威胁狩猎第一步

一、前言

尽管自动化安全工具以及第一层和第二层安全运营中心 (SOC) 分析师应该能够处理大约 80% 的威胁，但我们仍需对余下的20%保持高度的警惕。这部分未被充分覆盖的威胁中，往往隐藏着更为复杂且可能带来重大损失的潜在风险。

网络威胁狩猎为企业安全带来人为元素，补充了自动化系统的不足。凭借丰富的人为经验能够在威胁可能导致严重问题之前发现、记录、监控并消除威胁。

二、简单的ssh异常登陆检测

假设我们要构建一个最常见的异常检测场景：从海量的SSH连接日志中筛选出异常连接。

那么，如何界定SSH连接的异常性呢？我们可以从多个维度入手，比如时间维度（例如深夜的非正常工作时间段）、机器属性（如机器归属人、归属部门）以及历史记录等。这里，我们选择通过历史记录进行筛选，来做一个简单的实践。具体来说，我们可以统计今天的SSH连接记录，找出那些在过去一个月中从未出现过的连接，作为初步的异常检测。

虽然这个思路看似简单，但在实际操作中，却需要使用大数据处理组件来进行。毕竟，一个月的SSH连接数据量庞大，而且每条SSH日志除了包含IP等基本信息外，还包含其他丰富的附加信息，整体数据量大概10G。

我使用mac M3 36G机器上对10GB的ip.txt（模拟生成的）进行简单去重。

```python
import time
from collections import Counter

with open("ip.txt",encoding="gbk") as f:
    content = f.read()
    start_time = time.time()

    word_list = content.split(" ")
    word_counts = Counter(word_list)
    end_time = time.time()
    execution_time = end_time - start_time
    print(f"Execution time: {execution_time} seconds")
    print(word_counts)

```

使用python进行处理，内存直接爆炸了，就更不用说处理时间方面了（脚本都没跑完）

那如何解决呢?

三、站在巨人的肩膀上spark

剖析一下为何此次操作会以失败告终：原因在于，我试图一次性将10GB的庞大数据文件全部加载到内存中，随后使用Python进行split、Counter等处理操作，这无疑导致内存使用量急剧飙升，最终因内存耗尽而引发程序崩溃。

为了优化这一流程，我们应当采取分段读取的策略，即每次仅读取文件的一部分数据，并对其进行相应的计算处理。同时，我们还可以利用多线程技术来充分压榨CPU的性能，从而提升运算效率（当然，这一切都需要在严格的内存管理之下进行）。

事实上，上述这些繁琐的步骤已经有人为我们提前做好了：Apache Spark 是一种用于大数据工作负载的分布式开源处理系统。它使用内存中缓存和优化的查询执行方式，可针对任何规模的数据进行快速分析查询。

比如我们生成的一个亿ssh连接日志

```
import random

with open("temp.csv","w") as f:
    for _i in range(1,10000000):
        host_ip = "10.0.{}.{}".format(str(random.randint(0, 255)),str(ra
        dst_ip = "10.0.{}.{}".format(str(random.randint(0, 255)), str(ra
        # print(f"log_service,1734770440,aegis-log-login,22,{host_ip},{d
        f.write(f"log_service,1734770440,aegis-log-login,22,{host_ip},{d
```

```
cat login.csv >> login.csv
cat login.csv >> login.csv
cat login.csv >> login.csv
cat login.csv >> login.csv
cat login.csv >> login.csv
cat login.csv >> login.csv
cat login.csv >> login.csv
cat login.csv >> login.csv
cat login.csv >> login.csv
cat login.csv >> login.csv

```

比如我们的ssh login日志文件是login.csv（csv）格式，将dst_ip > host_ip作为一条记录
注意：这里还需要使用hadoop，因为涉及到文件的共享处理的问题，而spark又比较好支持haddop。

```
String inputPath = "hdfs://127.0.0.1:9000/login.csv";

```

```
JavaPairRDD<String, Integer> counts = textFile.
        flatMap(new FlatMapFunction<String, String>() {
            @Override
            public Iterator<String> call(String s) throws Exception {
                String[] columns = s.split(",\\s*");
                if (columns.length < 4) {
                    // 如果列数不足4，则跳过这一行
                    return null;
                }
                String joinedKey =  columns[5]+ ">" + columns[4];
                return Arrays.asList(joinedKey).iterator();
            }
        })
        .mapToPair(new PairFunction<String, String, Integer>() {
            @Override
            public Tuple2<String, Integer> call(String word) throws Exception {
                return new Tuple2<>(word, 1);
            }
        })
        .reduceByKey(new Function2<Integer, Integer, Integer>() {
            @Override
            public Integer call(Integer a, Integer b) throws Exception {
                return a + b;
            }
        });

```

统计dst_ip > host_ip的次数，并且进行排序

```
//先将key和value倒过来，再按照key排序
JavaPairRDD<Integer, String> sorts = counts
        //key和value颠倒，生成新的map
        .mapToPair(tuple2 -> new Tuple2<>(tuple2._2(), tuple2._1()))
        //按照key倒排序
        .sortByKey(false);

```

最终保存结果

```
//分区合并成一个，再导出为一个txt保存在hdfs
javaSparkContext.parallelize(all).coalesce(1).saveAsTextFile(outputPath);
```

最后搭建spark docker镜像里面，这里我们将内存限制了4G。

```yaml
version: '3.8'

services:
  spark:
    image: docker.io/bitnami/spark:3.5
    environment:
      - SPARK_MODE=master
      - SPARK_RPC_AUTHENTICATION_ENABLED=no
      - SPARK_RPC_ENCRYPTION_ENABLED=no
      - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
      - SPARK_SSL_ENABLED=no
      - SPARK_USER=spark
    ports:
      - '8080:8080'
      - '7077:7077'
  spark-worker:
    image: docker.io/bitnami/spark:3.5
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark:7077
      - SPARK_WORKER_MEMORY=4G
      - SPARK_WORKER_CORES=1
      - SPARK_RPC_AUTHENTICATION_ENABLED=no
      - SPARK_RPC_ENCRYPTION_ENABLED=no
      - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
      - SPARK_SSL_ENABLED=no
      - SPARK_USER=spark

```

13.73GB，一个亿的数据量，spark在一个4G内存，一个核心CPU的容器中101秒完成从hadoop读取文件，并且对文件进行解析、计算。

Workers (1)

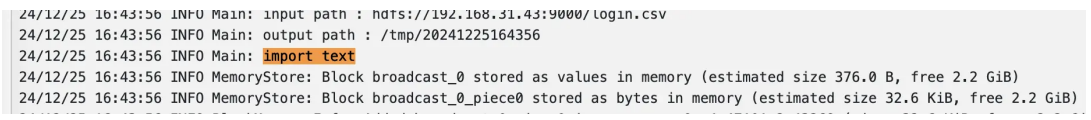
Worker Id	Address	State	Cores	Memory	Resources
worker-20241225164204-172.23.0.4-43613	172.23.0.4:43613	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Running Drivers (0)

Submission ID	Submitted Time	Worker	State	Cores	Memory	Resources	Main Class	Duration



```
24/12/25 16:43:56 INFO Main: input path : hdfs://192.168.31.43:9000/login.csv
24/12/25 16:43:56 INFO Main: output path : /tmp/20241225164356
24/12/25 16:43:56 INFO Main: import text
24/12/25 16:43:56 INFO MemoryStore: Block broadcast_0 stored as values in memory (estimated size 376.0 B, free 2.2 GiB)
24/12/25 16:43:56 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in memory (estimated size 32.6 KiB, free 2.2 GiB)
```

```
24/12/25 16:45:37 INFO DAGScheduler: Job 3 finished: collect at Main.java:87, took 14.234965 s
24/12/25 16:45:37 INFO Main: top 10 word :
10.0.156.206>10.0.217.223       30
10.0.40.190>10.0.180.27 30
10.0.101.35>10.0.233.97 30
10.0.223.181>10.0.5.204 30
10.0.224.88>10.0.161.77 30
10.0.156.91>10.0.90.88  30
10.0.104.16>10.0.12.48  30
10.0.81.72>10.0.30.226  20
10.0.194.152>10.0.146.142       20
10.0.211.37>10.0.70.99  20
```

```
drwxr-xr-x 2 1001 root 4.0K Dec 25 16:45 .
drwxrwxrwt 1 root root 4.0K Dec 25 16:45 ..
-rw-r--r-- 1 1001 root    8 Dec 25 16:45 ._SUCCESS.crc
-rw-r--r-- 1 1001 root 2.2M Dec 25 16:45 .part-00000.crc
-rw-r--r-- 1 1001 root    0 Dec 25 16:45 _SUCCESS
-rw-r--r-- 1 1001 root 279M Dec 25 16:45 part-00000
root@0ee4e47101c2:/tmp/20241225164356#
```

```
root@0ee4e47101c2:/tmp/20241225164356# head part-00000
(30,10.0.156.206>10.0.217.223)
(30,10.0.40.190>10.0.180.27)
(30,10.0.101.35>10.0.233.97)
(30,10.0.223.181>10.0.5.204)
(30,10.0.224.88>10.0.161.77)
(30,10.0.156.91>10.0.90.88)
(30,10.0.104.16>10.0.12.48)
(20,10.0.81.72>10.0.30.226)
(20,10.0.194.152>10.0.146.142)
(20,10.0.211.37>10.0.70.99)
```

最后一个简单的脚本，即可筛选出的想要的登陆数据

```python
test_list = [
    "10.0.156.206>10.0.217.223",
    "10.0.156.206>10.0.217.224"
]

content = ""
with open("part-00000") as f:
    content = f.read()

for _test in test_list:
    if _test in content:
        print(_test + " normal")
    else:
        print(_test + " abnormal")

```

```
10.0.156.206>10.0.217.223 normal
10.0.156.206>10.0.217.224 abnormal
```

检测的结果并不是认为异常的，还需要结合多维度的日志才能得出最终结论。

四、总结

这种规模庞大的离线数据处理场景其实屡见不鲜，上述例子只是其中最为基础的一个。同样地，诸如进程的异常派生、网络的异常连接等复杂问题，也都可以运用类似的思路来解决。（在处理这些场景时，采用Apache Spark作为解决方案，是一个不错的选择。）