Global Information Assurance Certification Paper

Building a Custom SIEM Integration for an API-Based Log Source – Azure AD Graph Sign-in Events

GIAC (GSEC) Gold Certification

Author: Jason Mihalow, jason.mihalow@mheducation.com
Advisor: Hamed Khiabani, Ph.D.
Accepted: February 2, 2018

Abstract

Enterprise security breaches can quickly paralyze operations and cripple the ability to do business if security teams are not adequately equipped to collect all critical log data from the services an organization uses. Vendors lead us to believe that we are comprehensively covered with their "out-of-the box" log source integrations. It can be challenging for security professionals to find issues with these integrations and it is usually not until a security incident that we realize that crucial log data is missing. This paper takes a critical look at a hidden gap in "out-of-the-box" integrations in SIEM platforms for API log sources, which we, as security professionals, rely on for our detection and analysis of security incidents. As organizations turn from on premises log sources with push style log delivery methods to cloud-based solutions where logs are pulled from an API endpoint, new issues arise that have not been seen before. These issues can lead to undetected gaps of missing data between the true record of API log data and what is found in the SIEM platform.

1. Introduction

When an attacker is actively exploiting an organization's environment the security team lives and dies by the accuracy and timeliness of the logs they can to collect. The faster they know about an incident and understand what occurred, the faster they can respond it and mitigate any further damage. Today's organizations are increasingly moving to cloud-based versions of their business-critical services. Collecting logs from cloud-based services presents new challenges to monitoring and incident response efforts and, if these challenges are not detected, they could blind an organization to the critical logs necessary for adequate detection and response. Our vendors would have us believe that their "out-of-the-box" integrations for cloud services comprehensively cover our monitoring needs but that assurance appears to be far from the truth and the cloud service providers share some that blame.

1.1. Syslog Style Log Delivery

The standard for log delivery for most on premises appliances and systems involves using the syslog protocol. These systems typically have a syslog configuration page in the user interface where a logging destination can be set. Some systems might even allow for the configuration of additional options like what type of logs will be sent to the destination server. Once enabled the logs are sent to the destination server as they happen, in as close to real-time as possible. The monitoring of these services is straightforward in our SIEM solutions which will either have a native parser or a user-created parser that deciphers the field values in the syslog packets appropriately. The administrator does not need to create a custom way to get the logs from the service or appliance to the SIEM.

Jason Mihalow, Jason.mihalow@mheducation.com

1.2. API-Based Log Delivery

Today many companies are turning from on premises to cloud-based solutions for their services. Cloud-based solutions are more cost-effective in some situations and maintenance of the product is typically the responsibility of the service provider which eases the need to hire local support staff to maintain the solution. The problem with this change in service delivery is that we can no longer push the logs directly to the SIEM from the service. These services are disconnected from our physical environments and information security analysts typically aren't interested in the raw system logs. Information security analysts are more interested in the logs from the application or service that is provided than the underlying hardware logs. Some most critical of these logs are success and failure login events to the application. If an organization does not use an identity provider, such as Okta, to centralize the authentication of its cloud-based applications, then the authentication logs for the application only exists in that service. Cloud-based applications and services utilize what is called an Application Programming Interface (API) to deliver logs and enable data exchange with outside processes and services. The API is a web service that allows for reporting of the logs from the cloud service or to issue commands to control the cloud service.

1.3. Push to Pull

The SIEM products that are available in today's market all have a varying degree of native integrations for the vast amount of different log sources that an organization might use. When sending logs to the SIEM, the on premises solutions all typically use a syslog configuration which is a push type of delivery. The logs are pushed from the

service in question to the logging destination. As organizations move to cloud services, such as Microsoft Office 365, the SIEM vendors have created native integrations for these new API-based logs sources so that SIEM administrators do not have to create them on their own. API-based log sources are "polled" or "queried" to pull the logs from the service to the logging destination. When configuring these log sources in the SIEM; there are a variety of new fields to configure. The SIEM is no longer waiting for the logs to be delivered to it; it actively sends a query to the web-based API endpoint and receives the logs in return. The SIEM may have a system default polling period or the SIEM log source may have a configurable one. This polling period is the amount of time between each successive API query to the API endpoint. Retrieving logs from and API has moved the log delivery method from a push to a pull where we are no longer receiving events as they occur but asking for chunks of time worth of logs, such as the last 10 minutes, and hoping that each chunk of results contains all the logs that exist for the requested period.

1.4. Log Aggregation and Share Resources

There is also a change in log source behavior between a syslog log source and an API log pull. With the syslog style of log delivery, the systems or services send the logs as they were happening to a SIEM. With cloud services, the systems must send logs to a logging facility which in turn presents the data to the customer via the API. The cloud service will usually also have an application web portal that is used to control the service where the same logs can be reviewed. Where the ambiguity begins is when we think about the arrival of cloud logs in the logging service. The cloud service is a distributed service. A distributed service has multiple entry points, usually in regional locations, that are all used simultaneously. Having multiple, regional entry points is beneficial for

service; users can be driven to a regional location for access. Additionally, the cloud services are typically multi-tenant which logically separates data and access from other tenants that also use the same service. However even though data may be logically separated between organizations, the service is being shared on the same backend systems. Because customers are sharing the underlying infrastructure one server could be under more load than another and, although two events were simultaneously, the logs of those events can arrive at the logging facility at different times. The impact of this is that the log results for a query that covers the creation time of the two simultaneous logs might contain only one of those logs.

1.5. Ambiguity in Timeliness of Log Delivery

When we combine both the fact that the logs from all the regions of systems that run a cloud service are aggregated to a logging facility with the fact that one server in a cluster could experience more load than another causing sequential logs to be delivered out-of-order, it is easy to understand that there is some ambiguity in the timeliness of log delivery. While all the logs in the API service eventually reach the logging facility and are presented in the UI and the API endpoint, they can arrive out-of-order. All cloud services are, inescapably, locked into this mold because they all have multiple regions hosting the same service and all need to aggregate their logs. The following is a quote that is posted on a Microsoft informational site regarding the "content blobs" that contain the operational logs of the Office 365 service when obtained through the Microsoft Management API endpoint.

> The content blobs are creating by collecting and aggregating actions and events across multiple servers and datacenters. As a result of this distributed process, the

> actions and events contained in the content blobs will not necessarily appear in the order in which they occurred. One content blob can contain actions and events that occurred prior to the actions and events contained in an earlier content blob. We are working to decrease the latency between the occurrence of actions and events and their availability within a content blob, but we cannot guarantee that they appear sequentially. ("Use the Office 365 Management Activity API," 2017)

Microsoft has confirmed that logs do not arrive in the API endpoint sequentially when using the Management API. However, upon investigating further it was discovered that this held true for the Azure AD Graph API (graph.windows.net) (Baldwin et. al., 2017) as well as the Microsoft Graph API (graph.microsoft.com) ("You can use the Microsoft Graph API," n.d.). All of which are used to monitor the Office 365 services.

1.6. API Drift

When a security team is facing a situation of an unsupported API service where it is necessary to create a method to deliver the logs from an API-based source to a logging server or SIEM, then this ambiguity in the availability of the logs in the API endpoint must be considered in the design of any solution. There will always be a fluctuating gap of time between the time of executing an API query and the timestamp of the latest log from the results of that query, defined in this paper as API drift. API drift is caused, at least in part, because the log of any event must first be aggregated to the logging facility and then be presented in the API before it can be found when querying an API endpoint. It takes time for the logs to reach the log aggregation point and be made available to the API endpoint. A complication comes in because the API drift can vary due outside events such as a sudden traffic increase that drastically increases the number of logs that

must be delivered to the API tenants. Any custom script that was going to poll for the logs from the API endpoint cannot simply ask for the last ten minutes of logs and then in another ten minutes ask to do the same thing with the expectation that the combined result would contain all the logs in those two periods. Since there is a gap in time, API drift, between the execution time of the query and the latest timestamp from the query results, polling the API in blocks of non-overlapping segments of time would cause the log data to be missing some events. The logs closest to the execution time of the query are still making their way through the log aggregation point and are not yet available in the API endpoint. If non-overlapping query periods were used, the SIEM would never know it was missing logs.

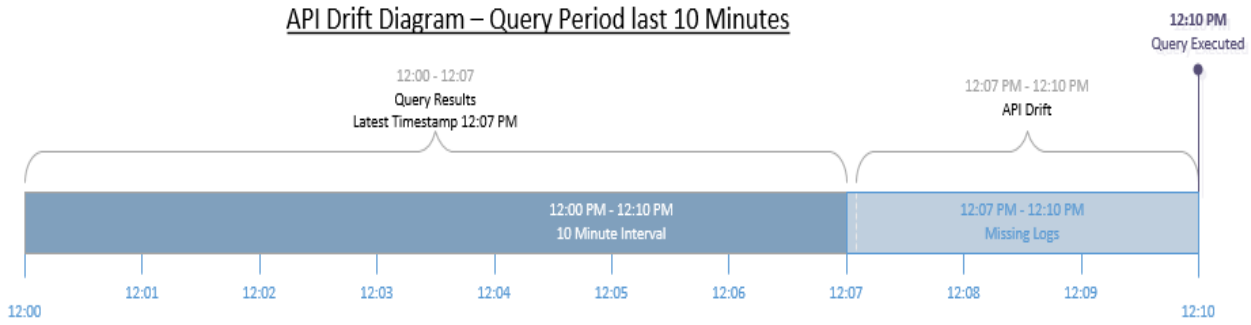

Figure 1 – Graphical demonstration of the term API drift.

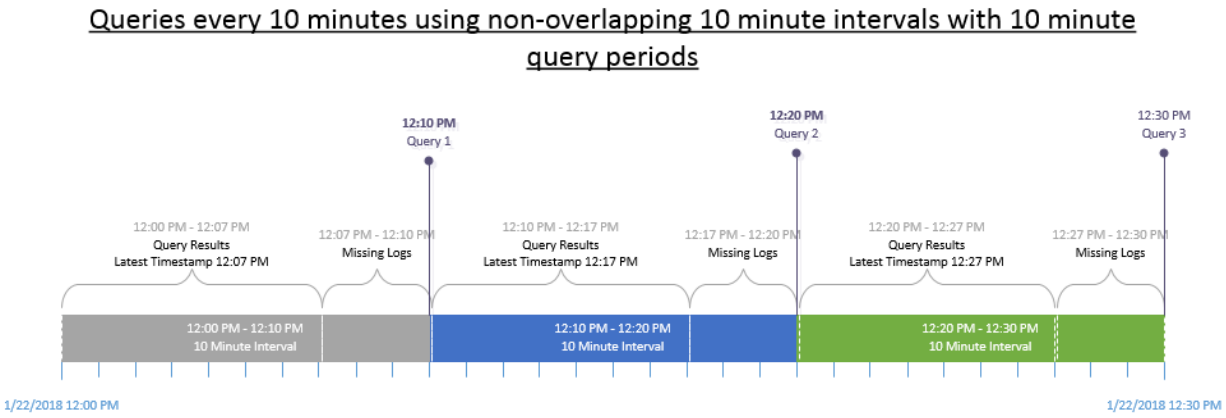

Figure 2 – Graphical demonstration of non-overlapping query intervals.

Jason Mihalow, Jason.mihalow@mheducation.com

1.7. Validate Native Integrations

Validating the data logged in the SIEM by a native "out-of-the-box" parser is another reason to create a custom log integration. The service providers have provided instructions to the SIEM developers on how to create the scripts the SIEM uses to poll the API endpoints and retrieve the logs for the service. However, as with any service, issues do exist that can lead to scenarios where the API drift has become longer than the configured polling period in the SIEM. If this were to happen, the SIEM would, in effect, not receive any logs from the API endpoint but everything would appear to be working. This scenario would be critical to a security team trying to respond to an incident that depends on the logs from an API endpoint. It is sometimes impossible to know exactly how a native integration was written and if it is susceptible to gaps in log data. At the very least, a log export should be done from the web UI of any cloud-based service and compared with the data that is logged by the SIEM to make sure that the SIEM native integration isn't missing any data.

1.8. Collecting Logs for an Office 365 Tenant

Microsoft Office 365 has numerous APIs that can be used to monitor the services they provide which can be quite confusing (Shuai, 2017). There are also multiple ways to access the Office 365 services. Portal.office.com has a web version of all the Microsoft apps. It is also typical for users to connect their mobile devices and fat clients on their workstations to the same services. One API location is the Microsoft Azure AD Graph API (https://graph.windows.net) (Baldwin et. al., 2017). This API contains the authentication logs for the web apps that are behind portal.office.com. The logs presented in this API are also viewable in the Azure portal, https://portal.azure.com. The

sign-in logs reflect the success and failure attempts against the portal.office.com web apps. Another API for Office 365 services is the Management API which exists at manage.office.com ("Use the Office 365 Management Activity API," 2017). This API endpoint has logs that are broken up into the different Microsoft Office 365 workloads (Azure Active Directory, Exchange, SharePoint, OneDrive). The logs from the Management API endpoint are also found in the Microsoft Search and Compliance portal under the audit logs section. Another API is the Microsoft Graph API which is located at https://graph.microsoft.com ("You can use the Microsoft Graph API," n.d). This API is a close duplicate of the manage.office.com API as Microsoft has acknowledged (Shuai, 2017). The SIEM vendors may have a built-in, native integration for the Microsoft Office 365 API logs. Personal experience has shown instances where this integration only utilizes one of the API endpoints. While Microsoft support engineers say that the data source behind the APIs is the same, personal experience has also taught that the APIs do present unique views of that data and there are, consequentially, some logs that are presented by one API and not the other.

1.9. Details of Custom Integration for Azure AD Graph API Sign-in Logs

The goal of this paper is to systematically walk through a script that is a working solution to get the Microsoft Azure AD Graph sign-in logs out of the API and sent to a SIEM (Flores et. al., 2017). The setup requirements of the script can be found in Appendix B. The core design of my script comes from the Microsoft sample of how to poll the Azure AD Graph API endpoint for the sign-in events (Altimore, Tillman, & Vilcinskas, 2017). This custom log source integration will concentrate on these same

sign-in logs, which are for the web applications behind portal.office.com. Although the script concentrates on one API, the framework of the script can be used to create custom API integrations for other services. The framework of the script can also be used as a method of validating the logs from the API log sources that are using native, out-of-the-box integrations. The resulting customized solution is written in Microsoft Powershell because the sample the script started from were also written in Powershell (Altimore, Tillman, and Vilcinskas, 2017). The script is meant to reside on a server and be scheduled to be executed with the local task scheduler. When the script is executed it has a defined or variable amount of time it will query for, depending on how you configure the script. When executed it will send a query to the Azure AD Graph API and ask for a segment of time or query period, such as the last one-hour worth of logs. Each log in the results has a unique ID value associated with it which is used to keep track of which logs have already been sent to the SIEM during successive executions of the script. The script will look for a log file of all the ID values from the previous script execution each time it runs. If it finds this file, it will compare the ID values from the results of the current script execution with the ID values in the file. If it finds a unique ID value, then the script will send that log to the SIEM because it knows that this log hasn't been sent previously to the SIEM. This is to eliminate sending duplicate logs to the SIEM. If the script doesn't find a file with the IDs from the previous execution, then it will send all the logs from the current query results to the SIEM and then write their ID values to the log file, to be read in and compared during the next execution. The script is meant to be scheduled to execute at a configured interval, making sure that the query time is well above any API drift that could be experienced and that the query times will overlap with

each execution of the script. Since Microsoft has clearly stated that logs will arrive out of order, the query time periods must overlap to give the best chance to capture any logs that arrive out of order and to account for any API drift that exists (Harvey, Krishnamoorthy, Tillman, & Vilcinskas, 2017) ("Use the Office 365 Management Activity API," 2017).

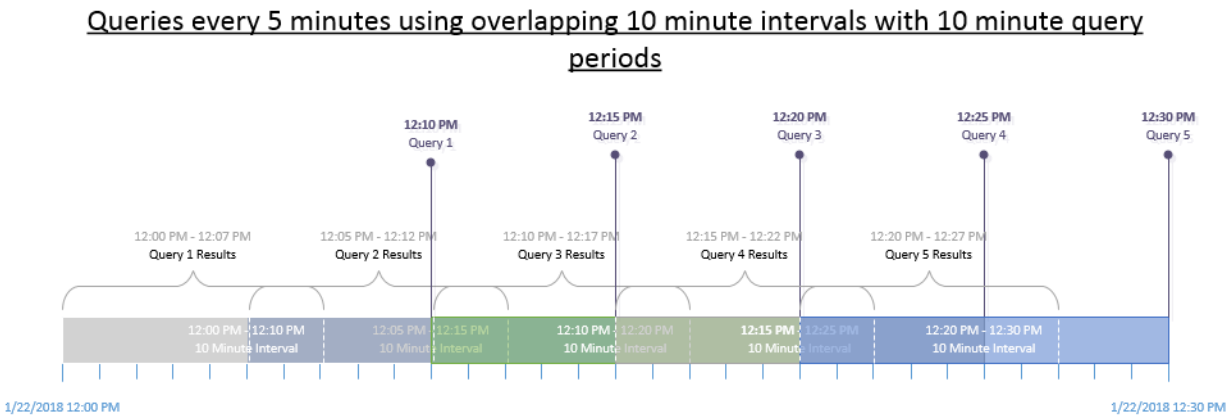

Figure 3 – Graphical demonstration of overlapping query intervals.

1.10. API Drift and Delayed Log Delivery Across All APIs

The findings are that all three Microsoft Office APIs exhibit the same ambiguity in log delivery to constantly varying degrees. There are events that occur on the Office 365 services that puts unplanned for load on the Microsoft systems. While this load in traffic, be it brute-force login attempts for business account take overs or a large increase in legitimate traffic from a valid tenant, does not impact the availability of the service in question but it does impact the delivery of the logs of those services. This is natural because the more logs that must be aggregated, the longer delay in the availability of the logs in the API endpoint. Attacks on another tenant will not impact the availability of that service for your organization but, it can delay the log delivery for both organizations because both organizations share the aggregating log service. This delay in log delivery due to service load is exhibited in increasing API drift. As the load goes up on the

service and the logs availability in the API is delayed, the API drift begins to grow longer. Personal experience with the mange.office.com API logs, specifically successful logins reported by the Azure Active Directory workload, show events being consistently delivered weeks after they occurred. This can be a nightmare for a security team if an incident occurs on an organization's Office 365 tenant and the logs from the incident are still being delivered weeks after it had occurred.

2.0 Main Part

2.1. Azure Registered Application – Prerequisite for API Access

The way that access to the APIs works is that a registered application must be created in the Azure portal (Caserio et. al., 2017) (Altimore et. al, 2017). This application is then granted rights to view the logs and an API key is assigned to it. The API key and the various other pieces of tenant information are what are used as a username/password in the script to gain an access token that can then be used to access the API endpoint.

2.2. OAuth2 Authentication Token Variables

The first step in the script process is to get an OAuth token that can be used to access the Azure AD Graph API endpoint (Altimore et. al, 2017). The script uses the API key, $ClientSecret, and the Microsoft tenant ID, $ClientID, to receive an access token that is used to access the Graph API (Altimore, Tillman, & Vilcinskas, 2017). The $loginURL variable is the web location of the login service that will issue the script the OAuth token and the $resource variable is the Microsoft resource that the token will be valid for. This section, in Figure 4, was taken directly from the Microsoft sample the script is based (Altimore, Tillman, & Vilcinskas, 2017).

```
# This script will require the Web Application and permissions setup in Azure Active Directory
$ClientID       =  # Should be a ~35 character string insert your info here
$ClientSecret   =  # Should be a ~44 character string insert your info here
$loginURL       = "https://login.microsoftonline.com/"
$tenantdomain   = # For example, contoso.onmicrosoft.com
$resource       = "https://graph.windows.net"
```

Figure 4 – This script sample is included in Appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.3. Keeping Track of Time

The next step is to initialize three time variables to keep track of time. The first variable, $universalcurrent_time, is the current time in UTC. The second variable, $query_minutes, is the amount of time the script queries the API endpoint for and is a negative number because the date is in the past. The third variable, $querytime, is the current date minus $query_minutes, formatted for the API to understand it. Always remember to use the ".ToUniversalTime()" function to put the localized results of the cmdlet, get-date, into the UTC time zone. Switching to universal time was one nuance that the base Microsoft sample did not consider.

```
#this is the current time in UTC timezone
$universaltime_current = (get-date).ToUniversalTime()

#assign the amount of minute behind the current you want to start with
$query_minutes = -360

#put the total amount of time being used into a format the API service will accept
$querytime = "{0:s}" -f  (((get-date).AddMinutes($query_minutes)).ToUniversalTime()) + "Z"
```

Figure 5 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.4. Sending the Logs to the SIEM

The next section of code, Figure 6, is used to define the logging destination. The intention was to avoid using external libraries for any part of the script so it would all fit in one file and be easier to understand. This section uses native .NET libraries to open a TCP socket with the configured destination server, $dstserver, over the destination port, $dstport. The script creates a new object, $writer, that is the place to send the text that makes up the log. The script also enables the autoflush function so that the TCP buffer

does not have to be manually emptied to send the logs. The system automatically empties the buffer and sends it to our destination server whenever it is full. TCP was chosen as the transport for the data because a TCP packet can hold four times as much text as a UDP one. It was of critical importance to make sure that the log was received in its entirety by the SIEM to try and avoid truncation of the log data. A sample piece of code posted to Stackoverflow was the source of this idea (Chrbolka, 2015).

```
#SIEM destination IP and port; open TCP stream to SIEM
$dstserver = "127.0.0.1"
$dstport = "1468"
$tcpConnection = New-Object System.Net.Sockets.TcpClient($dstserver, $dstport)
$tcpStream = $tcpConnection.GetStream()
$writer = New-Object System.IO.StreamWriter($tcpStream)
$writer.AutoFlush = $true
```

Figure 6 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.5. Obtaining an OAuth2 Authentication Token from the Login Service

Figure 7 is a copy of code taken from the original Microsoft sample that was used as the basis of this script. The script creates a variable called $body that contains the info required in the HTTP body of the web request used to get an authentication token. The second variable, $oath, is results of the actual web request that uses the cmdlet "Invoke-RestMethod" to send the formatted web request to the Microsoft login URL, $loginURL.

```
# Get an Oauth 2 access token based on client id, secret and tenant domain
$body       = @{grant_type="client_credentials";resource=$resource;client_id=$ClientID;client_secret=$ClientSecret}
$oauth      = Invoke-RestMethod -Method Post -Uri $loginURL/$tenantdomain/oauth2/token?api-version=1.0 -Body $body
```

Figure 7 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.6. Was a Token Received Successfully?

The next section tests if a token was successfully received in the $oauth.access_token variable. If it is not null, then the script successfully received a

token. If this variable is null, the script then sends a syslog message to the SIEM warning that the script was unable to get an access token. There could be many reasons why the request for a token would fail, such as the API key being expired or Microsoft experiencing an issue with the API endpoint. It is critical to alert if the script failed to get a token because that means the script is not sending logs to the SIEM and the SIEM is starting to miss events.

```
if ($oauth.access_token -ne $null)
{
```

Figure 8 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.7. Web Headers for API Query

A successfully obtained token from the Microsoft authentication service can then be used to query the Azure AD Graph API endpoint. The next step is to create a variable, $headerParams, that has the web headers required to use in the API request as shown in Figure 9.

```
$headerParams = @{'Authorization'="$($oauth.token_type) $($oauth.access_token)"}
```

Figure 9 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.8. Set the URL for the API Query

The next variable, $url, is the web URL that is used to query the Azure AD Graph API endpoint. The URL is formatted to include the $tenantdomain variable and utilize the $querytime date that was created during the beginning of the script to filter the results. Sign-in events are accessed from the "/activities/signinEvents" location which is used in Figure 10.

```
#the url creation for the Windows Graph API along with our time frame delimitor
$url = "https://graph.windows.net/$tenantdomain/activities/signinEvents?api-version=beta&`$filter=signinDateTime ge $querytime"
```

Figure 10 – This script sample is included in Appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.9. Arrays to Keep Track of Things

In Figure 11 the script initializes three arrays. The first array, $myReport, stores the current page of results returned from the API query. This page of results has numerous log entries contained within in it. Each API endpoint has a different number of results returned per page. The default, and max, for the Azure AD Graph API is 1000. The next array variable, $DateTimeReport, is used to store the timestamps from all the logs in the current query. The $lastrun_ids array is used to store all the id values from the previous execution of the script.

```
#initialize array that will hold the results from the API query
$myReport = @()

#initialize array that will hold all of the timestamps from logs of the current query
$DateTimeReport = @()

#initialize array to hold ids values from the last execution of the script
$lastrun_ids = @()
```

Figure 11 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.10. Check for IDs from Previous Script Execution

The next step, Figure 12, is to check if a file exists with the list of unique ID values from the logs of the previous script execution. If this file exists, then read it into the variable $lastrun_ids. Reading the file contents into a variable readies the old file for deletion. The script creates a new file with the ID values of the current query results later in the process but, first the old file is deleted. The script uses the process of reading in the old ID values and updating the file with the new ID values to try and eliminate sending duplicate logs to the SIEM. This process is done to compensate for executing the script in overlapping queries which is used to fill in the missing logs caused by the API drift and sequential logs being delivered out-of-order.

```
#test if a file exists with all the ids that were in $myReport from the last run
if (test-path C:\scripts\azure\logs\current_ids.txt)
{
    #assign all the ids from the last run file to a variable
    $lastrun_ids = Get-Content C:\scripts\azure\logs\current_ids.txt

    #delete the existing last run file of ids
    remove-item -Path C:\scripts\azure\logs\current_ids.txt

}
```

Figure 12 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.11. Pagination

The next lesson to learn is what is called pagination. When querying an API endpoint, the results do not return in one blob of all the logs. The results come back in multiple pages of logs. Each page of results has a "next link" value that tells the requestor what the URL is to retrieve the next page of results. Different APIs use different methods of denoting the next page of results. I recommend checking the documentation for any different API services the script is adapted to support. The script uses a loop to process each page of results while the "next link" value is not null. Figure 13 shows that there is an error trap built in to test if the results from the query are empty. If the results are empty, the script sends a warning message to the logging destination to alert that something is wrong and the API data should be investigated. There are two nested "do" loops. The first one is used to loop through the process of comparing the IDs from the current script execution results to the previous and sending the unique logs to the SIEM. The second one is used to control getting each page of results. The second “do” loop also uses a "try and catch" to avoid a crash if the script API queries get throttled. The variable $errorflag is set to $true prior to trying to obtain a page of results. If the "Invoke-WebRequest" cmdlet returns an error, stored in the variable $error, the script prints the error to the screen. An improvement could be to send the error or a field

of the error to the SIEM instead of just printing to the screen which would be missed if you weren't looking watching the execution of the script.

```
#pagination - process each page of results
do
{
    #set the error flag to makes sure we only proceed when we have positive results
    $errorflag = $true

    #loop to get the next page of results
    do
    {
        #try to get the next page of results
        try
        {
            #query the API and assign the results to $myReport; skip to catch if we get an error results
            $myReport = (Invoke-WebRequest -UseBasicParsing -Headers $headerParams -Uri $url)

            #if we are here then we didn't get an error; set the errorflag to $false so we can exit the loop
            $errorflag = $false
```

Figure 13 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.12. Query Throttling

There are limits on the number of successive queries that can be sent to the API endpoint before being throttled ("Throttling limits the number of concurrent," n.d.). After all, even the API endpoint is a shared resource amongst tenants. Query totals are, usually, counted on a per minute basis. The design of this script is to process each page of results, one page at a time as they are received. This process is as opposed to collecting all pages of results first and then processing them all at once. Processing each page of results as they are received allows for gaps in time between successive API queries when the results of the query come back in multiple pages. Each additional page of results is an additional query that counts against the throttling threshold. Sending each page of results through the process as they are received helps to avoid being throttled by spacing the successive queries apart.

2.13. Error Trap for Empty Results

There are scenarios where the script can obtain a valid access token, query the API endpoint, and receive results without any logs. Everything appears to be working

but, there won't be any logs sent to the SIEM. The next step is to test for this scenario and alert the SIEM when experiencing it. As can be seen in Figure 14, the script creates a new array, $emptytest, that then is assigned the log contents of the current $myReport. If the first value in this new array is null, then the results were empty. In this case, the script sends a warning to the SIEM.

```
#need to test if the results were empty; initialize the array $emptytest
$emptytest = @()

#assign all the values in $myReport.Content to the array $emptytest
$emptytest = (($myReport.Content | convertfrom-json).value)

#there was no content in the current query; increase the querytime and alert the SIEM
if (!$emptytest[0])
{
    #send special syslog message to SIEM; API query results were empty; either a Microsoft API issue or a throttling issue.
    $line= "WARNING: API query had zero results.  Drift may be larger than query period"
    $writer.WriteLine($line)
```

Figure 14 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.14. Attempt to Automatically Deal with API Drift

Figure 15, shown below, is intentionally commented out because it is optional and does come with an inherent risk if it is enabled. The section attempts, crudely, to deal with changing API drift that may go above the configured polling period, $query_minutes. A lesson learned from the data used to create this script is that the API drift can go way above the average two hours quoted by Microsoft. When the API drift is greater than the script query period, the result will be a successful query but no logs are returned.

On the opposite side of that, if the query period is set to be too large then one execution of the script might take longer than the script execution interval due to having too many ID values to compare per execution, which again runs the risk of the SIEM missing logs. The script must complete in the script execution interval time and the query periods must overlap to account for the logs arriving out-of-order to give the best chance for a complete log history. The process of balancing the script execution duration

with the amount of time in the query period is a challenge that Figure 15 only partially addresses. This section, if uncommented out, responds to receiving a successful but empty query result by increasing the query period in multitudes of thirty minutes. This section assumes that an empty query response from the API is because the API drift is now longer than the configured initial query period, $query_minutes. If this happens, the script adjusts the query period and tries again. The script also sends a notice to the SIEM that this occurred so that this scenario can be detected and responded to it. The risk of enabling this segment is sending duplicate logs to the SIEM. The script is only keeping track of the ID values of the logs from the last script execution results. Duplicate logs can occur if the query period is adjusted between executions because the script queries beyond the earliest results from the previous successful execution.

The time adjustment has been set to thirty minute increments to try to keep the log duplication to a minimum. This section of code is not necessary but, if the user of the script values a complete log history as up-to-date as possible in the SIEM over having some duplicate logs, then this can help accomplish that. What this section does not deal with is the scenario of the script execution duration going beyond the script execution interval due to having too many ID values to compare between executions. An improvement here would be to test for the size of the log file that contains the IDs from the previous execution and factor that into the timing in some manner. The power of the hardware running this script dictates how large a file of ID values can be and still manage to entirely processes that file within the configured script execution interval.

```
<#
#This section tries to auto adjust the query time until we get results.  This runs the risk of duplication of logs in favor of having the most complete record.

#step back a half hour from the current query time
$query_minutes = $query_minutes - 30

#tell the SIEM about the adjustment
$line = "WARNING: Increasing the query minutes by 30; query_minues are now at $query_minutes"
$writer.WriteLine($line)

#set the querytime to use the new amount of minutes
$querytime = "{0:s}" -f  (((get-date).AddMinutes($query_minutes)).ToUniversalTime()) + "Z"

#set $url to be queried again with the new querytime
$url = "https://graph.windows.net/$tenantdomain/activities/signinEvents?api-version=beta&`$filter=signinDateTime ge $querytime"

#set $errorflag to true so we don't exit the loop and we try the query again
$errorflag = $true

#>
```

Figure 15 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.15. Catch Any Errors from the API Query

In Figure 16 the embedded "do" statement is finished out with a "catch" to catch any errors that arise from the attempted API query with the "Invoke-WebRequest" statement. The catch statement prints the error to the screen and pauses the script for 5 seconds. This error trap assumes that any error that was received from the previous attempt to get a page of results was due to the script being throttled. An improvement here could be trapping for different error messages and responding to them differently. The script keeps attempting to get a page of results from the API until it is successful, sleeping for 5 seconds between queries. We close off the embedded "do" loop with the "while" statement which causes the loop to execute until the variable, $errorflag, equals $false. The variable $errorflag becomes $false if the current API query successfully returns logs in the current page of results.

```
        #catch any error from the previous API query (Invoke-WebRequest) so we don't crash
        catch
        {
            #print the error to the screen
            $Error

            #sleep for 5 seconds and try the query again
            Start-Sleep -s 5
        }

    }while($errorflag -eq $true)
```

Figure 16 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.16. Send Only Unique Logs to the SIEM

Next, the script tests if the array with the log IDs from the previous execution exists. If the array does exist, then the script loops through each log in the current page of results. As the script iterates through the logs, it puts the timestamp from each log in an array named $DateTimeReport. Collecting the timestamps is done so the latest timestamp can be found to compute the API drift from the API query results. After keeping track of the timestamps, the script writes the ID values from all the logs from the current page of results to a new IDs file. The script uses the new file during the next script execution. The final step in this loop is to compare the ID values from the current script execution to the ID values from the previous execution. If any ID values are found to be unique, then the script sends them to the SIEM. The script uses the "Convert-toJson-Compress" function to format the log back in the JSON format and take all the additional blank spaces and lines out to fit the text into the TCP packet.

```
#if there are ids from a previous execution
if ($lastrun_ids)
{
    #loop through each log in the page of results which exists in $myReport.Content
    foreach ($event in (($myReport.Content | ConvertFrom-Json).value))
    {
        #put the event timestamp from the current lgo in the $DateTimeReport array
        [array]$DateTimeReport += $event.signinDateTime

        #send the event id to the new ids file
        $event.id | Out-File -Append -NoClobber -FilePath C:\scripts\azure\logs\current_ids.txt

        #test if the current id is in the list of ids from the previous execution
        $current_id = $event.id
        $unique = $lastrun_ids -notcontains "$current_id"

        #if the event was not in the list from the previous execution
        if ($unique)
        {
            #assign $line to the compressed json
            $line = ($event | Convertto-Json -Compress)

            #send $line to the SIEM view TCP
            $writer.Writeline($line)
        }

    }
}
```

Figure 17 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.17. Keep the Format in JSON

The log format, JSON, was kept intact when sending the logs to the SIEM. All data received from the API is formatted in JSON. There are usually two parts to a custom integration, creating the way to get the logs to the SIEM, which is the focus of the script, and creating the parser that processes the field values in those logs. While the SIEM vendor might not support a log source today, they could in the future. If the SIEM vendor does create an out-of-the-box integration for the new log source, they also create the parser for that log data. This new, native log integration must use the same API data formatted in JSON that your custom integration is receiving. It makes sense to keep it formatted in JSON because it is possible to use a native parser for a custom log source. Keeping the log data in JSON format cuts the maintenance in half by letting your SIEM vendor maintain the parser while you maintain only the log delivery method. Any new

fields that are added to the log source would be taken care of by updates to the native parser from the SIEM vendor.

2.18. No Previous IDs, Send All Logs to SIEM

The following section of the script, Figure 18, deals with the possibility that the script did not find a file with the ID values from a previous execution. In this instance, the script assumes that it is executing for the first time and no logs have been sent to the SIEM yet. Therefore, it sends all the IDs from the current page of results to the SIEM. The script also put the timestamps on the timestamp array and writes all the ID values out to a file for processing during the next script execution.

```
#there was not a file with the IDs from the previous execution
else
{
    #loop through each log in the page of results which exists in $myReport.Content
    foreach ($event in (($myReport.Content | ConvertFrom-Json).value))
    {
        #put the event timestamp from the current lgo in the $DateTimeReport array
        [array]$DateTimeReport += $event.signinDateTime

        #send the event id to the new ids file
        $event.id | Out-File -Append -NoClobber -FilePath C:\scripts\azure\logs\current_ids.txt

        #assign $line to the compressed json
        $line = ($event | Convertto-Json -Compress)

        #send $line to the SIEM view TCP
        $writer.Writeline($line)
    }
}

#update the $url variable with the URL of the next page of query results
$url = ($myReport.Content | ConvertFrom-Json).'@odata.nextLink'
```

Figure 18 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.19. Get the Next Page of Results

The next line in the script, Figure 19, is crucial for pagination. This line updates the URL variable with the URL for the next page of results. For the Microsoft Azure AD API the data is stored in the content blob under the "@odata.nextLink" value. With each page of results comes a URL for the next page of results that the script must use for the next query. We use this value to determine when there are no more pages of results. The

script ends the outer "Do" loop when this variable, $url, is equal to $null, meaning there are no more pages of results to process.

```
    }
    #update the $url variable with the URL of the next page of query results
    $url = ($myReport.Content | ConvertFrom-Json).'@odata.nextLink'

} while($url -ne $null)
```

Figure 19 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.20. Compute API Drift

This point in the script is the end of sending logs from our API query to the SIEM. The following sections of the script are used to compute the current API drift and alert the SIEM when that drift has gone above a configurable time value, such as approaching the query period, $query_minutes. The API drift is computed by sorting the array of timestamps descending and taking the top value. We then compute the time difference from the time of script execution to the latest timestamp from the current script execution results. If the user of this script does not enable the commented out dynamic query time adjustment section, which has a built-in warning when the script adjusts the query time, then this is an alarm that can be helpful if the API drift gets close to the query period. The script sends a warning message to the SIEM that can alert the SIEM operators that the script is running the risk of missing logs and the query period might need to be adjusted. If the API drift goes above the query period, the script works without error but there won't be any logs sent to the SIEM. The configured time gap is set at 45 minutes in this example. A user of this script would want to set this value to something near the value of $query_minutes to detect when the API drift was at a dangerous level.

```
#sort the DateTimeReport decending
$DateTimeReport = $DateTimeReport | sort -Descending

#the top array value is the latest timestamp; assign to timestamp type variable
[datetime]$latest_stamp = $DateTimeReport[0]

#find the differce in time between the latest_stamp and the current time; assign it to a variable
$diff = New-TimeSpan -Start $latest_stamp.ToUniversalTime() -End $universaltime_current

#test to write the current API drift to the screen
#write-output ($diff | select Hours,Minutes,Seconds)

#if the time differnce is greater than 45 minutes
if ((($diff.Minutes) -gt 45) -or (($diff.Hours) -ge 1))
{
    #send special syslog message to SIEM warning of the API drift
    $line= "WARNING: API Logs are " + ($diff | select Days,Hours,Minutes,Seconds) +" behind the current time."
    $writer.writeline($line)
}
```

Figure 20 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

2.21. Cleanup

The final section of code, Figure 21, closes the TCP connection with the SIEM to make sure the last data that is in the TCP buffer is flushed and sent to the SIEM. The last error trap, the else statement below, is used to send a warning message to the SIEM if the script fails to obtain an OAuth token from the login service. There could be many reasons why the script would fail to get a token but, this will, at least, get the notice to the SIEM team that the script needs investigating.

```
#close the TCP socket with the SIEM
$writer.Close()
$tcpConnection.Close()

}
#we failed to obtain an access token
else
{
    #let SIEM know we are having issues with the API authentication
    $line = "WARNING: API authentication issue.  We were unable to obtain an OAUTH token.  Check API key validity."
    $writer.WriteLine($line)
}
```

Figure 21 – This script sample is included in appendix A in Azure_Signin_Activity_to_Syslog.ps1

3.0. Conclusion

The authentication data presented by the Azure AD Graph API is a critical log source for any organization using Office 365. Attackers are aware that authentication logs for the Office 365 web apps portal, portal.office.com, are only stored in the Microsoft APIs if an organization does not federate their authentication. Attackers also know that authentication to portal.office.com utilizes the internal accounts of the owning tenant and that access to the web apps will give the attacker access to the same data as internal employees on their fat clients. This knowledge makes portal.office.com an enticing part of the attack surface for any Office 365 tenant. The security teams of these organizations can be unaware that they are missing incidents because their current SIEM solution has a native integration for Office 365 but, it uses an API other than the Azure AD Graph API. API drift and other scenarios that cause the SIEM to stop receiving logs from an API log source are not uniquely a Microsoft issue. It is an absolute necessity for security teams to take a critical look at the data that is coming from their API-based services to ensure they are not being blinded by either their SIEM's native integrations or interruptions caused by the service providers. Hopefully, the ideas set forth in this paper enable other information security professionals to not only integrate a log source they do not currently have and must support but, also to use the script as a template to validate the data coming from other "out-of-the-box" integrations.

Jason Mihalow, Jason.mihalow@mheducation.com

References

Altimore, et. al. (2017, February 8). Authorize access to web applications using OAuth 2.0 and Azure Active Directory. Retrieved from https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-protocols-oauth-code

Altimore, Tillman, & Vilcinskas (2017, October 18). Azure Active Directory sigin-in activity report API Samples. Retrieved from https://docs.microsoft.com/en-us/azure/active-directory/active-directory-reporting-api-sign-in-activity-samples

Baldwin, et.al. (2017, April 4). Azure Active Directory Graph API. Retrieved from https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-graph-api

Caserio, et. al. (2017, November 14). Prerequisites to access the Azure AD reporting API. Retrieved from https://docs.microsoft.com/en-us/azure/active-directory/active-directory-reporting-api-prerequisites-azure-portal

Chrbolka, Jan (2015, April). How to connect to TCP socket with powershell to send and receive data? Retrieved from https://stackoverflow.com/questions/29759854/how-to-connect-to-tcp-socket-with-powershell-to-send-and-receive-data

Flores, et. al. (2017, July 17). What is Azure Active Directory? Retrieved from https://docs.microsoft.com/en-us/azure/active-directory/active-directory-whatis

Harvey, Krishnamoorthy, Tillman, & Vilcinskas (2017, December 15). Azure Active Directory reporting latencies. Retrieved from https://docs.microsoft.com/en-us/azure/active-directory/active-directory-reporting-latencies-azure-portal

Shuai, Wu (2017, September 14). The difference between the tokens used by Microsoft Graph API and Azure AD Graph API. Retrieved from https://blogs.msdn.microsoft.com/wushuai/2016/09/14/the-difference-between-the-tokens-used-by-microsoft-graph-and-azure-ad-graph/

Throttling limits the number of concurrent. (n.d.). Microsoft Graph Throttling Guidance. Retrieved from https://developer.microsoft.com/en-us/graph/docs/concepts/throttling

Tillman, Solbakken Mellum, & Vilcinskas (2017, October 18). Azure Active Directory Sign-in Activity Report API Reference. Retrieved from

Jason Mihalow, Jason.mihalow@mheducation.com

https://docs.microsoft.com/en-us/azure/active-directory/active-directory-reporting-api-sign-in-activity-reference

Use the Office 365 management activity API. (2017, November 3). Office 365 Management Activity API Reference. Retrieved from https://msdn.microsoft.com/en-us/office-365/office-365-management-activity-api-reference

You can use the Microsoft Graph API. (n.d.). Overview of Microsoft Graph. Retrieved from https://developer.microsoft.com/en-us/graph/docs/concepts/overview

Appendix A

```
#Save as – Azure_Signin_Activity_to_syslog.ps1
#This script is meant to send the Azure signin activity logs to a SIEM via syslog.
#It was created by Jason Mihalow

# This script will require the Web Application and permissions setup in Azure Active Directory
$ClientID       =  # Should be a ~35 character string insert your info here
$ClientSecret   =  # Should be a ~44 character string insert your info here
$loginURL       = "https://login.microsoftonline.com/"
$tenantdomain   = # For example, contoso.onmicrosoft.com
$resource       = "https://graph.windows.net"

#this is the current time in UTC timezone
$universaltime_current = (get-date).ToUniversalTime()

#assign the amount of minute behind the current you want to start with
$query_minutes = -360

#put the total amount of time being used into a format the API service will accept
$querytime = "{0:s}" -f  (((get-date).AddMinutes($query_minutes)).ToUniversalTime()) + "Z"

#SIEM destination IP and port; open TCP stream to SIEM
$dstserver = "127.0.0.1"
$dstport = "1468"
$tcpConnection = New-Object System.Net.Sockets.TcpClient($dstserver, $dstport)
$tcpStream = $tcpConnection.GetStream()
$writer = New-Object System.IO.StreamWriter($tcpStream)
$writer.AutoFlush = $true

# Get an Oauth 2 access token based on client id, secret and tenant domain
$body       =
@{grant_type="client_credentials";resource=$resource;client_id=$ClientID;client_secret=$ClientSecret}
$oauth      = Invoke-RestMethod -Method Post -Uri $loginURL/$tenantdomain/oauth2/token?api-version=1.0
-Body $body

if ($oauth.access_token -ne $null)
{
   $headerParams = @{'Authorization'="$($oauth.token_type) $($oauth.access_token)"}

   #the url creation for the Windows Graph API along with our time frame delimiter
   $url = "https://graph.windows.net/$tenantdomain/activities/signinEvents?api-
version=beta&`$filter=signinDateTime ge $querytime"

   #initialize array that will hold the results from the API query
   $myReport = @()

   #initialize array that will hold all of the timestamps from logs of the current query
   $DateTimeReport = @()

   #initialize array to hold ids values from the last execution of the script
   $lastrun_ids = @()

   #test if a file exists with all the ids that were in $myReport from the last run
   if (test-path C:\scripts\azure\logs\current_ids.txt)
   {
      #assign all the ids from the last run file to a variable
      $lastrun_ids = Get-Content C:\scripts\azure\logs\current_ids.txt

      #delete the existing last run file of ids
      remove-item -Path C:\scripts\azure\logs\current_ids.txt
```

```
}

#pagination - process each page of results
do
{
   #set the error flag to makes sure we only proceed when we have positive results
   $errorflag = $true

   #loop to get the next page of results
   do
   {
      #try to get the next page of results
      try
      {
         #query the API and assign the results to $myReport; skip to catch if we get an error results
         $myReport = (Invoke-WebRequest -UseBasicParsing -Headers $headerParams -Uri $url)

         #if we are here then we didn't get an error; set the errorflag to $false so we can exit the loop
         $errorflag = $false

         #need to test if the results were empty; initialize the array $emptytest
         $emptytest = @()

         #assign all the values in $myReport.Content to the array $emptytest
         $emptytest = (($myReport.Content | convertfrom-json).value)

         #there was no content in the current query; increase the querytime and alert the SIEM
         if (!$emptytest[0])
         {
            #send special syslog message to SIEM; API query results were empty; either a Microsoft API
issue or a throttling issue.
            $line= "WARNING: API query had zero results.  Drift may be larger than query period"
            $writer.WriteLine($line)

            <#
            #This section tries to auto adjust the query time until we get results.  This runs the risk of
duplication of logs in favor of having the most complete record.

            #step back a half hour from the current query time
            $query_minutes = $query_minutes - 30

            #tell the SIEM about the adjustment
            $line = "WARNING: Increasing the query minutes by 30; query_minues are now at
$query_minutes"
            $writer.WriteLine($line)

            #set the query time to use the new amount of minutes
            $querytime = "{0:s}" -f  (((get-date).AddMinutes($query_minutes)).ToUniversalTime()) + "Z"

            #set $url to be queried again with the new query time
            $url = "https://graph.windows.net/$tenantdomain/activities/signinEvents?api-
version=beta&`$filter=signinDateTime ge $querytime"

            #set $errorflag to true so we don't exit the loop and we try the query again
            $errorflag = $true

            #>
         }
      }
      #catch any error from the previous API query (Invoke-WebRequest) so we don't crash
      catch
```

```
    {
        #print the error to the screen
        $Error

        #sleep for 5 seconds and try the query again
        Start-Sleep -s 5
    }

}while($errorflag -eq $true)


#if there are ids from a previous execution
if ($lastrun_ids)
{
    #loop through each log in the page of results which exists in $myReport.Content
    foreach ($event in (($myReport.Content | ConvertFrom-Json).value))
    {
        #put the event timestamp from the current lgo in the $DateTimeReport array
        [array]$DateTimeReport += $event.signinDateTime

        #send the event id to the new ids file
        $event.id | Out-File -Append -NoClobber -FilePath C:\scripts\azure\logs\current_ids.txt

        #test if the current id is in the list of ids from the previous execution
        $current_id = $event.id
        $unique = $lastrun_ids -notcontains "$current_id"

        #if the event was not in the list from the previous execution
        if ($unique)
        {
            #assign $line to the compressed json
            $line = ($event | Convertto-Json -Compress)

            #send $line to the SIEM view TCP
            $writer.Writeline($line)
        }

    }
}

#there was not a file with the IDs from the previous execution
else
{
    #loop through each log in the page of results which exists in $myReport.Content
    foreach ($event in (($myReport.Content | ConvertFrom-Json).value))
    {
        #put the event timestamp from the current lgo in the $DateTimeReport array
        [array]$DateTimeReport += $event.signinDateTime

        #send the event id to the new ids file
        $event.id | Out-File -Append -NoClobber -FilePath C:\scripts\azure\logs\current_ids.txt

        #assign $line to the compressed json
        $line = ($event | Convertto-Json -Compress)

        #send $line to the SIEM view TCP
        $writer.Writeline($line)
    }
}

#update the $url variable with the URL of the next page of query results
$url = ($myReport.Content | ConvertFrom-Json).'@odata.nextLink'
```

```
    } while($url -ne $null)

#sort the DateTimeReport descending
$DateTimeReport = $DateTimeReport | sort -Descending

#the top array value is the latest timestamp; assign to timestamp type variable
[datetime]$latest_stamp = $DateTimeReport[0]

#find the difference in time between the latest_stamp and the current time; assign it to a variable
$diff = New-TimeSpan -Start $latest_stamp.ToUniversalTime() -End $universaltime_current

#test to write the current API drift to the screen
#write-output ($diff | select Hours,Minutes,Seconds)

#if the time difference is greater than 45 minutes
if ((($diff.Minutes) -gt 45) -or (($diff.Hours) -ge 1))
{
    #send special syslog message to SIEM warning of the API drift
    $line= "WARNING: API Logs are " + ($diff | select Days,Hours,Minutes,Seconds) +" behind the current
time."
    $writer.writeline($line)
}

#close the TCP socket with the SIEM
$writer.Close()
$tcpConnection.Close()

}
#we failed to obtain an access token
else
{
    #let SIEM know we are having issues with the API authentication
    $line = "WARNING: API authentication issue.  We were unable to obtain an OAUTH token.  Check API
key validity."
    $writer.WriteLine($line)
}
```

Appendix B

Script Installation Instructions

1) Create new directory on server – C:\scripts\azure\logs
2) Copy script to C:\script\azure
3) Edit all the critical variables in the script (eg. $query_minutes, $dstserver, $dstport)
4) Using the task scheduler, schedule a new task to execute the new script. Schedule this to be execute at the desired execution interval (eg. every 3 minutes)

Remember that the configured query interval in the script ($query_minutes) should be greater than the script execution internal. The idea is to have overlapping query results. Remember to make sure that the script execution is completing in the execution interval.

Jason Mihalow, Jason.mihalow@mheducation.com