Pointer: Hunting Cobalt Strike globally

By Pavel Shabarkin

Published: 2021-11-21 · Archived: 2026-04-05 22:28:12 UTC

14 min read

Sep 16, 2021

Introduction

Cobalt Strike is a commercial, full-featured, remote access tool that bills itself as “adversary simulation software designed to execute targeted attacks and emulate the post-exploitation actions of advanced threat actors”. Cobalt Strike’s interactive post-exploit capabilities cover the full range of ATT&CK tactics, all executed within a single, integrated system.

In addition to its own capabilities, Cobalt Strike leverages the capabilities of other well-known tools such as Metasploit and Mimikatz.

Cobalt Strike is a legitimate security tool used by penetration testers and red teamers to emulate threat actor activity in a network. However, lately, this tool has been hijacked and abused by cybercriminals.

Our goal was to develop a tool to help identify default Cobalt Strike servers exposed on the Internet. We strongly believe that understanding and mapping adversaries and their use of Cobalt Strike can improve defenses and boost organization detection & response controls. Blocking, mapping and tracking adversaries is a good start.

Press enter or click to view image in full size

Pointer logo

Tool Development

A review of existing Cobalt Strike detection tools and public research showed that current tools can only scan a small number of potential Cobalt Strike instances (1–5k hosts). Our goal was to increase the scanning capabilities and validate several million potential Cobalt instances in less than an hour.

To achieve the above goal within a reasonable timeframe and on a small budget, it was necessary to adapt and scale the current understanding of the Cobalt Strike hunting methodology. The following content assumes an understanding of what Cobalt Strike is and how to locate and identify Cobalt strike instances. Before going into the details of the tool and their components, let’s take a look at the general architecture.

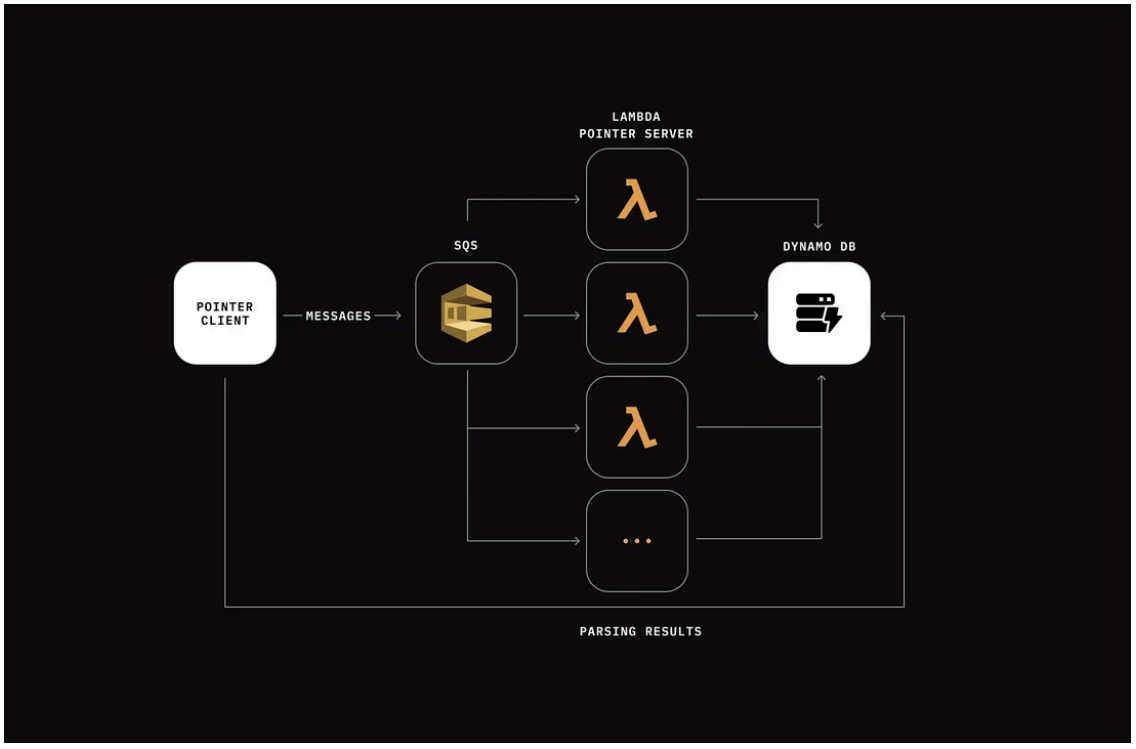
Architecture review

Scanning a large number of hosts in a reasonable amount of time does not scale and has physical, cost and power limitations. Unless you have a great home lab and the bandwidth to support it, personal computing cannot really solve the scaling problem, so the decision was made to use AWS to affordably scale and achieve the desired goals.

General architecture review

The tool is developed and heavily based on AWS SQS, Lambda and DynamoDB.

Press enter or click to view image in full size



The Pointer client parses the local json file with a list of IPs, optimally splits them into packets (10–20 IPs), and then adds the packets to be processed to the SQS queue.

The SQS queue is setup to invoke a lambda function for each packet in the queue. The lambda function (Pointer server) performs the actual scanning of the provided packet of IPs and saves results to DynamoDB.

In cases where Lambda fails or throws an error, packets are returned to the SQS queue and will wait for a retry.

If the packet fails a second time, a new Lambda function is launched that logs the failed packet to DynamoDB for further analysis and rescan each IP individually to locate the failed IPs.

Code Review

The scan functionality of the “Pointer server” consists of 4 parts:

1. Port Scanning (Port Workers)
2. HTTP Webservice scan (HTTP Workers)

- Certificate parsing
- JARM parsing

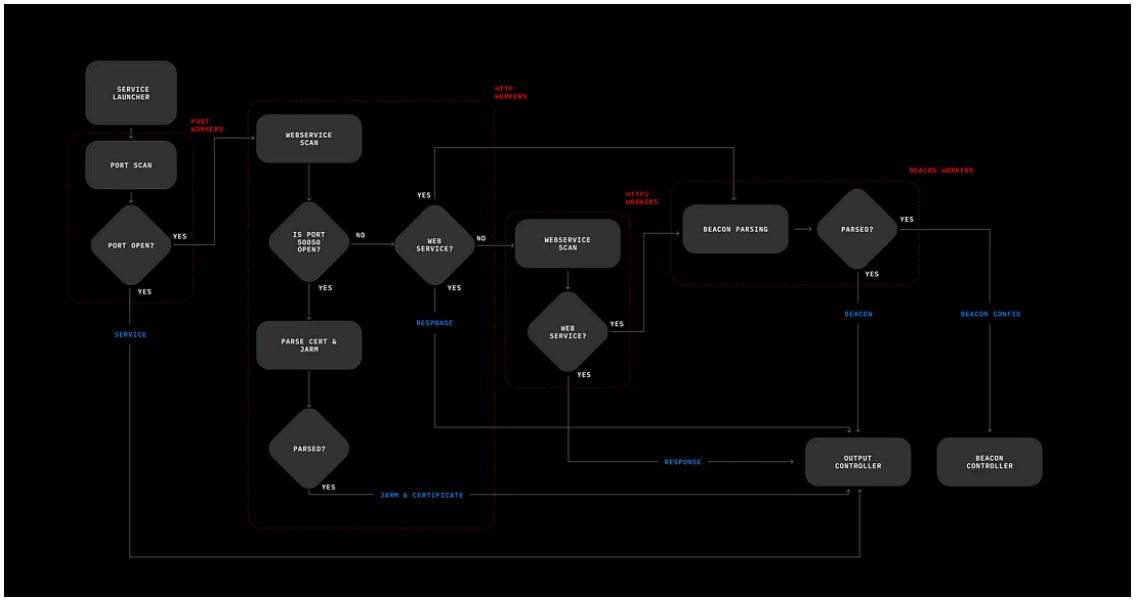
3. HTTPS Webservice scan (HTTPS Workers)

4. Beacon Parsing (Beacon Workers)

The tool was designed with an asynchronous approach to IP processing. Each scan probe stands as an independent unit, which is then processed by a Worker. The probes include a port scanning, Certificate Issuer parsing, JARM parsing, webservice scanning, and Beacon parsing. Once each probe is completed, the result is sent to the

corresponding controller, which writes the result to the global map. After all scan workers are done, the data is sorted and ordered before being combined into the `Target` structure. Overall, this reduces the number of delays since each service(ip:port) has its own scan pipeline.

Press enter or click to view image in full size



Internal architecture of Lambda function

Detailed review

Initially the lambda function launches Port, HTTP, HTTPS, and Beacon workers. The number of workers depends on the level of internal concurrency (Internal concurrency is the controllable CLI parameter). Each type of worker is portioned accordingly to the required power resources. Portioning has been calculated based on the number of probes each worker performs in average.

Each targeted IP address is scanned for 27 predefined ports, this list includes common ports on which Cobalt Strike beacons are hosted. The “launcher” sends service (ip:port) to the Port Workers through `portChannel` Golang channel.

Press enter or click to view image in full size

```
for _, ip := range targets.Ips {
    sort.Responses[ip] = make(map[string]string)
    for _, port := range settings.Ports {
      portChannel <- fmt.Sprintf("%s:%d", ip, port)
    }
}
```

Code snippet of the Service Launcher

Port workers then scan the individual ports. If a port is open, the worker sends the service to the HTTP Worker and Output controller through `httpChannel` and `outputChannel` Golang channels. If the port is closed the Port Worker exits the function.

Press enter or click to view image in full size

```
var portWG sync.WaitGroup
  for i := 0; i < int(settings.Concurrency*37/100); i++ {
    portWG.Add(1)
    go func() {
      for srv := range portChannel {
        conn, err := net.DialTimeout("tcp", srv, time.Millisecond*time.Duration(settings.PortDelay))
        if err != nil {
          continue
        }
        conn.Close()
        httpChannel <- srv
        outputChannel <- "Service|" + srv
      }
      portWG.Done()
    }()
  }
```

Code snippet of Port Worker

All workers send results through a single Golang channel, `outputChannel`, which are then processed by the output controller and saved to the global map (`Sorter` struct).

Each result produced by the workers has its own type tag (Ex: `"Service|"`, `"Certificate|"`, `"Jarm|"`, …), ensuring that the `ValidateOutput` function can sort the results based on their types.

Press enter or click to view image in full size

```
var outputWG sync.WaitGroup
  for i := 0; i < int(settings.Concurrency*5/100); i++ {
    outputWG.Add(1)
    go func() {
      for o := range outputChannel {
        utils.ValidateOutput(o, sort)
      }
      outputWG.Done()
    }()
  }
```

Code snippet of Output Controller

The HTTP worker waits for IP and port tuple (service) to be provided by the Port Worker via the `httpChannel`. If the HTTP Worker receives port 50050 it attempts the following actions:

- Parse the certificate issuer -> identifying the default self-signed Cobalt certificate
- Parse the JARM signature -> detecting malicious JARM signatures

For other services, it performs a web request to analyse response behaviour. Beacon's HTTP/HTTPS indicators are controlled by a malleable C2 profile, if the server uses the default malleable C2 profile, it responds with a 404 status code and 0 content-length for requests made to the root web endpoint. (http://domain.com/)

If the request to the targeted web service fails, HTTP Worker sends the service through `httpsChannel` channel further to the HTTPS Worker to perform the web request through HTTPS protocol.

Press enter or click to view image in full size

```
var httpWG sync.WaitGroup
  for i := 0; i < int(settings.Concurrency*28/100); i++ {
    httpWG.Add(1)

    go func() {
      for host := range httpChannel {
        response.Ports += 1
        if utils.CheckPort(host, 50050) {
          outputChannel <- "Certificate|" + host + "|" + GetTeamServerCertificate(host)
          outputChannel <- "Jarm|" + host + "|" + CheckJARM(host)
          continue
        }

        url := "http://" + host
        ok, resp := WebRequest(settings.HttpDelay, url)
        if ok {
          response.Services += 1
          if resp == "404/0" {
            beaconChannel <- url
          }
          outputChannel <- "Response|" + url + "|" + resp
          continue
        }
        httpsChannel <- host
      }
      httpWG.Done()
    }()
  }
```

Code snippet of the HTTP Worker

Being inspired by the “Analyzing Cobalt Strike for Fun and Profit” research and its corresponding tool for cobalt strike beacon parsing (developed using Python), we integrated the similar logic into our tool for beacon parsing (developed using Golang).

The guy, who researched how the beacon is packed, how to parse the beacon, how to decrypt the beacon, and how to work with that in general, you did the good job a big thank you!

All identified web services that have been configured with default malleable C2 profile are sent to the Beacon Workers. The Beacon Worker attempts to parse the beacon config. If the parsing succeeds, Beacon Worker sends the `CobaltStrikeBeaconStruct` struct to the Beacon controller through `beaconStructChannel` channel, and the beacon location URI to the output controller through `outputChannel` channel.

Press enter or click to view image in full size

```
var beaconWG sync.WaitGroup
  for i := 0; i < int(settings.Concurrency*25/100); i++ {
    beaconWG.Add(1)

    go func() {
      for url := range beaconChannel {
        beacon, beaconUri := GetBeaconConfig(settings.HttpBeaconDelay, url)
        if beacon != nil {
          response.Beacons += 1

          beaconStructChannel <- utils.CobaltStrikeBeaconStruct{
            Uri:          beaconUri,
            BeaconConfig: beacon,
          }
          outputChannel <- "Beacon|" + beaconUri
        }
      }
      beaconWG.Done()
    }()
  }
```

Code snippet of Beacon Worker

Press enter or click to view image in full size

```
var beaconStructWG sync.WaitGroup
  beaconStructWG.Add(1)
  go func() {
    for b := range beaconStructChannel {
      BeaconConfigs = append(BeaconConfigs, b)
    }
    beaconStructWG.Done()
  }()
```

Code snippet of Beacon Controller

When all workers finish the scans, the `Sort` method maps all gathered scan results sent to the output controller into the array of `CobaltStrikeStruct` type:

Press enter or click to view image in full size

```
type CobaltStrikeStruct struct {
  Ip          string
  Ports       []string
  Responses   map[string]string
  Jarm        string
  Certificate string
  Beacons     []string
  Probability float32
}
```

Code snippet of `CobaltStrikeStruct data type`

The `Probability` field is assigned when the `Voter` function calls the internal method `Vote` for each `CobaltStrikeStruct` object within the array.

In case the certificate issuer matches the default Cobalt Strike self-signed certificate, the `Vote` method gives 100% probability that it is the Cobalt Strike server. The same applies if the Beacon Worker successfully parses the beacon config hosted on the web service.

Default web service response and malicious JARM signature results cannot give us confidence in assigning the probability rate. Because other web services can respond with 0 content length and 404 status code, and servers can be configured with the same TLS options (if you don't understand what JARM is). If the `Vote` method matches only those two indicators, it assigns the 70% probability to the object.

If none of these meet our requirements, it is probably not a Cobalt Strike server. But, again, this tool targets only Cobalt Strike servers with default malleable C2 profile configurations.

Press enter or click to view image in full size

```
func (this *CobaltStrikeStruct) Vote() {
  if this.Certificate == "Major Cobalt Strike" {
    this.Probability = 1.0
    return
  } else if this.Beacons != nil {
    this.Probability = 1.0
    return
  } else if this.matchJarm() && this.matchResponse() {
    this.Probability = 0.7
    return
  } else {
    this.Probability = 0.0
  }
}
```

Code snippet of the Vote method

DynamoDB Component

We chose DynamoDB service to store scan results. DynamoDB can handle more than 10 trillion requests per day and support peaks of more than 20 million requests per second. That is what we needed 100%! We wanted to scan 20000–25000 targets per 60 seconds, which is about 40k-50k writing requests to the database.

On the first implementation, the Output and Beacon workers exceeded DynamoDB rate limits because they performed a write request to the DynamoDB table for each target object separately, and, in addition, we used the default DynamoDB configuration. The default capacity configuration could not handle that many requests, but by increasing the capacity we would pay more money for autoscaling during constant scanning. Further examination of the AWS documentation revealed that AWS had implemented the batch write to DynamoDB. For each lambda invocation, we have 10–20 targets (depending on the packet size) to scan, so this should reduce the number of requests to DynamoDB tables by a factor of 10–20.

We found that DynamoDB's `BatchWriteItemInput` function allows writing up to 25 items and up to 16 Mb in one request. The batch write implementation significantly decreased the number of requests and removed the rate limiting issue at the default configuration level. We did not have to pay for unnecessary autoscaling.

This method has the disadvantage that if one of the items in the batch fails to be written, the whole batch will not be saved. (The partition key must be unique and not exist in the table, but this is suitable in our case, as we filter our targets by unique values before launching the scans).

Press enter or click to view image in full size

```go
func WriteBatchTarget(svc *dynamodb.Client, targets []CobaltStrikeStruct) {

  request := make(map[string][]types.WriteRequest)

  for _, target := range targets {
    av, err := attributevalue.MarshalMap(target)
    if err != nil {
      log.Fatalf("Got error marshalling scan item: %s", err)
    }

    request[TableTargets] = append(request[TableTargets], types.WriteRequest{
      PutRequest: &types.PutRequest{
        Item: av,
      }})

  }

  input := &dynamodb.BatchWriteItemInput{
    RequestItems: request,
  }

  _, err := svc.BatchWriteItem(context.TODO(), input)
  if err != nil {
    fmt.Println(err.Error())
  }
}
```

Code snippet of the WriteBatchTarget function

Also, for unpredictable cases where the default capacity configuration cannot handle a large number of requests, we configure autoscaling:

AWS Console → DynamoDB → choose the Table → Edit Capacity → Read / Write Capacity increase to 10–15. To enable autoscaling we should give the required permissions for the DynamoDB service role .

Lambda Component

AWS Lambda is an interesting service. We wanted to try Lambda as a core service for our scans, however we did not want to get a crazy paycheck at the end of the month, so we had several things to figure out:

1. How much memory to allocate for Lambda execution
2. What default timeout to set for Lambda execution
3. How to manage Lambda concurrency
4. What internal concurrency would be suited for our model;
5. What request timeouts would be suited for our model
6. What packet size would be suited for our model

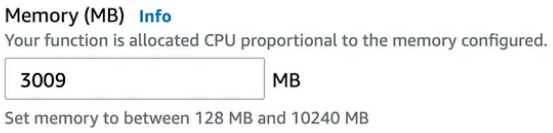
And the most difficult question — How to setup everything the way it would be efficient, cheap, and with minimum loss rate?

Lambda memory allocation

It was interesting to research how AWS allocates memory and CPU for Lambda functions, because it is physically impossible to divide 1/10 of the CPU. But it can allocate 1/10 of the time of the CPU to a single function, and you can have 10 of them working at the same time to share the same CPU core (check this research, it explains how AWS Lambda allocates CPU).

The only controllable parameter in AWS for Lambda functions is memory usage:

Press enter or click to view image in full size



Example of the memory configuration in AWS Console

We designed our model with a multithreaded architecture — the more cores we have, the better performance we can potentially obtain. But the nasty thing here is what the price of this luxury is.)))

We cannot directly control the number of cores we want to use. The CPU performance scales with the memory configuration. Lambda functions used to always have 2 vCPU cores, regardless of the allocated memory. The rest of the cores are throttled at certain memory configurations. By increasing the memory allocation, we obtain more cores. I found the research that discovered how the number of vCPUs and multithreaded computation power vary depending on the memory configuration.

The price for using the AWS Lambda function is based on the function runtime (in milliseconds) multiplied by the allocated memory (fixed prices per Mb). So, allocating 3008MB for the Lambda function, we get 2 vCPUs, and

allocating 3009MB we get 3vCPUs. By allocating 3009MB memory, we could gain more performance, at almost the same price.)))

According to the research, we get better performance gain for multithreading with each spike transition (jump in cores). But for our model we do not need more than 3 cores, 3009MB is enough for our purposes.

Memory	vCPUs
128 - 3008 MB	2
3009 - 5307 MB	3
5308 - 7076 MB	4
7077 - 8845 MB	5
8846+ MB	6

Correlation between Lambda memory configuration and number of cores

By the way, we decided to measure the computational power ourselves, and the practical tests showed that the power spike between 3008–3009 MB is bigger than between 5307–5308 MB. This once again confirms that the 3009MB memory configuration is the best choice for us.

Press enter or click to view image in full size

```go
package main

import (
  "log"
  "runtime"
  "time"
)

// Print system resource usage every 2 seconds.
func System() {
  mem := &runtime.MemStats{}

  for {
    cpu := runtime.NumCPU()
    log.Println("CPU:", cpu)

    rot := runtime.NumGoroutine()
    log.Println("Goroutine:", rot)

    // Byte
    runtime.ReadMemStats(mem)
    log.Println("Memory:", mem.Alloc)

    time.Sleep(2 * time.Second)
    log.Println("-------")
  }
}

func main() {
  go System()

  // Simulate never ending client requests.
  for {
    // Pretend like each request task (goroutine) takes 10 seconds.
    go func() {
      time.Sleep(10 * time.Second)
    }()

    // Wait 1 second before creating a new request.
    time.Sleep(time.Second)
  }
}
```

Code snippet for measuring the multithreaded computation power

Internal parameter tuning

As we got a solid understanding of how many resources to use, we started tuning and suiting other parameters for our model.

In my opinion, the lifetime of the Pointer lambda function should not be more than 60 seconds, because otherwise it will not be a true server-less tool with easy management, stable to errors and autoscaled architecture.

With a memory configuration of 3009 Mb and a default timeout of 60 seconds for one Lambda execution, we could scan from 10–20 targets in a single packet.

In case the lambda execution fails, we do not want to rescan all the targets inside the packet again. By having less number of targets in the packet, we minimize the probability that the packet will be crashed. Therefore, the optimal size, in my opinion, is 10–20 targets.

By having less number of targets inside the packet, we are minimising the chance that the packet will be crashed. In case the lambda execution fails, we must rescan all targets inside the packet (even those that have been successfully scanned).

When we defined lambda configuration parameters, the rest of the parameters were tuned with a big number of tests:

```
Targets/per packet  20    (Items)
Concurrency         140   (Items)
```

```
Lambda Memory       3009  (Mb)
Lambda Timeout      60    (sec)
Http timeout        4     (sec)
Port timeout        2     (sec)
Beacon timeout      10    (sec)
```

Lambda Concurrency

AWS Lambda provides autoscaling for function instances. But we simply cannot deploy as many instances as we want. We are limited by the AWS region quota (All the lambda functions of an account can use the pool of 1000 unreserved concurrent executions). Thus, having only 1 deployed function, we could get 1000 concurrent executions at the same time.

Dorking potential Cobalt Strike servers through Shodan, we could retrieve around 200–300k potential targets. However, we are designing the tool to scan 2M-10M targets. For example, 2M targets is about 100k packets (20 targets per packet), which means 100k lambda function invocations. If Lambda function is invoked 100k times, the lambda puller would process only 1k requests at a time, and the rest would be just throttled. So even if we increase the AWS region quota, it will not be enough.

So the question arises — how we can manage the invocation process? The answer is simple — SQS.

SQS Component

Configuration

Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications. This means we can send all our packets to the queue and SQS will manage the process of lambda invocation. The SQS management can be configured according to our needs:

1. We can control the number of retries.

- We can configure the maximum number of retries the SQS would perform, if the batch of messages(packets) fails
- The SQS sends messages packed in the batches, we can control the number of messages inside the batch we want to pass to the Lambda function, so having 1 message in the batch will equal 1 message.
- In our model we decided that If message fails more than once, it would be sent to the Dead-Letter-Queue (DLQ). We designed the Dead-Letter-Queue (DLQ) to redirect the failed messages to the Lambda function with the same logic as the core one, but before scanning activities it writes the the failed packet to the DynamoDB table and rescans each target separately.

2. Visibility timeout

- The visibility timeout sets the length of time that a message received from the queue (by one lambda function) will not be visible to the lambda function again. If the lambda function fails to process and delete

the message before the visibility timeout expires, the message becomes visible to the lambda function again.

3. SQS batch size

- We configured SQS batch size to a single message.

SQS & Lambda autoscaling

For standard queues, Lambda uses long polling to poll the queue until it becomes active. When messages are available, Lambda reads up to 5 batches and sends them to our lambda function. If messages are still available, Lambda increases the number of processes that read batches up to 60 more instances per minute. The maximum number of batches that can be processed simultaneously by event source mapping is 1000. This means that the full power we can get after 16 minutes of continuous scanning.

Results

At the first launch, when we ran a scan for 160k targets, we were able to identify 1,700 Cobalt Strike servers and parse 1,400 of their beacon configurations within 40 minutes. The Pointer tool can produce best performance results if the target size exceeds 500k. Scanning 160k targets took a little longer because 1000 concurrent lambda executions were achieved only after 30 minutes when the tool was launched. For the current implementation, the cost of scanning 250k targets is about 20$, however we are looking for a solution that will make it cheaper.

Targets table [sample]

We have developed 2 tables, first one for identified Cobalt Strike servers, and the second for parsed beacon configurations. Identified Cobalt Strike servers can be described by 7 features:

- IP address is a unique sorting key
- probability that it’s the actual cobalt strike server (easier filtering)
- JARM signature
- Certificate Issuer
- Opened Ports
- Response behaviour
- Links to the beacon configurations that we parsed and saved to another table

There is an example of the cobalt strike server table:

Press enter or click to view image in full size

ip	probability	jarm	certificate	ports	responses	beacons
99.79.101.225	1	07d14d16d21d21d00042d41d00041de5fb3038104f457d92ba02e9311512c2	Major Cobalt Strike	22,80,443,50050	"http://99.79.101.225:80":"404/0";"https://99.79.101.225:443":"404/0"	http://99.79.101.225:80/aaa9;https://99.79.101.225:443/aaa9
99.79.101.225	1	07d14d16d21d21d00042d41d00041de5fb3038104f457d92ba02e9311512c2	Major Cobalt Strike	80,22,443,50050	"http://99.79.101.225:80":"404/0";"https://99.79.101.225:443":"404/0"	http://99.79.101.225:80/aaa9;https://99.79.101.225:443/aaa9
98.142.143.100	1			80,443	"http://98.142.143.100:80":"404/0";"https://98.142.143.100:443":"404/0"	http://98.142.143.100:80/aaa9;https://98.142.143.100:443/aaa9
96.45.182.23	1			443,80	"http://96.45.182.23:80":"200/-1";"https://96.45.182.23:443":"404/0"	https://96.45.182.23:443/aaa9
96.45.180.42	1	07d14d16d21d21d07c07d14d07d21d9b2f5869a6985368a9dec764186a9175	Major Cobalt Strike	80,50050	"http://96.45.180.42:80":"404/0"	http://96.45.180.42:80/aaa9
96.45.180.42	1	07d14d16d21d21d07c07d14d07d21d9b2f5869a6985368a9dec764186a9175	Major Cobalt Strike	80,50050	"http://96.45.180.42:80":"404/0"	http://96.45.180.42:80/aaa9
96.45.179.18	1			443	"https://96.45.179.18:443":"404/0"	https://96.45.179.18:443/aaa9
96.44.160.141	1			22,443	"https://96.44.160.141:443":"404/0"	https://96.44.160.141:443/aaa9
95.85.67.149	1	07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1	Major Cobalt Strike	22,50050		

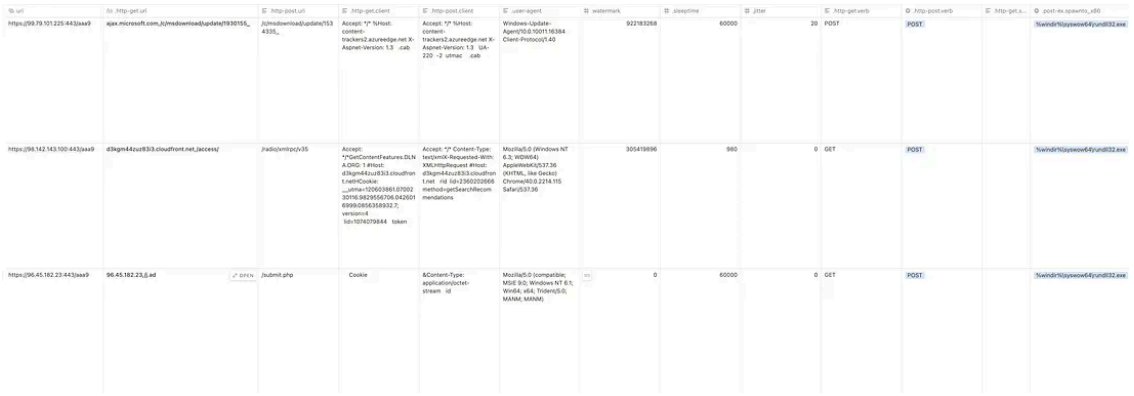
Table of parsed Cobalt Strike targets

Beacons table [sample]

The Beacon configuration table has the `uri` feature as a unique sorting key, when the rest of the features are the actual parsed beacon configurations.

Here is an example of the table with parsed beacon configurations:

Press enter or click to view image in full size



The full version of tables you can find here:

- Spreadsheet with 1709 Cobalt Strike servers
- Spreadsheet with 1473 Beacon configurations

Data analysis

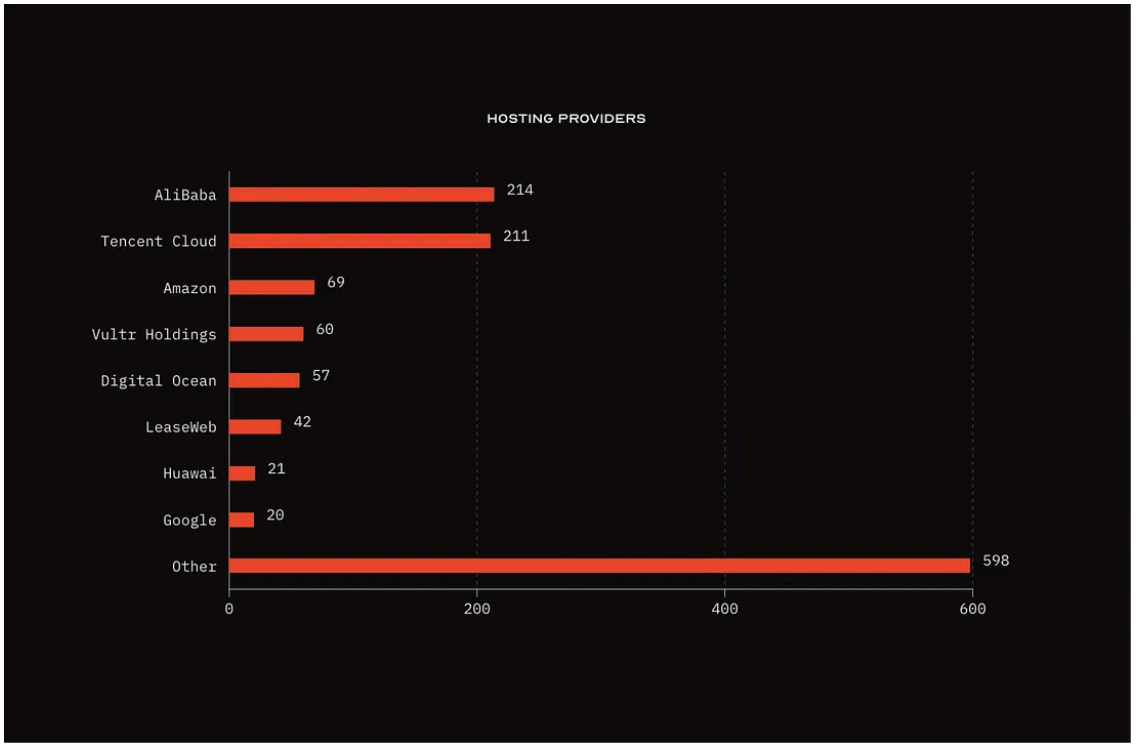
We are using collected data to map attackers infrastructure and understand how the attackers operate Cobalt Strike.

We know that threat intelligence groups are tracking specific ransomware groups with the help of Watermarks, For example:

- Sodinokibi (Watermark 452436291)
- APT 27 (Watermark 305419896).

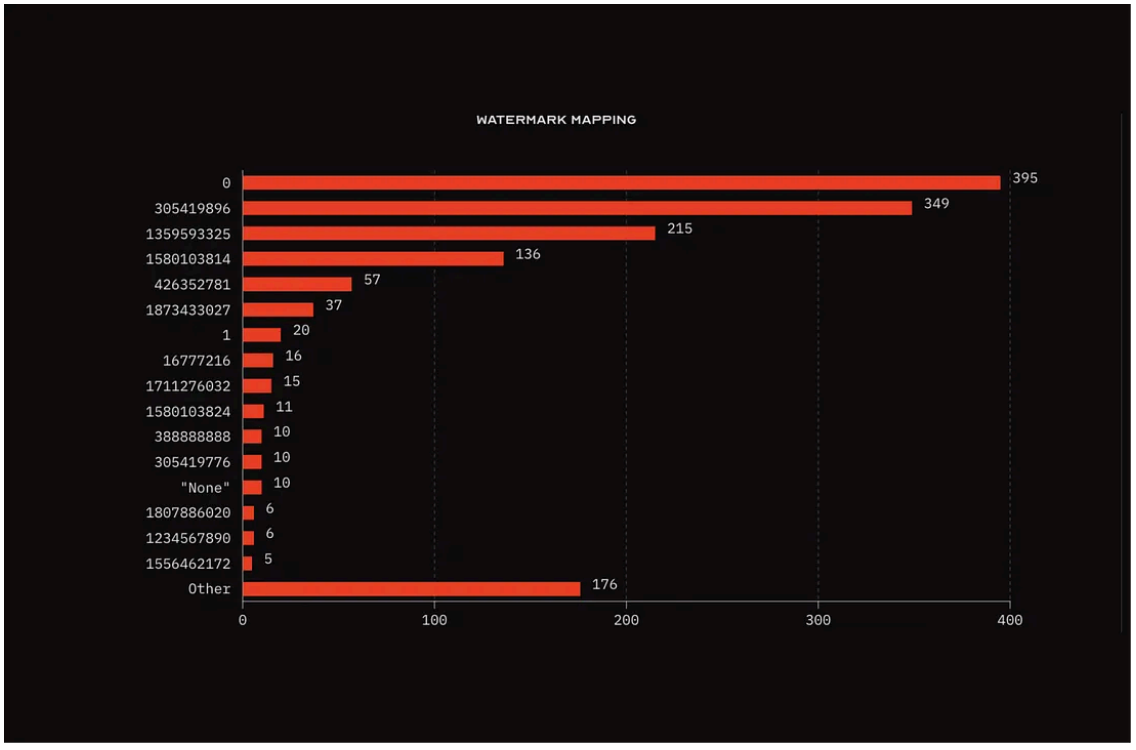
Based on the beacon's `spawnto` locations the blue teams can develop detection controls.

Press enter or click to view image in full size



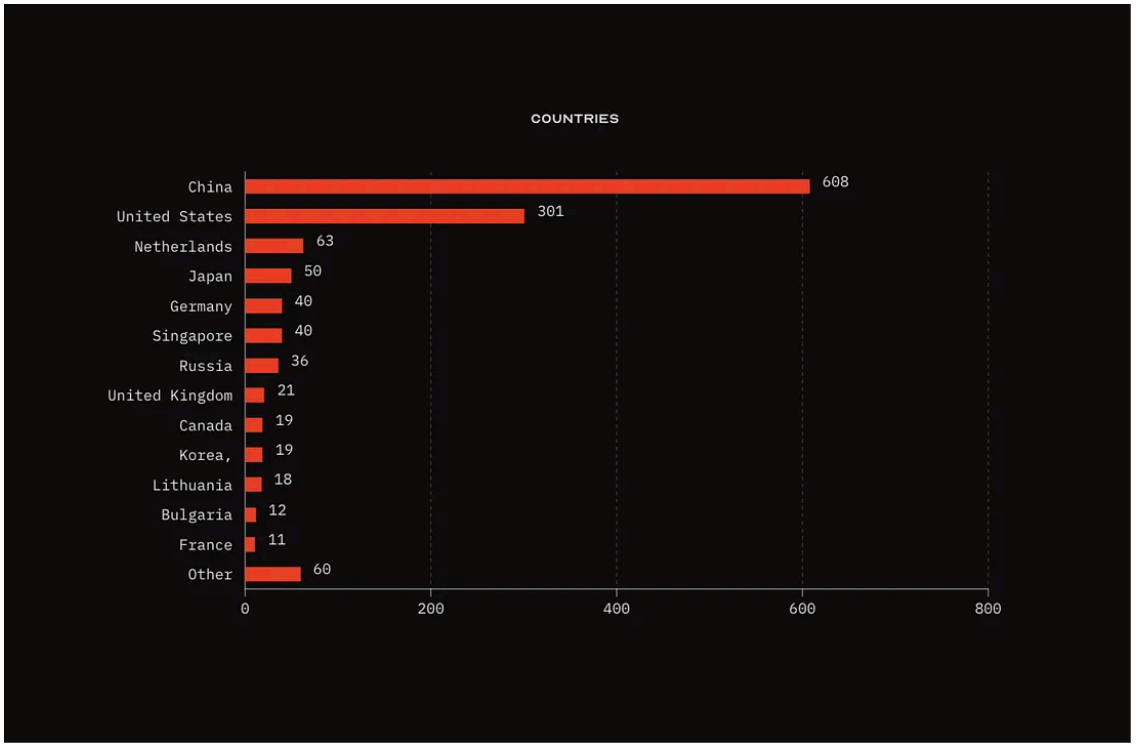
Location of servers (IP) (Hosting provides)

Press enter or click to view image in full size



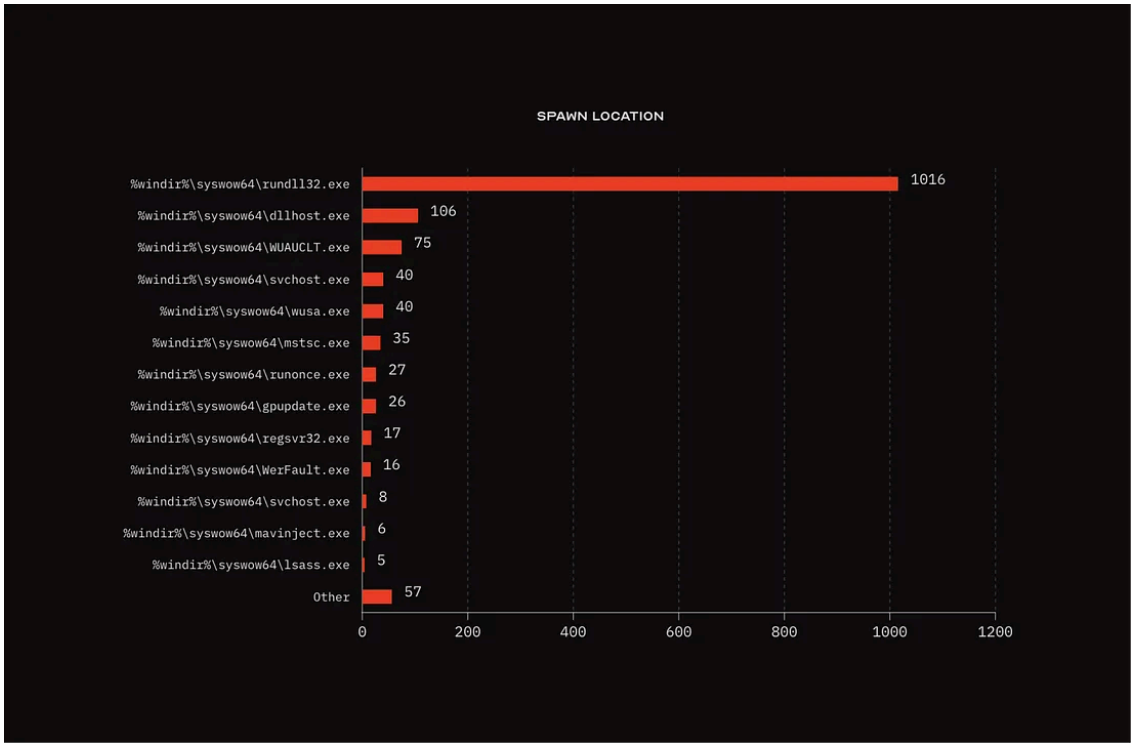
Watermarks

Press enter or click to view image in full size



Countries

Press enter or click to view image in full size



Spawn location

Press enter or click to view image in full size

Dork	Dorked results	Identified Cobalt Strike
hash:-2007783223 port:"50050"	1356	1014
ssl.jarm:07d14d16d21d21d00007d14d07d21d3fe87b802002478c27f1c0da514dbf80	3867	2
ssl.jarm:07d14d16d21d21d00042d41d00041d47e4e0ae17960b2a5b4fd6107fbb09 OPEN	1535	75
ssl.jarm:07d14d16d21d21d07c07d14d07d21d9b2f5869a6985368a9dec764186a9175	31887	114
ssl.jarm:07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1	5319	252
ssl.jarm:07d14d16d21d21d07c42d41d00041d58c7162162b6a603d3d90a2b76865b53	480	97
ssl.jarm:05d13d20d21d20d05c05d13d05d20dd7fc4c7c6ef19b77a4ca0787979cdc13	9154	1
ssl.jarm:07d0bd0fd06d06d07c07d0bd07d06d9b2f5869a6985368a9dec764186a9175	132	2
ssl.jarm:07d13d15d21d21d07c07d13d07d21dd7fc4c7c6ef19b77a4ca0787979cdc13	5379	5
ssl.jarm:07d14d16d21d21d00042d41d00041de5fb3038104f457d92ba02e9311512c2	588	117
ssl.jarm:2ad2ad16d2ad2ad00042d42d00042ddb04deffa1705e2edc44cae1ed24a4da	114	4
ssl.jarm:2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53	6502	4
"HTTP/1.1 404 Not Found" "Content-Length: 0" "Content-Type: text/plain"	110552	775

Sample of the Dork database (not completed)

Source: https://medium.com/@shabarkin/pointer-hunting-cobalt-strike-globally-a334ac50619a