Building an Open Source IDS IPS service for Gateway Load Balancer

aws.amazon.com/blogs/networking-and-content-delivery/building-an-open-source-ids-ips-service-for-gateway-load-balancer/

The Gateway Load Balancer (GWLB) service launched with support from the partner network. These partners provide networking appliances that enable customers to perform varying levels of packet inspection on flows that pass through them, taking action as necessary and as defined within their configuration. For a list of partners that support GWLB, refer to the following page. Whilst for some customers, using a partner supplied instance is a preferred choice (perhaps due to existing licensing, expertise or a specific capability), there is a segment of customers that wishes to benefit from all the capabilities that GWLB as a framework provides, but does not have any of the aforementioned considerations. For these customers, embracing open-source capabilities can make sense.

This blog provides the steps to create an open-source IDS/IPS service running in Docker containers, using Amazon Elastic Container Service (ECS) and Amazon Linux 2 (AL2). This service provides stateless packet inspection and logging, whilst leveraging the simplicity, elasticity and scalability that GWLB enables.

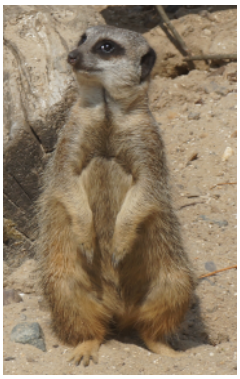
Figure 1, Meerkat

Meerkats (Suricata suricatta) are wonderful creatures. Amongst their accolades you'll discover that as well as being robust, tenacious and sporting high levels of intelligence, they are highly observant (care of their binocular vision), have developed advanced levels of vocalisation that they use to signal alarm and can physically dig their bodyweight in earth within seconds! Perhaps no surprise then that the Meerkat (Figure 1) is the namesake from which the popular open-source IDS/IPS service 'Suricata' takes its name. In a recent joint blog post the Open Information Security Foundation (OISF) and AWS discussed the importance of open-source security and how we have worked together to bring compatible Suricata rulesets to AWS Network Firewall. If you look at the source code for Suricata, you'll find that additional functions have been added to support the GENEVE protocol:

This can be seen here:

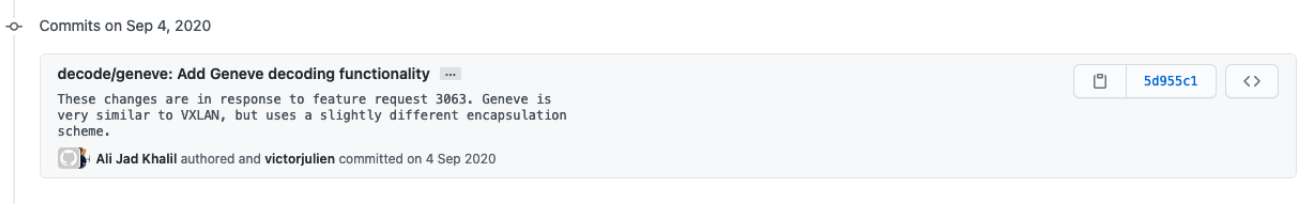

Figure 2, Code Snippet

The addition of these functions in the Suricata code, enable us to scale Suricata instances behind GWLB.

Deployment Overview

There are two key steps to the deployment:

The first step sets up a baseline Appliance VPC, Internet Gateway, NAT Gateways, S3 Buckets and SSM Parameters using a single CloudFormation template. This template will also set up a Code Pipeline which is made up of a code repository, build and deployment steps.

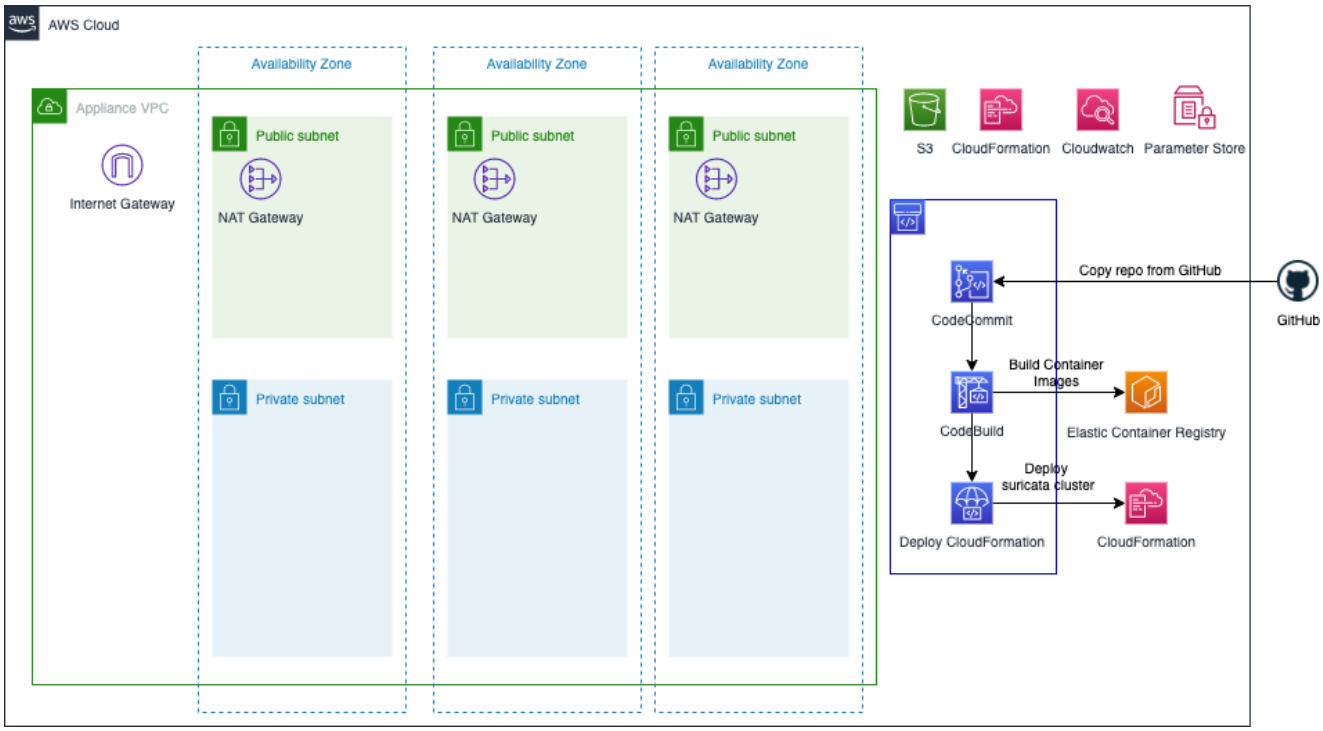

Figure 3, Baseline Appliance VPC

The second step deploys the inspection solution – you will release the pipeline that was built during the first step in order to do this. AWS Code Pipeline will deploy another CloudFormation template that builds and deploys a Suricata based packet inspection solution using GWLB and ECS. This is illustrated below:

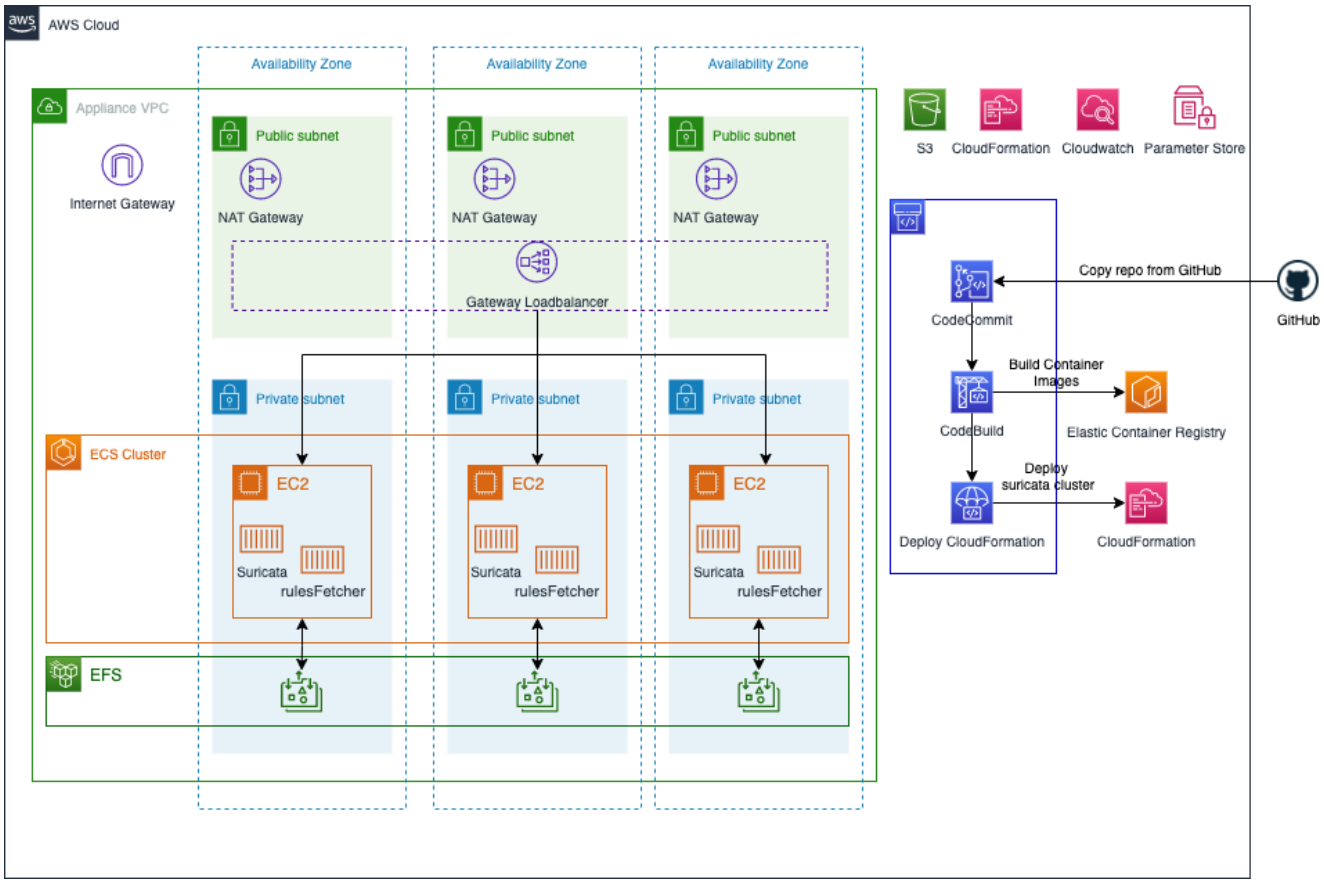

Figure 4, Inspection Solution

The pipeline creation and modification workflow is illustrated below:

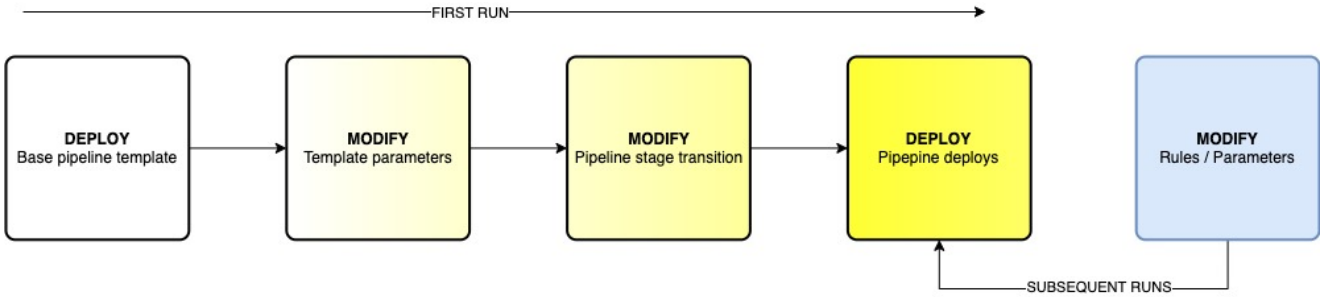

Figure 5, Pipeline Workflow

For more detailed instructions and descriptions of the parameters, you can reference the public readme.

You can choose to integrate this solution as appropriate for your requirements. Since the GWLB service uses PrivateLink endpoints for connectivity, it means that you can choose to deploy this solution in a centralised, decentralised pattern or a combination of both. For examples of GWLB implementation patterns, you can follow this guidance.

Solution Walkthrough

We wanted to take a moment to discuss the mechanics of the overall solution from the perspective of GWLB and how packets are inspected by the Suricata based instances.

We need to create a container image based on AL2 that holds the compiled Suricata code and rules along with any other parameters that are specific to the configuration. When the pipeline is triggered, CodeBuild pulls this public AL2 image, builds the Suricata and RulesFetcher containers and then stores them within a private ECR repository which is used by the ECS hosts. At the host level, the ECS worker nodes need to be configured to facilitate packet forwarding from GWLB to the Suricata container; for this there are a few elements of that warrant further discussion.

Packet path, Hooks and User space applications

In the Linux kernel, there are several hooks that allow actions on packets as they pass along the packet path, these are the netfilter hooks. Iptables provides a convenient interface into the netfilter framework and allows administrators to set rules for packets as they traverse these hooks. All packets that flow into, through, and out of Linux traverse these hooks.

The packet path inside Linux as it relates to the netfilter hooks is illustrated below:

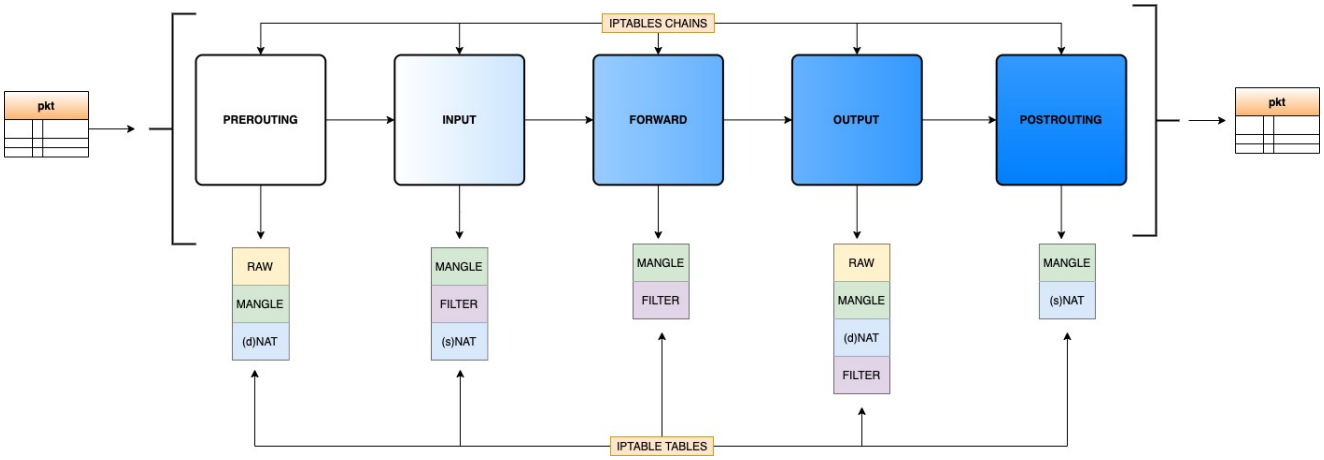

Figure 6, Packet Path

Iptables can be used to create rules for packet handling in any one of these chains and in any number of the tables that are processed. The specific chains that are traversed depend on the nature of the packet in the packet path. Since we are building Linux instances solely for the purpose of transparent packet inspection, then the traversal order of the chains will be as follows:

PREROUTING => FORWARD => POSTROUTING

GWLB uses GENEVE encapsulation and specific Type Length Values (TLV) in the GENEVE header to identify flows and maintain symmetry, it's important that we maintain these as the packets pass through the packet path. We also need to perform some source and

destination Network Address Translation (NAT) actions on the packets so that they are returned, back to the GWLB node that originally sent them to the inspection instance.

From a firewalling or filtering perspective, unless the packet is decapsulated or decoded, Iptables rules cannot take action on the passenger payload. However, since we know that Suricata understands the GENEVE protocol, we are able to route encapsulated packets to the Suricata engine and let it decode, inspect and take action on the passenger payload. Suricata is a user space application and so to invoke it in IPS mode (so that it sits in the packet path), we can use NFQUEUE. Essentially this tells Iptables to push packets that traverse the Forward chain to a queue number that Suricata is listening on. You can read more about NFQUEUE implementation in the Suricata documentation, here:

In summary of the above, the flow works like this:

1. Encapsulated packet arrives at the inspection instance from a GWLB node
2. Destination NAT is handled first by the NAT table in the PreRouting chain. This rewrites the destination of the packet to the GWLB node that delivered the packet
3. NFQUEUE is invoked next by the queue statement in the FILTER table within the Forward chain
4. The Suricata instance will receive the encapsulated payload and take actions based on the rules that have been created. Suricata is able to decode and read the passenger payload (Suricata puts the packet back in the packet path)
5. Source NAT is handled next by the NAT table in the PostRouting chain. This rewrites the source of the packet to the back-end instance that performed the packet filtration
6. The GENEVE encapsulated packet is then put back on the wire

No modifications are made to the original packet, but its contents are inspected.

The packet path modification for inline packet filtering is illustrated below:

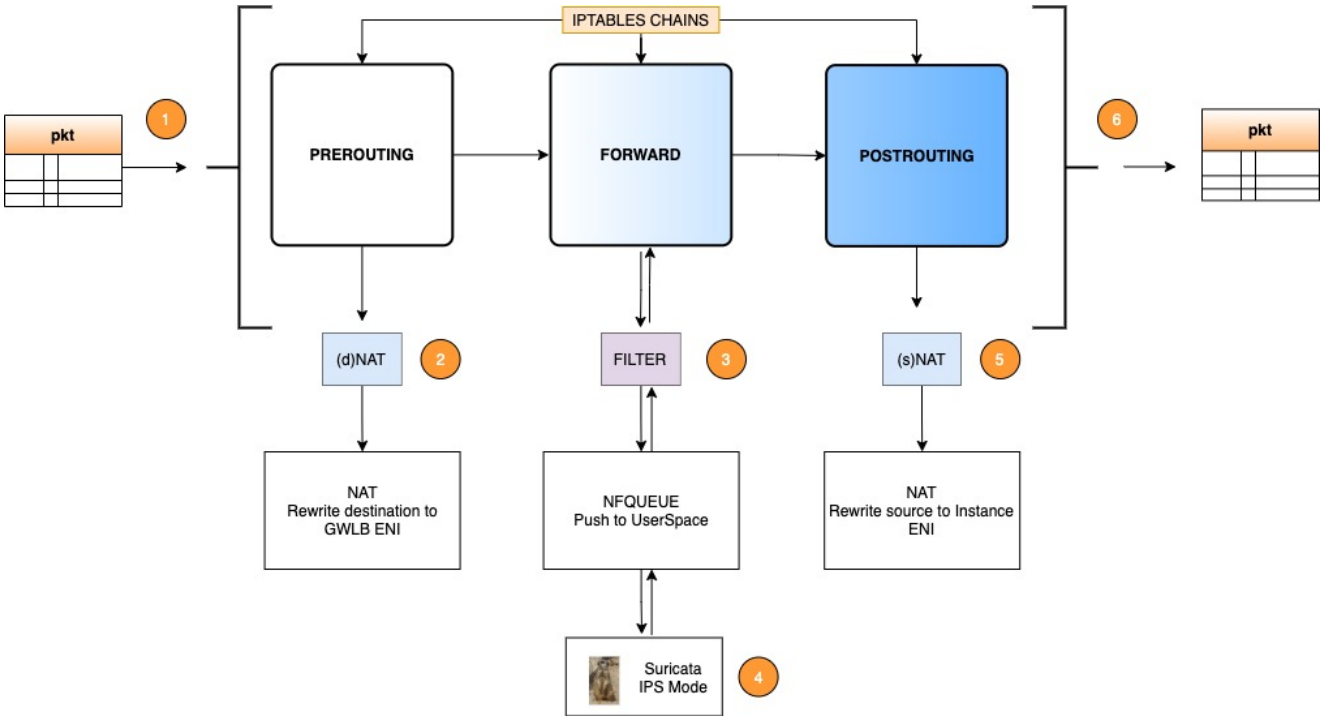

Figure 7, Inline Packet Filtering

Rule Creation and Monitoring

Container Static Rules

Static rules are built into the container image as part of the image build process by AWS Code Pipeline. You should use static rules when you want to keep your rules versioned together with the Suricata config and Suricata version or for rules that shall always be enforced and should not be removed. Rules in the static.rules file are NOT applied or updated on-the-fly. You need to rebuild and redeploy the Suricata container with the updated rules.

Rule Example 1 – Logging outbound DNS queries

With our solution deployed, let's make a change to the static.rules file. I'd like know about any DNS requests that are going beyond my VPC boundaries (external DNS requests) These rules are baked into the container image. I can make this change directly in the Code Commit console or I could subsequently clone the Code Commit repo and make the change in there and then commit the changes.

```
alert ip [%cidr%] any -> ! [%cidr%] 53 (msg:"external dns traffic
logged";sid:10000;rev:1;)
```

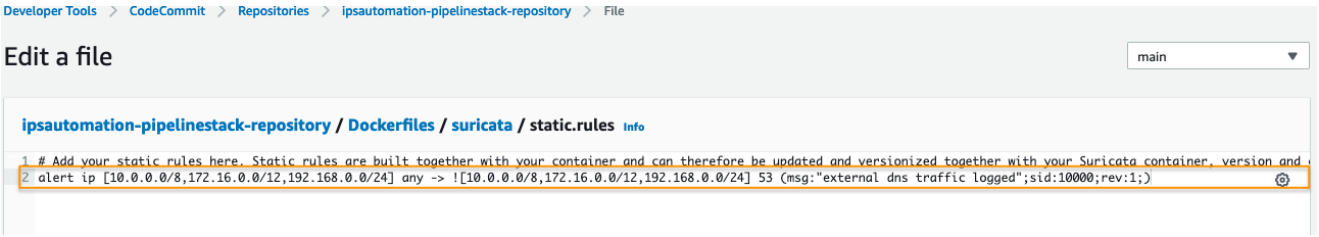

Figure 8, Static Rule

NOTE: Modifications to the container code will begin an ECS container replacement operation. During this time, clients may lose connectivity and will need to re-establish any connections. Modifications to dynamic rules are performed without interruption.

After a short while, pipeline should have deployed successfully – with your alerting rules baked in. We shall look at the logs that are generated little later.

Local Dynamic Rules

The Dynamic Rules should be used when you want to deploy and apply rules on-the-fly and don't want, or need to keep your rules versioned together with the Suricata config and Suricata version.
These rules are applied and updated without the need to rebuild or redeploy the Suricata container.

Dynamic rules are handled slightly differently to static rules. Whilst to Suricata they are just another rule file that is specified within its configuration file – this solution uses the RulesFetcher container to pull the dynamic rules from S3 and then run the Suricata-update daemon to load them into the engine. Dynamic rules are not tracked with the container image.

Modification of dynamic rules follows a similar process to static rules. Simply modify the dynamic.rules file in the Code Commit repo and commit the changes.

We have specified three rule entries here:

Rule Example 2 – This rule drops all ICMP traffic between two VPCs that are connected by a Transit Gateway (TGW)

> drop icmp [%cidr%] any <> [%cidr%] any (msg:"icmp traffic blocked";sid:10001;rev:1;)

Rule Example 3 – This rule drops all external connections to an Application Load Balancer, where the source IP address is identified as originating from within Great Britain (GB) *

> drop ip ![%cidr%] any -> [%cidr%] 80 (msg:"geo-ip rule GB";geoip:src,GB;sid:10002;rev:1;)

Rule Example 4 – This rule blocks access to a website based on the TLS information inside the certificate handshake

> drop tls [%cidr%] any -> any any (msg:"block access to social media websites";tls.sni; content:"facebook.com"; nocase; pcre:"/facebook.com$/" ;sid:10003;rev:1;)

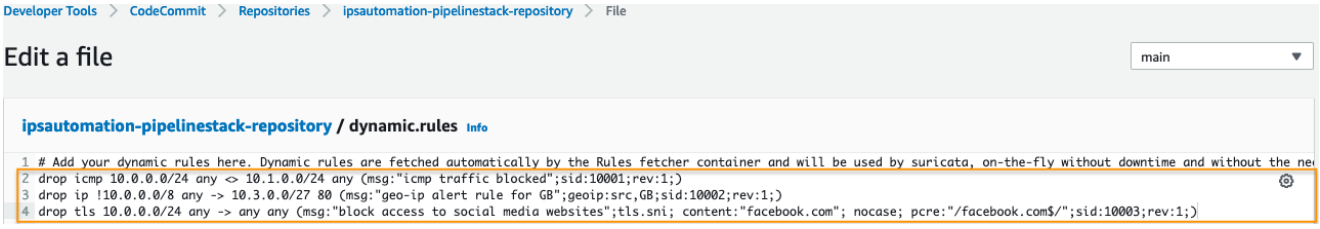

Figure 9, Dynamic Rules

**GeoIP functionality requires that prior to deployment, you must register with MaxMind and provide an API key during initial pipeline setup (Step 2). For more details on the parameter file that you need to modify, check the public readme documentation.*

External Dynamic Rules

This solution provides the ability to load third-party rules-sets into the configuration. For example, the rulesets provided in The Open Information Security Foundation rulesets list such as the popular: et/open ruleset. These rule-sets are managed by external parties and can be loaded automatically into the configuration. To reference an external rule-set, simply modify the CloudFormation stack template. The rules will be dynamically loaded.

Check the following *link* for further details on additional rulesets that can be loaded into the Suricata engine.

Logging and Validating the solution

You'll remember that we specified a static rule and three dynamic rules. Let's generate some traffic, monitor the behaviour, and trace the log entries.

Rule Example 1 – Logging outbound DNS queries

Let's perform a DNS query against an external DNS resolver, from one of our internal clients:

```
[ssm-user@ip-10-0-0-9 bin]$ nslookup
> server 8.8.8.8
Default server: 8.8.8.8
Address: 8.8.8.8#53
> www.bbc.com
Server:         8.8.8.8
Address:        8.8.8.8#53

Non-authoritative answer:
www.bbc.com     canonical name = www.bbc.com.pri.bbc.com.
www.bbc.com.pri.bbc.com canonical name = uk.www.bbc.com.pri.bbc.com.
Name:   uk.www.bbc.com.pri.bbc.com
Address: 212.58.233.250
Name:   uk.www.bbc.com.pri.bbc.com
Address: 212.58.237.250

```

Figure 10, Example DNS Query

Let's have a look at the Fast.log (this is where the alert will be generated) and also find the Flow statement in the Eve.log. If we search by the signature ID, we see entries in the Fast.Log file:

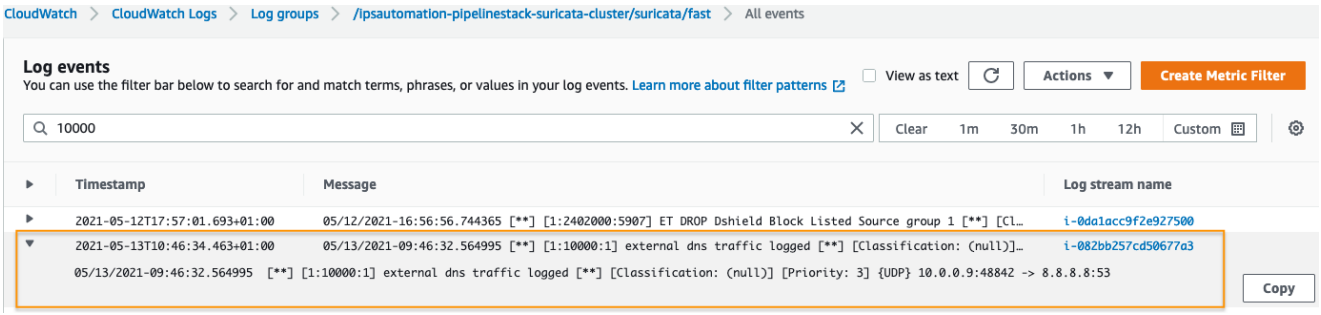

Figure 11, Fast.Log Analysis

Here is the Flow entry; notice the detail that it provides. You can see the tunneling that is happening from GWLB.

Figure 12, DNS Eve.Log

ICMP, GeoIP and TLS

In our dynamic rule file, we created rules to block ICMP packets, restrict access to a public Application Load Balancer (if the originating IP address was identified as being located somewhere in Great Britain) and prevent access to social media websites. The Signature IDs for those rules were “10001”, “10002” and “10003”, respectively. Let’s go and find the flow entries for those.

We can use JSON based matching to accomplish this in CloudWatch Logs. A query such as this one should bring back the matching entries:

{ $.alert.signature_id = %signatureid% }

Rule Example 2

Here’s the event that was generated when a client attempted to ping another – via a TGW connection. We can see that it was blocked – as expected.

Figure 13, ICMP Eve.Log

Rule Example 3

Here is another event that was generated when an entity within Great Britain attempted to connect to an ALB that is being protected by a GeoIP filtering rule:

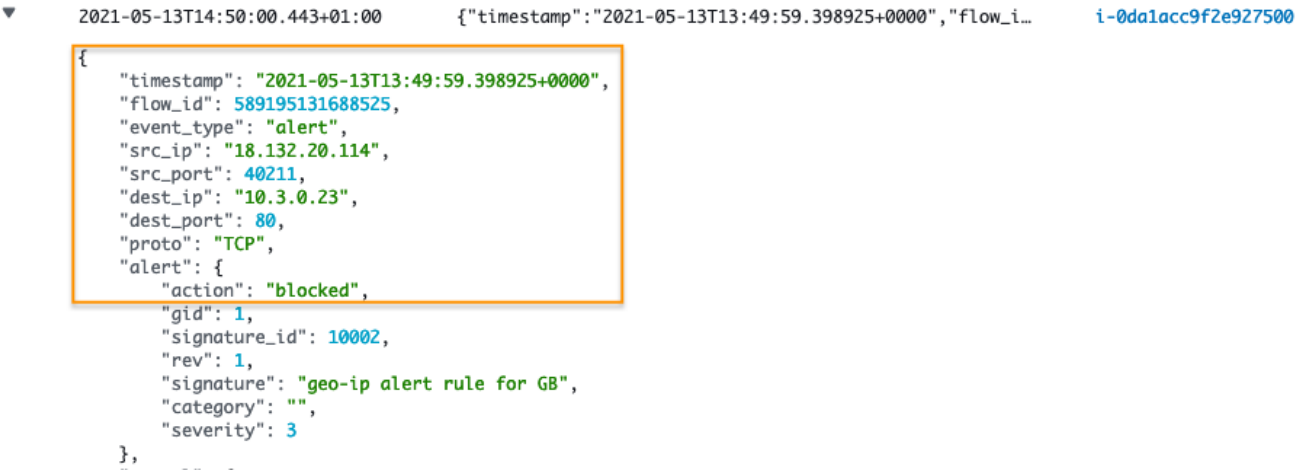

Figure 14, GeoIP Eve.Log

To test this yourself, simply spin up an EC2 instance in the blocked region and attempt a connection to the public facing IP address of the load balancer.

Rule Example 4

We also blocked access to social media sites. Let's search for the signature id to discover any activity for this rule.

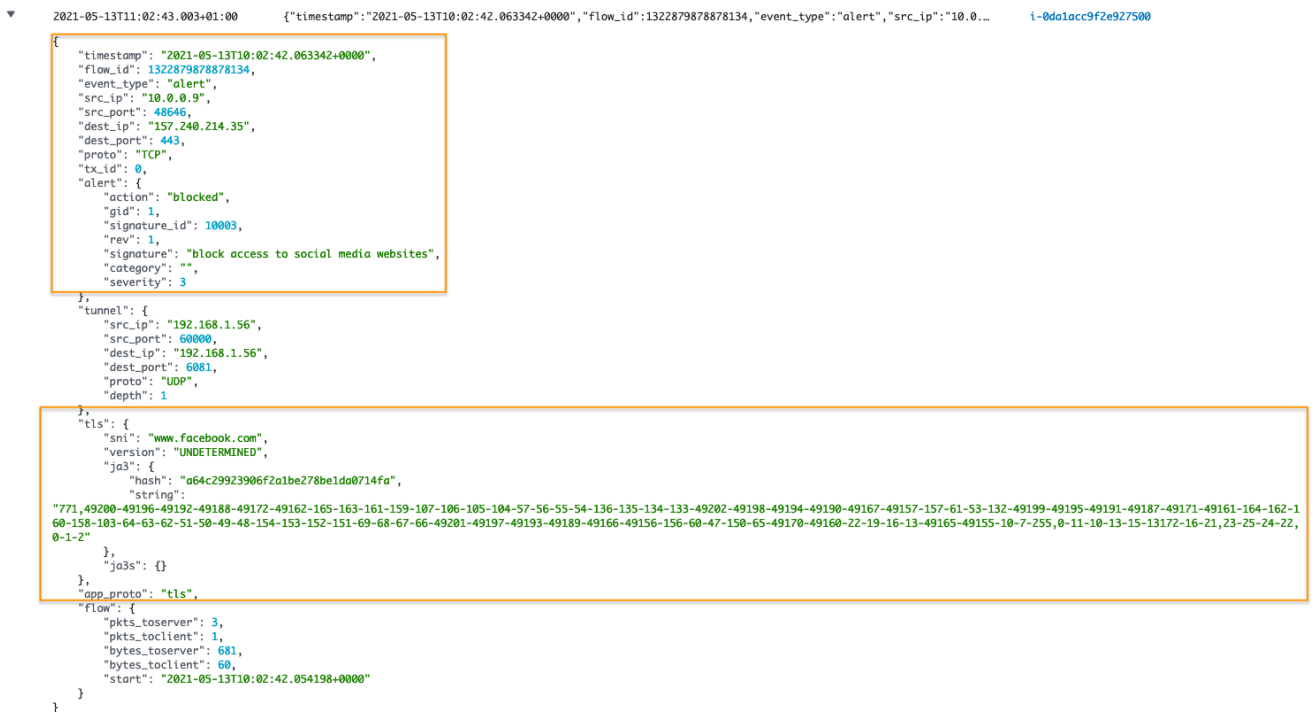

Figure 15, TLS Eve.Log

We can see the traffic was blocked, the signature description, and some detail regarding the fingerprinting of the TLS communications.

To test this yourself, a simple curl command can be used to generate some traffic. This will grab the headers only and report with enhanced detail. You should see that the TLS handshake is broken when you do this.

➜ ~ curl https://facebook.com -i -v

These rules are just examples so you can adjust the configuration to suit your deployment. Since you've built your own Suricata containers, there is more that you can do. This solution was compiled with the **LUA** scripting module. With this capability you can write more complex rules that provide advanced matching against malware. Additionally, you could use the packet capture capability so that you can debug the traffic that is flowing through your inspection instances. See the public readme for more details on this and how to enable it.

Clean-up

Clean-up is straightforward, you can delete the CloudFormation stack that was created by the pipeline, and then delete the stack that defined the pipeline itself. You'll be left with a couple of S3 buckets and ECR repositories that you can either choose to keep or delete manually.

Conclusion

So, there you have it. You built a GitOps driven IDS/IPS service using open-source code, on top of Gateway Load Balancer. You created rules that log particular types of traffic (DNS) as a baseline, added protections for your network traffic in blocking known protocols (ICMP) and restricted access to an Application Load Balancer based on Geographic location metadata. Finally, you prevented access to a social media website using data that is negotiated as part of a TLS handshake.

We also discovered a little more about just how impressive Meerkats are!

Author Bios

Adam Palmer

Adam Palmer is a Senior Specialist Network Solutions Architect at AWS. Prior to joining AWS, Adam worked as an Architect in the Financial Service sector; specializing in Networking, VMware, Microsoft platform and End-User Compute solutions. In his spare time, he can be found climbing mountain faces, wherever the weather is good!

Jesper Eneberg

Jesper Eneberg is a Senior Solutions Architect based in Sweden and is working with global Telecommunications customers at AWS. Jesper is an open source advocate and have a background in Operations and Infrastructure. Prior to joining AWS, he implemented DevOps and worked with DevOps organizations in both the Public and private sector