

Elastic API & Threat Detection Rules

WE Innovate X Zero\$exploit

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

Prepared by : [Omar Hassan](#)



zero\$exploit
Cyber Security trusted partner



EG|CERT



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***



The screenshot displays the Elasticsearch Dev Tools interface. The top navigation bar includes 'Console', 'Search Profiler', 'Grok Debugger', and 'Painless Lab' (marked as BETA). The 'Console' tab is active, showing a 'Shell' view with a code editor and a 'History' view below it. In the code editor, a PUT request is being constructed for the 'lab_demo' index. The request body is a JSON object defining the index's settings and mappings. A red circle highlights the 'Run' button (a play icon) in the top right corner of the code editor. The 'History' view below shows the response to the PUT request, which is a JSON object indicating successful index creation with 'acknowledged' and 'shards_acknowledged' set to true, and the 'index' name 'lab_demo'.

```
1 PUT lab_demo
2 {
3   "settings": { "number_of_shards": 1, "number_of_replicas": 0 },
4   "mappings": {
5     "properties": {
6       "source": { "type": "keyword" },
7       "env": { "type": "keyword" },
8       "level": { "type": "keyword" },
9       "message": { "type": "text" },
10      "@timestamp": { "type": "date" }
11    }
12  }
13 }
14
```

```
1 {
2   "acknowledged": true,
3   "shards_acknowledged": true,
4   "index": "lab_demo"
5 }
```

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** , and make sure to run each request individually.*

Console Search Profiler Grok Debugger Painless Lab BETA

Shell History Config

```
1 POST lab_demo/_doc
2 {
3   "source": "app1", "env": "dev", "level": "INFO",
4   "message": "hello from app1", "@timestamp": "2025-08-24T13:00:00Z"
5 }
6
7 POST lab_demo/_doc
8 {
9   "source": "app2", "env": "prod", "level": "WARN",
10  "message": "app2 warning", "@timestamp": "2025-08-24T13:05:00Z"
11 }
12
```

Console Search Profiler Grok Debugger Painless Lab BETA

Shell History Config

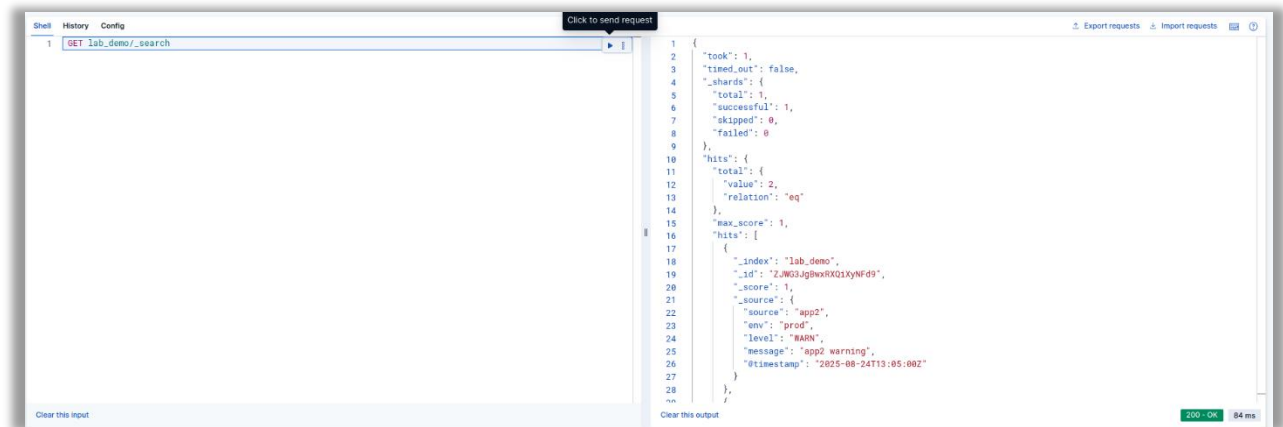
```
1 POST lab_demo/_doc
2 {
3   "source": "app1", "env": "dev", "level": "INFO",
4   "message": "hello from app1", "@timestamp": "2025-08-24T13:00:00Z"
5 }
6
7 POST lab_demo/_doc
8 {
9   "source": "app2", "env": "prod", "level": "WARN",
10  "message": "app2 warning", "@timestamp": "2025-08-24T13:05:00Z"
11 }
12
```

```
1 {
2   "_index": "lab_demo",
3   "_id": "XZWF3JgBwxRXQiXyKlcl",
4   "_version": 1,
5   "result": "created",
6   "_shards": {
7     "total": 1,
8     "successful": 1,
9     "failed": 0
10  },
11   "_seq_no": 0,
12   "_primary_term": 1
13 }
```

```
1 {
2   "_index": "lab_demo",
3   "_id": "ZJWG3JgBwxRXQiXyNFd9",
4   "_version": 1,
5   "result": "created",
6   "_shards": {
7     "total": 1,
8     "successful": 1,
9     "failed": 0
10  },
11   "_seq_no": 1,
12   "_primary_term": 1
13 }
```

Then we test if our documents by retrieving it by this query :

GET lab_demo/_search



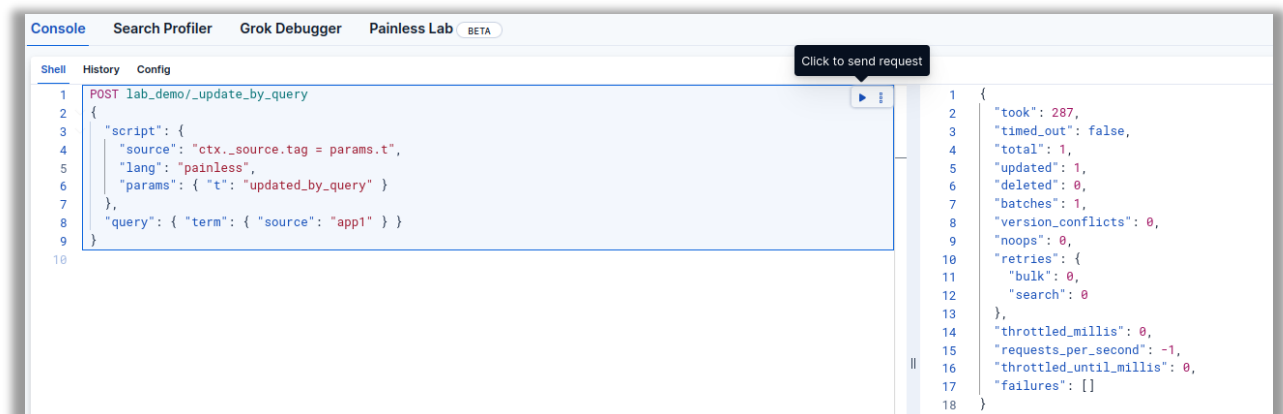
The screenshot shows a REST client interface with a 'Shell' tab. The input field contains the URL 'GET lab_demo/_search'. A 'Click to send request' button is visible. The response is displayed in a JSON format on the right side of the interface. The status bar at the bottom right indicates '200 - OK' and '84 ms'.

```
1 GET lab_demo/_search
2
3 {
4   "took": 1,
5   "timed_out": false,
6   "_shards": {
7     "total": 1,
8     "successful": 1,
9     "skipped": 0,
10    "failed": 0
11  },
12  "hits": {
13    "total": {
14      "value": 2,
15      "relation": "eq"
16    },
17    "max_score": 1,
18    "hits": [
19      {
20        "_index": "lab_demo",
21        "_id": "2_WG3Jg8xRX01xNfD9",
22        "_score": 1,
23        "_source": {
24          "source": "app2",
25          "env": "prod",
26          "level": "WARN",
27          "message": "app2 warning",
28          "@timestamp": "2025-08-24T13:05:00Z"
29        }
30      }
31    ]
32  }
33 }
```

200-OK means everything is working as we intended

PHASE 3 : Updating & Deleting documents by query

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

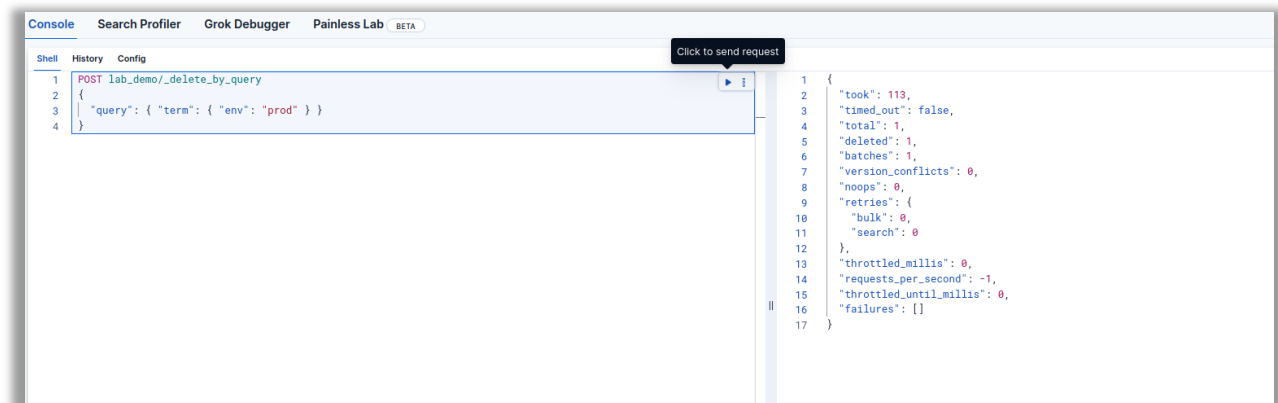


The screenshot shows a REST client interface with a 'Console' tab. The input field contains a POST request to 'lab_demo/_update_by_query' with a JSON body. A 'Click to send request' button is visible. The response is displayed in a JSON format on the right side of the interface. The status bar at the bottom right indicates '200 - OK'.

```
1 POST lab_demo/_update_by_query
2 {
3   "script": {
4     "source": "ctx._source.tag = params.t",
5     "lang": "painless",
6     "params": { "t": "updated_by_query" }
7   },
8   "query": { "term": { "source": "app1" } }
9 }
10
```

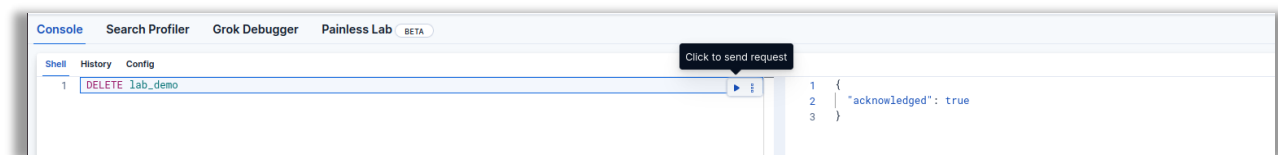
```
1 {
2   "took": 287,
3   "timed_out": false,
4   "total": 1,
5   "updated": 1,
6   "deleted": 0,
7   "batches": 1,
8   "version_conflicts": 0,
9   "noops": 0,
10  "retries": {
11    "bulk": 0,
12    "search": 0
13  },
14  "throttled_millis": 0,
15  "requests_per_second": -1,
16  "throttled_until_millis": 0,
17  "failures": []
18 }
```

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



The screenshot shows the DevTools console in Kibana. The left pane contains a REST client request: `POST lab_demo/_delete_by_query` with a body `{ "query": { "term": { "env": "prod" } } }`. The right pane shows the response: `{ "took": 113, "timed_out": false, "total": 1, "deleted": 1, "batches": 1, "version_conflicts": 0, "noops": 0, "retries": { "bulk": 0, "search": 0 }, "throttled_millis": 0, "requests_per_second": -1, "throttled_until_millis": 0, "failures": [] }`. A "Click to send request" button is visible above the request.

PHASE 4 : Deleting the Index after our test



The screenshot shows the DevTools console in Kibana. The left pane contains a REST client request: `DELETE lab_demo`. The right pane shows the response: `{ "acknowledged": true }`. A "Click to send request" button is visible above the request.

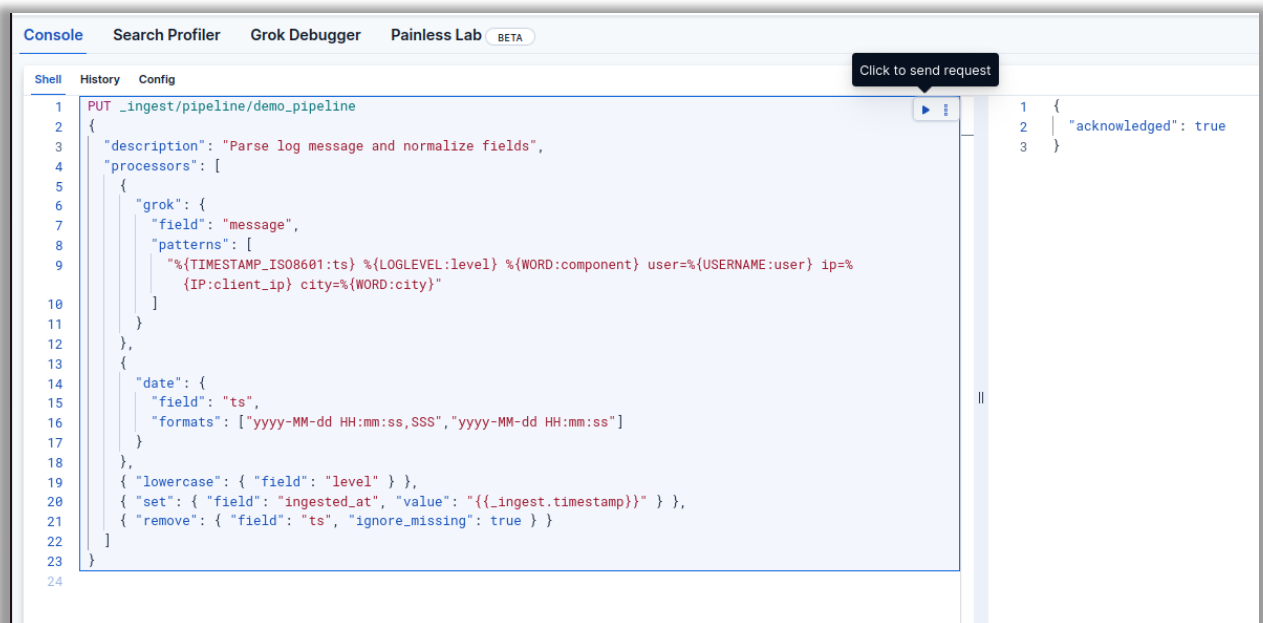
Summary

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 Pipeline

PHASE 1 : Creating a pipeline

Create an pipeline : Management > Dev tools > console



The screenshot shows the Elasticsearch Dev Tools console with the 'Console' tab selected. The left pane contains a JSON configuration for an ingestion pipeline named 'demo_pipeline'. The configuration includes a description, a grok processor to parse log messages, a date processor to convert timestamps, a lowercase processor for the 'level' field, a set processor to add an 'ingested_at' field, and a remove processor to delete the 'ts' field. The right pane shows the response to the PUT request, which is a simple acknowledgment object: { "acknowledged": true }.

```
1 PUT _ingest/pipeline/demo_pipeline
2 {
3   "description": "Parse log message and normalize fields",
4   "processors": [
5     {
6       "grok": {
7         "field": "message",
8         "patterns": [
9           "%{TIMESTAMP_ISO8601:ts} %{LOGLEVEL:level} %{WORD:component} user=%{USERNAME:user} ip=%
10            {IP:client_ip} city=%{WORD:city}"
11         ]
12       }
13     },
14     {
15       "date": {
16         "field": "ts",
17         "formats": ["yyyy-MM-dd HH:mm:ss,SSS", "yyyy-MM-dd HH:mm:ss"]
18       }
19     },
20     { "lowercase": { "field": "level" } },
21     { "set": { "field": "ingested_at", "value": "{{_ingest.timestamp}}" } },
22     { "remove": { "field": "ts", "ignore_missing": true } }
23   ]
24 }
```

```
1 {
2   "acknowledged": true
3 }
```

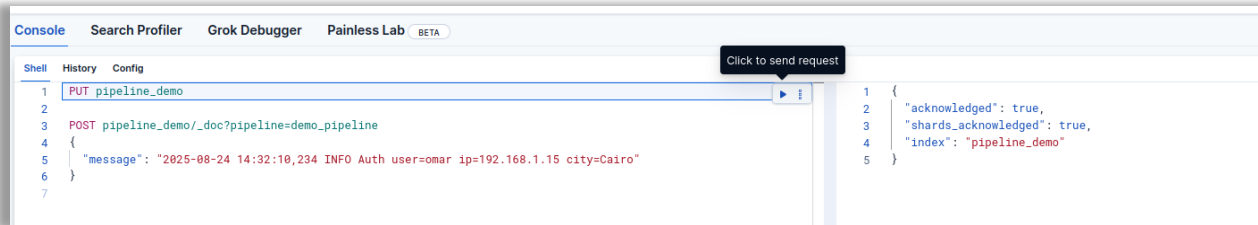
But why those processors?

I chose these processors for the ingestion pipeline because each one serves a useful purpose.

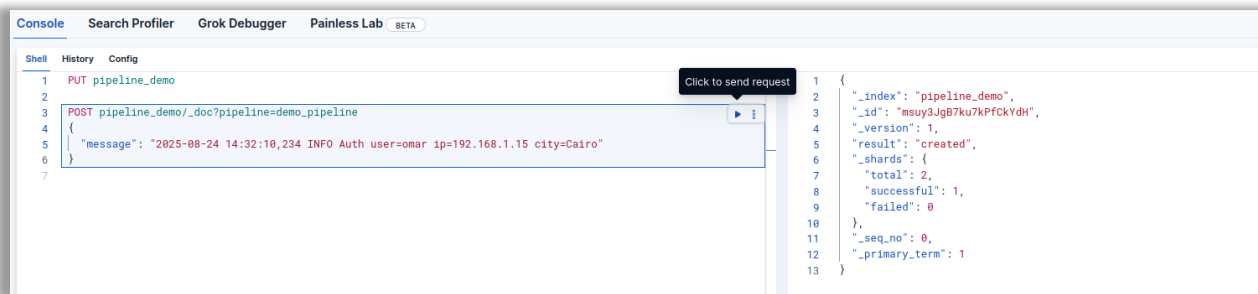
Grok is used to parse the message field and extract structured fields such as user, IP, city, and level. **Date** converts the string-based timestamp into a proper Elasticsearch date so it can be used in time-based queries and visualizations. **Lowercase** ensures the level field is always stored in 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

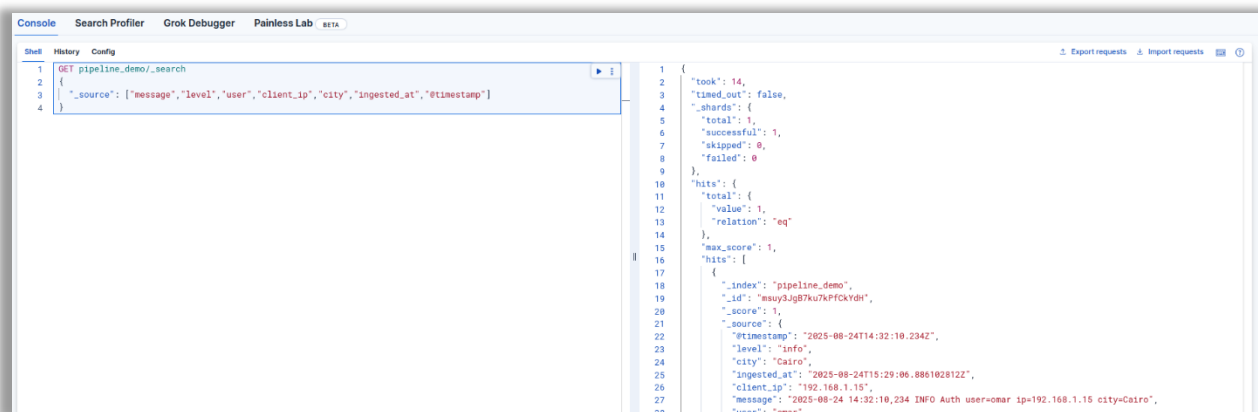


The screenshot shows a console interface with a 'Shell' tab. A PUT request is entered: `PUT pipeline_demo`. A button 'Click to send request' is visible. The response on the right is a JSON object: `{ "acknowledged": true, "shards_acknowledged": true, "index": "pipeline_demo" }`.



The screenshot shows the console with a POST request: `POST pipeline_demo/_doc?pipeline=demo_pipeline`. The request body is: `{ "message": "2025-08-24 14:32:10,234 INFO Auth user=omar ip=192.168.1.15 city=Cairo" }`. The response on the right is a JSON object: `{ "_index": "pipeline_demo", "_id": "msuy3JgB7ku7kPFckYdH", "_version": 1, "result": "created", "shards": { "total": 1, "successful": 1, "failed": 0 }, "_seq_no": 0, "_primary_term": 1 }`.

PHASE 3 : Verifying our work



The screenshot shows the console with a GET request: `GET pipeline_demo/_search`. The request body is: `{ "_source": ["message", "level", "user", "client_ip", "city", "ingested_at", "timestamp"] }`. The response on the right is a JSON object: `{ "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" } }] }`.


```
1 {
2   "took": 14,
3   "timed_out": false,
4   "_shards": {
5     "total": 1,
6     "successful": 1,
7     "skipped": 0,
8     "failed": 0
9   },
10  "hits": {
11    "total": {
12      "value": 1,
13      "relation": "eq"
14    },
15    "max_score": 1,
16    "hits": [
17      {
18        "_index": "pipeline_demo",
19        "_id": "msuy3JgB7ku7kPfCkYdH",
20        "_score": 1,
21        "_source": {
22          "@timestamp": "2025-08-24T14:32:10.234Z",
23          "level": "info",
24          "city": "Cairo",
25          "ingested_at": "2025-08-24T15:29:06.886102812Z",
26          "client_ip": "192.168.1.15",
27          "message": "2025-08-24 14:32:10,234 INFO Auth user=omar ip=192.168.1.15 city=Cairo",
28          "user": "omar"
29        }
30      }
31    ]
32  }
33 }
```

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

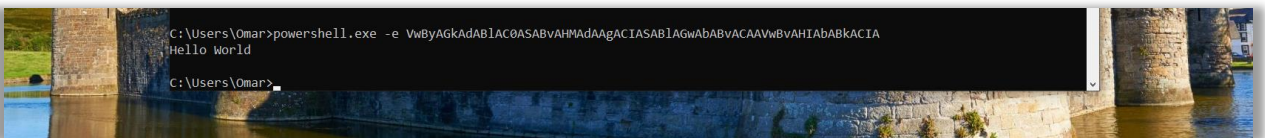
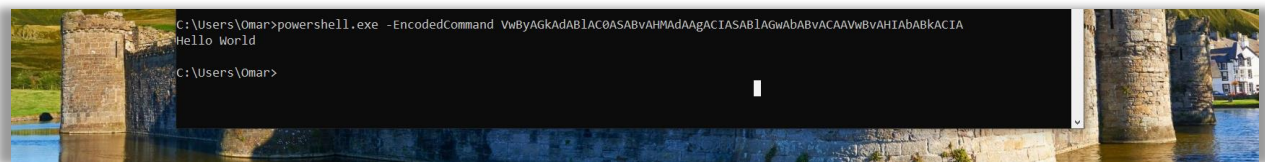
Run these Base64 Encoded PowerShell command on CMD

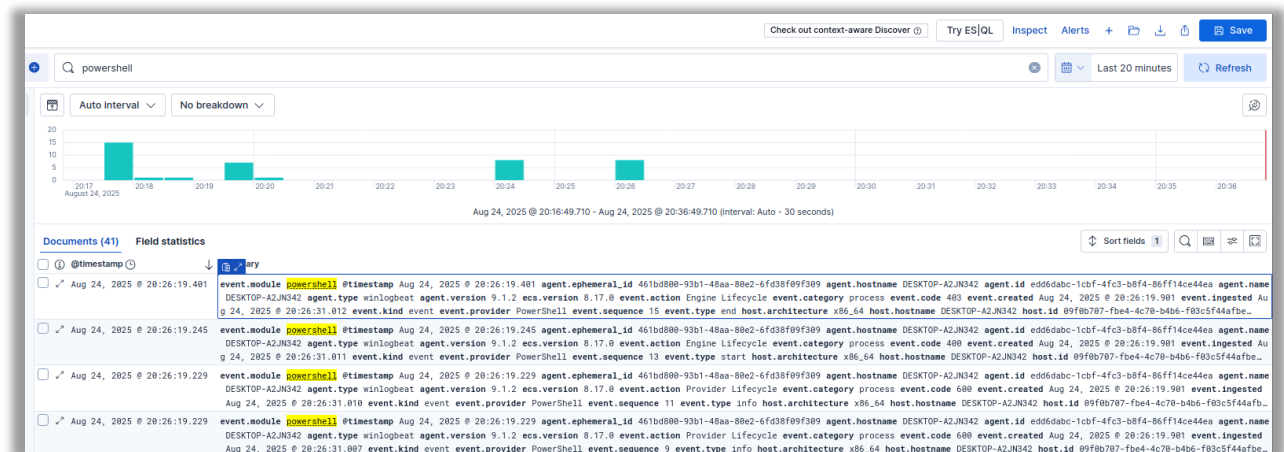
```
powershell.exe -EncodedCommand
```

```
VwByAGkAdABlAC0ASABvAHMAdAAgACIASABlAGwAbABvACA AVwBvAHlAbABkACIA
```

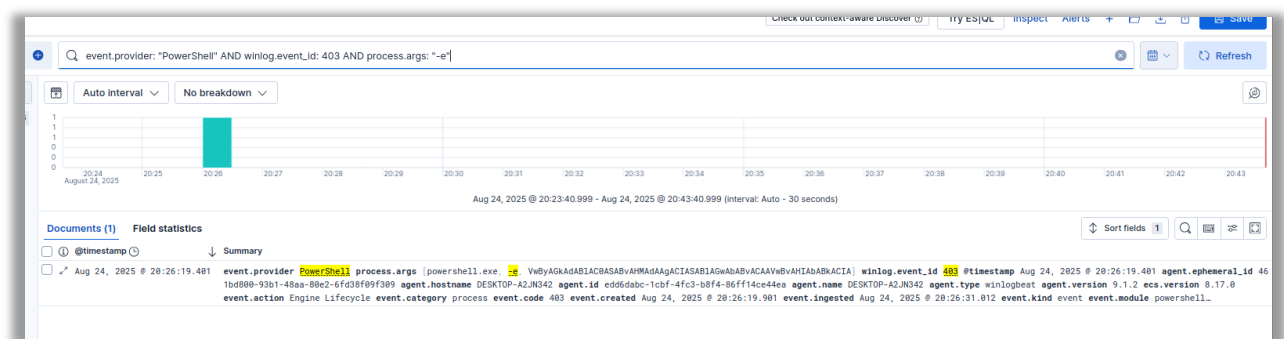
```
powershell.exe -e
```

```
VwByAGkAdABlAC0ASABvAHMAdAAgACIASABlAGwAbABvACA AVwBvAHlAbABkACIA
```





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)

< Rules

Create new rule

Rule preview

1 Define rule

Rule type

Custom query

Use KQL or Lucene to detect issues across indices.

✓ Selected

Machine Learning

Access to ML requires a [Platinum subscription](#).

Unavailable

Threshold

Aggregate query results to detect when number of matches exceeds threshold.

Select

Event Correlation

Use Event Query Language (EQL) to match events, generate sequences, and stack data.

Select

Indicator Match

Use indicators from intelligence sources to detect matching events and alerts.

Select

New Terms

Find documents with values appearing for the first time.

Select

ES|QL

Rule preview

Rule preview reflects the current configuration of your rule settings and exceptions, click refresh icon to see the updated preview.

Select a preview timeframe

Last 1 hour

Refresh

☐ Show Elasticsearch requests, ran during rule executions

Source

Use Kibana Data Views or specify individual index patterns as your rule's data source to be searched.

Index Patterns

Data View

Index patterns

winlogbeat-*

Reset to default index patterns

Custom query

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

Import query from saved timeline

Suppress alerts by

Select a field

Select field(s) to use for suppressing extra alerts

Per rule execution

Per time period

5 Minutes

If a suppression field is missing

Suppress and group alerts for events with missing fields

Do not suppress alerts for events with missing fields

Required fields

Optional

2 About rule

Name

Suspicious PowerShell Encoded Command Execution

Description

Detects execution of PowerShell with encoded commands, which is commonly used by attackers to obfuscate malicious payloads and evade detection.

Default severity

Select a severity level for all alerts generated by this rule.

Medium

☐ Severity override

Use source event values to override the default severity.

Default risk score

Select a risk score for all alerts generated by this rule.

0 25 50 75 100

65

☐ Risk score override

Use a source event value to override the default risk score.



Tags

attack.execution attack.t1059.001 tool.powershell

Optional

Suspicious PowerShell Encoded Command Execution

Created by: elastic on Aug 24, 2025 @ 21:16:57.884 Updated by: elastic on Aug 24, 2025 @ 21:16:57.884

Last response: ● succeeded at Aug 24, 2025 @ 21:17:00.866   Notify when alerts generated

About

Description

Detects execution of PowerShell with encoded commands, which is commonly used by attackers to obfuscate malicious payloads and evade detection.

Severity

● Medium

Risk score

65

Max alerts per run

100

Tags

attack.execution attack.i1059.001 tool.powershell

Definition

Index patterns

winlogbeat-*

Custom query

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

Custom query language

KQL

Rule type

Query

Timeline template

None

Schedule

The screenshot shows the Splunk Alerts interface. At the top, there are tabs for 'Status' (open), 'Severity' (Medium), 'User', and 'Host'. Below these, there are tabs for 'Summary', 'Trend', 'Counts', and 'Treemap'. The 'Summary' tab is active, showing a 'Severity levels' section with a donut chart indicating 3 alerts at the 'Medium' severity level. To the right, the 'Alerts by name' section shows a table with one alert: 'Suspicious PowerShell Encoded Command Execution' with a count of 3. Further right, the 'Top alerts by' section shows a table with one alert: 'desktop-a2jn342' with a count of 100%.

Severity	Count
Medium	3

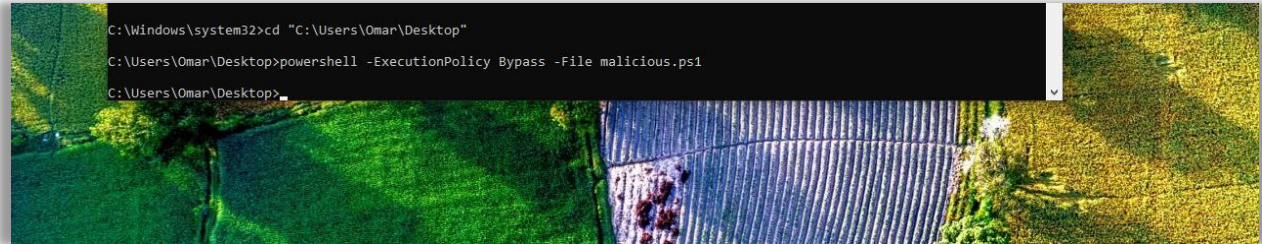
Rule name	Count
Suspicious PowerShell Encoded Command Execution	3

host.name	Count
desktop-a2jn342	100%

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*
```

A screenshot of the Elastic SIEM rule configuration page for a rule named 'PowerShell Execution Policy Bypass Attempt'. The page is divided into two main sections: 'About' and 'Definition'.
About Section:
- **Description:** Detects attempts to bypass PowerShell execution policy restrictions using the -ExecutionPolicy Bypass parameter, often used to execute unauthorized scripts.
- **Severity:** High (indicated by a red dot).
- **Risk score:** 75.
- **Max alerts per run:** 100.
- **Tags:** attack.t1059.001, attack.defense_evasion.
Definition Section:
- **Index patterns:** winlogbeat-*
- **Custom query:** 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*
- **Custom query language:** KQL
- **Rule type:** Query
- **Timeline template:** None
At the bottom right, a notification states: 'PowerShell Execution Policy Bypass Attempt was created'.A screenshot of the Elastic SIEM Alerts page. The page shows a summary of alerts and a list of alerts by name.
Summary Section:
- **Severity levels:** A donut chart showing 6 alerts. The chart is divided into two segments: High (red) and Medium (yellow).
- **Alerts by name:** A table showing the count of alerts for each rule name.

Rule name	Count
PowerShell Execution Policy Bypass Attempt	3
Suspicious PowerShell Encoded Command Execution	3

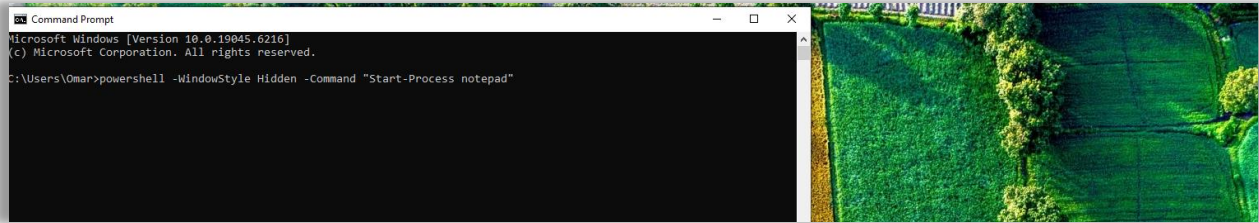
- **Top alerts by:** A table showing the top alerts by host name.

host.name	Count
desktop-x2n342	100%

At the bottom, there is a status bar showing 'Columns: 12', 'Sort fields: 1', '6 alerts', and 'Fields'.

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*)
```

PowerShell Hidden Window Execution
Created by: elastic on Aug 24, 2025 @ 22:49:07:612 Updated by: elastic on Aug 24, 2025 @ 22:49:07:612
Last response: Notify when alerts generated

About

Description
Detects PowerShell execution with hidden window style parameter, often used by attackers to conceal malicious activity from the user.

Severity
Medium

Risk score
60

Max alerts per run
100

Tags
attack.t1564.003 attack.defense_evasion tool.powershell

Definition

Index patterns
winlogbeat-*

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

Custom query language
KQL

Rule type
Query

Timeline template
None

Alerts Assignees Manage rules

Status: open Severity User Host

Summary Trend Counts Treemap

Severity levels

Levels	Count
High	4
Medium	4

8 alerts

Alerts by name

Rule name	Count
PowerShell Execution Policy Bypass Attempt	4
Suspicious PowerShell Encoded Command Execution	3
PowerShell Hidden Window Execution	1

Top alerts by host_name

host_name	Count
desktop-a2n342	100%

Columns: 12 Sort fields: 1 8 alerts Fields: @timestamp Rule Assignees Severity Risk Score Reason

الحمد لله العاكس المبین