Creating a Collector Initiated Subscription

You can subscribe to receive events on a local computer (the event collector) that are forwarded from remote computers (the event sources) by using a collector-initiated subscription. In a collector-initiated subscription, the subscription must contain a list of all the event sources. Before a collector computer can subscribe to events and a remote event source can forward events, both computers must be configured for event collecting and forwarding. For more information about how to configure the computers, see Configure Computers to Forward and Collect Events.

The following code example follows a series of steps to create a collector initiated subscription:

To create a collector initiated subscription

1. Open the subscription by providing the subscription name and access rights as parameters to the **EcOpenSubscription** function. For more information about access rights, see **Windows Event Collector Constants**.
2. Set the properties of the subscription by calling the **EcSetSubscriptionProperty** function. For more information about subscription properties that can be set, see the **EC_SUBSCRIPTION_PROPERTY_ID** enumeration.
3. Save the subscription by calling the **EcSaveSubscription** function.
4. Close the subscription by calling the **EcClose** function.

For more information about adding an event source, see Adding an Event Source to an Event Collector Subscription.

The following C++ code example shows how to create a collector-initiated subscription:

C++

```
#include <windows.h>
#include <iostream>
using namespace std;
#include <string>
#include <xstring>
#include <conio.h>
#include <EvColl.h>
#include <vector>
#include <wincred.h>
#pragma comment(lib, "credui.lib")
#pragma comment(lib, "wecapi.lib")

// Track properties of the Subscription.
typedef struct _SUBSCRIPTION_COLLECTOR_INITIATED
{
    std::wstring Name;
    std::wstring Description;
    std::wstring URI;
    std::wstring Query;
    std::wstring DestinationLog;
    std::wstring Password;
    std::wstring UserName;
    EC_SUBSCRIPTION_CONFIGURATION_MODE ConfigMode;
    EC_SUBSCRIPTION_DELIVERY_MODE DeliveryMode;
    EC_SUBSCRIPTION_TYPE SubscriptionType;
    DWORD MaxItems;
```

```
    DWORD MaxLatencyTime;
    DWORD HeartbeatInerval;
    EC_SUBSCRIPTION_CONTENT_FORMAT ContentFormat;
    EC_SUBSCRIPTION_CREDENTIALS_TYPE CredentialsType;
    BOOL SubscriptionStatus;
} SUBSCRIPTION_COLLECTOR_INITIATED;

// Subscription Information
DWORD GetProperty(EC_HANDLE hSubscription,
                  EC_SUBSCRIPTION_PROPERTY_ID propID,
                  DWORD flags,
                  std::vector<BYTE>& buffer,
                  PEC_VARIANT& vProperty);


void __cdecl wmain()
{
    LPVOID lpwszBuffer;
    DWORD dwRetVal = ERROR_SUCCESS;
    EC_HANDLE hSubscription = 0;
    EC_VARIANT vPropertyValue;
    std::vector<BYTE> buffer;
    PEC_VARIANT vProperty = NULL;
    SUBSCRIPTION_COLLECTOR_INITIATED sub;

    sub.Name = L"TestSubscription-CollectorInitiated";
    sub.Description = L"A subscription that collects events that are published in\n" \
        L"the Microsoft-Windows-TaskScheduler/Operational log and forwards them\n" \
        L"to the ForwardedEvents log.";
    sub.URI = L"http://schemas.microsoft.com/wbem/wsman/1/windows/EventLog";
    sub.Query = L"<QueryList>" \
        L"<Query Path=\"Microsoft-Windows-TaskScheduler/Operational\">" \
        L"<Select>*</Select>" \
        L"</Query>" \
        L"</QueryList>";
    sub.DestinationLog = L"ForwardedEvents";
    sub.ConfigMode = EcConfigurationModeCustom;
    sub.MaxItems = 5;
    sub.MaxLatencyTime = 10000;
    sub.HeartbeatInerval = 10000;
    sub.DeliveryMode = EcDeliveryModePull;
    sub.ContentFormat = EcContentFormatRenderedText;
    sub.CredentialsType = EcSubscriptionCredDefault;
    sub.SubscriptionStatus = true;
    sub.SubscriptionType = EcSubscriptionTypeCollectorInitiated;

    std::wstring eventSource = L"localhost";
    BOOL status = true;
    PEC_VARIANT vEventSource = NULL;
    DWORD dwEventSourceCount;
    EC_VARIANT vSourceProperty;

    // Create a handle to access the event sources array.
    EC_OBJECT_ARRAY_PROPERTY_HANDLE hArray = NULL;

    // The subscription name, URI, and query string must be defined to create
    // the subscription.
```

```
    if ( sub.Name.empty() || sub.URI.empty() || sub.Query.empty() )
    {
        dwRetVal = ERROR_INVALID_PARAMETER;
        goto Cleanup;
    }

    // Step 1: Open the Event Collector subscription.
    hSubscription = EcOpenSubscription(sub.Name.c_str(),
        EC_READ_ACCESS | EC_WRITE_ACCESS,
        EC_CREATE_NEW);
    if ( !hSubscription)
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    // Step 2: Define the subscription properties.
    // Set the Description property that contains a description
    // of the subscription.
    vPropertyValue.Type = EcVarTypeString;
    vPropertyValue.StringVal = sub.Description.c_str();
    if (!EcSetSubscriptionProperty(hSubscription,
        EcSubscriptionDescription,
        NULL,
        &vPropertyValue))
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    // Set the subscription type property (collector initiated).
    vPropertyValue.Type = EcVarTypeUInt32;
    vPropertyValue.UInt32Val = sub.SubscriptionType;
    if (!EcSetSubscriptionProperty(hSubscription,
        EcSubscriptionType,
        NULL,
        &vPropertyValue))
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    // Set the URI property that specifies the URI of all the event sources.
    vPropertyValue.Type = EcVarTypeString;
    vPropertyValue.StringVal = sub.URI.c_str();
    if (!EcSetSubscriptionProperty(hSubscription,
        EcSubscriptionURI,
        NULL,
        &vPropertyValue))
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    // Set the Query property that defines the query used by the event
    // source to select events that are forwarded to the event collector.
    vPropertyValue.Type = EcVarTypeString;
```

```
        vPropertyValue.StringVal = sub.Query.c_str();
        if (!EcSetSubscriptionProperty(hSubscription,
            EcSubscriptionQuery,
            NULL,
            &vPropertyValue))
        {
            dwRetVal = GetLastError();
            goto Cleanup;
        }

        // Set the Log File property that specifies where the forwarded events
        // will be stored.
        vPropertyValue.Type = EcVarTypeString;
        vPropertyValue.StringVal = sub.DestinationLog.c_str();
        if (!EcSetSubscriptionProperty(hSubscription,
            EcSubscriptionLogFile,
            NULL,
            &vPropertyValue))
        {
            dwRetVal = GetLastError();
            goto Cleanup;
        }

        // Set the ConfigurationMode property that specifies the mode in which events
        // are delivered.
        vPropertyValue.Type = EcVarTypeUInt32;
        vPropertyValue.UInt32Val = sub.ConfigMode;
        if (!EcSetSubscriptionProperty(hSubscription,
            EcSubscriptionConfigurationMode,
            NULL,
            &vPropertyValue))
        {
            dwRetVal = GetLastError();
            goto Cleanup;
        }

        // If the Configuration Mode is Custom, set the DeliveryMode, DeliveryMaxItems,
        // HeartbeatInterval, and DeliveryMaxLatencyTime properties.
        if ( sub.ConfigMode == EcConfigurationModeCustom)
        {
            // Set the DeliveryMode property that defines how events are delivered.
            // Events can be delivered through either a push or pull model.
            vPropertyValue.Type = EcVarTypeUInt32;
            vPropertyValue.UInt32Val = sub.DeliveryMode;
            if (!EcSetSubscriptionProperty(hSubscription,
                EcSubscriptionDeliveryMode,
                NULL,
                &vPropertyValue))
            {
                dwRetVal = GetLastError();
                goto Cleanup;
            }

            // Set the DeliveryMaxItems property that specifies the maximum number of
            // events that can be batched when forwarded from the event sources.
            vPropertyValue.Type = EcVarTypeUInt32;
            vPropertyValue.UInt32Val = sub.MaxItems;
```

```
            if (!EcSetSubscriptionProperty(hSubscription,
                EcSubscriptionDeliveryMaxItems,
                NULL,
                &vPropertyValue))
            {
                dwRetVal = GetLastError();
                goto Cleanup;
            }

            // Set the HeartbeatInterval property that defines the time interval, in
            // seconds, that is observed between the heartbeat messages.
            vPropertyValue.Type = EcVarTypeUInt32;
            vPropertyValue.UInt32Val = sub.HeartbeatInerval;
            if (!EcSetSubscriptionProperty(hSubscription,
                EcSubscriptionHeartbeatInterval,
                NULL,
                &vPropertyValue))
            {
                dwRetVal = GetLastError();
                goto Cleanup;
            }

            // Set the DeliveryMaxLatencyTime property that specifies how long, in
            // seconds, the event source should wait before forwarding events.
            vPropertyValue.Type = EcVarTypeUInt32;
            vPropertyValue.UInt32Val = sub.MaxLatencyTime;
            if (!EcSetSubscriptionProperty(hSubscription,
                EcSubscriptionDeliveryMaxLatencyTime,
                NULL,
                &vPropertyValue))
            {
                dwRetVal = GetLastError();
                goto Cleanup;
            }
        }

        // Set the ContentFormat property that specifies the format for the event content.
        vPropertyValue.Type = EcVarTypeUInt32;
        vPropertyValue.UInt32Val = sub.ContentFormat;
        if (!EcSetSubscriptionProperty(hSubscription,
            EcSubscriptionContentFormat,
            0,
            &vPropertyValue))
        {
            dwRetVal = GetLastError();
            goto Cleanup;
        }

        // Set the CredentialsType property that specifies the type of credentials
        // used in the event subscription.
        vPropertyValue.Type = EcVarTypeUInt32;
        vPropertyValue.UInt32Val = sub.CredentialsType;
        if (!EcSetSubscriptionProperty(hSubscription,
            EcSubscriptionCredentialsType,
            0,
            &vPropertyValue))
        {
```

```
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    // Set the Enabled property that is used to enable or disable the subscription
    // or to obtain the current status of a subscription.
    vPropertyValue.Type = EcVarTypeBoolean;
    vPropertyValue.BooleanVal = sub.SubscriptionStatus;
    if (!EcSetSubscriptionProperty(hSubscription,
        EcSubscriptionEnabled,
        0,
        &vPropertyValue))
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    // Get the user name and password used to connect to the event sources
    wcout << "Enter credentials used to connect to the event sources. " << endl <<
        "Enter user name: " << endl;
    wcin >> sub.UserName;
    cout << "Enter password: " << endl;

    wchar_t c;
    while( (c = _getwch()) && c != '\n' && c != '\r' && sub.Password.length() < 512)
    {sub.Password.append(1, c);}

    // Set the CommonUserName property that is used by the local and remote
    // computers to authenticate the user with the source of the events. This
    // property is used across all the event sources available for this subscription.
    vPropertyValue.Type = EcVarTypeString;
    vPropertyValue.StringVal = sub.UserName.c_str();
    if (!EcSetSubscriptionProperty(hSubscription,
        EcSubscriptionCommonUserName,
        NULL,
        &vPropertyValue))
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    // Set the CommonPassword property that is used by the local and remote
    // computers to authenticate the user with the source of the events.
    // Use Credential Manager Functions to handle Password information.
    vPropertyValue.Type = EcVarTypeString;
    vPropertyValue.StringVal = sub.Password.c_str();
    if (!EcSetSubscriptionProperty(hSubscription,
        EcSubscriptionCommonPassword,
        NULL,
        &vPropertyValue))
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    //  When you have finished using the credentials,
    //  erase them from memory.
```

```
        sub.UserName.erase();
        sub.Password.erase();


        //---------------------------------------------
        // Add event sources.
        // Ensure that a handle to the event sources array has been obtained.

        //Initialize the Event Sources Array
        dwRetVal = GetProperty( hSubscription,
            EcSubscriptionEventSources,
            0,
            buffer,
            vEventSource);

        if (vEventSource->Type != EcVarTypeNull  &&
            vEventSource->Type != EcVarObjectArrayPropertyHandle)
        {
            dwRetVal = ERROR_INVALID_DATA;
            goto Cleanup;
        }

        hArray = (vEventSource->Type == EcVarTypeNull)? NULL:
            vEventSource->PropertyHandleVal;
        if(!hArray)
        {
            dwRetVal = ERROR_INVALID_DATA;
            goto Cleanup;
        }
        if (!EcGetObjectArraySize(hArray, &dwEventSourceCount))
        {
            dwRetVal = GetLastError();
            goto Cleanup;
        }

        // Step 3: Add a new event source to the event source array.
        if (!EcInsertObjectArrayElement(hArray,
            dwEventSourceCount))
        {
            dwRetVal = GetLastError();
            goto Cleanup;
        }

        // Set the properties of the event source
        // Set the EventSourceAddress property that specifies the address
        // of the event forwarding computer, this property can be localhost
        // or a fully-qualified domain name.
        vSourceProperty.Type = EcVarTypeString;
        vSourceProperty.StringVal = eventSource.c_str();
        if (!EcSetObjectArrayProperty( hArray,
            EcSubscriptionEventSourceAddress,
            dwEventSourceCount,
            0,
            &vSourceProperty))
        {
            dwRetVal = GetLastError();
            goto Cleanup;
```

```
    }

    // Set the EventSourceEnabled property that enables the event source
    // to forward events.
    vSourceProperty.Type = EcVarTypeBoolean;
    vSourceProperty.BooleanVal = status;
    if (!EcSetObjectArrayProperty(hArray,
        EcSubscriptionEventSourceEnabled,
        dwEventSourceCount,
        0,
        &vSourceProperty))
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    //------------------------------------------------
    // Step 3: Save the subscription.
    // Save the subscription with the associated properties
    // This will create the subscription and store it in the
    // subscription repository
    if( !EcSaveSubscription(hSubscription, NULL) )
    {
        dwRetVal = GetLastError();
        goto Cleanup;
    }

    // Step 4: Close the subscription.
Cleanup:
    if(hSubscription)
        EcClose(hSubscription);
    if(hArray)
        EcClose(hArray);

    if (dwRetVal != ERROR_SUCCESS)
    {
        FormatMessageW( FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
            NULL,
            dwRetVal,
            0,
            (LPWSTR) &lpwszBuffer,
            0,
            NULL);

        if (!lpwszBuffer)
        {
            wprintf(L"Failed to FormatMessage.  Operation Error Code: %u." \
                L"Error Code from FormatMessage: %u\n", dwRetVal, GetLastError());
            return;
        }

        wprintf(L"\nFailed to Perform Operation.\nError Code: %u\n" \
            L" Error Message: %s\n", dwRetVal, lpwszBuffer);

        LocalFree(lpwszBuffer);
    }
}
```

```
DWORD GetProperty(EC_HANDLE hSubscription,
                  EC_SUBSCRIPTION_PROPERTY_ID propID,
                  DWORD flags,
                  std::vector<BYTE>& buffer,
                  PEC_VARIANT& vProperty)
{
    DWORD  dwBufferSize, dwRetVal = ERROR_SUCCESS;
    buffer.resize(sizeof(EC_VARIANT));

    if (!hSubscription)
        return ERROR_INVALID_PARAMETER;

    // Get the value for the specified property.
    if (!EcGetSubscriptionProperty(hSubscription,
        propID,
        flags,
        (DWORD) buffer.size(),
        (PEC_VARIANT)&buffer[0],
        &dwBufferSize) )
    {
        dwRetVal = GetLastError();

        if (ERROR_INSUFFICIENT_BUFFER == dwRetVal)
        {
            dwRetVal = ERROR_SUCCESS;
            buffer.resize(dwBufferSize);

            if (!EcGetSubscriptionProperty(hSubscription,
                propID,
                flags,
                (DWORD) buffer.size(),
                (PEC_VARIANT)&buffer[0],
                &dwBufferSize))
            {
                dwRetVal = GetLastError();
            }
        }
    }

    if (dwRetVal == ERROR_SUCCESS)
    {
        vProperty = (PEC_VARIANT) &buffer[0];
    }
    else
    {
        vProperty = NULL;
    }

    return dwRetVal;
}
```

Validate that the subscription works correctly

1. On the event collector computer complete the following procedure:

    a. Run the following command from an elevated privilege command prompt to get the runtime status of the subscription:

    wecutil gr <*subscriptionID*>

    b. Verify that the event source has connected. You might need to wait until the refresh interval specified in the policy is over after you create the subscription for the event source to be connected.
    c. Run the following command to get the subscription information:

    wecutil gs <*subscriptionID*>

    d. Get the DeliveryMaxItems value from the subscription information.
2. On the event source computer, raise the events that match the query from the event subscription. The DeliveryMaxItems number of events must be raised for the events to be forwarded.
3. On the event collector computer, validate that the events have been forwarded to the ForwardedEvents log or to the log specified in the subscription.

Related topics

Configure Computers to Forward and Collect Events
Adding an Event Source to an Event Collector Subscription
Windows Event Collector Reference