Elastic API & Threat Detection Rules

WE Innovate X Zero$ploit

Supervised by : Eng.Zeyad Mazen & Eng.Adel Ahmed

Prepared by : Omar Hassan

Required Tasks

- *(Creating – Inserting – Updating – Deleting) documents in Elastic*
- *Ingestion Pipeline*
- *Threat Detection rules for suspicious windows events*

Prerequisites

- VMware / Virtual Box
- Windows 10/11 ISO – Ubuntu (20.0/22.0/24.0) ISO
- 16 GB RAM – 60 GB Disk Space
- 4 CPU Cores
- ElasticSearch & Kibana Configured/Running
- WinLogBeat Configured/Running or any other logshipper on windows

---

Ubuntu Machine

Setting	Recommended
RAM	5-6 GB
Disk	20-30 GB
CPU	2-3 Cores
Network	NAT

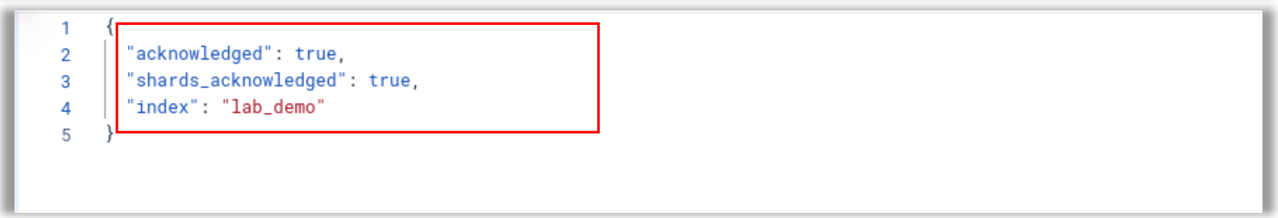
Windows Machine

Setting	Recommended
RAM	2-3 GB
Disk	30-40 GB
CPU	1-2 Cores
Network	NAT

(Create - Read - Update - Delete) With Elasticsearch

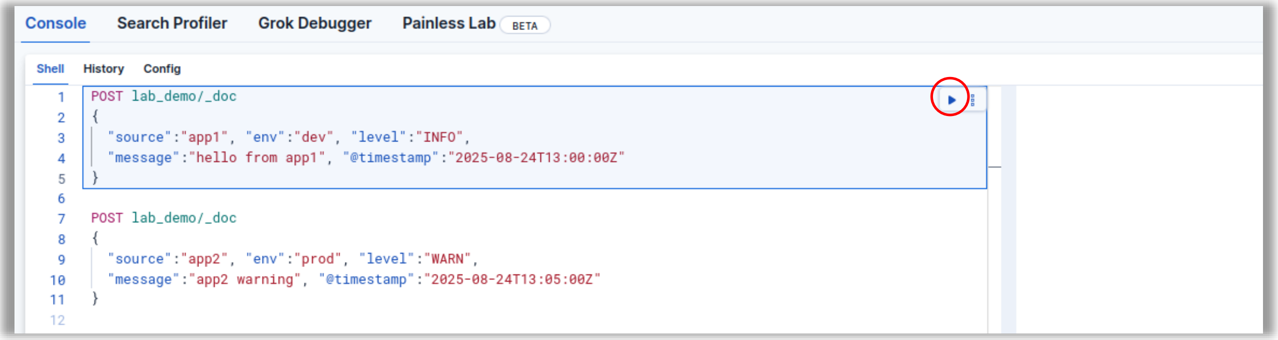
PHASE 1 : Creating Index

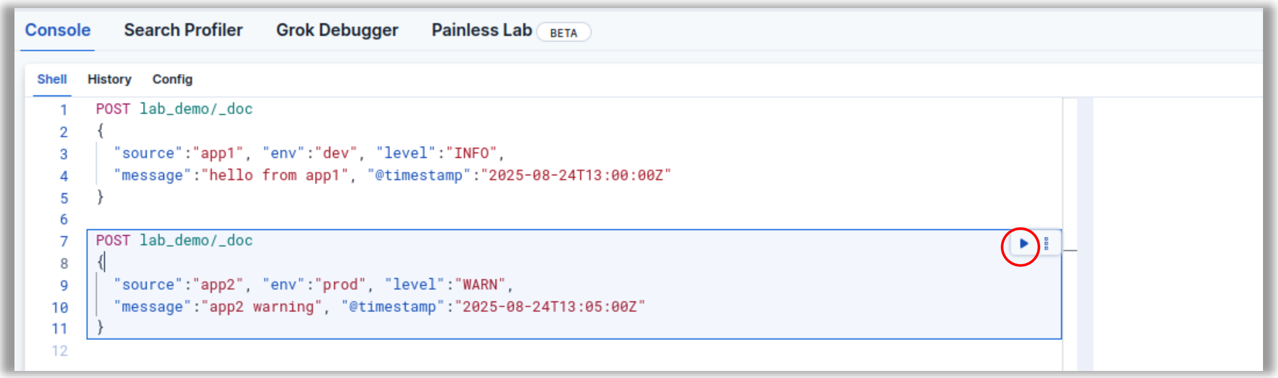
__Create an Index__ : Management > Dev tools > console
This allows us to use Elastic’s API & creating an __HTTP PUT request__



PHASE 2 : Inserting Documents

Again in the same place : Management > Dev tools > console
We will be inserting two documents just for display
One will be __INFO__ & the other will be __WARN__ <u>, and make sure to run each request individually.</u>

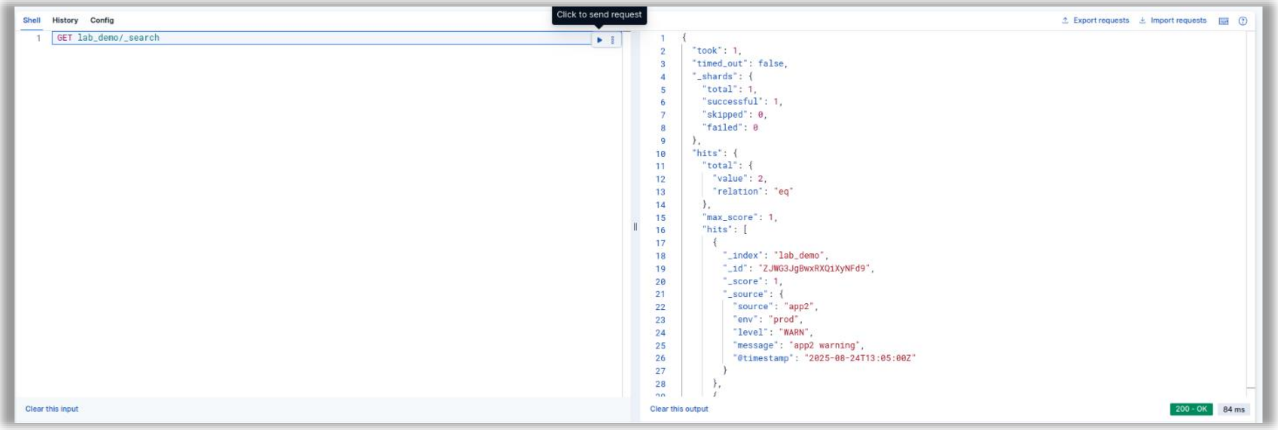

Console
Search Profiler
Grok Debugger
Painless Lab BETA
Shell
History
Config
POST lab_demo/_doc
{
"source":"app1", "env":"dev", "level":"INFO",
"message":"hello from app1", "@timestamp":"2025-08-24T13:00:00Z"
}
POST lab_demo/_doc
{
"source":"app2", "env":"prod", "level":"WARN",
"message":"app2 warning", "@timestamp":"2025-08-24T13:05:00Z"
}


Console
Search Profiler
Grok Debugger
Painless Lab BETA
Shell
History
Config
POST lab_demo/_doc
{
"source":"app1", "env":"dev", "level":"INFO",
"message":"hello from app1", "@timestamp":"2025-08-24T13:00:00Z"
}
POST lab_demo/_doc
{
"source":"app2", "env":"prod", "level":"WARN",
"message":"app2 warning", "@timestamp":"2025-08-24T13:05:00Z"
}


{
"_index": "lab_demo",
"_id": "XZWF3JgBwxRXQiXyklcL",
"_version": 1,
"result": "created",
"_shards": {
"total": 1,
"successful": 1,
"failed": 0
},
"_seq_no": 0,
"_primary_term": 1
}


{
"_index": "lab_demo",
"_id": "ZJWG3JgBwxRXQiXyNFd9",
"_version": 1,
"result": "created",
"_shards": {
"total": 1,
"successful": 1,
"failed": 0
},
"_seq_no": 1,
"_primary_term": 1
}

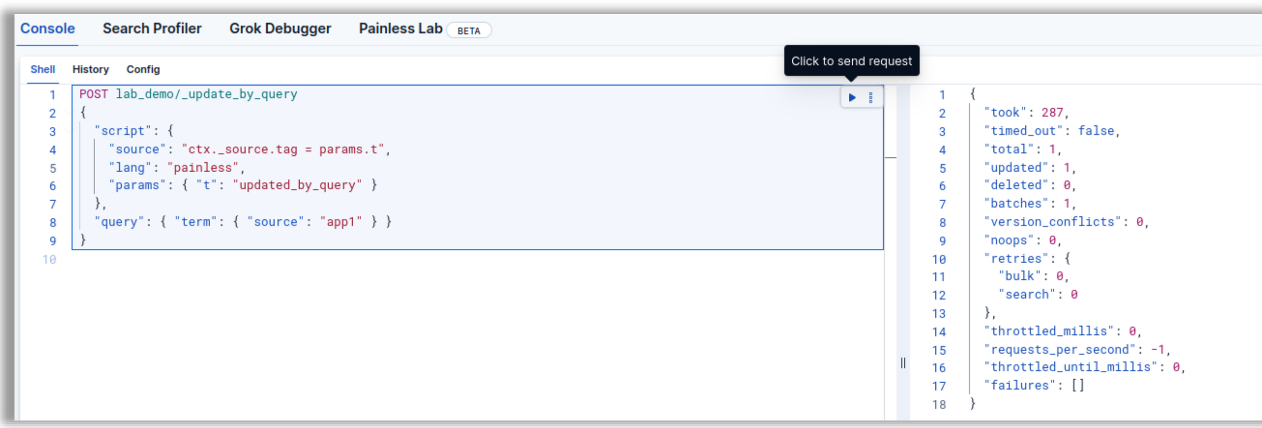
Then we test if our documents by retrieving it by this query :
`GET lab_demo/_search`



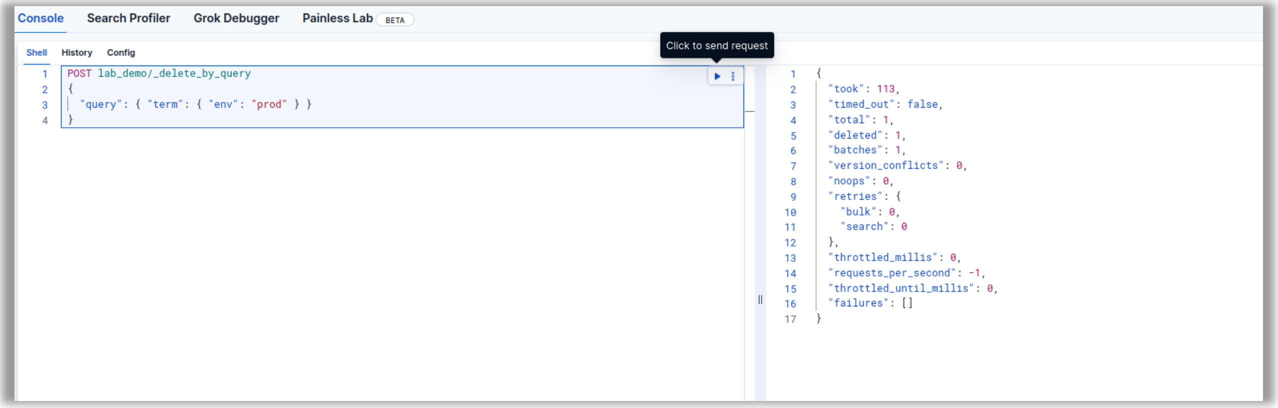
200-OK *means everything is working as we intended*

PHASE 3 : Updating & Deleting docuements by query

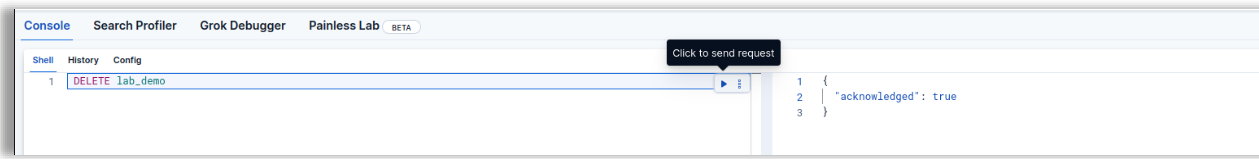
Updating and adding a new field called tag , a message of **200 – OK** should appear at the bottom right.



Deleting every document where **env=prod** , a message of **200 – OK** should appear at the bottom right.



PHASE 4 : Deleting the Index after our test



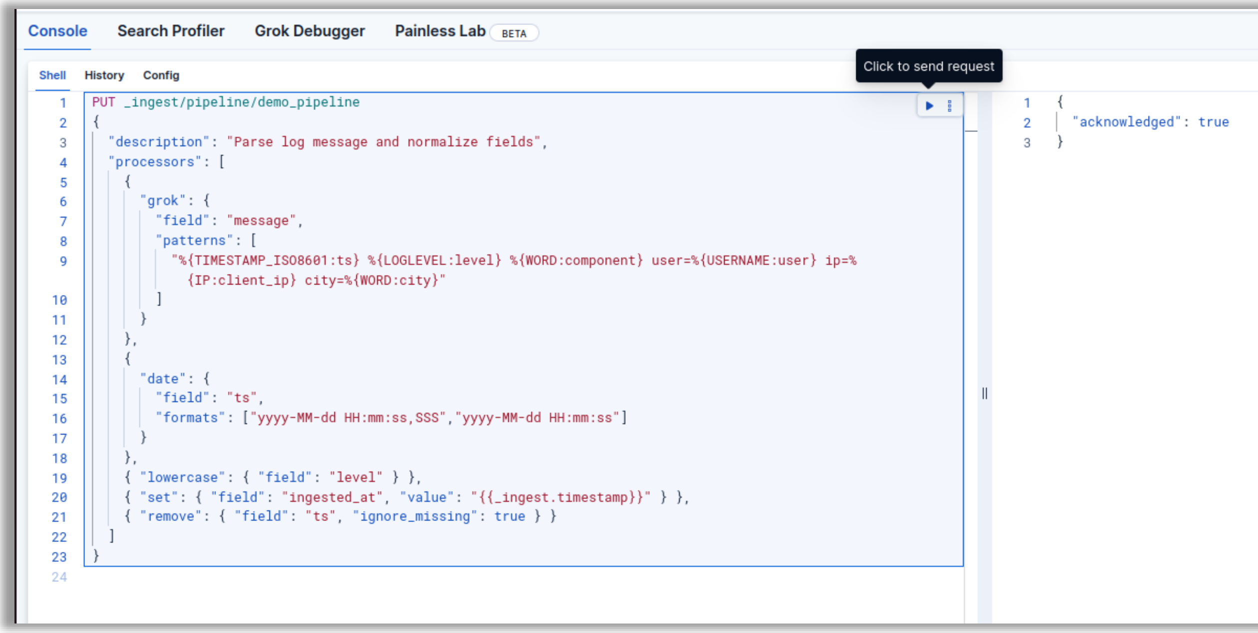
Summary

We started by opening Kibana and using the Dev Tools console, which allows us to run Elasticsearch API requests and view responses directly. First, we created an index called ***lab_demo*** *with one shard and no replicas, since this was just a small test, and we defined mappings for each field, such as keyword, text, and date. Next, we inserted two documents: one representing a development info message from app1 and another representing a production warning from app2, then confirmed they were stored by running a search query. After that, we performed an update by query, where we targeted documents with source = app1 and added a new field called tag with the value updated_by_query. We then deleted all documents where the environment was prod, which removed the app2 document, leaving only the app1 record. Finally, we deleted the entire lab_demo index to clean up the cluster after testing.*

Ingestion Pipline

PHASE 1 : Creating a pipline

Create an pipline *: Management > Dev tools > console*

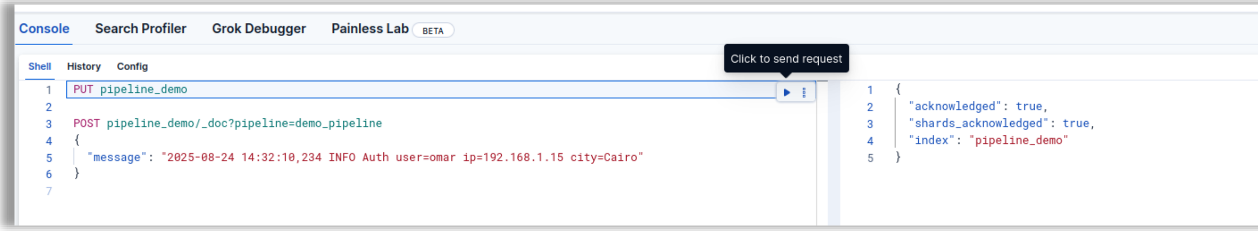


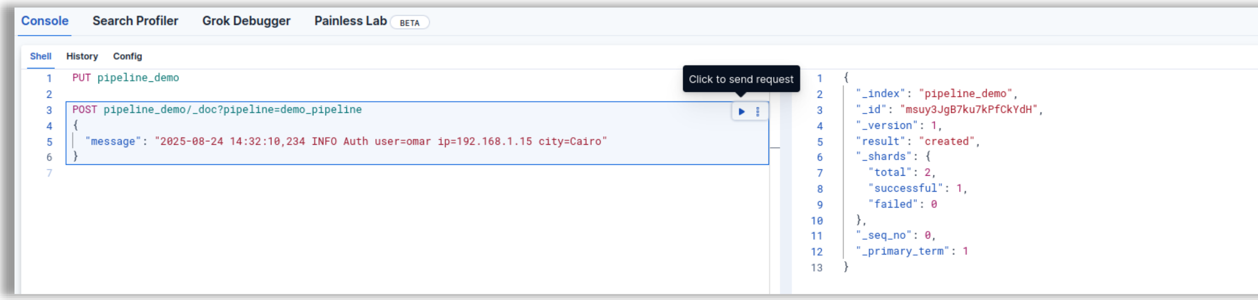
But why those processors?

*I chose these processors for the ingestion pipeline because each one serves a useful purpose. **Grok** is used to parse the message field and extract structured fields such as user, IP, city, and level. **Date** converts the string-based timestamp into a proper Elasticsearch date so it can be used in time-based queries and visualizations. **Lowercase** ensures the level field is always stored in lowercase (info/warn), making searches and aggregations more consistent. **Set** adds a new field called ingested_at to record the actual ingestion time of the document. Finally, **Remove** deletes the original ts field since it's a duplicate and no longer needed.*

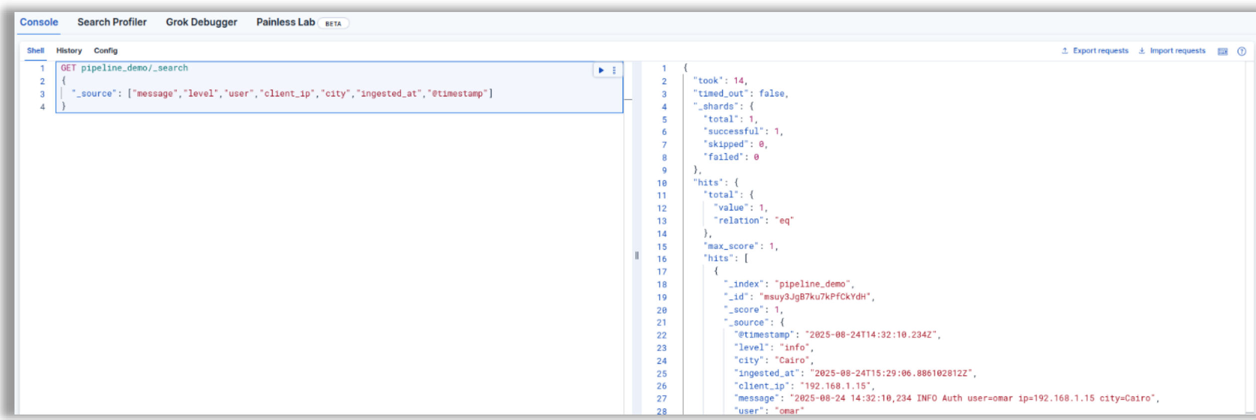
PHASE 2 : Creating an index & ingesting with pipeline

Run each query separately





PHASE 3 : Verifying our work



```json
{
  "took": 14,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": {
      "value": 1,
      "relation": "eq"
    },
    "max_score": 1,
    "hits": [
      {
        "_index": "pipeline_demo",
        "_id": "msuy3JgB7ku7kPfCkYdH",
        "_score": 1,
        "_source": {
          "@timestamp": "2025-08-24T14:32:10.234Z",
          "level": "info",
          "city": "Cairo",
          "ingested_at": "2025-08-24T15:29:06.886102812Z",
          "client_ip": "192.168.1.15",
          "message": "2025-08-24 14:32:10,234 INFO Auth user=omar ip=192.168.1.15 city=Cairo",
          "user": "omar"
```


Clear this output

200 - OK 155 ms

Summary

*In this part, we worked with an **ingest pipeline** to process data as it enters Elasticsearch. First, we created a pipeline called **demo_pipeline** that uses several processors: a **grok** processor to parse the raw message and extract fields such as timestamp, log level, component, user, IP, and city; a **date** processor to convert the extracted timestamp into a proper Elasticsearch date; a **lowercase** processor to normalize the log level field; a **set** processor to add an ingested_at field showing the ingestion time; and finally, a **remove** processor to delete the original timestamp field since it was redundant. Next, we created a new index named **pipeline_demo** and ingested a sample log document into it while applying the pipeline, which automatically parsed and enriched the data. Finally, we verified the results with a search query and confirmed that the fields were extracted and normalized as expected.*

Threat detection rules for suspicious windows events

Scenarios Covered

- **Powershell with Encoded Command** – Detects when PowerShell is run with the EncodedCommand flag, which is often used by attackers to hide malicious commands.
- **Powershell Execution ByPass** – Detects PowerShell execution with the -ExecutionPolicy Bypass flag, which allows scripts to run regardless of system policy and is commonly abused by attackers.
- **Powershell Hidden Window** – Detects when PowerShell is launched with the -WindowStyle Hidden flag, which attackers use to conceal malicious activity from the user.

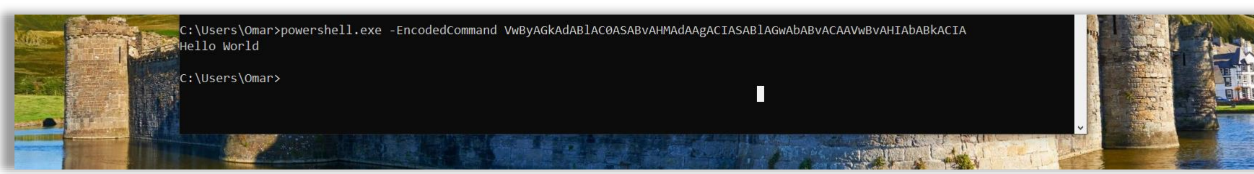
Make sure the windows machine that has the logshipper is running & is sending logs to proceed.

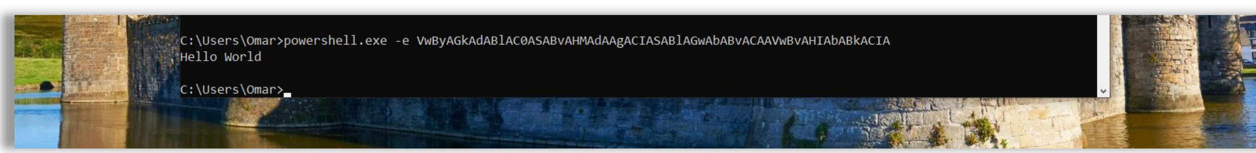
Scenario 1 : PowerShell with Encoded Command

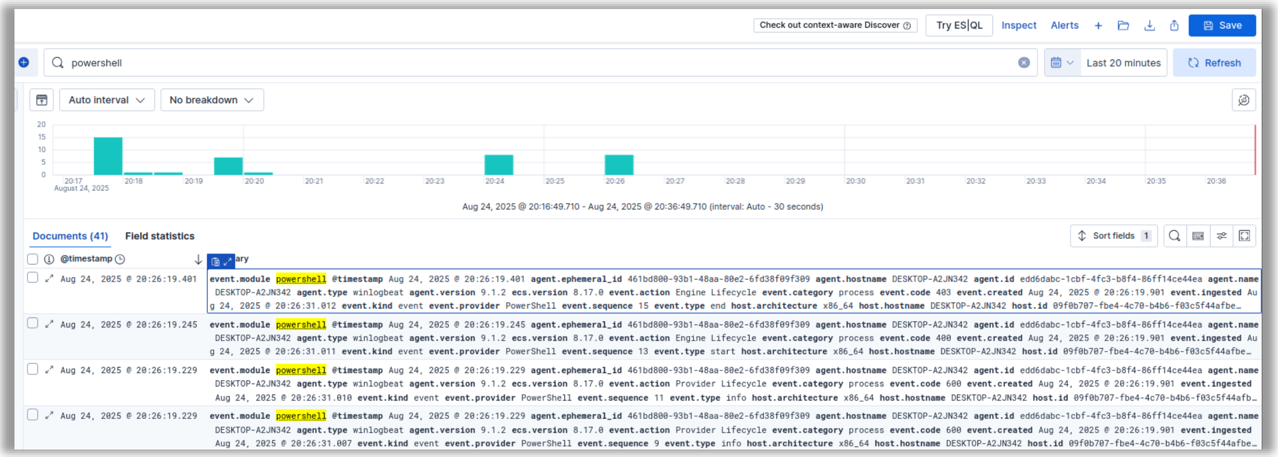
Run these Base64 Encoded PowerShell command on CMD

```
powershell.exe -EncodedCommand
VwByAGkAdABlAC0ASABvAHMAdAAgACIASABlAGwAbABvACAAVwBvAHIAbABkACIA
```

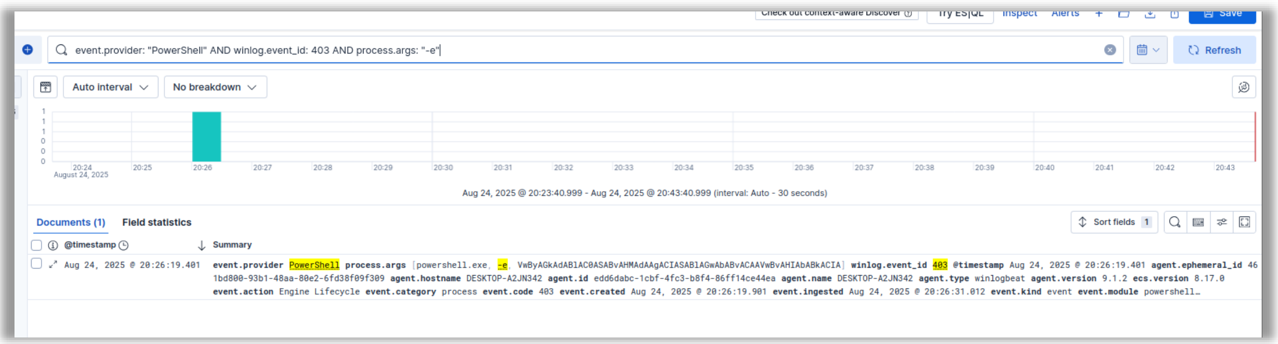
```
powershell.exe -e
VwByAGkAdABlAC0ASABvAHMAdAAgACIASABlAGwAbABvACAAVwBvAHIAbABkACIA
```







Wait approximately 5 minutes for logs to be shipped to Elastic & search powershell , then we can expand the log to see the fields and based on that we write our kql query.

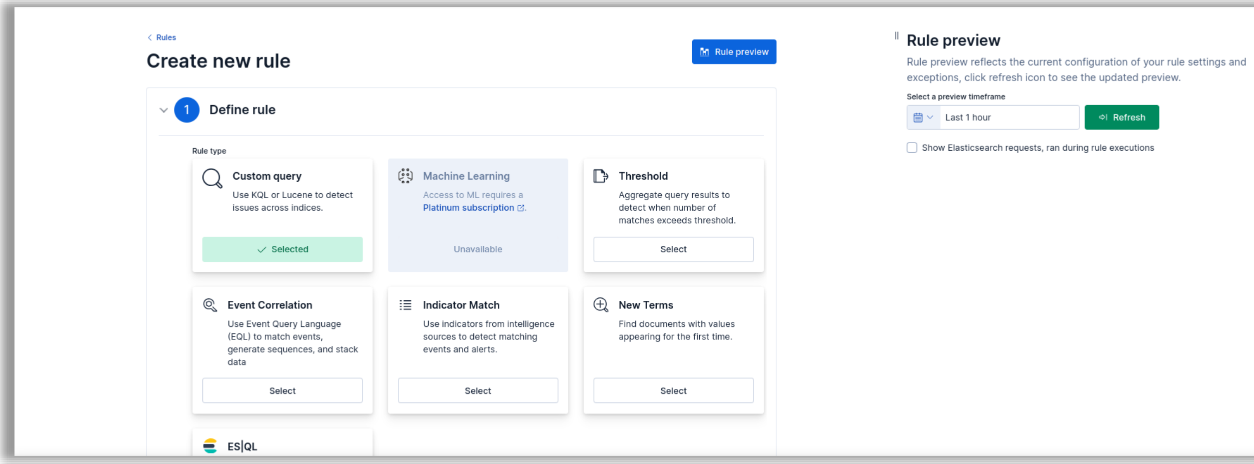


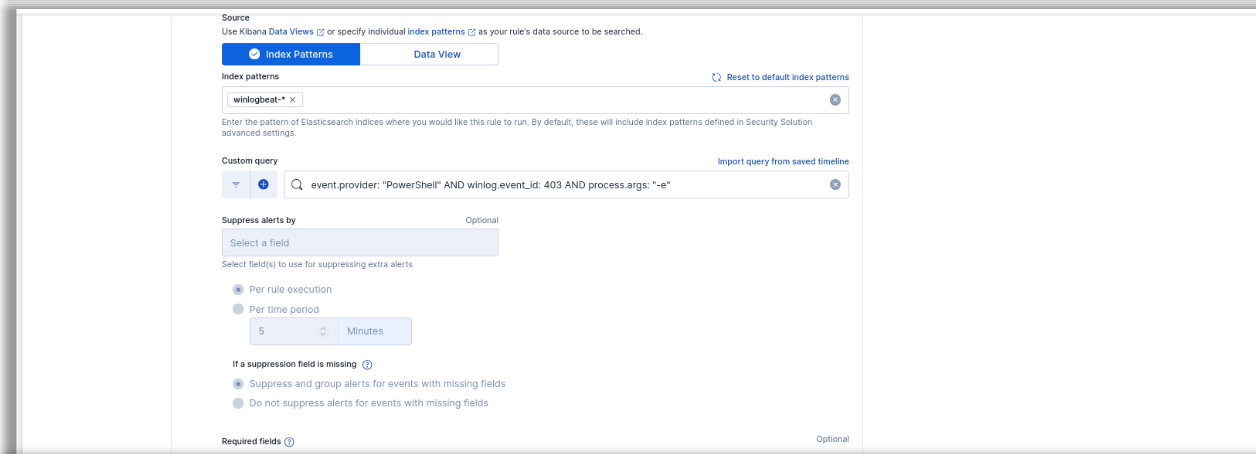
My rule :

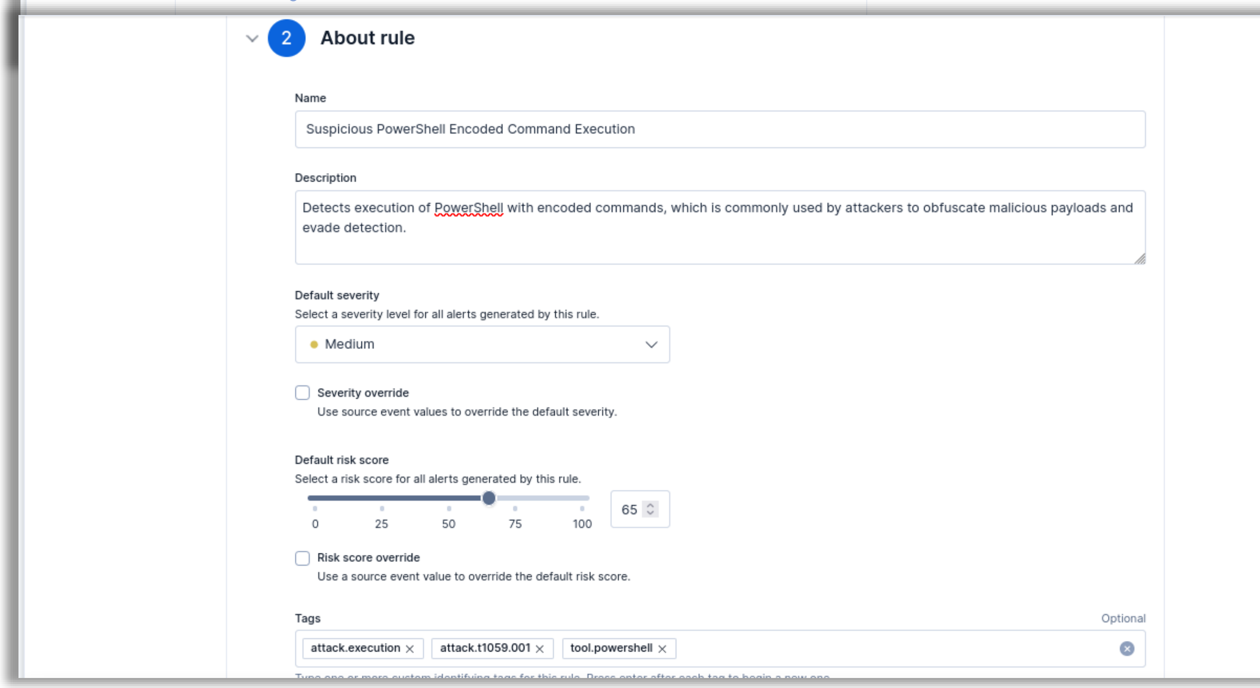
```
event.provider: "PowerShell" AND winlog.event_id: 403 AND
process.args: "-e"
```

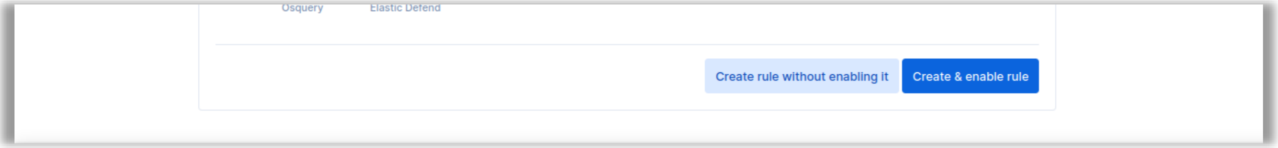
You must test your rule first in the discover tab before writing the alert rule , I will be displaying how to create the alert rule for this example only , you can follow the workflow through the next scenarios.

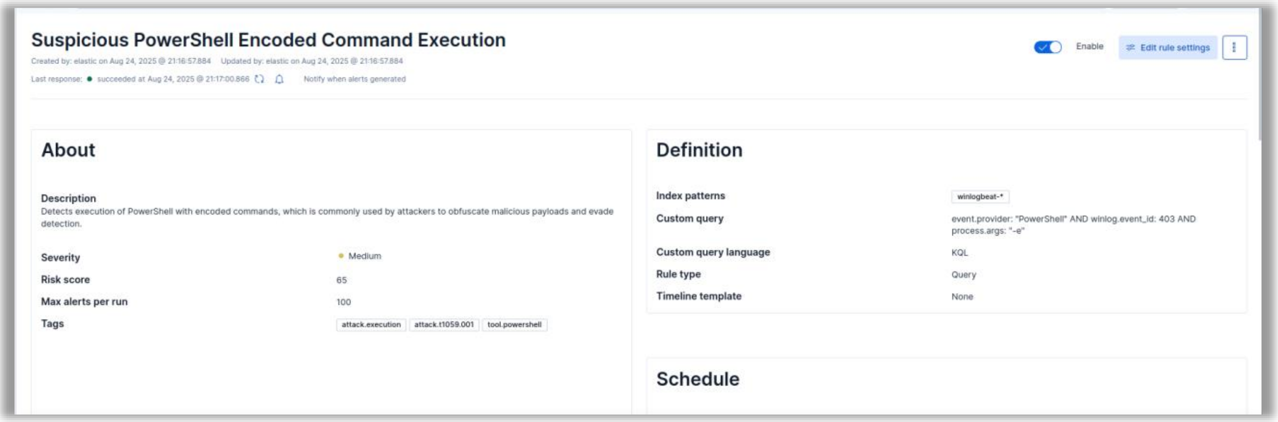
Creating a rule through security > rules > Detection Rules (SIEM)



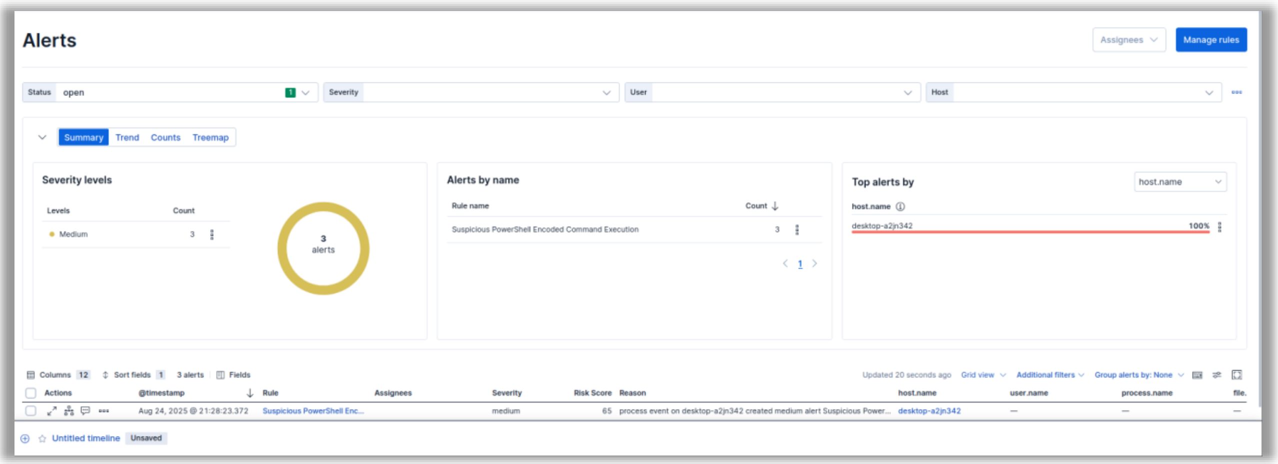








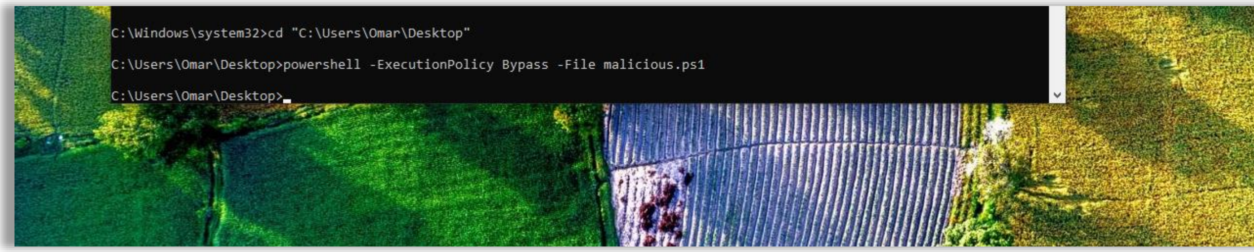
Wait for a few minutes for alerts to appear



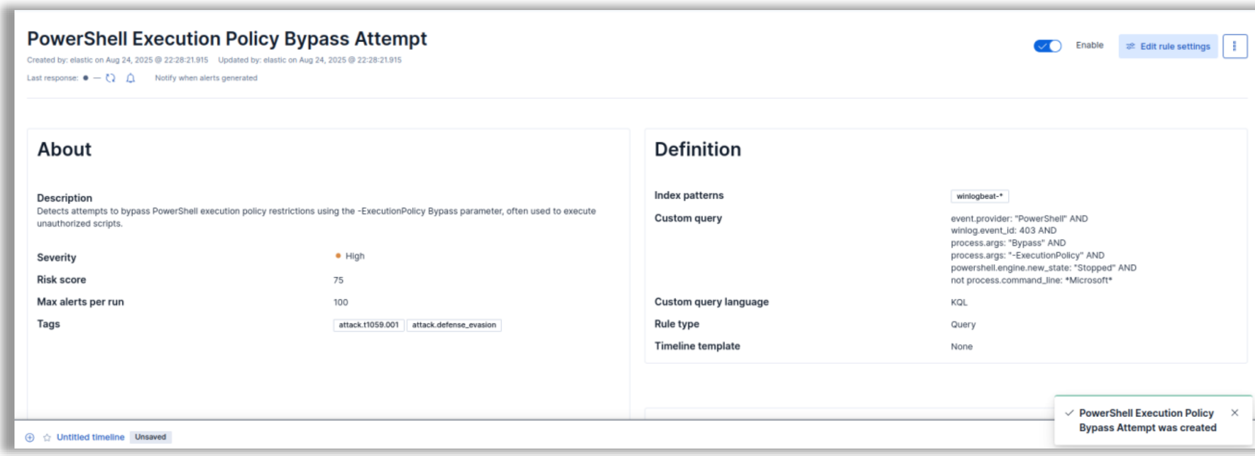
Scenario 2 : PowerShell Execution ByPass

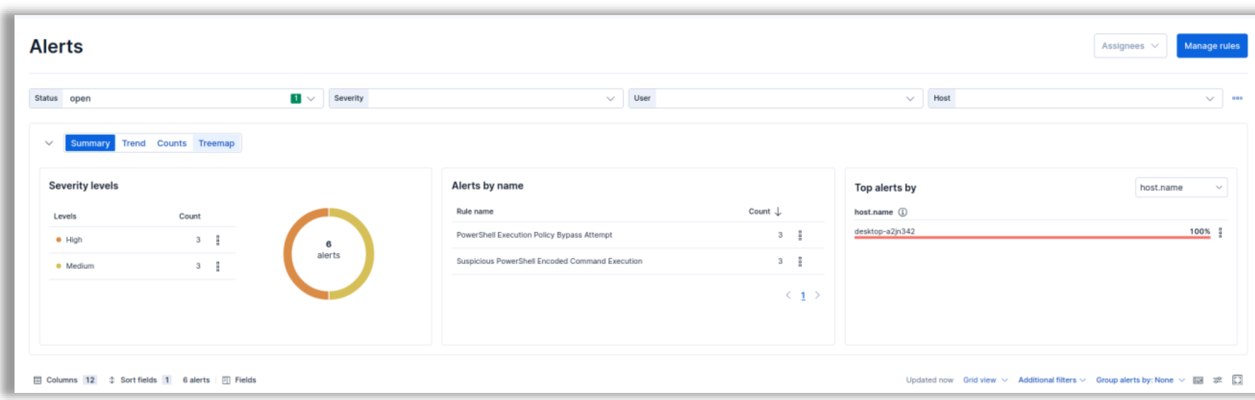
Create a malicious.ps1 file to test the command

```
powershell -ExecutionPolicy Bypass -File malicious.ps1
```



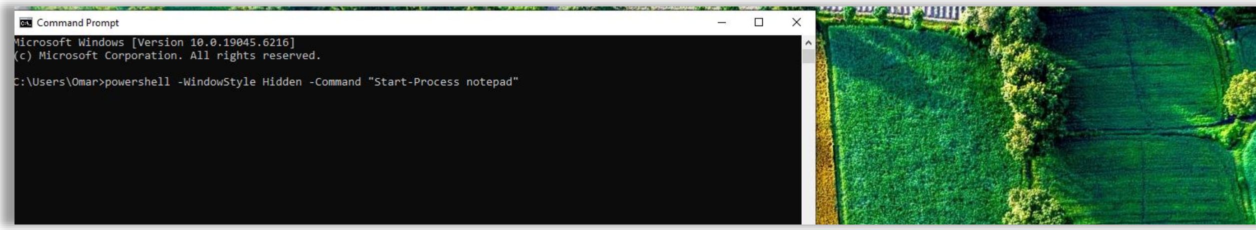

```
event.provider: "PowerShell" AND
winlog.event_id: 403 AND
process.args: "Bypass" AND
process.args: "-ExecutionPolicy" AND
powershell.engine.new_state: "Stopped" AND
not process.command_line: *Microsoft*
```



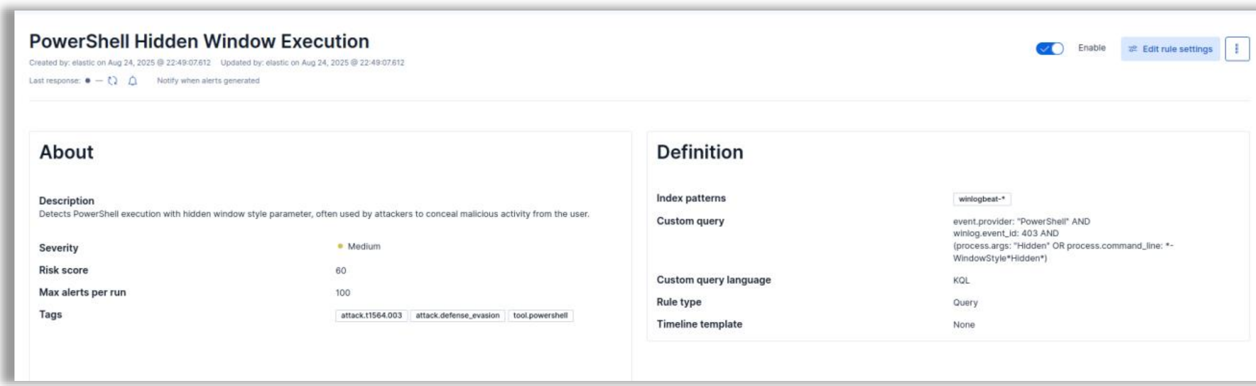


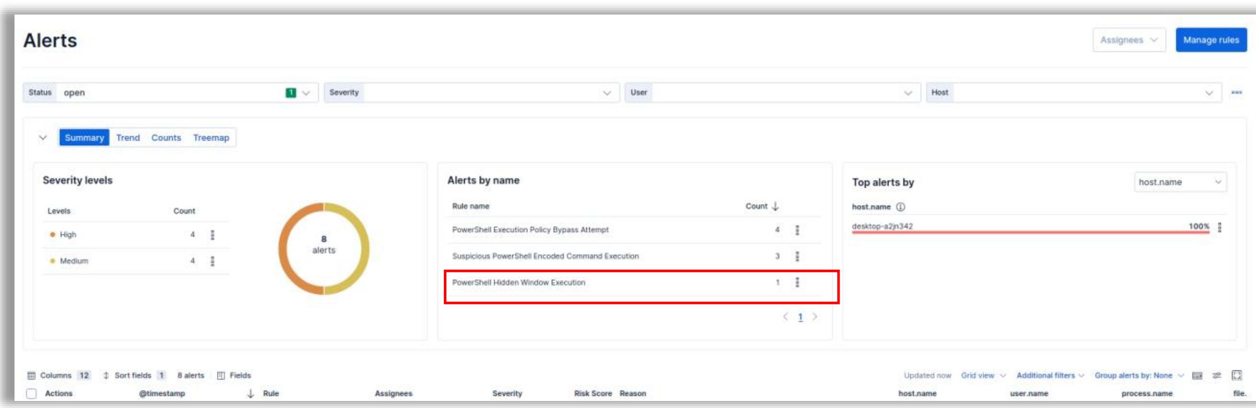
Scenario 3 : PowerShell Hidden Window

```
powershell -WindowStyle Hidden -Command "Start-Process notepad"
```




```
event.provider: "PowerShell" AND
winlog.event_id: 403 AND
(process.args: "Hidden" OR process.command_line: *-
WindowStyle*Hidden*)
```





الحمد لله رب العالمين