest": {"operationNam
[0]16AC.15BC::  131497929218227069 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "PartB_OutgoingServiceRequest": {"operationNam
[1]16AC.15BC::  131497929218250143 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.0", "PartB_OutgoingServiceRequest": {"operationNam
[1]16AC.1474::  131497929218260270 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.8", "PartB_OutgoingServiceRequest": {"operationNam
[0]16AC.15BC::  131497929218265140 [Microsoft-Windows-Store/OutgoingServiceRequest/]{"__TlgCV__": "VEG25Q9DRkmp/Qc/.9", "PartB_OutgoingServiceRequest": {"operationNam
[0]15F4.0CEC::  131497929222567726 [Microsoft.Windows.FaultReporting/AppCrashEvent/]{"ProcessId": 5804, "ProcessCreateTime": "09/13/2017 17:33:26.181", "ExceptionCode
[1]15F4.0CEC::  131497929222939394 [Microsoft.Windows.WindowsErrorReporting/WerReportCreate/]{"HResult": "0x00000000", "ReportId": "a60574c6-98a9-11e7-b901-080027d84a
[1]15F4.0CEC::  131497929223823719 [Microsoft.Windows.WindowsErrorReporting/WerReportSubmit/]{"ReportId": "a60574c6-98a9-11e7-b901-080027d84a38", "IntegratorReportId"
[0]15F4.0CEC::  131497929224346655 [Microsoft.Windows.WindowsErrorReporting/WerReportQueued/]{"ReportId": "a60574c6-98a9-11e7-b901-080027d84a38"}
[0]15F4.0CEC::  131497929224388524 [Microsoft.Windows.WindowsErrorReporting/WerEvent/]{"BucketId": "", "BucketTable": "", "BucketHash": "", "EventName": "MoAppCrash",
Total of 17 Messages from 4 Buffers
```

Figure 18: Telemetry data delivered by Diagtrack-Listener

The telemetry data temporarily stored in the kernel buffers is periodically flushed to the DiagTrack service. The flushed data is stored in memory buffers allocated as part of the DiagTrack's memory. After the DiagTrack service has the telemetry data in its memory buffers, it converts the data into JavaScript Object Notation (JSON) format. In this way, the service prepares the data for sending to Microsoft (see Section 2). DiagTrack then stores the converted data in a separate memory segment by invoking the `Microsoft::Diagnostics::CringBufferEventStore::AddData` function. Therefore, by observing the data processed by this function starting at a given point in time, we can view all of the telemetry data that is ultimately sent to Microsoft.

30 https://msdn.microsoft.com/en-us/library/windows/desktop/aa363687(v=vs.85).aspx [Retrieved:]
31 https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-wmitrace-logdump [Retrieved: 9/8/2017]

The telemetry data processed in `AddData`, already into JSON format, can be viewed by displaying the data stored at the memory address pointed by the function's second parameter. To automate this activity, we developed an extension of the API monitor we presented in Section 2.2. This extension is placed in the Appendix, section 'API Monitor: Extension AddData'. Figure 19 depicts a snippet of the output of this extension.

```
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:01:58.9621247Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:01:58.9621300Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:01:58.9621323Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:01:58.9621342Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Microsoft.Windows.ContentDeliveryManager.ProcessCreativeEvent","time":"2017-09-13T18:02:00
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Microsoft.Windows.FaultReporting.AppCrashEvent","time":"2017-09-13T18:02:05.9000942Z","epo
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Microsoft.Windows.WindowsErrorReporting.WerEvent","time":"2017-09-13T18:02:06.0752897Z","e
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Microsoft.Windows.ContentDeliveryManager.ProcessCreativeEvent","time":"2017-09-13T18:02:13
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:02:59.4689844Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:02:59.4689993Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:02:59.4690012Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:02:59.4690031Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:02:59.4690046Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:02:59.4690061Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:02:59.4690076Z","epoch":
@@AddDataToRingBufferA@@ {"ver":"2.1","name":"Win32kTraceLogging.AppInteractivitySummary","time":"2017-09-13T18:04:04.8102943Z","epoch":
```

Figure 19: Telemetry data processed in Microsoft::Diagnostics::CRingBufferEventStore::AddData

We observed that `AddData` invokes the `deflate_slow` function. This function compresses telemetry data in JSON format (see Figure 19) using the `DEFLATE` algorithm.[32] Figure 20 depicts compressed telemetry data, which is stored at a location in memory pointed to by a parameter of `deflate_slow`.

```
0:011> db poi(25a0640afd0+ 0x10) L0x100
0000025a`091362b0  5d 52 4d 6f db 30 0c bd-0f d8 8f a8 ae 8b 5c 7d  ]RMo.0........\}
0000025a`091362c0  59 92 75 f3 27 30 60 dd-0e ed 5a 60 97 41 95 e8  Y.u.'0`...Z`.A..
0000025a`091362d0  d5 68 12 67 92 92 06 28-f2 df 27 a7 d9 50 4c 07  .h.g...(..'..PL.
0000025a`091362e0  59 34 f9 1e c9 47 be a2-03 04 64 10 2b 28 5a a1  Y4...G....d.+(Z.
0000025a`091362f0  ad dd 40 36 6e 26 17 e6-38 8f a9 78 98 b6 7e 7e  ..@6n&..8..x..~~
0000025a`09136300  89 c5 f7 9d b7 09 8a 6f-c1 3d 41 4c c1 a6 39 14  .......o.=AL..9.
0000025a`09136310  1d 24 70 69 9a b7 19 98-a6 33 90 11 aa 30 a9 30  .$pi.....3...0.0
[...]
```

Figure 20: Compressed telemetry data

We observed that `AddData` invokes functions for storing the telemetry data compressed by `deflate_slow` in the following memory-mapped files:[33] `%ProgramData%\Microsoft\Diagnosis\Events_CostDeferred.rbs`, `%ProgramData%\Microsoft\Diagnosis\Events_Normal.rbs`, `%ProgramData%\Microsoft\Diagnosis\Events_NormalCritical.rbs`, and `%ProgramData%\Microsoft\Diagnosis\Events_Realtime.rbs` (see '.rbs files' in Figure 17). Figure 21 depicts a

```
[...]
000028b0: 0000 0e02 0000 0100 0000 005d 524d 6fdb  ...........]RMo.
000028c0: 300c bd0f d88f a8ae 8b5c 7d59 9275 f327  0........\}Y.u.'
000028d0: 3060 dd0e ed5a 6097 4195 e8d5 6812 6792  0`...Z`.A...h.g.
000028e0: 9206 28f2 df27 a7d9 504c 0759 34f9 1ec9  ..(..'..PL.Y4...
000028f0: 47be a203 0464 102b 285a a1ad dd40 366e  G....d.+(Z...@6n
00002900: 2617 e638 8fa9 7898 b67e 7e89 c5f7 9db7  &..8..x..~~.....
00002910: 098a 6fc1 3d41 4cc1 a639 141d 2470 699a  ..o.=AL..9..$pi.
00002920: b719 98a6 3390 11aa 30a9 30a3 7754 1946  ....3...0.0.wT.F
00002930: 4d29 0a49 9994 82fd c851 ee3e c7dc aa2f  M).I.....Q.>.../
[...]
```

Figure 21: A snippet of the content of Events_Realtime.rbs

32 https://tools.ietf.org/html/rfc1951 [Retrieved: 26/10/2017]

33 https://docs.microsoft.com/en-us/dotnet/standard/io/memory-mapped-files

snippet of the content of the `Events_Realtime.rbs` file, which we displayed using the `xxd` utility. This file contains the same data that `deflate_slow` had compressed (see Figure 20).

We observed that the DiagTrack service implements the sending of telemetry data to Microsoft in the `WINHTTP!WinHttpWriteData` function. The sent data can be viewed by setting a breakpoint at this function and displaying the contents of its second parameter. As an example, such data is depicted in Figure 22. The data sent to Microsoft can be found in the previously mentioned memory-mapped files. For example, Figure 23 depicts telemetry data stored in `Events_NormalCritical.rbs`, which is identical to the data depicted in Figure 22. This shows that the telemetry data originating from the kernel buffers managed by the 'Diagtrack-Listener' ETW session, and later processed in the `AddData` function, is the actual data sent to Microsoft.

```
0:009> db 00000082`8a77e060
00000082`8a77e060  75 52 5d 6f db 20 14 7d-9f b4 3f c1 73 6c 81 8d  uR]o. .}..?.sl..
00000082`8a77e070  09 f1 5b 12 3b 6a a5 6d-9d e6 b4 93 f6 52 51 fb  ..[.;j.m.....RQ.
00000082`8a77e080  26 45 25 c6 03 9c 66 ab-f2 df 07 f9 68 d3 26 7b  &E%...f.....h.&{
00000082`8a77e090  b1 e1 9e 7b ee c7 e1 bc-a0 35 18 94 a3 24 26 68  ...{.....5...$&h
00000082`8a77e0a0  80 5a b1 02 7f 99 83 9a-2a 09 ad ab fe b4 ee 11  .Z......*.......
00000082`8a77e0b0  9c ac e3 2b 10 c6 4d 40-b8 fb cc e7 39 b9 cb 4b  ...+..M@....9..K
00000082`8a77e0c0  30 19 46 78 14 91 d1 9c-64 79 32 ca 53 16 d3 6c  0.Fx....dy2.S..l
00000082`8a77e0d0  44 79 46 7f f9 2c e8 74-fd e8 d3 28 27 98 30 16  DyF..,.t...('.0.
```

Figure 22: Telemetry data being sent to the Microsoft's back-end infrastructure

```
[...]
0000cdb0: 0000 0000 0037 0200 0001 0000 0000 7552  .....7........uR
0000cdc0: 5d6f db20 147d 9fb4 3fc1 736c 818d 09f1  ]o. .}..?.sl....
0000cdd0: 5b12 3b6a a56d 9de6 b493 f652 51fb 2645  [.;j.m.....RQ.&E
0000cde0: 25c6 039c 66ab f2df 07f9 68d3 267b b1e1  %...f.....h.&{..
0000cdf0: 9e7b eec7 e1bc a035 1894 a324 2668 805a  .{.....5...$&h.Z
0000ce00: b102 7f99 839a 2a09 adab feb4 ee11 9cac  ......*.........
0000ce10: e32b 10c6 4d40 b8fb cce7 39b9 cb4b 3019  .+..M@....9..K0.
0000ce20: 4678 1491 d19c 6479 32ca 5316 d36c 4479  Fx....dy2.S..lDy
0000ce30: 467f f92c e874 fde8 d328 2798 3016 8816  F..,.t...('.0...
0000ce40: 7e7f eb57 2827 03b4 5062 6951 9e64 7c80  ~..W('..PbiQ.d|.
0000ce50: b43f a09f b26d f4b3 45e1 7ab7 1b8a e018  .?...m..E.z.....
0000ce60: c784 a6a3 d47f 398e c5aa 6174 6120 3696  ......9...ata 6.
0000ce70: dc1b 5020 2cc4 6488 8798 4509 c634 f4dc  ..P ,.d...E..4..
[...]
```

Figure 23: Telemetry data stored in Events_NormalCritical.rbs

As we mentioned, the `DEFLATE` compressed data is also stored in memory-mapped files. We developed a proof of concept to decompress a snippet of the stored data. The code of this proof of concept, implemented in the Python3 programming language, is placed in the Appendix, section 'RBS Decompression Script'.[34] Figure 24 depicts decompressed telemetry data.

```
⇒  ./decompress_rbs_files.py Events_NormalCritical.rbs
{"ver":"2.1","name":"TelClientSynthetic.HeartBeat_5","time":"2017-09-07T09:49:12.4269452Z","epoch":"43405810","seqNum":1,"fl
ags":258,"os":"Windows","osVer":"10.0.14393.1480.amd64fre.rs1_release.170706-2004","ext":{"utc":{"cat":140737488355328,"flag
s":100663856},"device":{"localId":"s:08DBAC8A-AEE5-4D8B-83BE-6A6588DFBD3F","deviceClass":"Windows.Desktop"}},"data":{"Previo
usHeartBeatTime":"2017-09-07T09:22:42.0828031Z","EtwDroppedCount":0,"ConsumerDroppedCount":0,"DecodingDroppedCount":0,"Throt
tledDroppedCount":0,"DbDroppedCount":0,"EventSubStoreResetCounter":0,"EventSubStoreResetSizeSum":0,"CriticalOverflowEntersCo
unter":0,"EnteringCriticalOverflowDroppedCounter":0,"UploaderDroppedCount":0,"InvalidHttpCodeCount":0,"LastInvalidHttpCode":
0,"MaxInUseScenarioCounter":0,"LastEventSizeOffender":"","SettingsHttpAttempts":4,"SettingsHttpFailures":0,"VortexHttpAttemp
ts":10,"EventsUploaded":44,"DbCriticalDroppedCount":0,"VortexHttpFailures4xx":0,"VortexHttpFailures5xx":0,"VortexFailuresTim
eout":0,"CensusTaskEnabled":1,"CensusExitCode":0,"CensusStartTime":"2017-09-07T08:45:21.506","FullTriggerBufferDroppedCount"
:0}}
```

Figure 24: Decompressed telemetry data stored in Events_NormalCritical.rbs

We now discuss the persistence of telemetry data delivered by the 'Diagtrack-Listener' ETW session. Before being delivered to the DiagTrack service, this data is temporarily stored in the kernel buffers managed by the

34 https://zlib.net/ [Retrieved: 26/10/2017]

‘Diagtrack-Listener’ ETW session (see Section 1.3.1). Figure 25 depicts properties of ‘Diagtrack-Listener’ related to the persistence of logged data. The variable `BufferSize` stores the size of the kernel buffers. The `MaximumBuffers` variable stores the maximum number of kernel buffers that can be managed by the ETW session. The `FlushTimer` variable stores the time (in miliseconds) over which the data is kept in the kernel buffers before it is flushed. The `LoggerMode` variable stores the type of ETW session – 0x8800110 indicates that the ‘Diagtrack-Listener’ ETW session delivers log data in real-time.

```
lkd> dt nt!_WMI_LOGGER_CONTEXT 0xFFFFCC02667F0040
[...]
   +0x004 BufferSize        : 0x10000
[...]
   +0x00c LoggerMode        : 0x8800110
[...]
   +0x0e0 FlushTimer        : 0x12c
[...]
   +0x0f4 BuffersAvailable  : 0n4
   +0x0f8 NumberOfBuffers   : 0n4
   +0x0fc MaximumBuffers    : 6
[...]
```

Figure 25: Properties of ‘Diagtrack-Listener’

The telemetry data originating from the kernel buffers managed by ‘Diagtrack-Listener’ is received by the DiagTrack service by invoking a callback function when logged events are available. As mentioned in this section, the DiagTrack service then stores received telemetry data in memory-mapped files. In the context of the DiagTrack service, telemetry data is managed by the service itself by applying memory-management practices at application level. We observed that the memory-mapped files are of a fixed, relatively small size (in the order of kilobyte (KB)), storing several hundred events. New telemetry data overwrites data already stored in these files if the maximum file size is reached.

2.3 Telemetry: Network Interface

In this section, we discuss the way in which logged telemetry data is sent to Microsoft. We focus our discussion on the network interface between the DiagTrack service and this infrastructure (see [ERNW WP2], Section 2.4.2).

We analyzed the network interface in focus using the `Microsoft Message Analyzer` utility. This utility can be configured to perform network traffic sniffing by enabling the provider with GUID `2ed6006e-4729-4609-b423-3ee7bcd678ef`. This enabled us to analyze the network data exchanged between the DiagTrack service and Microsoft. Figure 26 depicts a snippet of the output of the `Microsoft Message Analyzer`. It shows that DiagTrack and Microsoft communicate over a Transport Layer Security (TLS)-protected network interface, exchanging Hypertext Transfer Protocol (HTTP)-formatted messages (see column ‘*Module*’ and ‘*Summary*’ in Figure 26).[35]

Source	Destination	Module	Summary
192.168.0.48	192.168.0.1	DNS	Query Operation, QResult: NoError, Query ID: 0xF84F, OpCode: NoError, Query Name: v10.vortex-win.data.microsoft.com,
192.168.0.48	192.168.0.1	DNS	Query Operation, QResult: NoError, Query ID: 0x365C, OpCode: NoError, Query Name: v10.vortex-win.data.microsoft.com,
192.168.0.1	192.168.0.48	DNS	Response, RCode: NoError, Query ID: 0xF84F
192.168.0.1	192.168.0.48	DNS	Response, RCode: NoError, Query ID: 0x365C, Answers: [40.77.226.250]
192.168.0.48	db5.vortex.data.microsoft.com.akadns.net	TCP	Flags: ......S., SrcPort: 51496, DstPort: HTTPS(443), Length: 0, Seq Range: 2264009943 - 2264009944, Ack: 0, Win: 64
db5.vortex.data.microsoft.com.akadns.net	192.168.0.48	TCP	Flags: ...A..S., SrcPort: HTTPS(443), DstPort: 51496, Length: 0, Seq Range: 612108317 - 612108318, Ack: 2264009944,
db5.vortex.data.microsoft.com.akadns.net	192.168.0.48	TCP	Flags: ...A..S., SrcPort: HTTPS(443), DstPort: 51496, Length: 0, Seq Range: 612108317 - 612108318, Ack: 2264009944,
192.168.0.48	db5.vortex.data.microsoft.com.akadns.net	TCP	Flags: ...A...., SrcPort: 51496, DstPort: HTTPS(443), Length: 0, Seq Range: 2264009944 - 2264009944, Ack: 612108318,
192.168.0.48	db5.vortex.data.microsoft.com.akadns.net	TLS	Records: [Handshake: [Client Hello]]
db5.vortex.data.microsoft.com.akadns.net	192.168.0.48	TLS	Records: [Handshake: [Server Hello, Certificate, Server Key Exchange, Server Hello Done]]
db5.vortex.data.microsoft.com.akadns.net	192.168.0.48	TCP	Flags: ...A...., SrcPort: HTTPS(443), DstPort: 51496, Length: 1400, Seq Range: 612109718 - 612111118, Ack: 22640101
db5.vortex.data.microsoft.com.akadns.net	192.168.0.48	TCP	Flags: ...AP..., SrcPort: HTTPS(443), DstPort: 51496, Length: 967, Seq Range: 612111118 - 612112085, Ack: 226401016
192.168.0.48	db5.vortex.data.microsoft.com.akadns.net	TLS	Records: [Handshake: [Client Key Exchange], ChangeCipherSpec, Handshake(Encrypted)]
db5.vortex.data.microsoft.com.akadns.net	192.168.0.48	TLS	Records: [ChangeCipherSpec, Handshake(Encrypted)]

Figure 26: A snippet of the output of Microsoft Message Analyzer

35 https://msdn.microsoft.com/en-us/library/windows/desktop/aa380516(v=vs.85).aspx [Retrieved: 26/10/2017]

In order to characterize the network traffic exchanged between DiagTrack and Microsoft, we used the `Moloch` framework.[36] The DiagTrack service was running on Windows 10, without additional software installed. We monitored the network interface of DiagTrack for 48 hours and observed that within this time period, connections to the IP addresses `40.77.226.249` and `40.77.226.250` were established. The DiagTrack service may establish network connections to other hosts that are part of Microsoft's back-end infrastructure. Multiple names of such hosts are hardcoded in the `diagtrack.dll` library file. Table 5 depicts the names (column '*Hostname*'), the IP addresses (column '*IP address*'), and the geographical location of these hosts (column ' *Location*').

Hostname	**IP address**	**Location**
geo.settings-win.data.microsoft.com.akadns.net, db5-eap.settings-win.data.microsoft.com.akadns.net, settings-win.data.microsoft.com, db5.settings-win.data.microsoft.com.akadns.net, asimov-win.settings.data.microsoft.com.akadns.net	40.77.226.249	Ireland, Dublin
db5.vortex.data.microsoft.com.akadns.net, v10-win.vortex.data.microsoft.com.akadns.net, geo.vortex.data.microsoft.com.akadns.net, v10.vortex-win.data.microsoft.com	40.77.226.250	Ireland, Dublin
us.vortex-win.data.microsoft.com	13.92.194.212	Virginia (US), Boston
eu.vortex-win.data.microsoft.com	52.178.38.151	Netherlands, Amsterdam
vortex-win-sandbox.data.microsoft.com	52.229.39.152	California (US), Los Angeles
alpha.telemetry.microsoft.com	52.183.114.173	California (US), Los Angeles
oca.telemetry.microsoft.com	13.78.232.226	Wyoming (US), Cheyenne

Table 5: Information on hosts hardcoded in diagtrack.dll

Figure 27 and Figure 28 depict the frequency and intensity of the communication between DiagTrack and the Microsoft's back-end infrastructure over a 3-hour time interval. DiagTrack communicated with the hosts with IP addresses `40.77.226.250` (see Figure 27) and `40.77.226.249` (see Figure 28). The dark grey bar shows the amount of sent data. The light grey bar shows the amount of received data. In terms of communication frequency, we observed that at approximately every 20 to 25 minutes, a connection was established to `40.77.226.250` and `40.77.226.249`, respectively.

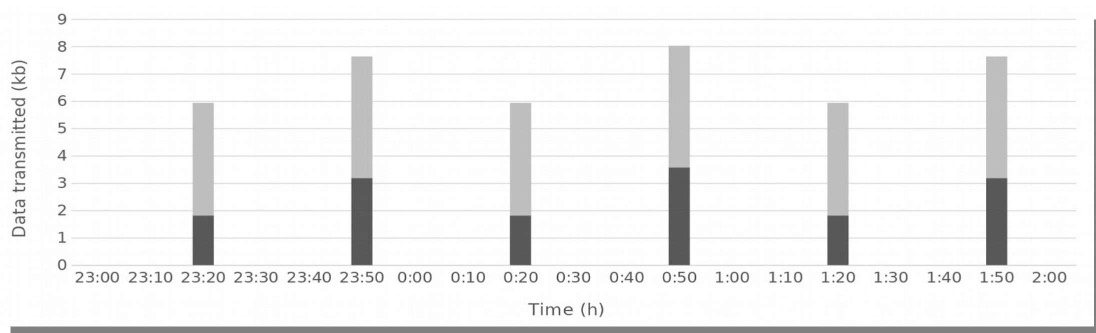


Figure 27: Communication frequency and intensity (host: 40.77.226.250)

36 https://github.com/aol/moloch [Retrieved: 11/04/2018]

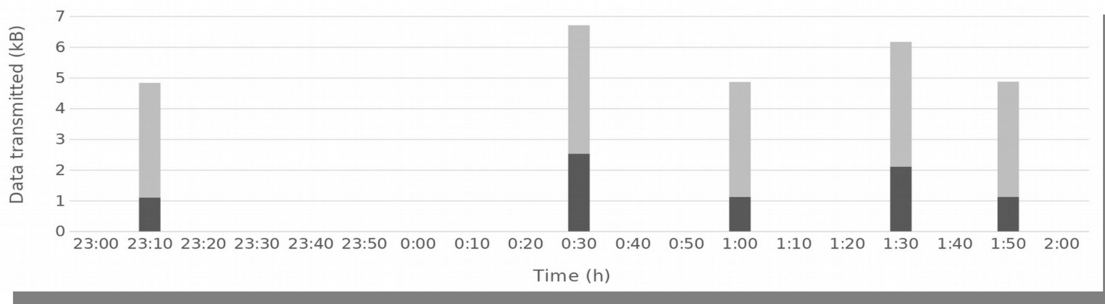


Figure 28: Communication frequency and intensity (host: 40.77.226.249)

We observed that collected telemetry data was sent to the host with IP address `40.77.226.250` (see Section 2.2.1). In addition, we observed that settings data (e.g., the file `%ProgramData%\Microsoft\Diagnosis\DownloadedSettings\utc.app.json`, see Section 2.2) was downloaded from the host with IP address `40.77.226.249`.

Access to certain resources (such as settings-win.data.microsoft.com) requires authentication against an external Microsoft identity provider. The service uses the additional authority parameter "consumers", which indicates that it uses the Microsoft Account (MSA) authentication type. The used IdentityString is randomly generated does correlate with an actual Microsoft Account. Analysis of the handling of this randomly generated IdentityString is out of scope of this work package. Once the service has access to the MSA account options, it can request the web token to access the web resources. The web token is Base64 encoded, it typically contains information to validate the caller and to ensure that the caller has the proper permissions to perform the operation they're requesting. The following figure shows a cached msatoken which DiagTrack uses to perform web operations. This identity provider can not be used to access resources on the Windows 10 client.

Event Name	IdentityString (Field 3)	ReturnedAccountId (Field 6)
Microsoft.Windows.DiagTrack/MSAUserIdentityProvider_CachePopulateResult/	5888a0da8c214d9a	000300000B340078
Microsoft.Windows.DiagTrack/MSAUserIdentityProvider_CachePopulateResult/	t=GwA2ARR4BAAUW6CljNsPH2VM5ReSSi...	000300000B340078

Figure 29: Cached MSA token

We now focus on the TLS-based protection of the communication interface between DiagTrack and Microsoft's back-end infrastructure. In this context, DiagTrack plays the role of a client and hosts that are part of the Microsoft's infrastructure play the role of servers.

Before any data exchange over a TLS-secured network interface, a communication session must be established. This typically involves server authentication, negotiation of a cipher suite used for data encryption, and negotiation of a session encryption key. During session establishment, DiagTrack sends a 'client hello' message to the server, along with the supported cipher suites (see '*cipher_suites*' in Figure 30). The server responds by sending a 'server hello' message. This message specifies a cipher suite and a chain of certificates for authentication (see '*cipher_suite*' and '*certificate_list*' in Figure 31). In the Appendix, in the section 'Server Authentication Certificate: Leaf', 'Server Authentication Certificate: Intermediate', and 'Server Authentication Certificate: Root', we present the leaf, intermediate, and root certificate provided by the host with IP address `40.77.226.250`.

[0]	Client Hello
msg_type	client_hello(1) (0x01)
length	207 (0x000000CF)
body	ClientHello{client_version=TLS 1.2,random=Random{gmt_u…
client_version	TLS 1.2
random	Random{gmt_unix_time=1522246294,random_bytes=binary[24…
session_id	SessionID{length_in_bytes=0,session_id=nothing}
cipher_suites_le…	42 (0x002A)
cipher_suites	[TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_ECDHE_ECD…
[0]	TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384(49196) (0xC02C)
[1]	TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256(49195) (0xC02B)
[2]	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384(49200) (0xC030)
[3]	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256(49199) (0xC02F)
[4]	TLS_DHE_RSA_WITH_AES_256_GCM_SHA384(159) (0x009F)
[5]	TLS_DHE_RSA_WITH_AES_128_GCM_SHA256(158) (0x009E)
[6]	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384(49188) (0xC024)
[7]	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256(49187) (0xC023)
[8]	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384(49192) (0xC028)
[9]	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256(49191) (0xC027)
[10]	TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA(49162) (0xC00A)
[11]	TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA(49161) (0xC009)
[12]	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA(49172) (0xC014)
[13]	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA(49171) (0xC013)
[14]	TLS_RSA_WITH_AES_256_GCM_SHA384(157) (0x009D)
[15]	TLS_RSA_WITH_AES_128_GCM_SHA256(156) (0x009C)
[16]	TLS_RSA_WITH_AES_256_CBC_SHA256(61) (0x003D)
[17]	TLS_RSA_WITH_AES_128_CBC_SHA256(60) (0x003C)
[18]	TLS_RSA_WITH_AES_256_CBC_SHA(53) (0x0035)
[19]	TLS_RSA_WITH_AES_128_CBC_SHA(47) (0x002F)
[20]	TLS_RSA_WITH_3DES_EDE_CBC_SHA(10) (0x000A)

Figure 30: A portion of a 'client hello' message

[0]	Handshake: [Server Hello, Certificate, Server Key Excha…
type	handshake(22) (0x16)
version	TLS 1.2
length	3762 (0x0EB2)
fragment	[Server Hello,Certificate,Server Key Exchange,Server He…
[0]	Server Hello
msg_type	server_hello(2) (0x02)
length	85 (0x00000055)
body	ServerHello{server_version=TLS 1.2,random=Random{gmt_un…
server_version	TLS 1.2
random	Random{gmt_unix_time=1522246294,random_bytes=binary[167…
session_id	SessionID{length_in_bytes=32,session_id=binary[85,57,0,…
cipher_suite	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384(49192) (0xC028)
compression_method	NULL(0) (0x00)
extensions_length…	13 (0x000D)
extensions	[extension_type: extended_master_secret,extension_type:…
[1]	Certificate
msg_type	certificate(11) (0x0B)
length	3332 (0x00000D04)
body	Certificate{certificate_list_length_in_bytes=3329,certi…
certificate_list_…	3329 (0x00000D01)
certificate_list	[ASN1Cert{length_in_bytes=1567,x509_cert=Certificate{Tb…
[0]	ASN1Cert{length_in_bytes=1567,x509_cert=Certificate{Tbs…
[1]	ASN1Cert{length_in_bytes=1756,x509_cert=Certificate{Tbs…

Figure 31: A portion of a 'server hello' message

With the goal to observe the actual encrypted data exchanged between DiagTrack and Microsoft, we used the `mitmproxy` utility. This utility enables users to view encrypted data sent over TLS channels, since it is capable of acting as the man-in-the-middle between two end-points participating in a TLS connection.[37] We captured transfered network packets over a 12-hour period, as well as the negotiated TLS session keys. We analyzed the captured packets with the `wireshark` utility.

37 https://docs.mitmproxy.org/stable/howto-transparent/ [Retrieved: 11/04/2018]

Protocol	Info
TCP	52896 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
TCP	443 → 52896 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
TCP	52896 → 443 [ACK] Seq=1 Ack=1 Win=525568 Len=0
TLSv1.2	Client Hello
TCP	443 → 52896 [ACK] Seq=1 Ack=409 Win=30336 Len=0
TLSv1.2	Server Hello
TLSv1.2	Certificate, Server Key Exchange, Server Hello Done
TCP	52896 → 443 [ACK] Seq=409 Ack=2195 Win=525568 Len=0
TLSv1.2	Client Key Exchange, Change Cipher Spec, Finished
TCP	443 → 52896 [ACK] Seq=2195 Ack=502 Win=30336 Len=0
TLSv1.2	New Session Ticket, Change Cipher Spec, Finished
TLSv1.2	[SSL segment of a reassembled PDU]
TCP	52896 → 443 [RST, ACK] Seq=1722 Ack=2453 Win=0 Len=0

Figure 32: TLS secure session negotiate with 40.77.226.250

When DiagTrack sends telemetry data to the Microsoft's back-end, it establishes a TLS connection to the host with IP `40.77.226.250`. DiagTrack then sends a POST request (see '*[SSL segment of a reassembled PDU]*'[38] in Figure 32 and Figure 33). After sending the POST request, it resets the connection by sending a TCP RST packet without waiting a reply from the server. We analyzed this anomalous behavior and observed that it is due to failed verification of the certificate of our `mitmproxy` server. We remind that this server acts as the man-in-the-middle. Server certificate verification is done against a pinned server certificate for preventing man-in-the-middle attacks.[39] We discuss the implemented certificate pinning mechanism later in this section.

```
POST /collect/v1 HTTP/1.1
Cache-Control: no-cache
Connection: Keep-Alive
Content-Type: application/x-json-stream; charset=utf-8
Content-Encoding: deflate
Accept: application/json
Accept-Encoding: gzip, deflate
User-Agent: MSDW
X-UploadTime: 2018-03-29T17:46:29.656Z
X-AuthMsaDeviceTicket: t=[...]
X-Tickets: "53435919"="p:t=[...]"
Content-Length: 2225
Host: v10.vortex-win.data.microsoft.com
```

Figure 33: A POST request

When DiagTrack receives settings data from the Microsoft's back-end, it establishes a TLS connection to the host with IP `40.77.226.249`. Once a connection to this host is established, the DiagTrack service sends a GET request (see Figure 34) and receives a `HTTP2` packet as a response.[40] This packet contains a data frame populated with settings data. After the packet is received, the connection between DiagTrack and `40.77.226.249` is terminated.

38 Protocol Data Unit (PDU).

39 https://docs.microsoft.com/en-us/windows/access-protection/enterprise-certificate-pinning [Retrieved: 26/10/2017]

40 https://tools.ietf.org/html/rfc7540 [Retrieved: 11/04/2018]

```
HEADERS[3]: GET /settings/v3.0/utc/app?expId=RS%3AF09%2CFX%3A20F36D18&sku=4&deviceClass
HEADERS[3]: 200 OK, DATA[3]
DATA[3]
DATA[3]
DATA[3]
DATA[3]
DATA[3] (application/json)
```

Figure 34: The exchange between DiagTrack and 40.77.226.249

```
0000  7b 22 71 75 65 72 79 55 72 6c 22 3a 22 2f 73 65   {"queryUrl":"/se
0010  74 74 69 6e 67 73 2f 76 33 2e 30 2f 75 74 63 2f   ttings/v3.0/utc/
0020  61 70 70 22 2c 22 73 65 74 74 69 6e 67 73 22 3a   app","settings":
0030  7b 22 55 54 43 3a 3a 3a 45 4e 44 50 4f 49 4e 54   {"UTC:::ENDPOINT
0040  2e 54 45 4c 45 4d 45 54 52 59 2e 41 53 4d 2d 57   .TELEMETRY.ASM-W
0050  49 4e 44 4f 57 53 53 51 22 3a 22 74 65 6c 65 6d   INDOWSSQ":"telem
[...]
ced0  46 49 43 41 54 49 4f 4e 52 45 47 49 53 54 52 41   FICATIONREGISTRA
cee0  54 49 4f 4e 55 52 49 22 3a 22 68 74 74 70 73 3a   TIONURI":"https:
cef0  2f 2f 73 65 74 74 69 6e 67 73 2d 77 69 6e 2e 64   //settings-win.d
cf00  61 74 61 2e 6d 69 63 72 6f 73 6f 66 74 2e 63 6f   ata.microsoft.co
cf10  6d 2f 72 65 67 69 73 74 65 72 63 68 61 6e 6e 65   m/registerchanne
cf20  6c 2f 76 31 2e 30 2f 22 7d 7d                     l/v1.0/"}}
```

Figure 35: Part of the data frame stored in the HTTP2 packet

We now focus on the mechanism for establishing TLS connections that is implemented as part of the DiagTrack service. This includes the certificate verification and pinning mechanism.

We observed that the implementation of the mechanism for establishing TLS connections is based on the Microsoft Windows HTTP Service (WinHTTP) framework.[41] In accordance with this framework, a TLS connection is established by invoking the following functions in a sequential order: `WinHttpOpen`, `WinHttpConnect`, `WinHttpOpenRequest`, `ChttpRequest::SendRequestWithRetry`, and `ChttpRequest::CheckCertForMicrosoftRoot` (see Figure 36). `WinHttpOpen` initializes basic WinHTTP structures. `WinHttpConnect` initializes connection-specific data structures that contain data such as an IP address and port. `WinHttpOpenRequest` initializes request-specific data structures that contain data such as HTTP headers. `ChttpRequest::SendRequestWithRetry` establishes the TLS connection. This involves sending the 'client hello' message, and receiving and processing the 'server hello' message. The chain of certificates for server authentication is stored in the 'server hello' message. In addition, `ChttpRequest::SendRequestWithRetry` sends the initial request to the server after a TLS connection has been established.

The chain of certificates for server authentication, received in `ChttpRequest::SendRequestWithRetry`, is verified in `ChttpRequest::CheckCertForMicrosoftRoot` (see Figure 37). The `WinHTTPQueryOption` function extracts the chain from the 'server hello' message, and the `CertGetCertificateChain` function verifies the chain. The `CertVerifyCertificateChainPolicy` function, when the `dwFlags` variable is set to `0x20000`, verifies the public key of the root certificate against a hash of a public key issued by the Microsoft Root Certificate Authority 2011. This hash is hardcoded in the `crypt32.dll` file.

If the verification of the public key stored in the root certificate fails, `CheckCertForMicrosoftRoot` returns an error code `CERT_E_UNTRUSTEDROOT (0x800B0109)`.[42] In that case, if the initial request sent by `ChttpRequest::SendRequestWithRetry` is a POST request, the DiagTrack service sends a TCP RST packet to terminate the connection without waiting a reply from the server. For example, this takes place when the DiagTrack service sends telemetry data to the Microsoft's back-end (see Figure 33). If the initial

41 https://msdn.microsoft.com/en-us/library/windows/desktop/aa382925%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396 [Retrieved: 28/3/2018]

42 https://msdn.microsoft.com/en-us/library/windows/desktop/aa377188(v=vs.85).aspx [Retrieved: 11/04/2018]

request sent by `ChttpRequest::SendRequestWithRetry` is a GET request, the DiagTrack service receives the data originating from the server. However, the DiagTrack service does not process the received data. For example, this takes place when the DiagTrack service receives settings data from the Microsoft's back-end (see Figure 34). This indicates that the DiagTrack service does not process settings data received from unknown servers, but only from servers that can present a valid certificate signed by Microsoft.

```
1   __int64 __fastcall CHttpRequest::CreateConnectionAndSendRequestImpl([...],FlagsBitMask ,[...])
2   {
3       [...]
4       hSession = WinHttpOpen(L"MSDW", [...]);
5       *(_QWORD *)hInternet = hSession;
6       [...]
7
8       hConnect = WinHttpConnect(hSession, pServerName, nServerPort, [...]);
9       *((_QWORD *)hInternet + 1) = hConnect;
10      [...]
11
12      hRequest = WinHttpOpenRequest(hConnect, pVerb, pObjectName, [...]);
13      *((_QWORD *)hInternet + 2) = hRequest;
14      [...]
15
16      returnCode = CHttpRequest::SendRequestWithRetry(hInternet, [...]);
17      [...]
18
19      if ( returnCode >= 0 )
20      {
21          if ( !(FlagsBitMask & 0x40) )
22              goto LABEL_113;
23          [...]
24          returnCode = CHttpRequest::CheckCertForMicrosoftRoot(hRequest);
25          [...]
26
27  LABEL_113:
28          returnCode = 0;
29          goto LABEL_110;
30          [...]
31      }
32  LABEL_110:
33      return returnCode;
34  }
```

Figure 36: Establishment of a TLS connection (WinHTTP)

```
1   __int64 __fastcall CHttpRequest::CheckCertForMicrosoftRoot(hRequest)
2   {
3       [...]
4       WinHttpQueryOption(hRequest, dwOption, &pCertContext, [...])
5       [...]
6
7       CertGetCertificateChain(0i64, pCertContext, [...], &pChainContext)
8       [...]
9
10      pPolicyPara.dwFlags = 0x20000;
11      CertVerifyCertificateChainPolicy(pPolicyOID, pChainContext, &pPolicyPara, &pPolicyStatus)
12      [...]
13
14      if ( pPolicyStatus.dwError )
15          returnCode = ERROR_HR_FROM_WIN32(pPolicyStatus.dwError);
16      else
17      returnCode = 0;
18      [...]
19
20      return returnCode;
21  }
```

Figure 37: The CheckCertForMicrosoftRoot function

Relevant events related to certificate operations (e.g., certificate verification) are logged by the ETW provider with GUID `5bbca4a8-b209-48dc-a8c7-b23d3e5216fb`. This provider is registered under the name

`Microsoft-Windows-CAPI2`.[43] It logs data related to the certificate verification process described in this section. The data logged by this provider can be viewed using the `Event Viewer` utility (see Figure 38).

Error	30/03/2018 08:32:57	CAPI2	30	Verify Chain Policy
Information	30/03/2018 08:32:57	CAPI2	11	Build Chain
Information	30/03/2018 08:32:57	CAPI2	90	X509 Objects
Information	30/03/2018 08:32:57	CAPI2	10	Build Chain

Figure 38: Logged events related to certificate operations

When the certificate verification process implemented in the `CHttpRequest::CheckCertForMicrosoftRoot` function (see Figure 37) takes place, the following events are logged in a sequential order: Event ID 10, Event ID 11, Event ID 30, and Event ID 90 (see Figure 38). The events with IDs 10, 11, and 90 are generated in the `CertGetCertificateChain` function. The event with ID 10 indicates that the extraction of the certificate chain from the 'server hello' message has begun. The event with ID 90 shows the actual certificates that are part of this chain. The event with ID 11 indicates the validity of the certificate chain. The event with ID 30 is generated in the `CertVerifyCertificateChainPolicy` function. It indicates the validity of the root certificate. Figure 39 depicts a logged event with ID 30 indicating that the root certificate is not valid (see '*Result*' in Figure 39, value `800B0109` – `CERT_E_UNTRUSTEDROOT`).

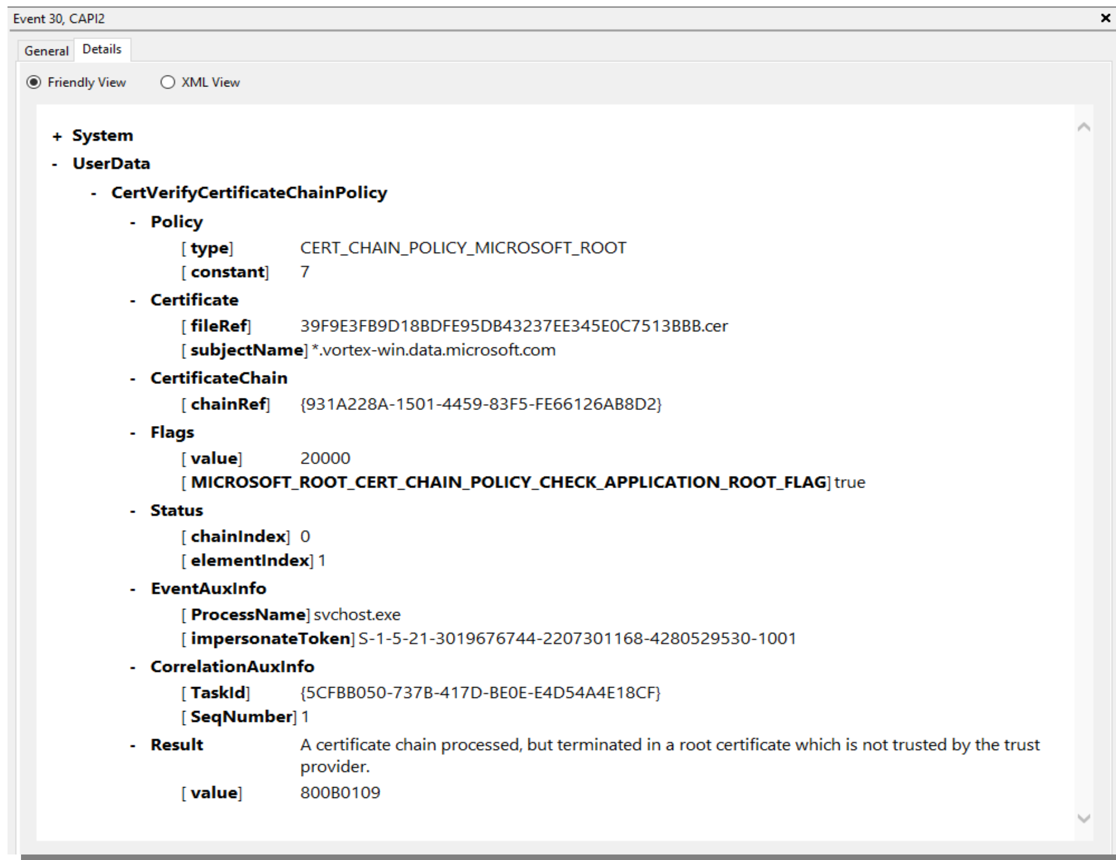


Figure 39: Detailed information about the event with ID 30

The certificate verification process described in this section can be disabled by setting the registry value `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\`

43 https://blogs.msdn.microsoft.com/benjaminperkins/2013/09/30/enable-capi2-event-logging-to-troubleshoot-pki-and-ssl-certificate-issues/ [Retrieved: 9/8/2017]

`DiagTrack\TestHooks\SkipMicrosoftRootCertCheck` to the `REG_DWORD` value `0x1`.[44] This disables the invocation of the `CHttpRequest::CheckCertForMicrosoftRoot` function.

2.4 Telemetry: Monitoring Activities

In this section, we propose an approach for monitoring activities of Telemetry on a given instance of Windows 10. These activities include sending of telemetry data to Microsoft and activities affecting the platform on which Telemetry operates. This includes, for example, creation of files for configuration and/or logging purposes (see Section 2.1 and Section 2.3).

The approach we propose may be implemented in the form of a framework. Figure 40 depicts the architecture of this framework. The framework performs monitoring of Telemetry activities on a given system, based on external sources of logs of such activities (*log sources* in Figure 40). We provide examples of such sources later in this section. The sources of Telemetry activities populate with log data *activity pools*. The log data can be in the form of log files or real-time log feeds (see, for example, Section 1.3).

Activity pools may obtain logs that contain not only Telemetry activities, but also activities of other system components. Therefore, we introduce the concept of *filters*, where each filter is associated with a given activity pool. The purpose of a filter is filtering out the Telemetry activities logged as part of the data with which the associated activity pool is populated, and presenting the filtered data to framework users. This may be in the form of charts, tables, and/or pop-up notifications (*output* in Figure 40).

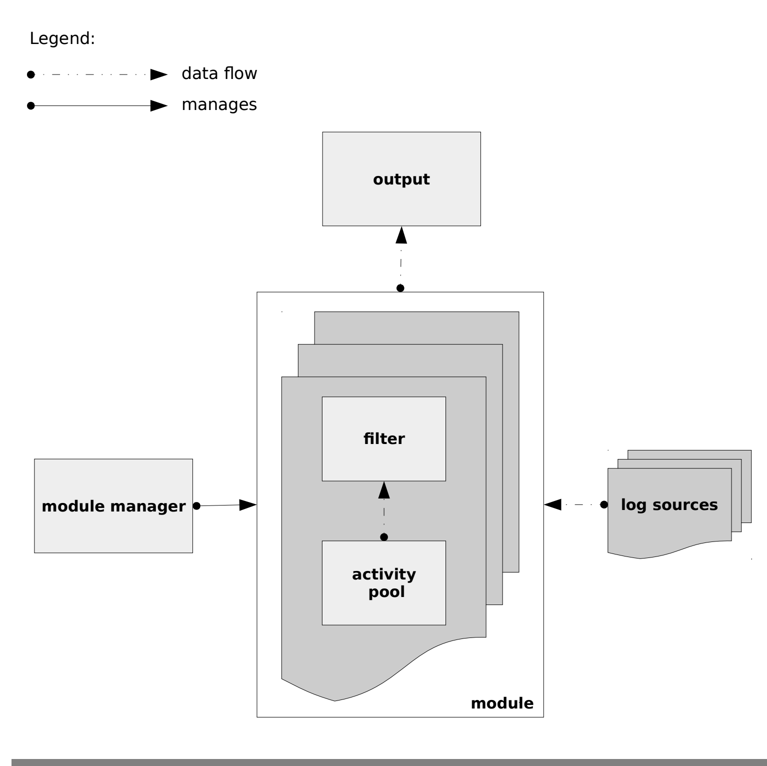


Figure 40: Activity monitoring architecture

The association between an activity pool and a filter is logically structured into a single framework *module*. Multiple modules are centrally managed by the framework component, which we refer to as the *module manager*.

The modular design of the framework we propose makes its monitoring capabilities extensible and fitting to any usage scenario. We emphasize that this framework is meant for observing and analyzing specific

44 https://msdn.microsoft.com/en-us/library/windows/desktop/ms724884(v=vs.85).aspx [Retrieved: 26/10/2017]

activities of the Telemetry component that are relevant to a given study. This makes the modular design of the framework crucial.

With respect to the sources of log data populating activity pools, we categorize activity pools as follows:

- Activity pools populated by ETW: These activity pools are ETW sessions, which are entities consuming data logged by ETW providers (see Section 1.3.1). The ETW providers are the external sources of logs of Telemetry activities, which populate activity pools.

  An example of an ETW provider populating an activity pool is the provider registered under the name `Microsoft-Windows-DNS-Client` (GUID: `1C95126E-7EEA-49A9-A3FE-A378B03DDB4D`). This ETW provider logs data containing network activities related to the Domain Name System (DNS) client implemented as part of Windows 10.[45] An example relevant activity logged by `Microsoft-Windows-DNS-Client` is name querying. Monitoring this activity is useful, for example, for users of the framework to observe the Telemetry component communicating with a specific server that is part of the Microsoft's back-end infrastructure (see Section 2.3).

  `Microsoft-Windows-DNS-Client` logs name querying activities in log entries with the Event ID 3010. The field `QueryName` of this Event ID contains the queried name. The field `Execution ProcessID` contains the ID of the process performing the name query activity. Once an activity pool is populated with log entries with the Event ID 3010, a framework user can apply a filter taking into account the values of the `QueryName` and `Execution ProcessID` fields. The filter helps to identify the log entries that contain the server name of interest, stored in the QueryName field, and the process ID of the DiagTrack service, stored in the `Execution ProcessID` field. Therefore, framework users are able to observe Telemetry communicating with a specific server that is part of the Microsoft's back-end infrastructure.

  In addition to Microsoft-Windows-DNS-Client, the ETW providers registered under the names `Microsoft-Windows-Security-Auditing` (GUID: `54849625-5478-4994-A5BA-3E3B0328C30D`) and `Microsoft-Windows-WinINet-Capture` (GUID: `A70FF94F-570B-4979-BA5C-E59C9FEAB61B`) are examples of ETW providers populating activity pools. The former is relevant for observing the creation of the DiagTrack service, whereas the latter for observing the data exchange between this service and the Microsoft's back-end infrastructure (see Section 2.3).

- Activity pools populated by APIs: These activity pools are populated by APIs producing log data. An example API is `System.IO`, which declares a class `FileSystemWatcher`.[46] This class implements methods for logging activities modifying the file-system. This is useful, for example, for users of the framework we propose to observe the Telemetry component affecting the platform on which it operates. This includes creation of files for configuration or logging purposes (see, for example, Section 3.1);

- Activity pools populated by external tools: These activity pools are populated by tools producing log data. An example tool is Wireshark, which logs network traffic.[47] This is useful, for example, for users of the framework we propose to observe the data exchange between Telemetry and the Microsoft's back-end infrastructure. .

45 https://technet.microsoft.com/en-us/library/cc766230%28v=ws.10%29.aspx [Retrieved: 26/10/2017]
46 https://msdn.microsoft.com/en-us/library/system.io.filesystemwatcher(v=vs.110).aspx [Retrieved: 26/10/2017]
47 https://www.wireshark.org/ [Retrieved: 26/10/2017]

3 Configuration and Logging Capabilities

In this section, we provide an overview of the capabilities of Windows 10 for configuring the Telemetry component (Section 3.1) and logging events related to this component (Section 3.2). In Work Package 11 [ERNW WP11], we provide recommendations for configuring Telemetry for the purpose of hardening platform security. In [ERNW WP4.1], we discuss the configuration of the Telemetry component in more detail and provide a guideline for disabling the Telemetry component.

3.1 Configuration Capabilities

As we mentioned in Section 2.1, the DiagTrack service is the core building block of the Telemetry component. As any service deployed in Windows 10, the DiagTrack service can be managed using the `Services` utility (executable: `services.msc`) or by modifying registry values located at `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\DiagTrack`.

We now focus on configuring Telemetry levels (see Section 2.1). In their essence, the Telemetry levels are used for specifying what, and how many, ETW providers are actively used for logging telemetry data (see Section 2.2). These levels are the following:

- `Security`: This level specifies the use of ETW providers for logging security-relevant activities, for example, detection of a malicious software;[48]
- `Basic`: This level specifies the use of the same providers as the 'Security' level plus additional providers used for logging device information;
- `Enhanced`: This level specifies the use of the same providers as the 'Security' and 'Basic' levels plus additional providers used for logging system usage and performance data (e.g., a list of running applications);
- `Full`: This level specifies the use of the same providers as the 'Security', 'Basic', and 'Enhanced' levels plus additional providers used for logging data relevant for troubleshooting (e.g., application crash information).

The 'Security' Telemetry level is available only in the Enterprise edition of Windows 10. Any Telemetry level listed above can be configured using the built-in `Settings` utility. This can be done by clicking on the button `Privacy` and then on `Feedback&diagnostics`. A Telemetry level can also be configured by setting one of the values listed in Table 6 (column 'Value') at the registry value `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\DataCollection\AllowTelemetry`. Alternatively, a Telemetry level can be configured by editing the 'Allow Telemetry' setting located at the policy path: `Computer Configuration` -> `Administrative Templates` -> `Windows Components` -> `Data Collection and Preview Builds`. This activity results in modification of the registry value `HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows\DataCollection\AllowTelemetry`.

Telemetry Level	Value
Security	0
Basic	1
Enhanced	2
Full	3

Table 6: Values of Telemetry levels

48 https://www.microsoft.com/en-us/windows/windows-defender [Retrieved: 9/8/2017]

We observed that values stored at the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack` are used for configuring the operation of the DiagTrack service. For example, the registry value `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\LastSuccessfullUploadTime` stores the last time the DiagTrack service has uploaded data. `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\TestHooks\SkipMicrosoftRootCertCheck` stores a flag for disabling root certificate verification (see Section 2.3). Figure 41 depicts the DiagTrack service reading data stored at `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack` for configuration purposes. Figure 41 depicts a snippet of the output of the `Process Monitor` utility.

RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\TraceManager	C:\Windows\System32\svchost.exe -k utcsvc
RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\TraceManager	C:\Windows\System32\svchost.exe -k utcsvc
RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\SettingsRequests	C:\Windows\System32\svchost.exe -k utcsvc
RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\SettingsRequests	C:\Windows\System32\svchost.exe -k utcsvc
RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\TelemetryNamespaces	C:\Windows\System32\svchost.exe -k utcsvc
RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\TelemetryNamespaces	C:\Windows\System32\svchost.exe -k utcsvc
RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\HeartBeats	C:\Windows\System32\svchost.exe -k utcsvc
RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\HeartBeats	C:\Windows\System32\svchost.exe -k utcsvc
RegCreateKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\LocalSettings	C:\Windows\System32\svchost.exe -k utcsvc
RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Diagnostics\DiagTrack\LocalSettings	C:\Windows\System32\svchost.exe -k utcsvc

Figure 41: The DiagTrack service reading configuration data stored in the registry

In addition to data stored in the system's registry, we observed that the DiagTrack service uses data downloaded from the Microsoft's back-end infrastructure for configuration purposes. This data is downloaded from https://settings-win.data.microsoft.com/settings/v2.0/ and is stored in configuration files located in the directory `%ProgramData%\Microsoft\Diagnosis`. Figure 42 depicts the DiagTrack service opening the directory for the purpose of reading a configuration file. Figure 42 depicts a snippet of the output of the `Process Monitor` utility.

CreateFile	C:\ProgramData\Microsoft\Diagnosis	C:\Windows\System32\svchost.exe -k utcsvc
QueryBasicInformationFile	C:\ProgramData\Microsoft\Diagnosis	C:\Windows\System32\svchost.exe -k utcsvc

Figure 42: The DiagTrack service reading configuration files

An example configuration file read by the DiagTrack service is `%ProgramData%\Microsoft\Diagnosis\DownloadedSettings\utc.app.json` (see Section 2.2). A portion of this file is depicted in Figure 43. It contains data structured into a typical JSON format. For example, the attributes `UTC:::GROUPDEFINITION.AI` and `UTC:::PROVIDERDEFINITION.0BD3506A-9030-4F76-9B88-3E8FE1F7CFB6` of the `settings` element store GUIDs of ETW providers delivering telemetry data.

The DiagTrack service obtains telemetry data from the ETW sessions 'Diagtrack-Listener' and 'Autologger-Diagtrack-Listener' (see Section 2.2). These ETW sessions can be configured using the `Performance Monitor` utility. We discussed this topic in [ERNW WP2], Section 2.4.2.

```
{
    "refreshInterval": "80",
    "queryUrl": "/settings/v2.0/utc/app",
    "settings": {
        [...]
        "UTC:::CATEGORYDEFINITION.MS.CRITICALDATA": "140737488355328",
        [...]
        "UTC:::GROUPDEFINITION.AI": "0d943590-b235-5bdb-f854-89520f32fc0b",
        [...]
        "UTC:::MINISLOTENABLED": "1",
        "UTC:::ONESETTINGSVERSION": "3",
        [...]
        "UTC:::PROVIDERDEFINITION.0BD3506A-9030-4F76-9B88-3E8FE1F7CFB6": "0BD3506A-9030-4F76-9B88-3E8FE1F7CFB6",
        [...]
    }
}
```

Figure 43: A portion of the utc.app.json file

3.2 Logging Capabilities

Windows 10 uses ETW for logging events related to Telemetry ([ERNW WP2], Section 2.5). We analyzed the functionalities for logging events related to the Diagtrack service implemented in `diagtrack.dll` (see Section 2.1). We identified 4 ETW providers with the GUIDs `43ac453b-97cd-4b51-4376-db7c9bb963ac`, `da995380-18dc-5612-a0db-161c5db2c2c1`, `56dc463b-97e8-4b59-e836-ab7c9bb96301`, and `6489b27f-7c43-5886-1d00-0a61bb2a375b`. We observed that these providers are registered when the DiagTrack service is initialized. These providers are attached to an ETW session named `EventCollector_DiagTrack`. It is used for logging internal activities of the DiagTrack service. `EventCollector_DiagTrack` is created using a definition (profile) file conform to the `Windows Performance Recorder`, part of the `Windows Performance Toolkit`.[49] A part of this file is depicted in Figure 44 , whereas the complete file is placed in the Appendix, section 'Windows Performance Recorder Profile'.

```
<Profile
    Description="Trace for diagnosing escalation issues in DiagTrack"
    DetailLevel="Verbose"
    Id="SelfDiagnosisTrace.Verbose.File"
    LoggingMode="File"
    Name="SelfDiagnosisTrace"
    >
    <Collectors>
        <EventCollectorId Value="EventCollector_DiagTrack">
            <EventProviders>
            <EventProvider
                Name="Microsoft-Windows-UniversalTelemetryClient"
                Id="Microsoft-Windows-UniversalTelemetryClient"
                />
            <EventProvider
                Name="Microsoft-Windows-Diagtrack"
                Id="Microsoft-Windows-Diagtrack"
                />
            <EventProvider
                Name="43ac453b-97cd-4b51-4376-db7c9bb963ac"
                Id="43ac453b-97cd-4b51-4376-db7c9bb963ac"
                />
            <EventProvider
                Name="da995380-18dc-5612-a0db-161c5db2c2c1"
                Id="da995380-18dc-5612-a0db-161c5db2c2c1"
                />
            </EventProviders>
        </EventCollectorId>
    </Collectors>
</Profile>
```

Figure 44: A portion of a Windows Performance Recorder custom profile

The ETW providers with GUIDs `43ac453b-97cd-4b51-4376-db7c9bb963ac` and `da995380-18dc-5612-a0db-161c5db2c2c1` are not registered under names. The providers with GUIDs `56dc463b-97e8-4b59-e836-ab7c9bb96301` and `6489b27f-7c43-5886-1d00-0a61bb2a375b` are registered under the names `Microsoft-Windows-Diagtrack` and `Microsoft-Windows-UniversalTelemetryClient`, respectively. This can be verified by issuing the command `logman query providers`. This command displays names and GUIDs of ETW providers.

Since the ETW providers with GUIDs `56dc463b-97e8-4b59-e836-ab7c9bb96301` and `6489b27f-7c43-5886-1d00-0a61bb2a375b` are registered under names, we can extract the event IDs under which the providers may log events. This can be done by using the `wevtutil` utility, as described in ([ERNW WP2], Section 2.5.3). We observed that the Event IDs under which these providers may log events do not contain descriptions and therefore, we do not present these IDs.

We emphasize that detailed information about the events logged by all of the ETW providers mentioned in this section can be obtained by using the `Windows Performance Recorder` utility. This utility can be

49 https://docs.microsoft.com/en-us/windows-hardware/test/wpt/recording-with-custom-profiles [Retrieved: 11/04/2018]

used to start the ETW session `EventCollector_DiagTrack` based on its definition file (see Appendix, section 'Windows Performance Recorder Profile'). The actual logged data can then be viewed using the `Windows Performance Analyzer` utility. This includes overview on relevant events structured into categories, such as processing configuration files and enabling ETW providers (see Figure 45).

Line #	Provider Name	Provider Id	Task Name	Count
1	Microsoft.Windows.DiagTrack	43ac453b-97cd-4b51-4376-db7c9bb963ac		10.556
2			Setting not recognized	2.649
3			ScenarioManager_DumpTriggerSettings	2.585
4			SettingsManager_DumpSettings	1.021
5			Utils_GetRegistryKey	777
6			ResultMacro_LogError	577
7			Utils_SetRegistryKey	534
8			EtwSession_DisableProvider	461
9			ETWConsumer_ModernAppProviderLoaded	404
10			SettingsEndpoint_MillisToNextDownload	156
11			FilterFactory_CreateFilter	133
12			Utils_OpenKey	117
13			EtwSession_EnableProvider	111
14			ScenarioManager_AcceptProviderSetting	83
15			ScenarioManager_SkippingIndividualEventSetting_SampledOut	77
16			ScenarioManager_AcceptProviderSettingFromGroup	76
17			Utils_CreateRegistryKey	72
18			LoadSettingsFromFile	48

Start: 32.530824909s
End: 45.457899500s
Duration: 12.927074591s

Figure 45: Event categories

Figure 46 depicts detailed information on a specific event. The detailed information that can be viewed with the `Windows Performance Analyzer` utility are useful for obtaining in-depth knowledge of the inner working principles of the DiagTrack service.

Provider Id	Event Name	Event Type	ThreadId	FileName (Field 1)
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\utc.app.json
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\telemetry.ASM-Wi
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\TELEMETRY.ASM-\
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\telemetry.P-ARIA-
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\telemetry.P-ARIA-
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\telemetry.P-ARIA-
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\telemetry.P-ARIA-
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\telemetry.P-ARIA-
43ac453b-97cd-4b51-4376-db7c9bb963ac	Microsoft.Windows.DiagTrack/LoadSettingsFromFile/	TraceLogging	5.232	C:\ProgramData\Microsoft\Diagnosis\DownloadedSettings\telemetry.P-ARIA-

Figure 46: Detailed event information logged by Microsoft-Windows-Diagtrack

Appendix

Tools

Tool	Availability and Description
`IDA`	*Availability:* https://www.hex-rays.com/products/ida/index.shtml [Retrieved: 7/5/2017] *Description*: A disassembly and debugging framework.
`Performance Monitor`	*Availability:* Distributed with Windows 10 *Description*: A utility displaying information on how processes affect the computer's performance (e.g., network and central processing unit (CPU) utilization).
`Sysinternals Suite` (includes `Process Monitor`, `Process Explorer`, and `Strings`)	*Availability:* https://technet.microsoft.com/de-de/sysinternals/bb545021.aspx [Retrieved: 7/5/2017] *Description*: A suite of tools for analyzing the Windows system (e.g., analyzing operation of processes, services, and enumeration of loaded libraries by processes).
`wevtutil`	*Availability:* Distributed with Windows 10 *Description*: A tool for querying running logging mechanisms.
Windows Debugger (`windbg`)	*Availability:* https://developer.microsoft.com/en-us/windows/hardware/download-windbg [Retrieved: 7/5/2017] *Description*: A debugger for the Windows system.
`xperf`	*Availability:* *https://docs.microsoft.com/en-us/windows-hardware/test/wpt/* [Retrieved: 26/10/2017] *Description*: A performance monitoring command line tool that produce in-depth performance profiles of Windows operating systems and applications.
`Windows Performance Analyzer`	*Availability:* *https://docs.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-analyzer* [Retrieved: 26/10/2017] *Description*: A performance monitoring tool that produce in-depth performance profiles of Windows operating systems and applications.
`xxd`	*Availability:* Distributed with Linux *Description*: A tool for viewing hex dumps of a given file or standard output.
`Microsoft Message Analyzer`	*Availability:* *https://technet.microsoft.com/en-us/library/jj649776.aspx*

	[Retrieved: 26/10/2017] *Description*: A tool to trace and assess system events and other messages from Windows components.
`yara`	*Availability*: *https://virustotal.github.io/yara/* [Retrieved: 26/10/2017] *Description*: A tool to search for patterns in a given file or process memory.
`mitmproxy`	*Availability*: *https://mitmproxy.org/* [Retrieved: 11/04/2018] *Description*: A tool for debugging web communications.
`wireshark`	*Availability*: *https://www.wireshark.org/* [Retrieved: 11/04/2018] *Description*: A network traffic sniffer.
`Moloch`	*Availability*: *https://github.com/aol/moloch* [Retrieved: 11/04/2018] *Description*: A network traffic sniffing framework.

API Monitor: Main Module

```
$$ ** API Monitor: Main Module
$$ ** Script usage:     "$$>a<[path_to_script_file]"
$$ *** Set options:     "$$>a<[path_to_script_file] set [LogFilePath] [ExtensionPath]"
$$ *** List options:    "$$>a<[path_to_script_file] opt"
$$ *** Init extensions: "$$>a<[path_to_script_file] init [ExtensionDefinitionFileName]"
$$
$$ [LogFilePath]: path to the log file
$$ [ExtensionPath]: path to an extension
$$ [ExtensionDefinitionFileName]: file name of ExtensionDefinition
$$
$$ ** Example: "$$>a<C:\APIMon\ApiMon.wds set C:\APIMon\output C:\APIMon\extension"
$$ ** Example: "$$>a<C:\APIMon\ApiMon.wds opt"
$$ ** Example: "$$>a<C:\APIMon\ApiMon.wds init ExtensionDefinition.wds"


.catch
{
        .if ${/d:$arg1} == 0
    {
        .printf "**********************************\n";
                .printf "** Script usage: API Monitor\n";
        .printf "*** Set Options:       ApiMon.wds set [LogFilePath] [ExtensionPath]\n";
        .printf "*** List Options:      ApiMon.wds opt\n";
        .printf "*** Init Extensions:   ApiMon.wds init [ExtensionDefinitionFileName]\n";
        .printf "**********************************\n";
                .leave;
        }

    .if @@masm(not($scmp("${$arg1}", "set")))
    {
        aS LogFilePath @"${$arg2}";
        aS ExtensionPath @"${$arg3}";
        .leave;
    }

    .if @@masm(not($scmp("${$arg1}", "opt")))
    {
```

```
        .printf "** List Options: ****************\n";
        al;
        .printf "********************************\n";
                .leave;
    }

        .if @@masm(not($scmp("${$arg1}", "init")))
    {

                .logopen /t ${LogFilePath};

                .if @@c++(((unsigned char *)&@$PEB->BeingDebugged)[0]) == @@masm(0y1)
        {
                        r? $t5 = @@c++((unsigned long *)&@$PEB->BeingDebugged);
                        eb $t5 @@masm(0y0);
                }

                .if @@c++(((unsigned char *)&@$PEB->NtGlobalFlag)[0]) ==
@@masm(0y1110000)
        {
                        r? $t6 = @@c++((unsigned long *)&@$PEB->NtGlobalFlag);
                        eb $t6 @@masm(0y0);
                }

        $$><"${ExtensionPath}\${$arg2}"

        .printf "** Init Extension: **************\n";
        bl;
        .printf "********************************\n";

        sxi ld;
                .leave;
    }
}
```

API Monitor: Extension Definition

```
$$ ** API Monitor: Extension Definition
$$ ** Script usage:     "$$><${ExtensionPath}\[ExtensionFileName]"

$$><"${ExtensionPath}\ExtensionEnableTraceEx2.wds";
$$><"${ExtensionPath}\ExtensionAddDataToRingBufferA.wds";
```

API Monitor: Extension EnableTraceEx2

```
$$ ** API Monitor: Extension EnableTraceEx2

.catch
{

        .if ${/d:$arg1} == 0
    {
                bu sechost!EnableTraceEx2 @"$$>a<${$arg0} EnableTraceEx2";
                .leave;
        }

        .if @@masm($spat("${$arg1}", "EnableTraceEx2"))
    {
        .if (@rdx !=0)
        {
            $$
            r? $t0 = @@c++(*(nt!_GUID *) @rdx)
            r? $t1 = @@c++((unsigned long) @$t0.Data1)
```

```
            r? $t2 = @@c++((unsigned short) @$t0.Data2)
            r? $t3 = @@c++((unsigned short) @$t0.Data3)

            r? $t4 = @@c++((unsigned char) @$t0.Data4[0])
            r? $t5 = @@c++((unsigned char) @$t0.Data4[1])
            r? $t6 = @@c++((unsigned char) @$t0.Data4[2])
            r? $t7 = @@c++((unsigned char) @$t0.Data4[3])
            r? $t8 = @@c++((unsigned char) @$t0.Data4[4])
            r? $t9 = @@c++((unsigned char) @$t0.Data4[5])
            r? $t10 = @@c++((unsigned char) @$t0.Data4[6])
            r? $t11 = @@c++((unsigned char) @$t0.Data4[7])

            $$ print json stream
             .printf /D "<col fg=\"srcspid\">@@EnableTraceEx2@@ %08x-%04x-%04x-%02x%02x-
%02x%02x%02x%02x%02x%02x @@EnableTraceEx2@@\n</col>",@$t1, @$t2, @$t3, @$t4, @$t5, @$t6,
@$t7, @$t8, @$t9, @$t10, @$t11;
             gc;
        }
    }
}
```

API Monitor: Extension AddDataToRingBufferA

```
$$ ** API Monitor: Extension AddDataToRingBufferA

.catch
{

        .if ${/d:$arg1} == 0
    {
        bu diagtrack!Microsoft::Diagnostics::CRingBufferEventStore::AddData @"$$>a<$
{$arg0} AddDataToRingBufferA";
                .leave;
        }

        $$
    .if @@masm($spat("${$arg1}", "AddDataToRingBufferA"))
    {
        .if (@rdx !=0)
        {
            $$ calc offset uploadData (format: json stream)
            r? $t0 = @@masm(@rdx + 0x8);
            r? $t1 = @@masm(poi($t0));

            $$
            .printf /D "<col fg=\"srcspid\">@@AddDataToRingBufferA@@ %ma
@@AddDataToRingBufferA@@\n</col>", @$t1;
            gc;
        }
    }
}
```

Yara Rule

```
rule hardcodedProviderGuids
{
        meta:
                description = "Searches for the GUIDs that are still enabled without the
utc.app.json file."

        strings:
```

```
                $a1 = {D6 2C FB 22 7B 0E 2B 42 A0 C7 2F AD 1F D0 E7 16}
                $a2 = {50 A0 F4 96 31 7E 3C 45 88 BE 96 34 F4 E0 21 39}
                $a3 = {3A 3B 1C 33 05 20 C2 44 AC 5E 77 22 0C 37 D6 B4}
                $a4 = {33 BE B6 A0 59 B9 C3 50 3C 92 84 51 B6 F9 65 C3}
                $a5 = {94 FF 39 28 12 8F 1B 4E 82 E3 AF 7A F7 7A 45 0F}
                $a6 = {83 B3 E9 DB F3 7C 31 43 91 CC A3 CB 16 A3 B5 38}
                $a7 = {69 DC 9F A1 26 A6 89 52 BE 4D 1F 50 8A 8C 9A 3B}


        condition:
                all of them
}
```

RBS Decompression Script

```
#!/usr/bin/env python3
# coding: utf-8
import zlib
import sys

d = zlib.decompressobj(wbits=-zlib.MAX_WBITS)

with open(str(sys.argv[1]), 'rb') as fp:

    fp.seek(0x52)

    for chunk in iter(lambda: fp.read(1024), b''):
        try:
            sys.stdout.buffer.write(d.decompress(chunk))
        except Exception as e:
            print(e)
            break
```

External Executables

Executable	Description
`%SystemRoot%\System32\telsvc.exe`	No description available
`%SystemRoot%\SysWow64\dtdump.exe`	No description available
`%SystemRoot%\SysWow64\RdrLeakDiag.exe, %SystemRoot%system32\RdrLeakDiag.exe`	No description available
`%SystemRoot%\system32\appidtel.exe`	No description available

%SystemRoot%\system32\disksnapshot.exe	No description available
%SystemRoot%\system32\bcdedit.exe	A tool for managing the Boot Configuration Database (BCD); https://technet.microsoft.com/en-us/library/cc709667(v=ws.10).aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\dxdiag.exe	A tool for collecting information on devices; *https://support.microsoft.com/en-us/help/4028644/windows-open-and-run-dxdiagexe*
%SystemRoot%\system32\dispdiag.exe	A tool for collecing and logging information on displays; https://technet.microsoft.com/de-de/library/ff920169(v=ws.10).aspx [Retrieved: 26/10/2017]
%ProgramFiles%\internet explorer\iediagcmd.exe	No description available
%SystemRoot%\system32\icacls.exe	A tool for displaying and modifying access control lists; us/library/cc753525(v=ws.11).aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\licensingdiag.exe	No description available
%SystemRoot%\system32\ipconfig.exe	A tool for displaying network information and configuring network settings; https://technet.microsoft.com/en-us/library/cc940124.aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\msinfo32.exe	A tool for displaying information about the hardware and software enviroment deployed on a platform; https://support.microsoft.com/en-us/help/300887/how-to-use-system-information-msinfo32-command-line-tool-switches [Retrieved: 26/10/2017]
%SystemRoot%\system32\logman.exe	A tool for configuring, and displaying information about, the ETW environment; https://technet.microsoft.com/en-us/library/bb490956.aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\netsh.exe	A tool for displaying network information and configuring network settings; https://technet.microsoft.com/en-us/library/bb490939.aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\netcfg.exe	A tool for installing the Windows preinstallation environment, a lightweight version of Windows; https://technet.microsoft.com/en-us/library/hh875638(v=ws.11).aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\route.exe	A tool for displaying and modifying the platform's IP routing table; https://technet.microsoft.com/en-us/library/ff961510(v=ws.11).aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\powercfg.exe	A tool for configuring power settings (e.g., configuring the platform's standby mode); https://technet.microsoft.com/en-us/library/cc748940(v=ws.10).aspx [Retrieved: 26/10/2017]

%SystemRoot%\system32\stordiag.exe	No description available
%SystemRoot%\system32\settingsynchost.exe	No description available
%SystemRoot%\system32\verifier.exe	A tool for detecting and troubleshooting driver issues; https://support.microsoft.com/en-us/help/244617/using-driver-verifier-to-identify-issues-with-windows-drivers-for-adva [Retrieved: 26/10/2017]
%SystemRoot%\system32\tracelog.exe	A tool for managing ETW environment (e.g., activation and deactivation of ETW sessions); https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/tracelog [Retrieved: 26/10/2017]
%SystemRoot%\system32\whoami.exe	A tool for displaying information on the user currently logged on to the system; https://technet.microsoft.com/en-us/library/cc771299(v=ws.11).aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\wevtutil.exe	A tool for managing the EventLog environment; https://technet.microsoft.com/en-us/library/cc732848(v=ws.11).aspx [Retrieved: 26/10/2017]
%SystemRoot%\system32\wscollect.exe	No description available

Server Authentication Certificates: Leaf

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            33:00:00:01:10:a7:d3:81:42:ef:d2:92:79:00:00:00:00:01:10
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, ST = Washington, L = Redmond, O = Microsoft Corporation, CN =
Microsoft Secure Server CA 2011
        Validity
            Not Before: Oct 18 17:29:38 2017 GMT
            Not After : Jan 18 17:29:38 2019 GMT
        Subject: C = US, ST = WA, L = Redmond, O = Microsoft Corporation, OU = Microsoft,
CN = *.vortex-win.data.microsoft.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:e0:ec:10:36:8a:cb:9e:e3:fe:a8:63:02:22:54:
                    8f:07:b3:8b:b7:bb:a5:f0:d0:23:19:33:f5:77:2c:
                    89:e1:11:1f:f8:78:16:42:f8:df:51:25:76:7d:42:
                    50:15:4c:e4:b7:81:1b:f4:f8:e2:b1:bb:87:1e:44:
```

```
                39:c6:18:28:06:a0:7d:1b:20:a4:ff:5c:bf:95:05:
                e9:a3:1f:41:36:ed:de:3f:5c:56:da:a9:52:d6:1a:
                91:b3:ff:65:ba:e8:e2:90:92:14:ea:f5:67:e3:6e:
                49:5c:51:4a:c2:c6:c4:17:ce:02:99:4e:50:e4:4d:
                f2:c2:aa:0c:70:17:43:40:f1:79:a5:ed:56:db:16:
                b3:b2:c0:2a:a2:ca:2b:fc:1d:63:68:7f:cc:a8:db:
                90:00:d5:32:a1:93:4c:c6:bb:a2:87:ef:71:be:74:
                35:10:da:7e:e5:27:40:ec:16:62:72:02:97:2b:10:
                ad:c0:28:66:1a:4a:06:6c:9d:9b:8e:ae:5b:75:53:
                f4:15:ab:1b:82:46:ff:c7:ad:b7:eb:e9:3a:04:6c:
                6f:3c:87:43:81:27:db:1d:95:34:a0:42:0d:83:04:
                4d:10:0d:a4:a9:83:f9:e6:b3:99:a0:c0:aa:3c:37:
                c1:c6:0a:15:b1:8a:48:f5:fe:bf:dd:ac:21:b1:c5:
                d9:3d
            Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Non Repudiation, Key Encipherment, Data Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client Authentication
            X509v3 Subject Key Identifier:
                57:1E:69:00:F7:1A:9F:36:BD:47:F0:B9:28:49:BB:D5:57:7D:88:DD
            X509v3 Subject Alternative Name:
                DNS:*.vortex-win.data.microsoft.com, DNS:vortex-win.data.microsoft.com
            X509v3 Authority Key Identifier:
                keyid:36:56:89:65:49:CB:5B:9B:2F:3C:AC:42:16:50:4D:91:B9:33:D7:91

            X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://www.microsoft.com/pkiops/crl/MicSecSerCA2011_2011-10-18.crl

            Authority Information Access:
                CA Issuers -
URI:http://www.microsoft.com/pkiops/certs/MicSecSerCA2011_2011-10-18.crt

            X509v3 Basic Constraints: critical
                CA:FALSE
    Signature Algorithm: sha256WithRSAEncryption
         b7:d1:d2:75:4e:2f:b2:b6:29:28:56:92:1a:d4:db:12:46:7d:
         a4:1a:5e:ca:9b:9f:83:e9:4d:29:1d:16:06:0c:61:ef:78:31:
         d4:bc:f9:6c:3a:ed:0e:6c:60:d8:82:41:2d:70:32:9d:d0:72:
```

```
        1c:12:66:a1:44:cd:5e:00:19:38:01:ea:ac:63:ff:83:fc:cc:
        52:3a:81:6e:2c:bd:e9:b1:25:72:08:b7:07:00:38:da:cf:d9:
        de:14:cc:31:5b:87:1b:c6:97:c4:ec:76:55:c2:0f:ad:a6:ad:
        a8:a7:d8:f6:f4:c6:51:fc:bd:8f:5e:17:3d:ba:f0:40:4f:a1:
        90:f9:e2:5e:09:8f:0d:37:05:89:93:88:61:5c:ae:f1:f6:a9:
        0f:47:7d:ca:92:45:48:f5:ee:54:57:45:22:b6:cb:20:a1:3c:
        26:ea:f6:44:39:07:3a:cd:b1:d4:14:ec:5d:a4:0b:a0:1b:ec:
        32:87:3f:6a:57:5a:a3:37:81:22:de:1a:84:88:48:5d:f7:22:
        b4:b4:d5:77:76:db:03:ad:3d:07:54:8b:d2:7f:0b:ac:0d:b0:
        d9:3b:7b:30:10:95:76:c6:91:77:2d:fc:c9:88:62:c9:ef:8a:
        21:c8:9a:b5:70:68:c0:38:4d:6f:dc:e7:2e:f1:95:7f:a6:07:
        de:d6:75:1a:c5:2e:7e:e2:20:45:39:13:8f:e3:3a:e0:74:dd:
        c1:9d:b2:e8:66:dc:d4:f2:0a:6f:13:5b:b9:67:af:c4:4b:cc:
        25:4b:91:cc:d0:f0:e6:49:41:d8:1e:56:c0:6f:4b:6c:18:57:
        a2:0e:25:83:63:3f:e6:e9:13:e3:e9:39:b4:cf:f0:97:3d:ce:
        ae:2f:03:73:af:3e:f8:b0:2d:6a:cd:5a:0c:1a:3a:0b:a2:0e:
        62:e3:c4:49:19:1e:27:f7:68:7a:02:1f:a6:97:2d:1a:ae:01:
        65:64:9e:98:26:b1:f8:12:96:9b:1e:1f:b8:f9:c4:fd:d6:d1:
        da:7e:77:9a:46:ec:05:b3:e1:25:94:09:14:51:a0:7a:fa:1a:
        9e:5f:d1:c6:58:e9:44:d1:7c:d3:b4:9d:3e:52:76:a4:35:f0:
        75:f9:85:68:8b:42:29:b9:c2:2e:5e:84:29:8f:06:80:c2:30:
        11:32:aa:92:d1:fe:8e:53:64:af:2b:5a:aa:ad:6e:d5:3d:96:
        8c:d4:f9:9a:67:81:8c:d2:2c:77:ac:b1:ae:33:75:97:1c:96:
        08:cb:59:d6:2f:40:a0:e1:95:51:e8:89:8b:0d:30:d0:4b:45:
        12:8e:15:e0:f0:54:62:05:27:0f:33:1c:d6:26:b6:85:74:e4:
        04:be:aa:ae:1d:19:a2:08
```


```
-----BEGIN CERTIFICATE-----
MIIGGzCCBAOgAwIBAgITMwAAARCn04FC79KSeQAAAAABEDANBgkqhkiG9w0BAQsF
ADB+MQswCQYDVQQGEwJVUzETMBEGA1UECBMKV2FzaGluZ3RvbjEQMA4GA1UEBxMH
UmVkbW9uZDEeMBwGA1UEChMVTWljcm9zb2Z0IENvcnBvcmF0aW9uMSgwJgYDVQQD
Ex9NaWNyb3NvZnQgU2VjdXJlIFNlcnZlciBDQSAyMDExMB4XDTE3MTAxODE3Mjkz
OFoXDTE5MDExODE3MjkzOFowgYoxCzAJBgNVBAYTAlVTMQswCQYDVQQIEwJXQTEQ
MA4GA1UEBxMHUmVkbW9uZDEeMBwGA1UEChMVTWljcm9zb2Z0IENvcnBvcmF0aW9u
MRIwEAYDVQQLEwlNaWNyb3NvZnQxKDAmBgNVBAMMHyoudm9ydGV4LXdpbi5kYXRh
Lm1pY3Jvc29mdC5jb20wggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDg
7BA2isue4/6oYwIiVI8Hs4u3u6Xw0CMZM/V3LInhER/4eBZC+N9RJXZ9QlAVTOS3
gRv0+OKxu4ceRDnGGCgGoH0bIKT/XL+VBemjH0E27d4/XFbaqVLWGpGz/2W66OKQ
khTq9WfjbklcUUrCxsQXzgKZTlDkTfLCqgxwF0NA8Xml7VbbFrOywCqiyiv8HWNo
f8yo25AA1TKhk0zGu6KH73G+dDUQ2n7lJ0DsFmJyApcrEK3AKGYaSgZsnZuOrlt1
U/QVqxuCRv/Hrbfr6ToEbG88h0OBJ9sdlTSgQg2DBE0QDaSpg/nms5mgwKo8N8HG
ChWxikj1/r/drCGxxdk9AgMBAAGjggGDMIIBfzAOBgNVHQ8BAf8EBAMCBPAwHQYD
VR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMB0GA1UdDgQWBBRXHmkA9xqfNr1H
```

```
8LkoSbvVV32I3TBJBgNVHREEQjBAgh8qLnZvcnRleC13aW4uZGF0YS5taWNyb3Nv
ZnQuY29tgh12b3J0ZXgtd2luLmRhdGEubWljcm9zb2Z0LmNvbTAfBgNVHSMEGDAW
gBQ2VollSctbmy88rEIWUE2RuTPXkTBTBgNVHR8ETDBKMEigRqBEhkJodHRwOi8v
d3d3Lm1pY3Jvc29mdC5jb20vcGtpb3BzL2NybC9NaWNTZWNTZXJDQTIwMTFfMjAx
MS0xMC0xOC5jcmwwYAYIKwYBBQUHAQEEVDBSMFAGCCsGAQUFBzAChkRodHRwOi8v
d3d3Lm1pY3Jvc29mdC5jb20vcGtpb3BzL2NlcnRzL01pY1NlY1NlckNBMjAxMV8y
MDExLTEwLTE4LmNydDAMBgNVHRMBAf8EAjAAMA0GCSqGSIb3DQEBCwUAA4ICAQC3
0dJ1Ti+ytikoVpIa1NsSRn2kGl7Km5+D6U0pHRYGDGHveDHUvPlsOu0ObGDYgkEt
cDKd0HIcEmahRM1eABk4AeqsY/+D/MxSOoFuLL3psSVyCLcHADjaz9neFMwxW4cb
xpfE7HZVwg+tpq2op9j29MZR/L2PXhc9uvBAT6GQ+eJeCY8NNwWJk4hhXK7x9qkP
R33KkkVI9e5UV0UitssgoTwm6vZEOQc6zbHUFOxdpAugG+wyhz9qV1qjN4Ei3hqE
iEhd9yK0tNV3dtsDrT0HVIvSfwusDbDZO3swEJV2xpF3LfzJiGLJ74ohyJq1cGjA
OE1v3Ocu8ZV/pgfe1nUaxS5+4iBFOROP4zrgdN3BnbLoZtzU8gpvE1u5Z6/ES8wl
S5HM0PDmSUHYHlbAb0tsGFeiDiWDYz/m6RPj6Tm0z/CXPc6uLwNzrz74sC1qzVoM
GjoLog5i48RJGR4n92h6Ah+mly0argFlZJ6YJrH4EpabHh+4+cT91tHafneaRuwF
s+EllAkUUaB6+hqeX9HGWOlE0XzTtJ0+UnakNfB1+YVoi0IpucIuXoQpjwaAwjAR
MqqS0f6OU2SvK1qqrW7VPZaM1PmaZ4GM0ix3rLGuM3WXHJYIy1nWL0Cg4ZVR6ImL
DTDQS0USjhXg8FRiBScPMxzWJraFdOQEvqquHRmiCA==
-----END CERTIFICATE-----
```


Server Authentication Certificates: Intermediate

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            61:3f:b7:18:00:00:00:00:00:04
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, ST = Washington, L = Redmond, O = Microsoft Corporation, CN =
Microsoft Root Certificate Authority 2011
        Validity
            Not Before: Oct 18 22:55:19 2011 GMT
            Not After : Oct 18 23:05:19 2026 GMT
        Subject: C = US, ST = Washington, L = Redmond, O = Microsoft Corporation, CN =
Microsoft Secure Server CA 2011
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:d0:0b:c0:a4:a8:19:81:e2:36:e5:e2:aa:e5:f3:
                    b2:15:58:75:be:b4:e5:49:f1:e0:84:f9:bb:0d:64:
                    ef:85:c1:81:55:b8:f3:e7:f1:6d:40:55:3d:ce:8b:
                    6a:d1:84:93:f5:75:7c:5b:a4:d4:74:10:ca:32:f3:
```

```
                    23:d3:ae:ee:cf:9e:04:58:c2:d9:47:cb:d1:7c:00:
                    41:48:71:1b:01:67:17:18:af:c6:fe:73:03:7e:e4:
                    ef:43:9c:ef:01:71:2a:1f:81:26:43:77:98:54:57:
                    73:9d:55:2b:f0:9e:8e:7d:06:0e:ac:1b:54:f3:26:
                    f7:f8:23:08:22:8b:9e:06:1d:37:38:fd:72:d2:ca:
                    e5:63:c1:9a:5a:7d:b2:6d:b3:52:a9:6e:e9:ae:b5:
                    fc:8b:36:f9:9e:fa:f6:1c:58:1b:97:56:a5:11:e5:
                    b7:52:db:bb:e9:f0:54:bf:b4:ff:2c:6c:b8:5d:26:
                    ce:a0:0a:d7:df:93:ed:7f:dd:ac:f1:2c:73:1a:d9:
                    19:37:55:ba:dd:22:78:8e:a1:d4:9b:09:f8:07:22:
                    31:71:b0:94:ae:e0:b0:e7:26:44:57:90:81:97:15:
                    ce:61:ec:65:e2:4b:f1:85:52:16:32:f8:b5:78:aa:
                    7e:cd:4d:ec:83:21:a4:a8:9b:be:9a:6a:04:e0:a3:
                    1c:cd:56:18:6c:fd:6b:2f:42:3e:e2:37:f2:72:ab:
                    d0:78:73:72:7b:de:ec:00:58:e5:21:30:a3:08:3a:
                    99:ef:9f:c3:f7:7a:16:96:65:b5:c3:81:af:f4:39:
                    70:49:af:f6:a9:f6:6a:00:38:f9:b4:08:19:e0:1a:
                    35:a5:56:76:22:5f:6a:f2:69:ae:3e:ad:58:46:4d:
                    b8:54:f6:89:41:44:1e:72:b1:bc:12:27:53:d2:c1:
                    ff:b2:cd:50:98:1e:b5:f4:bb:b6:c2:82:39:d9:ac:
                    1b:f2:3b:27:84:6a:b0:c6:26:0b:d7:3a:10:e7:b3:
                    db:7c:d3:56:ac:53:4c:0b:fa:3b:31:37:74:d8:59:
                    2b:f9:00:79:19:06:7b:fd:1c:1d:42:d4:41:0d:2f:
                    05:0e:d5:6b:49:23:ff:cf:cd:f8:7a:82:cf:da:3c:
                    2d:df:e8:d8:12:04:18:ba:1e:88:77:b8:98:1f:10:
                    07:bb:c8:05:7e:0b:09:bf:6b:dd:e3:4e:5b:b0:f9:
                    c7:84:a6:3b:ca:4c:9f:5b:62:29:f7:c7:a2:a8:95:
                    88:70:2c:e5:c1:3f:3c:52:23:4f:40:9a:c3:31:85:
                    83:2f:bf:29:f1:1d:50:8f:21:96:07:ce:ef:f2:80:
                    c2:44:7d:9b:62:ef:2f:c3:77:89:ab:45:4d:53:3e:
                    02:79:d3
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            1.3.6.1.4.1.311.21.1:
                ...
            X509v3 Subject Key Identifier:
                36:56:89:65:49:CB:5B:9B:2F:3C:AC:42:16:50:4D:91:B9:33:D7:91
            1.3.6.1.4.1.311.20.2:
                .
.S.u.b.C.A
            X509v3 Key Usage:
                Digital Signature, Certificate Sign, CRL Sign
```

```
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Authority Key Identifier:
                keyid:72:2D:3A:02:31:90:43:B9:14:05:4E:E1:EA:A7:C7:31:D1:23:89:34

            X509v3 CRL Distribution Points:

                Full Name:

URI:http://crl.microsoft.com/pki/crl/products/MicRooCerAut2011_2011_03_22.crl

            Authority Information Access:
                CA Issuers -
URI:http://www.microsoft.com/pki/certs/MicRooCerAut2011_2011_03_22.crt

    Signature Algorithm: sha256WithRSAEncryption
         41:c8:61:c1:f5:5b:9e:3e:91:31:f1:b0:c6:bf:09:01:b4:9d:
         b6:90:74:d7:09:db:a6:2e:0d:9f:c8:e7:76:34:46:af:07:60:
         89:4c:81:b3:3c:d5:f4:12:35:75:c2:73:a5:f5:4d:84:8c:cb:
         a4:5d:af:bf:92:f6:17:08:57:42:95:72:65:05:76:79:ad:ee:
         d1:ba:b8:2e:54:a3:51:07:ac:68:eb:21:0c:e3:25:81:c2:cd:
         2a:f2:c3:ff:cf:c2:bd:49:18:9a:c7:f0:84:c5:f9:14:bc:6b:
         95:e5:96:ef:b3:42:d2:53:d5:4a:a0:12:c4:ae:12:76:53:09:
         56:0e:9d:f7:d3:a6:49:88:50:f2:8a:2c:97:20:a2:be:4e:78:
         ef:05:65:b7:4b:a1:16:88:de:31:c7:08:42:24:7c:a4:7b:9e:
         9d:bc:60:00:5e:62:97:e3:93:fc:a7:fe:5b:7b:25:df:e4:53:
         7f:4b:be:e6:3e:f0:db:01:79:42:1c:6e:85:6c:7d:b6:44:30:
         fb:a5:37:92:93:b2:a5:ee:20:ad:3f:53:d5:c9:f4:28:6b:57:
         c1:f8:1d:6a:b7:56:2a:b6:27:81:1c:a6:2d:9f:e7:f4:d0:31:
         83:97:a8:2a:b6:ac:be:1b:41:f5:e4:89:5f:56:fb:da:5a:d3:
         5e:7d:55:94:10:7e:53:57:f4:4a:3d:40:2a:c8:bd:67:9f:84:
         e1:10:ee:fd:da:6b:15:82:49:fc:46:1d:ff:45:06:74:9c:42:
         14:ed:c5:39:d3:b3:cd:0b:83:27:90:43:51:92:f2:44:82:ae:
         6e:9a:15:17:b2:19:fa:c7:45:6c:98:01:7b:bf:37:a9:b0:88:
         a4:92:bc:38:38:e0:1d:e4:7c:97:98:1a:2e:5f:ef:38:65:b7:
         35:2f:bd:7f:4f:21:fa:c4:8c:d2:6f:06:f9:49:35:ea:df:20:
         0f:25:aa:ea:60:ab:2c:1f:4b:89:fc:b7:fa:5c:54:90:4b:3e:
         a2:28:4f:6c:e4:52:65:c1:fd:90:1c:85:82:88:6e:e9:a6:55:
         dd:21:28:79:45:b0:14:e5:0a:cc:e6:5f:c4:bb:db:61:34:69:
         9f:ac:26:38:f7:c1:29:41:08:15:2e:4c:a0:f7:f9:0c:3e:de:
         5f:ab:08:09:2d:83:ac:ac:34:83:62:f4:c9:49:42:89:25:b5:
         6e:b2:47:c5:b3:39:a0:b1:20:1b:2c:b1:8e:04:6f:a5:30:49:
         1c:d0:46:e9:40:5b:f4:ad:6e:ba:db:82:4a:87:12:4a:80:09:
```

```
        4d:db:df:76:b9:05:5b:1b:e0:bb:20:70:5f:00:25:c7:d3:0e:
        fa:16:ad:7b:22:9e:71:08
-----BEGIN CERTIFICATE-----
MIIG2DCCBMCgAwIBAgIKYT+3GAAAAAAABDANBgkqhkiG9w0BAQsFADCBiDELMAkG
A1UEBhMCVVMxEzARBgNVBAgTCldhc2hpbmd0b24xEDAOBgNVBAcTB1JlZG1vbmQx
HjAcBgNVBAoTFU1pY3Jvc29mdCBDb3Jwb3JhdGlvbjEyMDAGA1UEAxMpTWljcm9z
b2Z0IFJvb3QgQ2VydGlmaWNhdGUgQXV0aG9yaXR5IDIwMTEwHhcNMTExMDE4MjI1
NTE5WhcNMjYxMDE4MjMwNTE5WjB+MQswCQYDVQQGEwJVUzETMBEGA1UECBMKV2Fz
aGluZ3RvbjEQMA4GA1UEBxMHUmVkbW9uZDEeMBwGA1UEChMVTWljcm9zb2Z0IENv
cnBvcmF0aW9uMSgwJgYDVQQDEx9NaWNyb3NvZnQgU2VjdXJlIFNlcnZlciBDQSAy
MDExMIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA0AvApKgZgeI25eKq
5fOyFVh1vrTlSfHghPm7DWTvhcGBVbjz5/FtQFU9zotq0YST9XV8W6TUdBDKMvMj
067uz54EWMLZR8vRfABBSHEbAWcXGK/G/nMDfuTvQ5zvAXEqH4EmQ3eYVFdznVUr
8J6OfQYOrBtU8yb3+CMIIoueBh03OP1y0srlY8GaWn2ybbNSqW7prrX8izb5nvr2
HFgbl1alEeW3Utu76fBUv7T/LGy4XSbOoArX35Ptf92s8SxzGtkZN1W63SJ4jqHU
mwn4ByIxcbCUruCw5yZEV5CBlxXOYexl4kvxhVIWMvi1eKp+zU3sgyGkqJu+mmoE
4KMczVYYbP1rL0I+4jfycqvQeHNye97sAFjlITCjCDqZ75/D93oWlmW1w4Gv9Dlw
Sa/2qfZqADj5tAgZ4Bo1pVZ2Il9q8mmuPq1YRk24VPaJQUQecrG8EidT0sH/ss1Q
mB619Lu2woI52awb8jsnhGqwxiYL1zoQ57PbfNNWrFNMC/o7MTd02Fkr+QB5GQZ7
/RwdQtRBDS8FDtVrSSP/z834eoLP2jwt3+jYEgQYuh6Id7iYHxAHu8gFfgsJv2vd
405bsPnHhKY7ykyfW2Ip98eiqJWIcCzlwT88UiNPQJrDMYWDL78p8R1QjyGWB87v
8oDCRH2bYu8vw3eJq0VNUz4CedMCAwEAAaOCAUswggFHMBAGCSsGAQQBgjcVAQQD
AgEAMB0GA1UdDgQWBBQ2VollSctbmy88rEIWUE2RuTPXkTAZBgkrBgEEAYI3FAIE
DB4KAFMAdQBiAEMAQTALBgNVHQ8EBAMCAYYwDwYDVR0TAQH/BAUwAwEB/zAfBgNV
HSMEGDAWgBRyLToCMZBDuRQFTuHqp8cx0SOJNDBaBgNVHR8EUzBRME+gTaBLhklo
dHRwOi8vY3JsLm1pY3Jvc29mdC5jb20vcGtpL2NybC9wcm9kdWN0cy9NaWNSb29D
ZXJBdXQyMDExXzIwMTFfMDNfMjIuY3JsMF4GCCsGAQUFBwEBBFIwUDBOBggrBgEF
BQcwAoZCaHR0cDovL3d3dy5taWNyb3NvZnQuY29tL3BraS9jZXJ0cy9NaWNSb29D
ZXJBdXQyMDExXzIwMTFfMDNfMjIuY3J0MA0GCSqGSIb3DQEBCwUAA4ICAQBByGHB
9VuePpEx8bDGvwkBtJ22kHTXCdumLg2fyOd2NEavB2CJTIGzPNX0EjV1wnOl9U2E
jMukXa+/kvYXCFdClXJlBXZ5re7RurguVKNRB6xo6yEM4yWBws0q8sP/z8K9SRia
x/CExfkUvGuV5Zbvs0LSU9VKoBLErhJ2UwlWDp3306ZJiFDyiiyXIKK+TnjvBWW3
S6EWiN4xxwhCJHyke56dvGAAXmKX45P8p/5beyXf5FN/S77mPvDbAXlCHG6FbH22
RDD7pTeSk7Kl7iCtP1PVyfQoa1fB+B1qt1YqtieBHKYtn+f00DGDl6gqtqy+G0H1
5IlfVvvaWtNefVWUEH5TV/RKPUAqyL1nn4ThEO792msVgkn8Rh3/RQZ0nEIU7cU5
07PNC4MnkENRkvJEgq5umhUXshn6x0VsmAF7vzepsIikkrw4OOAd5HyXmBouX+84
Zbc1L71/TyH6xIzSbwb5STXq3yAPJarqYKssH0uJ/Lf6XFSQSz6iKE9s5FJlwf2Q
HIWCiG7pplXdISh5RbAU5QrM5l/Eu9thNGmfrCY498EpQQgVLkyg9/kMPt5fqwgJ
LYOsrDSDYvTJSUKJJbVuskfFszmgsSAbLLGOBG+lMEkc0EbpQFv0rW6624JKhxJK
gAlN2992uQVbG+C7IHBfACXH0w76Fq17Ip5xCA==
-----END CERTIFICATE-----
```

Server Authentication Certificates: Root

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            3f:8b:c8:b5:fc:9f:b2:96:43:b5:69:d6:6c:42:e1:44
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, ST = Washington, L = Redmond, O = Microsoft Corporation, CN =
Microsoft Root Certificate Authority 2011
        Validity
            Not Before: Mar 22 22:05:28 2011 GMT
            Not After : Mar 22 22:13:04 2036 GMT
        Subject: C = US, ST = Washington, L = Redmond, O = Microsoft Corporation, CN =
Microsoft Root Certificate Authority 2011
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:b2:80:41:aa:35:38:4d:13:72:32:68:22:4d:b8:
                    b2:f1:ff:d5:52:bc:6c:c7:f5:d2:4a:8c:36:ee:d1:
                    c2:5c:7e:8c:8a:ae:af:13:28:6f:c0:73:e3:3a:ce:
                    d0:25:a8:5a:3a:6d:ef:a8:b8:59:ab:13:23:68:cd:
                    0c:29:87:d1:6f:80:5c:8f:44:7f:5d:90:01:52:58:
                    ac:51:c5:5f:2a:87:dc:dc:d8:0a:1d:c1:03:b9:7b:
                    b0:56:e8:a3:de:64:61:c2:9e:f8:f3:7c:b9:ec:0d:
                    b5:54:fe:4c:b6:65:4f:88:f0:9c:48:99:0c:42:0b:
                    09:7c:31:59:17:79:06:78:28:8d:89:3a:4c:03:25:
                    be:71:6a:5c:0b:e7:84:60:a4:99:22:e3:d2:af:84:
                    a4:a7:fb:d1:98:ed:0c:a9:de:94:89:e1:0e:a0:dc:
                    c0:ce:99:3d:ea:08:52:bb:56:79:e4:1f:84:ba:1e:
                    b8:b4:c4:49:5c:4f:31:4b:87:dd:dd:05:67:26:99:
                    80:e0:71:11:a3:b8:a5:41:e2:a4:53:b9:f7:32:29:
                    83:0c:13:bf:36:5e:04:b3:4b:43:47:2f:6b:e2:91:
                    1e:d3:98:4f:dd:42:07:c8:e8:1d:12:fc:99:a9:6b:
                    3e:92:7e:c8:d6:69:3a:fc:64:bd:b6:09:9d:ca:fd:
                    0c:0b:a2:9b:77:60:4b:03:94:a4:30:69:12:d6:42:
                    2d:c1:41:4c:ca:dc:aa:fd:8f:5b:83:46:9a:d9:fc:
                    b1:d1:e3:b3:c9:7f:48:7a:cd:24:f0:41:8f:5c:74:
                    d0:ac:b0:10:20:06:49:b7:c7:2d:21:c8:57:e3:d0:
                    86:f3:03:68:fb:d0:ce:71:c1:89:99:4a:64:01:6c:
                    fd:ec:30:91:cf:41:3c:92:c7:e5:ba:86:1d:61:84:
```

```
                c7:5f:83:39:62:ae:b4:92:2f:47:f3:0b:f8:55:eb:
                a0:1f:59:d0:bb:74:9b:1e:d0:76:e6:f2:e9:06:d7:
                10:e8:fa:64:de:69:c6:35:96:88:02:f0:46:b8:3f:
                27:99:6f:cb:71:89:29:35:f7:48:16:02:35:8f:d5:
                79:7c:4d:02:cf:5f:eb:8a:83:4f:45:71:88:f9:a9:
                0d:4e:72:e9:c2:9c:07:cf:49:1b:4e:04:0e:63:51:
                8c:5e:d8:00:c1:55:2c:b6:c6:e0:c2:65:4e:c9:34:
                39:f5:9c:b3:c4:7e:e8:61:6e:13:5f:15:c4:5f:d9:
                7e:ed:1d:ce:ee:44:ec:cb:2e:86:b1:ec:38:f6:70:
                ed:ab:5c:13:c1:d9:0f:0d:c7:80:b2:55:ed:34:f7:
                ac:9b:e4:c3:da:e7:47:3c:a6:b5:8f:31:df:c5:4b:
                af:eb:f1
            Exponent: 65537 (0x10001)
    X509v3 extensions:
        X509v3 Key Usage:
            Digital Signature, Certificate Sign, CRL Sign
        X509v3 Basic Constraints: critical
            CA:TRUE
        X509v3 Subject Key Identifier:
            72:2D:3A:02:31:90:43:B9:14:05:4E:E1:EA:A7:C7:31:D1:23:89:34
            ...
Signature Algorithm: sha256WithRSAEncryption
     7f:72:cf:0f:b7:c5:15:db:9b:c0:49:ca:26:5b:fe:9e:13:e6:
     d3:f0:d2:db:97:5f:f2:4b:3f:4d:b3:ae:19:ae:ed:d7:97:a0:
     ac:ef:a9:3a:a3:c2:41:b0:e5:b8:91:9e:13:81:24:03:e6:09:
     fd:3f:57:40:39:21:24:56:d1:10:2f:4b:40:a9:36:86:4b:b4:
     53:57:9a:fb:f1:7e:89:8f:11:fe:18:6c:51:aa:e8:ed:09:95:
     b5:e5:71:c9:a1:e9:87:75:a6:15:7f:c9:7e:37:54:5e:74:93:
     c5:c3:67:cc:0d:4f:6b:a8:17:0c:6d:08:92:7e:8b:dd:81:aa:
     2d:70:21:c3:3d:06:14:bb:bf:24:5e:a7:84:d7:3f:0f:21:22:
     bd:4b:00:06:db:97:1c:d8:5e:d4:c5:0b:5c:87:6e:50:a4:e8:
     c3:38:a4:fb:cb:2c:c5:92:66:9b:85:5e:cb:7a:6c:93:7c:80:
     29:58:5b:57:b5:40:69:ba:08:79:a6:64:62:15:9d:87:96:45:
     b5:66:23:20:03:8b:1c:73:a0:d3:a2:79:33:e0:50:59:86:db:
     2f:e5:02:25:ea:73:2a:9f:00:14:c8:36:c7:92:3b:e9:4e:00:
     ec:d8:56:09:b9:33:49:12:d2:54:0b:01:ab:ac:47:b6:91:29:
     7d:4c:b4:75:80:52:01:e8:ca:82:f6:9f:cc:ac:9c:8f:17:ea:
     2f:26:b0:ab:72:ac:0b:fe:9e:51:1e:c7:43:55:67:4f:51:b3:
     57:d6:b6:ec:ee:52:b7:3a:e9:4e:e1:d7:81:88:bc:4f:8e:75:
     bb:4b:a8:f0:35:aa:26:d4:67:67:49:b2:70:4c:3b:93:dc:1d:
     df:78:90:86:72:b2:38:a4:d1:dc:92:4d:c9:58:eb:2b:12:5c:
     d4:3b:ae:8c:6b:b0:83:e5:01:3f:f8:09:32:f6:93:35:34:22:
```

```
        af:dd:37:0d:77:09:80:2b:cd:48:00:f1:8c:99:19:47:05:01:
        e9:d1:bf:d1:4e:d0:e6:28:43:37:99:a4:0a:4a:08:d9:9a:71:
        73:d2:aa:cd:31:13:63:76:a1:37:6f:92:38:1e:7d:12:3c:66:
        32:e7:cb:6d:e1:fc:52:89:dd:ca:d6:66:05:9a:96:61:be:a2:
        28:c7:1c:a3:a7:36:50:3c:3a:a4:df:4a:6e:e6:87:3b:ce:eb:
        f0:e0:81:37:9d:13:3c:52:8e:bd:b9:1d:34:c6:1d:d5:0a:6a:
        3d:98:29:70:8c:89:2a:d1:ab:82:10:48:1f:dc:f4:ef:a5:c5:
        bb:55:1a:38:63:84:4e:b7:6c:ad:95:54:ec:65:22:10:49:17:
        b8:c0:1e:c7:0f:ac:54:47
-----BEGIN CERTIFICATE-----
```


MIIF7TCCA9WgAwIBAgIQP4vItfyfspZDtWnWbELhRDANBgkqhkiG9w0BAQsFADCB
iDELMAkGA1UEBhMCVVMxEzARBgNVBAgTCldhc2hpbmd0b24xEDAOBgNVBAcTB1Jl
ZG1vbmQxHjAcBgNVBAoTFU1pY3Jvc29mdCBDb3Jwb3JhdGlvbjEyMDAGA1UEAxMp
TWljcm9zb2Z0IFJvb3QgQ2VydGlmaWNhdGUgQXV0aG9yaXR5IDIwMTEwHhcNMTEw
MzIyMjIwNTI4WhcNMzYwMzIyMjIxMzA0WjCBiDELMAkGA1UEBhMCVVMxEzARBgNV
BAgTCldhc2hpbmd0b24xEDAOBgNVBAcTB1JlZG1vbmQxHjAcBgNVBAoTFU1pY3Jv
c29mdCBDb3Jwb3JhdGlvbjEyMDAGA1UEAxMpTWljcm9zb2Z0IFJvb3QgQ2VydGlm
aWNhdGUgQXV0aG9yaXR5IDIwMTEwggIiMA0GCSqGSIb3DQEBAQUAA4ICDwAwggIK
AoICAQCygEGqNThNE3IyaCJNuLLx/9VSvGzH9dJKjDbu0cJcfoyKrq8TKG/Ac+M6
ztAlqFo6be+ouFmrEyNozQwph9FvgFyPRH9dkAFSWKxRxV8qh9zc2AodwQO5e7BW
6KPeZGHCnvjzfLnsDbVU/ky2ZU+I8JxImQxCCwl8MVkXeQZ4KI2JOkwDJb5xalwL
54RgpJki49KvhKSn+9GY7Qyp3pSJ4Q6g3MDOmT3qCFK7VnnkH4S6Hri0xElcTzFL
h93dBWcmmYDgcRGjuKVB4qRTufcyKYMME782XgSzS0NHL2vikR7TmE/dQgfI6B0S
/Jmpaz6SfsjWaTr8ZL22CZ3K/QwLopt3YEsDlKQwaRLWQi3BQUzK3Kr9j1uDRprZ
/LHR47PJf0h6zSTwQY9cdNCssBAgBkm3xy0hyFfj0IbzA2j70M5xwYmZSmQBbP3s
MJHPQTySx+W6hh1hhMdfgzlirrSSL0fzC/hV66AfWdC7dJse0Hbm8ukG1xDo+mTe
acY1logC8Ea4PyeZb8txiSk190gWAjWP1Xl8TQLPX+uKg09FcYj5qQ1OcunCnAfP
SRtOBA5jUYxe2ADBVSy2xuDCZU7JNDn1nLPEfuhhbhNfFcRf2X7tHc7uROzLLoax
7Dj2cO2rXBPB2Q8Nx4CyVe0096yb5MPa50c8prWPMd/FS6/r8QIDAQABo1EwTzAL
BgNVHQ8EBAMCAYYwDwYDVR0TAQH/BAUwAwEB/zAdBgNVHQ4EFgQUci06AjGQQ7kU
BU7h6qfHMdEjiTQwEAYJKwYBBAGCNxUBBAMCAQAwDQYJKoZIhvcNAQELBQADggIB
AH9yzw+3xRXbm8BJyiZb/p4T5tPw0tuXX/JLP02zrhmu7deXoKzvqTqjwkGw5biR
nhOBJAPmCf0/V0A5ISRW0RAvS0CpNoZLtFNXmvvxfomPEf4YbFGq6O0JlbXlccmh
6Yd1phV/yX43VF50k8XDZ8wNT2uoFwxtCJJ+i92Bqi1wIcM9BhS7vyRep4TXPw8h
Ir1LAAbblxzYXtTFC1yHblCk6MM4pPvLLMWSZpuFXst6bJN8gClYW1e1QGm6CHmm
ZGIVnYeWRbVmIyADixxzoNOieTPgUFmG2y/lAiXqcyqfABTINseSO+lOAOzYVgm5
M0kS0lQLAausR7aRKX1MtHWAUgHoyoL2n8ysnI8X6i8msKtyrAv+nlEex0NVZ09R
s1fWtuzuUrc66U7h14GIvE+OdbtLqPA1qibUZ2dJsnBMO5PcHd94kIZysjik0dyS
TclY6ysSXNQ7roxrsIPlAT/4CTL2kzU0Iq/dNw13CYArzUgA8YyZGUcFAenRv9FO
0OYoQzeZpApKCNmacXPSqs0xE2N2oTdvkjgefRI8ZjLny23h/FKJ3crWZgWalmG+
oijHHKOnNlA8OqTfSm7mhzvO6/DggTedEzxSjr25HTTGHdUKaj2YKXCMiSrRq4IQ
SB/c9O+lxbtVGjhjhE63bK2VVOxlIhBJF7jAHscPrFRH

```
-----END CERTIFICATE-----
```

Windows Performance Recorder Profile

```xml
<?xml version='1.0' encoding='utf-8' standalone='yes'?>
<WindowsPerformanceRecorder
    Author="Dominik Phillips"
    Comments="Telemetry Profile"
    Company="ERNW"
    Copyright="ERNW"
    Team="Windows Security"
    Version="1.0"
    >
  <Profiles>
    <EventCollector
        Id="EventCollector_DiagTrack"
        Name="Trace"
        >
      <BufferSize Value="128"/>
      <Buffers Value="32"/>
    </EventCollector>
    <Profile
        Description="Trace for diagnosing escalation issues in DiagTrack"
        DetailLevel="Verbose"
        Id="SelfDiagnosisTrace.Verbose.File"
        LoggingMode="File"
        Name="SelfDiagnosisTrace"
        >
      <Collectors>
        <EventCollectorId Value="EventCollector_DiagTrack">
          <EventProviders>
            <EventProvider
              Name="Microsoft-Windows-UniversalTelemetryClient"
              Id="Microsoft-Windows-UniversalTelemetryClient"
              />
            <EventProvider
              Name="Microsoft-Windows-Diagtrack"
              Id="Microsoft-Windows-Diagtrack"
              />
            <EventProvider
              Name="43ac453b-97cd-4b51-4376-db7c9bb963ac"
              Id="43ac453b-97cd-4b51-4376-db7c9bb963ac"
              />
```

```
            <EventProvider
              Name="da995380-18dc-5612-a0db-161c5db2c2c1"
              Id="da995380-18dc-5612-a0db-161c5db2c2c1"
              />
          </EventProviders>
        </EventCollectorId>
      </Collectors>
    </Profile>
  </Profiles>
</WindowsPerformanceRecorder>
```

Reference Documentation

ERNW WP2	ERNW GmbH: Work Package 2: Analysis of Windows 10
ERNW WP4.1	ERNW GmbH: Turning Off Telemetry in Windows 10
Soulami 2012	Tarik Soulami: Inside Windows Debugging
Russinovich 2012	Russinovich, Mark E.; Solomon, David A.; Ionescu, Alex: Windows Internals, Part 2
ERNW WP5	ERNW GmbH: Work Package 5: Trusted Platform Module and Windows Boot
ERNW WP11	ERNW GmbH: Work Package 11: Hardening Guidelines

Keywords and Abbreviations

Abbreviations ... 63
application programming interface ... 6, 9, 11ff., 16, 23, 28, 39
Boot Configuration Database ... 49
Bundesamt für Sicherheit in der Informationstechnik ... 5, 10
*central pro*cessing unit ... 44
Domain Name System ... 39
dynamic-link library ... 6, 16, 25
Event Tracing for Windows ... 5ff., 20ff., 29f., 36, 39ff.
Extensible Markup Language ... 9
global unique identifier ... 6f., 9, 12, 18, 20, 22ff., 30, 39, 41f.
Hypertext Transfer Protocol ... 30, 34f.
input/output ... 6, 11, 15
Internet Protocol ... 12, 31f., 34f., 49, 59
JavaScript Object Notation ... 8, 27f., 41
kilobyte ... 30
long-term servicing branch ... 5, 10
Managed Object Format ... 12
Megabyte ... 27, 52f., 56
Microsoft Account ... 32
Protocol Data Unit ... 34
Transmission Control Protocol ... 12, 34f.
transport layer security ... 8, 17, 30, 32ff.
Windows software trace preprocessor ... 12