Alex Shah

EN.695.741.81.SP25 Information Assurance Analysis

Mod 10 Assignment – Splunk

April 9, 2025

1. Schema

Name	Type	Description	Reason
ip	String	IP address in dotted quad notation	Dotted quad notation requires string not Integer
datetime	datetime	A datetime stamp	Timestamps are standard types for events
uri	String	The requested resource URI	Combines GET/POST with URI strings
response	Integer	HTTP response code	Standard HTTP codes are three digits
size	Integer	Size of the response in bytes	Size in bytes can be represented as an Integer
ref	String	Referrer URL	URLs are strings
useragent	String	Useragent string	Useragents are strings

2. Script

I created a python script which defines the fields and types for the parsed http_log.txt records according to the schema. I read the file in with pandas and use the "|" delimiter to separate the parts of each record and store them in a dataframe so that the script arguments can return the requested matches and display the fields. I found that there were some lines with too many fields because they contain the delimiter character, so I created a count for when there were too few or too many fields. There was also a problem with some of the datetime stamps where they didn't make logical sense like February 29th, 2009 when 2009 was not a leap year which the datetime function caught, as well as a record where the seconds field was greater than 60. I counted those in the invalid records and skipped them.

```
alex@Mac Downloads % uv run --with pandas parsehttplog.py http_log.txt 200 ip datetime uri response size ref useragent

Record 200:
ip: 207.67.117.171
datetime: 2008-09-19 08:00:40-04:00
uri: GET /robots.txt HTTP/1.0
response: 200
size: 271
ref: -
useragent: larbin_2.6.3 (larbin2.6.3@unspecified.mail)

Summary:
Valid records: 21542
Invalid records: 11083
alex@Mac Downloads %
alex@Mac Downloads %
alex@Mac Downloads %
alex@Mac Downloads %
alex@Mac Downloads % uv run --with pandas parsehttplog.py http_log.txt 200 ip useragent

Record 200:
ip: 207.67.117.171
useragent: larbin_2.6.3 (larbin2.6.3@unspecified.mail)

Summary:
Valid records: 21542
Invalid records: 11083
alex@Mac Downloads %
alex@Mac Downloads %
alex@Mac Downloads %
alex@Mac Downloads %
alex@Mac Downloads % uv run --with pandas parsehttplog.py http_log.txt 1871 ip useragent
Record 1871 not found (corrupt or filtered).

Summary:
Valid records: 21542
Invalid records: 11083
alex@Mac Downloads %
```

Figure 1: Example use of the python script to parse records and answer queries

3. Problematic records

I looked through the records and found there were several patterns in URI or useragent containing suspicious strings like “cmd=” or sql attacks like “select * from”. So I made a list of patterns and as I parsed the records I saved when a record matched a suspicious pattern and counted how many were problematic. The whole file is parsed and the script returns counts for valid and invalid records and then returns the fields for the record number that the script arguments call for. Problematic and corrupt records were skipped when parsing the file, but I tried to keep the record numbers intact so that if you queried a record number that had been skipped, data from another record wouldn’t populate that index in the dataframe, the script would just return that it couldn’t be found. In the last example in Figure 1, record 1871 was removed because it had too many fields (contained delimiter characters) so when you query that record it shows that it was corrupt or filtered. I kept track of the reason that records were removed, which can be printed with the --verbose flag.

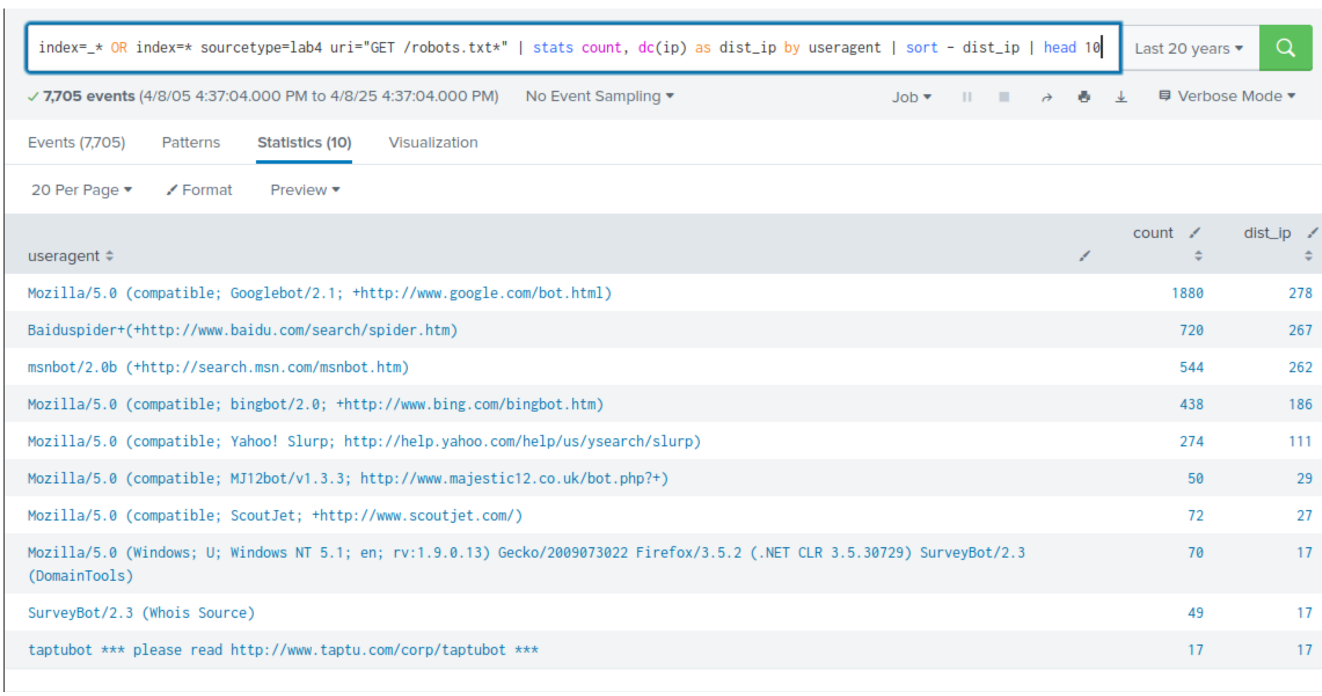
Analysis with Splunk:

4. 10 busiest spiders

The query I used is:

```
index=_* OR index=* sourcetype=lab4 uri="GET /robots.txt*" | stats count, dc(ip) as dist_ip by useragent | sort - dist_ip | head 10
```

This query searches for URI requests for robots.txt which are used in spiders, and shows the count of requests and number of distinct IP addresses made using that spider, sorted by the number of IP addresses and then limited to the top 10 busiest spiders.



useragent	count	dist_ip
Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)	1880	278
Baiduspider+(+http://www.baidu.com/search/spider.htm)	720	267
msnbot/2.0b (+http://search.msn.com/msnbot.htm)	544	262
Mozilla/5.0 (compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm)	438	186
Mozilla/5.0 (compatible; Yahoo! Slurp; http://help.yahoo.com/help/us/ysearch/slurp)	274	111
Mozilla/5.0 (compatible; MJ12bot/v1.3.3; http://www.majestic12.co.uk/bot.php?+)	50	29
Mozilla/5.0 (compatible; ScoutJet; +http://www.scoutjet.com/)	72	27
Mozilla/5.0 (Windows; U; Windows NT 5.1; en; rv:1.9.0.13) Gecko/2009073022 Firefox/3.5.2 (.NET CLR 3.5.30729) SurveyBot/2.3 (DomainTools)	70	17
SurveyBot/2.3 (Whois Source)	49	17
taptubot *** please read http://www.taptu.com/corp/taptubot ***	17	17

Figure 2: Top 10 busiest spiders

5. Masquerading as a spider

I created a query that contained instances of strings that bots would use but also contained suspicious patterns like SQL select statements or “cmd=” that would indicate malicious activity that could be someone masquerading as a spider. The results showed many requests masquering as Googlebot and others with various URI patterns from a few IP addresses.

```
index=_* OR index=* sourcetype=lab4 useragent="*bot*" OR useragent="*spider*" OR useragent="*crawler*" | search useragent="*<?system*" OR uri="*cmd=*" OR uri="*.php*" OR uri="/admin*" OR uri="*select * from*" | stats values(uri) as uris count by ip, useragent | sort -count | head 10
```

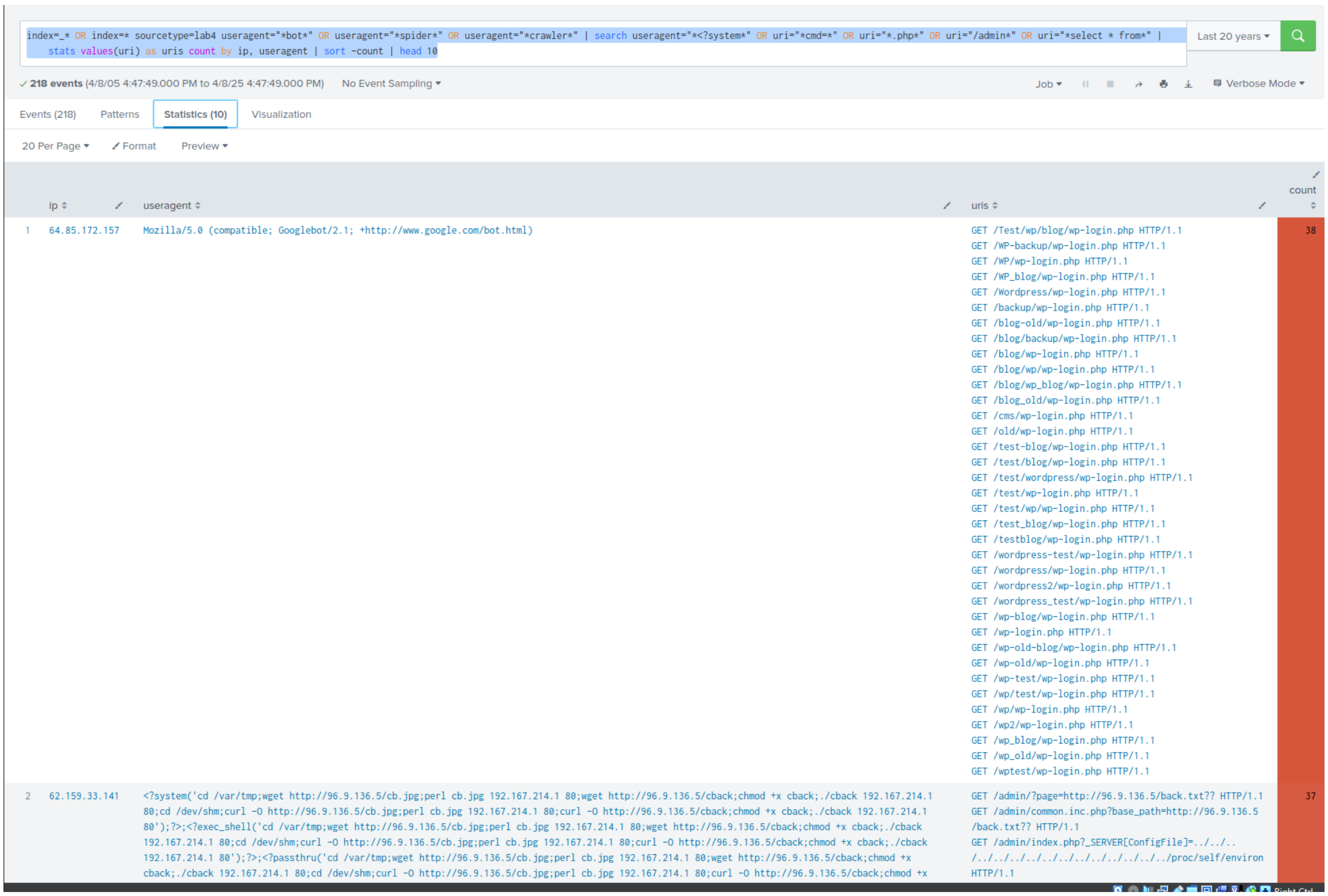


Figure 3: Requests from IPs masquerading as spiders

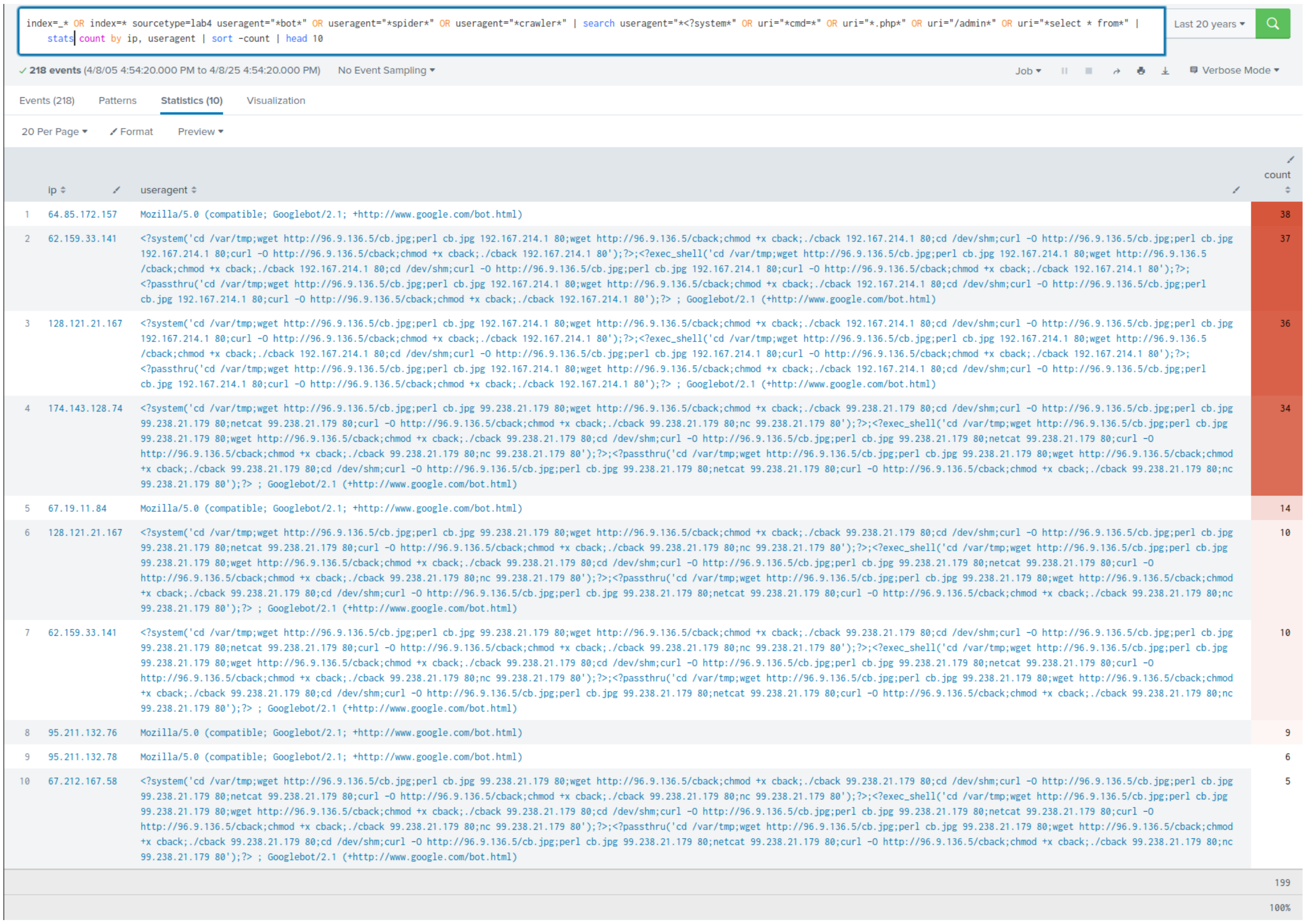


Figure 4: Limiting the query results to show more IP addresses, the useragents, and number of requests

6. Plot Googlebot activity

I inverted the selection using “NOT” to remove the fake spiders and selected for instances of Googlebot, then I used Splunk’s timechart to put the number of requests and number of IP addresses used into 1 month buckets and plotted them.

Query selecting Googlebot activity, showing IP, useragent, and request counts

```
index=_* OR index=* sourcetype=lab4 (useragent="*bot*" OR useragent="*spider*" OR useragent="*crawler*") AND useragent="*Googlebot*" | search NOT (useragent="*<?system*" OR uri="*cmd=*" OR uri="*.php*" OR uri="/admin*" OR uri="*select * from*") | stats count by ip, useragent | sort -count | head 10
```

Query using timehart to put activity into 1 month buckets

```
index=_* OR index=* sourcetype=lab4 useragent="*Googlebot*" NOT (useragent="*<?system*" OR
uri="*cmd=*" OR uri="*.php*" OR uri="/admin*" OR uri="*select * from*") | bin span=1mon _time
| stats count as req, dc(ip) as ips by _time | sort _time
```

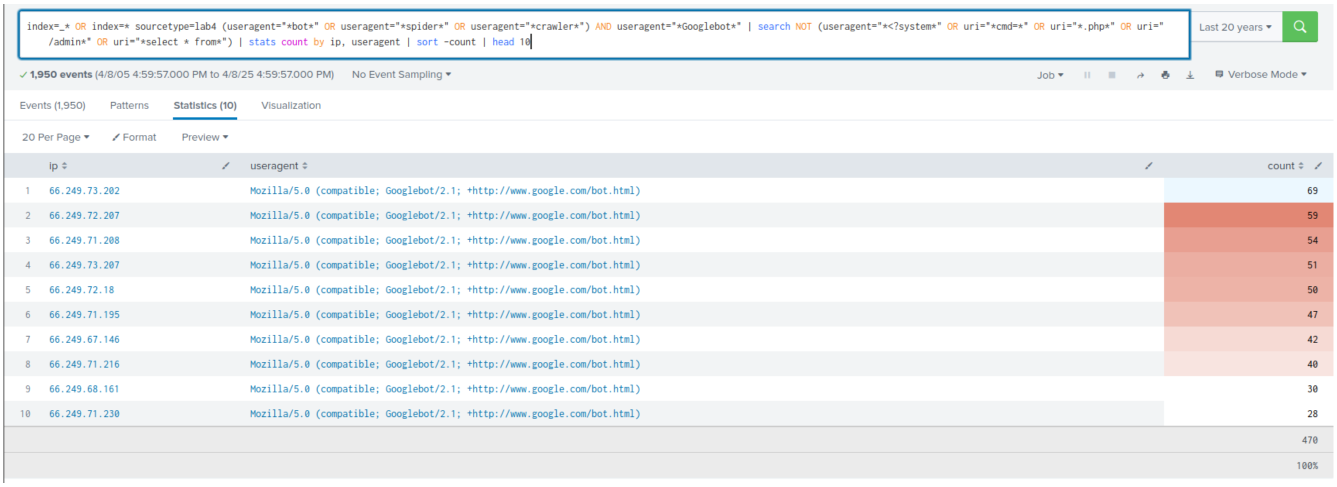


Figure 5: Selected Googlebot activity by IP, useragent, request count

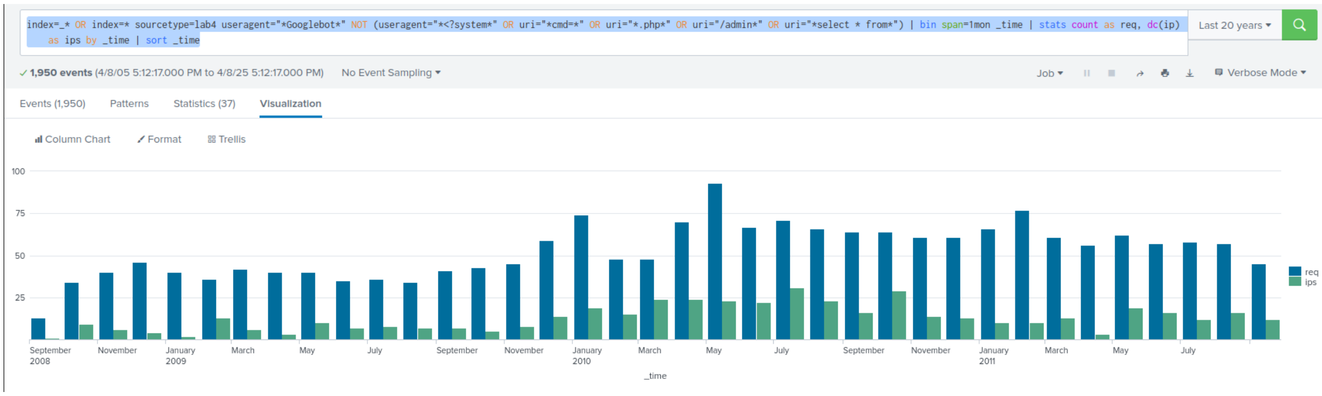
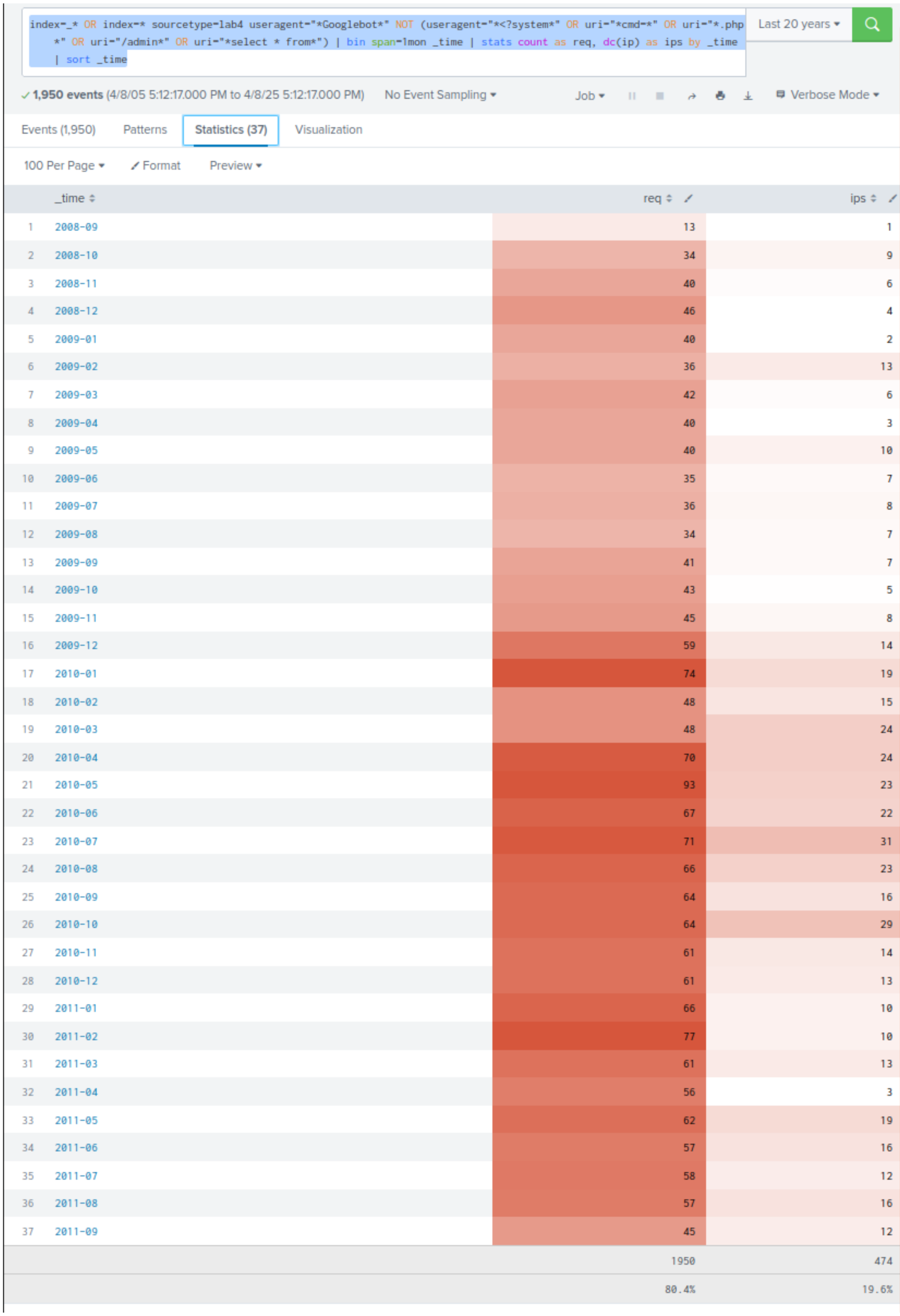


Figure 6: Timechart of Googlebot requests and number of distinct IPs by month



```
index=_* OR index=* sourcetype=lab4 useragent="*Googlebot*" NOT (useragent="*<?system*" OR uri="*cmd=*" OR uri="*.php
    *" OR uri="/admin*" OR uri="*select * from*") | bin span=1mon _time | stats count as req, dc(ip) as ips by _time
    | sort _time
```


	_time	req	ips
1	2008-09	13	1
2	2008-10	34	9
3	2008-11	40	6
4	2008-12	46	4
5	2009-01	40	2
6	2009-02	36	13
7	2009-03	42	6
8	2009-04	40	3
9	2009-05	40	10
10	2009-06	35	7
11	2009-07	36	8
12	2009-08	34	7
13	2009-09	41	7
14	2009-10	43	5
15	2009-11	45	8
16	2009-12	59	14
17	2010-01	74	19
18	2010-02	48	15
19	2010-03	48	24
20	2010-04	70	24
21	2010-05	93	23
22	2010-06	67	22
23	2010-07	71	31
24	2010-08	66	23
25	2010-09	64	16
26	2010-10	64	29
27	2010-11	61	14
28	2010-12	61	13
29	2011-01	66	10
30	2011-02	77	10
31	2011-03	61	13
32	2011-04	56	3
33	2011-05	62	19
34	2011-06	57	16
35	2011-07	58	12
36	2011-08	57	16
37	2011-09	45	12
		1950	474
		80.4%	19.6%

Figure 8: Heatmap of Googlebot requests and distinct IPs by month

7. Splunk advantages

Splunk's feature set covers more than data analysis and a query engine. It can be a strong addition to enterprise security monitoring and assist with operations or integrate with other products like SOAR solutions. For example, Splunk has real time monitoring capabilities which can be used like an SIEM to aggregate logs, generate reports, and send alerts for activities like login attempts, data exfiltration attempts, or warn about connections to malicious IPs or match other intelligence list patterns. It can also ingest more than network traffic like application logs cover all of an enterprise's activities that can send triggers through integrations to other systems. It can send alerts to firewalls, inform IDS/IPS systems, generate tickets, and otherwise connect multiple third party systems together. Splunk SOAR (or Splunk Phantom in older versions) are integrations and add ons for using Splunk to automate tasks through workbooks and playbooks that an enterprise can use to set up routines and protocols in response to activities like phishing attacks. Integrating third party systems is extremely beneficial in an enterprise environment that might use multiple products from different vendors, and Splunk can enable them to work together with a centralized data pool using add ons and pre made connectors for ingesting logs from different systems and use APIs to query and write to tools like firewalls and ticket systems. Splunk includes premade connectors, universal forwarders which can collect data from unsupported or custom applications, and the ability to write custom scripts and inputs. This makes Splunk a universal middleman system that can benefit security operations in the enterprise with real time analysis and automation with integrations to other security tools.

Sources

datetime — Basic date and time types. (n.d.). Python Documentation.
https://docs.python.org/3/library/datetime.html#datetime.datetime.strptime

Developing SOAR use cases using workbooks and playbooks. (2025, February 3). Splunk Lantern.

https://lantern.splunk.com/Security/UCE/Proactive_Response/Orchestrate_response_workflowsDeveloping_SOAR_use_cases_using_workbooks_and_playbooks

Splunk® Enterprise - Splunk Documentation. (2025). Splunk.com.

https://docs.splunk.com/Documentation/Splunk/8.1.0

Splunk® SOAR (On-premises) - Splunk Documentation. (2024). Splunk.com.

https://docs.splunk.com/Documentation/SOARonprem

Appendix

parsehttplog.py is also attached as a separate file to the assignment submission

```
"""
Filename: parsehttplog.py
Author: Alex Shah
Created: 2025-04-10
Description:
EN.695.741.81.SP25 Information Assurance Analysis
Mod 10 Splunk Assignment
Parse HTTP log script
Run with uv:
uv run --with pandas parsehttplog.py \
<filename> <record number> <field1> [<field2 ...] [--verbose]
"""

import sys
import pandas as pd
import re
from datetime import datetime

COLUMNS = ['ip', 'datetime', 'uri', 'response', 'size', 'ref', 'useragent']

SUSPICIOUS = [
r"<\?system", r"cmd=", r"\.php", r"/admin", r"select\s+\*\s+from",
r"curl", r"python-requests", r"nmap", r"sqlmap", r"wget"
]

def is_problematic(record):
```

```
issues = []

if not record['useragent'] or len(record['useragent']) < 5:
issues.append("useragent too short or missing")

for pattern in SUSPICIOUS:
if re.search(pattern, record['useragent'], re.IGNORECASE):
issues.append(f"Suspicious useragent: {pattern}")
if re.search(pattern, record['uri'], re.IGNORECASE):
issues.append(f"Suspicious uri: {pattern}")

return issues

def parse_datetime(date_str):
try:
return datetime.strptime(date_str, '%d/%b/%Y:%H:%M:%S %z')
except ValueError:
return None

def parse_log_file(filepath, verbose=False):
valid_records = []
corrupt_few, corrupt_many, problems = 0, 0, 0
problematic_lines = []

with open(filepath, encoding='utf-8') as f:
for line_num, line in enumerate(f, 1):
parts = line.strip().split('|')

if len(parts) < 7:
corrupt_few += 1
if verbose:
print(f"[Too Few Fields] Line {line_num}: {line.strip()}")
continue
if len(parts) > 7:
corrupt_many += 1
if verbose:
print(f"[Too Many Fields] Line {line_num}: {line.strip()}")
continue

try:
datetime_value = parse_datetime(parts[1])
if datetime_value is None:
raise ValueError("Invalid datetime format")

record = {
'ip': parts[0], # ip as string
'datetime': datetime_value, # datetime
'uri': parts[2], # uri as string
'response': int(parts[3]), # response as int
```

```
'size': int(parts[4]), # size as int
'ref': parts[5], # referrer as string
'useragent': parts[6] # useragent as string
}
except ValueError as e:
problems += 1
problematic_lines.append((line_num, str(e)))
continue

record['record_number'] = line_num

issues = is_problematic(record)
if issues:
problems += 1
problematic_lines.append((line_num, ' '.join(issues)))
continue

valid_records.append(record)

if valid_records:
df = pd.DataFrame(valid_records)
df['record_number'] = df['record_number'].astype(int)
df.set_index('record_number', inplace=True)
else:
df = pd.DataFrame(columns=COLUMNS)
#DEBUG
if verbose:
print(f"\nParsed {len(df)} valid records")
print(f"Corrupt (too few fields): {corrupt_few}")
print(f"Corrupt (too many fields): {corrupt_many}")
print(f"Problematic records: {problems}")
print("\nProblematic Lines:")
for line_num, reason in problematic_lines:
print(f"Line {line_num}: {reason}")
return df, corrupt_few, corrupt_many, problems

def show_record(df, record_num, fields):
if record_num not in df.index:
print(f"Record {record_num} not found (corrupt or filtered).")
return

print(f"\nRecord {record_num}:")
for field in fields:
if field in df.columns:
print(f"{field}: {df.at[record_num, field]}")

def main():
verbose = '--verbose' in sys.argv
filepath = sys.argv[1]
```

```
record_num = int(sys.argv[2])
fields = sys.argv[3:]

df, corrupt_few, corrupt_many, problems = parse_log_file(filepath, verbose)
show_record(df, record_num, fields)

print(f"\nSummary:")
print(f"Valid records: {len(df)}")
print(f"Invalid records: {corrupt_few + corrupt_many + problems}")

if __name__ == "__main__":
main()
```