Overview

This document describes the design and operation of an automated External Dynamic List (EDL) solution, built to enable rapid, hands-off blocking of malicious IPs in a firewall. The system leverages Azure Storage, Azure Functions, and SentinelOne integration to maintain a cloud-hosted, dynamically updated EDL.

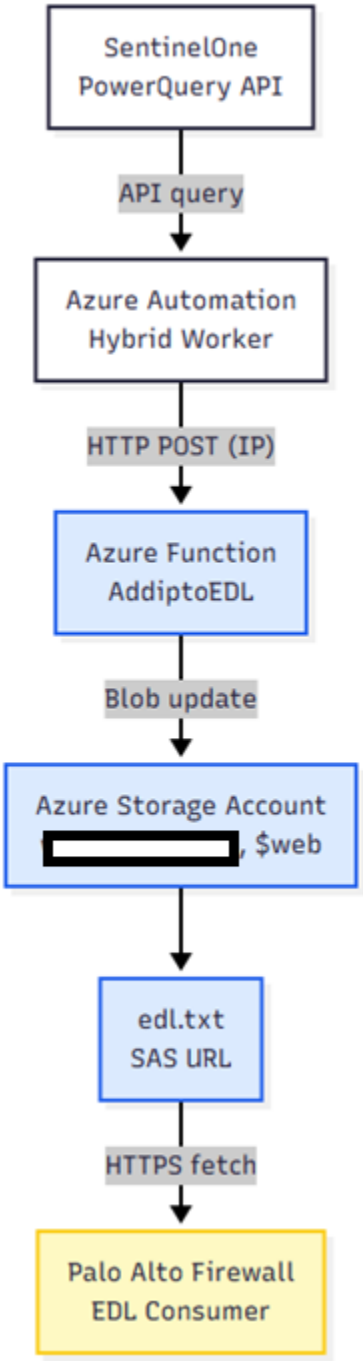
The flow is as follows.

A powershell script performs a Power Query to the SentinelOne datalake using an API

The returned data from the query is submitted to an Azure Function that checks it for duplicate entries and otherwise submits to a blob hosted in a Storage Container in your Azure tenant.

The blob is hosted using a “static website” configured on the storage account

The contents of the file are ingested by your Firewall to provide the contents of the External Dynamic (block)List.



Getting started

Using this step-by-step guide, we will be creating and configuring the components required for this setup.

We will be creating:

- **A SentinelOne API Token to use with our Collector**
- **A Parser to turn our log into fields we can query using Power Queries**
- **A SentinelOne Collector**
- **An Azure Storage Account**
- **A Storage Container**
- **A Blob (file)**
- **An Azure Function with HTTP trigger**
- **A Runbook that runs on a fixed interval**

Generate API Key

First generate a new API key in SentinelOne

Log into your SentinelOne Console (make sure you are using the Singularity Operations Center UI).

Using the left-hand navigation menu go to Policies and Settings.

Under category AI SIEM you should see API keys.

Generate a new KEY, give it Write access and store the KEY value somewhere so we can use it later to configure the SentinelOne Collector

Parser

We need a placeholder parser so we can include a reference for it in our Collector config.

Using the left-hand navigation menu in the SentinelOne Management console go to Policies and Settings.

Under category AI SIEM you should see Parsers

Click Add parser, give it a relevant name and accept the placeholder JSON value of the parser for now, we will be changing it later depending on our log source.

SentinelOne Collector

To port your own logs to SentinelOne so we can parse and query them we need to configure a SentinelOne Collector. There are several ways to do this.

SentinelOne Collector installation quick start

Our preferred installation method was using Docker, but I am sure you can get it to work on any platform.

The installation instructions for the Docker based install have changed, but because I am unfamiliar with them, I will share my installation method which still works.

- Install Docker on your OS of choice Install | Docker Docs
- Copy the docker-compose.yaml and syslog.yaml files from this repo to your docker host
- Alter the syslog.yaml file, configure the port it should listen on (default is udp-714)
- Alter the name, api-token that you generated in your S1 tentant and destination (default destination for EU customers is **xdr.eu1.sentinelone.net**)
- Alter the parser value to reflect the name of the custom parser as it exists in SentinelOne
- From the directory that contains the .yaml files to run **docker compose up -d** to compose and start the docker image.
- Configure the docker host (ip or fdqn) as syslog server on the device that you want to collect logs from.

Reach out for support from SentinelOne if you are having trouble getting it to work as my instructions are a little outdated.

Designing your Parser

With your logs now being streamed to the SentinelOne XDR Datalake, the EDL object created and having configured methods to access it we can move on to writing the parser to process the logs into fields that are easier to query and filter using Power Queries.

Each log source is going to have a slightly different logging format, so your parser is always going to be customized to your source.

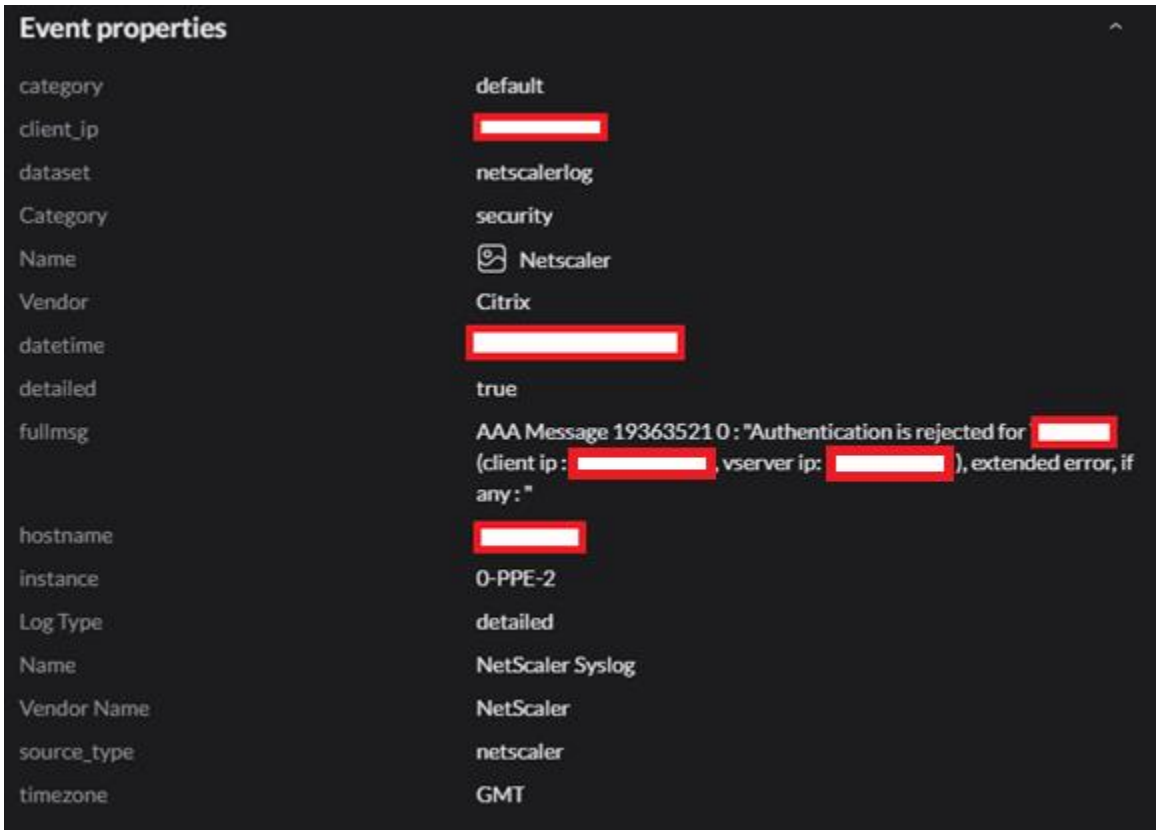
The parser uses REGEX queries to map chunks a line into separate fields and has some identifying properties that allow us to query just these logs later using our Power Query.

My tip would be to use the parser included in the repo as a reference and use your AI tool of choice to adapt it to the format of your log. Unless you are a REGEX expert, then feel free to have a crack at it yourself.

I find the documentation around the parsers a bit lacking, but SentinelOne has lots of examples that showcase the different ways to make your parser as clever as you need it to be.

Without a parser the logs end up in the datalake unprocessed and will just be a long line of text per event, making it nearly impossible to query.

A correctly parsed event has values for the fields as you configured them in your parser.



Design your Power Query

Once you can parse your events into fields we can query them to determine who is in violation of your intended thresholds. In my example it concerns a certain amount of rejected logins within a set timeframe.

In my example, the relevant bit of information is in the fullmsg field, it contains the fact that the login was rejected and it contains the originating IP of the login.

You can use Power Queries to further parse data from fields, in this case I grab the IP from the fullmsg field and map it to a "client ip" column. Not to be confused with the client_ip field of the parent event (small naming convention error on my part).

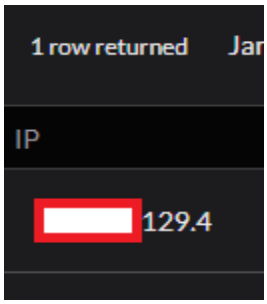
Once we grab the relevant bit of information from the field we can start to count how many events there are in the desired timespan. The timespan is important, as it will determine who gets blocked and who doesn't. In my example I set the timespan to 2 hours and the threshold to 10 rejected logins.

The resulting query looks like this (I added some comments, remove those before trying them yourself)

```
## Only retrieve events from my source that contain the word rejected in the fullmsg field
dataSource.category = 'security'
| filter( dataSource.name == 'netscaler' AND fullmsg contains:anycase( 'Rejected' ) )
## Grab the client IP by using the parse argument so we can output it later
| parse 'client ip : $client_ip$,' from fullmsg
##  Group amount of events by client IP
| group EventCount = count() by IP = client_ip
## Only show me IPs that have 10+ events for the configured timespan)
| filter( EventCount >= 10 )
## Return the list as just the IPs
| columns IP
```

Full query:

```
dataSource.category = 'security' | filter( dataSource.name == 'netscaler' AND fullmsg contains:anycase( 'Rejected' ) ) | parse 'client ip : $client_ip$,' from fullmsg | group EventCount = count() by IP = client_ip | filter( EventCount >= 10 ) | columns IP
```



We will need to save this query because we are going to use the S1 API to retrieve the response in our Runbook later.

Storage and EDL Hosting

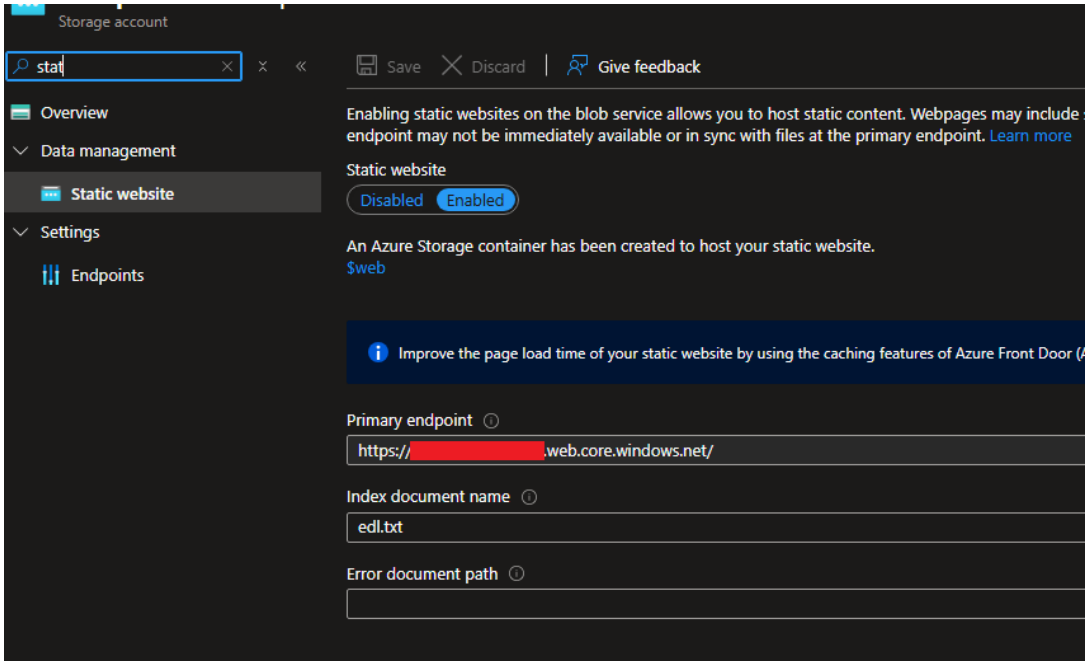
The EDL file is a plain text list of IP addresses, one per line.

We need 2 methods to access this file.
One that supports write access so we can use it in the Azure Function to write new IPs to the list. One to access the file using a static URL (read-only) so our firewall can ingest it as EDL.

For the write method our easiest option is using the access key for the storage account and using a python function to utilize it in the Function App, this is not the best method but will do for this example. Best practice would be to use a Managed Identity for the Function App that has permission on the blob, but this requires more setup.

For the read method this example is going to use the Static Website option on the storage account. This results in a URL that links directly to the file that ends in .txt as opposed to a long SAS URL that most firewalls do not support as EDL.

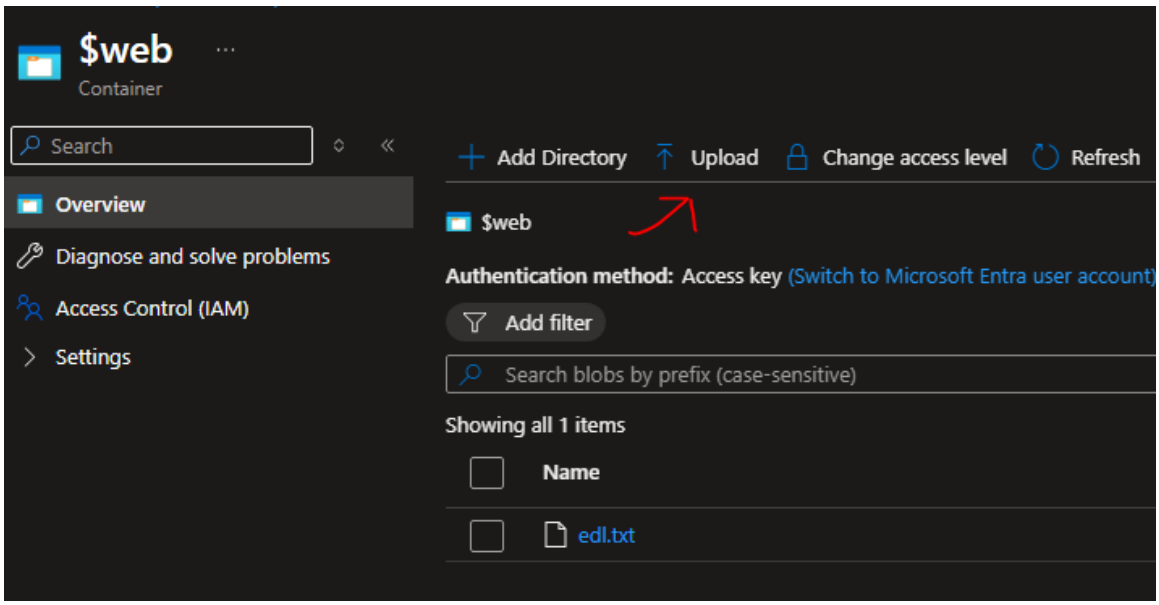
First create a new storage account and enable “Static Website” on it.
This creates a storage container called $web in which we create our blob that contains the EDL.



Note the Primary Endpoint URL, this url + the path to your EDL blob is what we configure in the firewall later. For example:

https://blobbyblob.web.core.windows.net/edl.txt

Upload an empty .txt file to the resulting $web container.
This is going to be the source of our EDL.



Set up the Azure Function

(Please note that the steps outlined are not exactly best practice, they are slightly simplified for a quick setup so you can try it out, I recommend optimizing for minimal access before going full PROD)

The Azure Function is going to receive the data returned by the Power Query runbook that we will be setting up in the next step.

Each IP returned by the power query the runbook job is going to be submitted individually to the Azure function.

The Azure function is going to check if the IP is already present on the list, if not, it's going to add it.
Before we begin, grab an Account Key from the storage account you created early (Access Keys blade)

The code from the repo is ready with a few modifications.

Add new Function App in Azure with a HTTP trigger

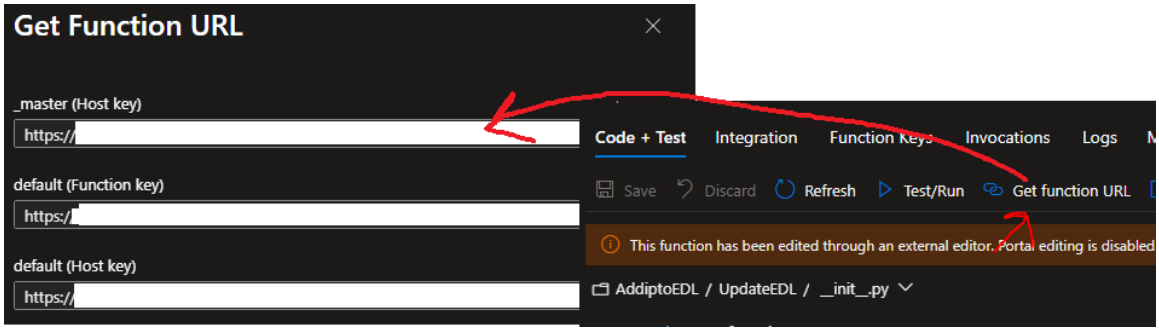
Change the container_name value to the name of the container in your Storage account.

Change blob name to the file you created to store your EDL

Create an Environment Variable in your Function App called AzureWebJobsStorage
and set it to the following value:
DefaultEndpointsProtocol=https;AccountName=NAME-OF-STORAGE-ACCOUNT;AccountKey=INSERT-ACCOUNT-KEY==;EndpointSuffix=core.windows.net

```
# The connection string is automatically pulled from the AzureWebJobsStorage setting
STORAGE_CONNECTION_STRING = os.getenv("AzureWebJobsStorage")
CONTAINER_NAME = '$web'
BLOB_NAME = 'edl.txt'
```

Once everything is created, grab the function URL that was generated, as we need it for the next step.



Using the SentinelOne API to perform Power queries

The runbook (see repo) that is going to perform the API call and pass the returned data to the Azure Function uses PowerShell but can easily be translated to Python.

For you to include it in your setup all you need to do is set the 3 variables below,

- SentinelOne API key we generated earlier
- SentinelOne Tenant URL, for EU that is usually **xdr.eu1.sentinelone.net**
- Azure Function URL we copied from the previous step

I use an Azure Automation Account for my runbook, so I use specific cmdlets that allow me to retrieve values for the variables from my vault, but you can use any method to set these 3 variables.

You need to edit the Power query you wish to run and the desired timespan for the data.

I have set the runbook to every hour and the timespan for the data set to 2 hours.

```
# === Load Automation Variables ===
$apiToken         = Get-AutomationVariable -Name "SentinelOneApiToken"
$tenantUrl        = Get-AutomationVariable -Name "SentinelOneTenantUrl"
$azureFunctionUrl = Get-AutomationVariable -Name "EDLAzureFunctionUrl"
```

```
# === PowerQuery ===
$query = @'
dataSource.category = 'security' | filter( dataSource.name ==
'@

# === Prepare Request ===
$url = "https://$tenantUrl/api/powerQuery"
$headers = @{
    "Authorization" = "Bearer $apiToken"
    "Content-Type"  = "application/json"
}
$body = @{
    "query"     = $query
    "startTime" = "2h"
} | ConvertTo-Json
```

Determines the data timespan

Sidenotes

EDL Update Logic (Azure Function)

Runtime: Python
Trigger: HTTP, secured with an App key (required in all requests).
Security: The HTTP trigger is IP-restricted; only calls from our on-premise network (where the Automation worker agent lives) are permitted.
Operation:

- Receives IP addresses via HTTP POST from the SentinelOne Datalake
- Checks if the IP is already present in edl.txt; if not, appends it to the file.
- Prevents duplicate entries to ensure a clean EDL.

SentinelOne Integration

Automation: PowerShell script, running on an Azure Automation Hybrid Worker.
Frequency: Once per hour.
Function:

- Queries SentinelOne's PowerQuery API
- When the rejected logins exceeds the set threshold (10) , the script POSTs the IP to the Azure Function.
- Job execution and results are logged for auditing.

EDL Consumption

The EDL is made available as static web object as this is compatible with most firewalls.

Security and Logging

No IP firewall on storage account due to Azure Functions access requirements.
Azure Function: IP-restricted and secured with an App key to prevent unauthorized updates.
Logging: All PowerShell job executions (including failed/successful EDL updates) are logged via Automation job logs.

Notable Design Choices

Deduplication logic in the Azure Function prevents EDL bloat and simplifies management.
Hybrid worker is used for seamless on-premise to cloud integration and secure execution.