AppSentinels.ai
Application Security Re-invented

AppSentinels API Security Platform

Sniffer Mode Deployment

Revision	Date Modified	Author	Comments
1.0	08-Jan-24		Initial Draft
1.1	01-Aug-24	Sachin	Merged documentation into single sniffer-based document
1.2	06-Aug-24	Arun	Merged AWS deployment details in this document
1.3	23-08-24	Sachin	Support for daemonset based sniffer
1.4	18-Oct-24	Sachin	Aarch/Arm image details and instance config
1.5	28-Oct-24	Sachin	Secure logging documentation
1.6	19-Feb-24	Sachin	Deployment with ingress and app sidecar
1.7	01-Jun-25	Sachin	Support for URI and hostname filtering
1.8	18-Jul-25	Sachin	Updated non-container support
1.9	3-Nov-25	Sagar	Updated URI regex and sampling config

Introduction

AppSentinels Sniffer Sensor deployment is Out-Of-Band deployment mode, without any modification required in the application configuration and application environment.

AppSentinels Sniffer Sensor (called sensor henceforth) has to be deployed on the Application Server as a docker instance and should be able to see the unencrypted traffic of incoming traffic and outgoing traffic to upstream services.

The Sniffer Sensor captures the HTTP traffic on the specified interface(s) and forwards the API traffic to the AppSentinels Edge Controller (called controller henceforth).

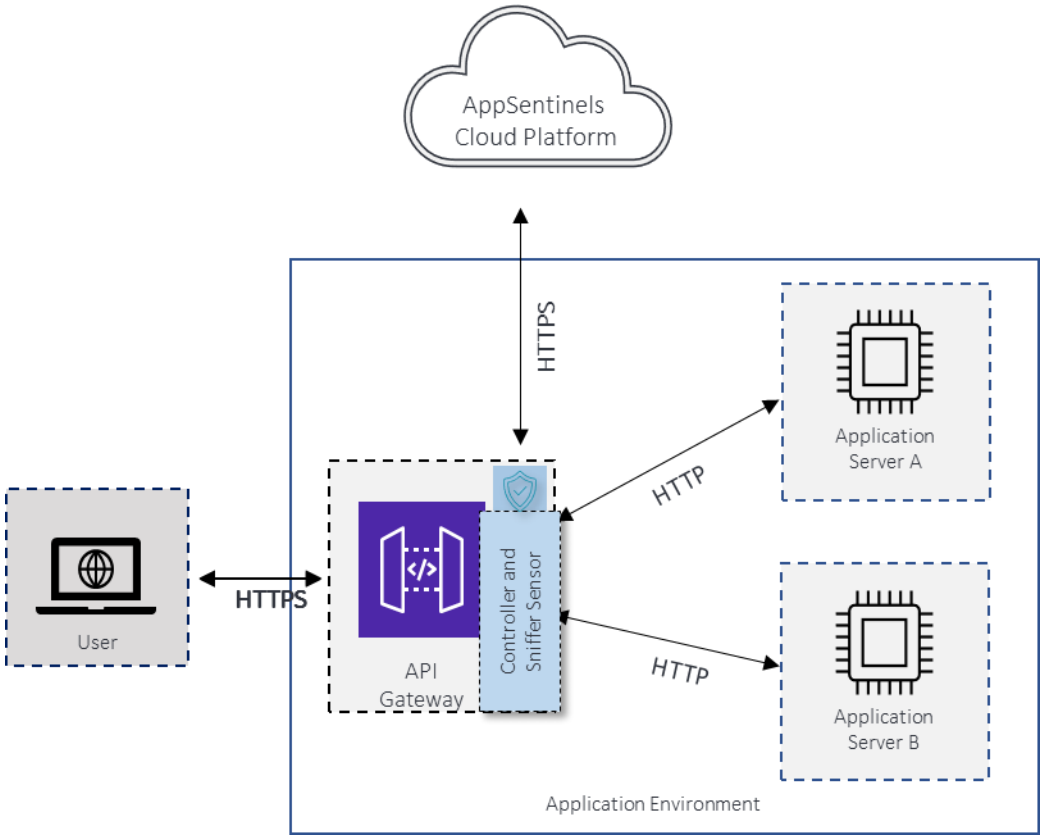
To take preventive action against Security Events and Threat Actors requires integration of Firewall or API Gateway with AppSentinels API Security Platform since this deployment is in Out-Of-Band mode.

Deployment Options

Integrated Controller and Sniffer Sensor Deployment

The Controller and Sensor runs as single docker instance on the application server. This deployment option is preferred, when one instance of Sensor and Controller is required.

Topology



Deployment Details

AppSentinels Integrated deployment captures the HTTP Traffic, processes the API traffic and forwards the meta data to the Cloud Platform for AI/ML processing. The Edge Controller detects the Security Events and takes preventive action using Firewall/API GW.

Pre-Requisites: -

- Linux OS with docker and docker-compose installed
- Minimum 2 vCPU (x86_64) & 4 GB RAM
- 50 GB disk space
- Network connectivity to AppSentinels API Security Platform

```yaml
version: "3.3"
services:
  onprem_controller:
    container_name: onprem-controller
    restart: on-failure:5
    image: appsentinels/ng-controller:latest
    hostname: appsentinels-opc-sniffer
    environment:
      - SAAS_API_KEY_VALUE=<your api key> # Add API key you got from Appsentinels support
      - APPLICATION_DOMAIN=<domain>       # Add the name of application or application group
      - ENVIRONMENT=<environment>         # Add environment, production or UAT or staging
      - SAAS_SERVER_NAME=in-cloud.appsentinels.ai
      - TAP_INTERFACE=<interface to sniff on>   #Add TAP device interface to sniff API traffic
      - TAP_FILTER=<bpf filter eg: port-3000-or-8090-and-tcp> #Add TAP filter details, refer documentation
    network_mode: "host"
    logging:
      driver: local
      options:
        max-size: 10m
    volumes:
      - /var/crash:/var/crash
      - /var/log/appsentinels:/var/log/appsentinels
```

Deploy AppSentinels Integrated Edge Controller and Sniffer Sensor as docker container by updating the above YAML configuration with values specific to application environment.

https://sample-config.appsentinels.ai/appsentinels-deployment/sample-docker-compose/docker-compose-tapmode.yaml

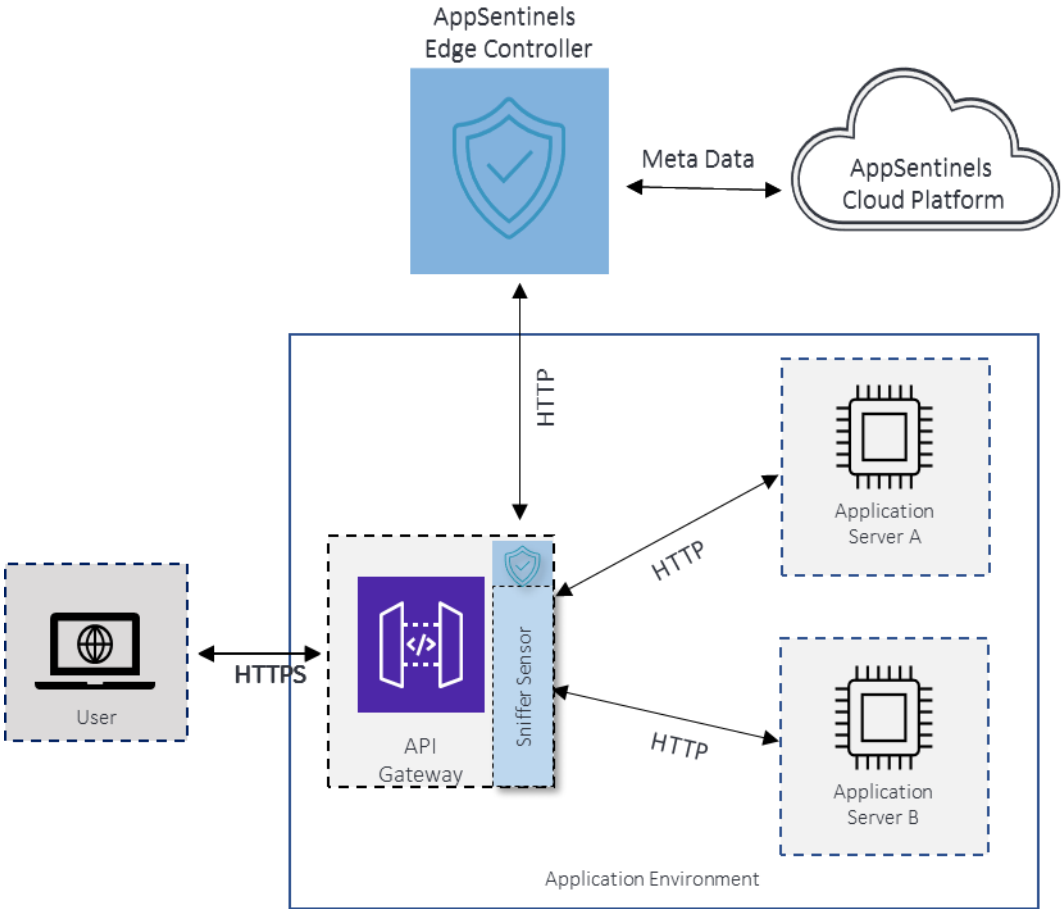
- ***Please note that when bringing up multiple controllers, ensure hostname isn't repeated.***
- ***Integrated controller and sniffer is supported only on x86-64 platforms***

For the TAP filter, set the filter as per Berkley Packet Filter.
For example, to only sniff API traffic for the service listening on port 8080, set the filter to port-8080-and-tcp.
To sniff on port 8081 as well, set the filter to port-8080-or-port-8081
To sniff on a range of ports say (8000 to 8080), set the filter to tcp-portrange-8000*8080

Controller and Sensor Deployed Separately

The Controller and Sensor are deployed separate docker instances. This deployment option is preferred, when multiple Sniffer Sensors are required to monitor different application services with one or more Controller.

Topology



Deploy Edge Controller

AppSentinels Edge Controller processes the API traffic data received from the Sensor and forwards the meta data to the Cloud Platform for AI/ML processing. The Edge Controller detects the Security Events and takes preventive action using Firewall/API GW.

Pre-Requisites: -

- Linux OS with docker and docker-compose installed
- Minimum 2 vCPU (x86_64) & 4 GB RAM
- 50 GB disk space
- Network connectivity to AppSentinels API Security Platform

While this document covers basic deployment using docker-compose, please refer to Edge Controller Deployment guide for other methods of deployment.

```yaml
version: "3.3"
services:
  appsentinels-edge-controller:
    container_name: appsentinels-edge-controller
    restart: on-failure:5
    image: appsentinels/ng-controller:latest
    hostname: appsentinels-edge-controller
    environment:
      - APPLICATION_DOMAIN=<your-app-domain>       # Add the name of application or application group
      - ENVIRONMENT=<environment>                  # Add environment, production or UAT or staging
      - SAAS_SERVER_NAME=in-cloud.appsentinels.ai
      - SAAS_API_KEY_VALUE=<your provided api key> # Add the name of application or application group
    ports:
      - "9004:9004"
      - "127.0.0.1:9101:9101"
    logging:
      driver: local
      options:
        max-size: 10m
    volumes:
      - /var/crash/:/var/crash
```

Deploy AppSentinels Edge Controller as docker container by updating the above YAML configuration with values specific to application environment.
https://sample-config.appsentinels.ai/appsentinels-deployment/k8/controller/edge-controller.yaml

- ***Please note that when bringing up multiple controllers, ensure hostname isn't repeated***
- ***Integrated controller and sniffer is supported only on x86-64 platforms***

Deploy Sniffer Sensor

```yaml
version: "3.3"
services:
  sniffer:
    container_name: sniffer-sensor
    restart: on-failure:5
    image: appsentinels/ng-controller:latest
    hostname: appsentinels-sniffer-sensor
    environment:
      - REMOTE_CONTROLLER_SERVER_NAME=<appsentinels-controller> # Add dns mapped hostname or IP of edge controller
      - REMOTE_CONTROLLER_SERVER_PORT=9004 # Add edge controller listening port to talk from this sensor
      - ENVIRONMENT=<environment> # Add environment, production or UAT or staging
      - SNIFFER_SENSOR_INSTANCE=<instance name> # Provide user identifiable instance name for visibility, eg: sniffer-sensor-dev-app-1
      - TAP_INTERFACE=<interface to sniff on> # Add TAP device interface to sniff api traffic
      - TAP_FILTER=<Add filter details>       # bpf filter with hyphens for spaces, eg: port-8080-or-3000-and-tcp> based on app
      - RELAY_PROTOCOL=http
    network_mode: "host"
    logging:
      driver: local
      options:
        max-size: 10m
    volumes:
      - /var/crash:/var/crash
      - /var/log/appsentinels:/var/log/appsentinels
```

Deploy AppSentinels Sniffer Sensor on the docker container by updating the above YAML configuration with values specific to application environment.
https://sample-config.appsentinels.ai/appsentinels-deployment/sample-docker-compose/docker-compose-sniffer-sensor-medium.yaml
https://sample-config.appsentinels.ai/appsentinels-deployment/sample-docker-compose/docker-compose-sniffer-sensor-small.yaml
For the TAP filter, set the filter as per Berkley Packet Filter.

For example, to only sniff API traffic for the service listening on port 8080, set the filter to port-8080-and-tcp.
To sniff on port 8081 as well, set the filter to port-8080-or-port-8081
To sniff on a range of ports say (8000 to 8080), set the filter to tcp-portrange-8000*8080

Secure Logging

Sniffer has the capability to perform logging onto edge controller via HTTPS. Setting the below environments will ensure this,

Environment	Default	Description
RELAY_PROTOCOL	http	Logging protocol onto controller. Alternate option is https
HTTPS_INSECURE_SKIP_VERIFY	false	Ignore controller cert validation in case RELAY_PROTOCOL is https

Resource Requirements

Sniffer sensor can be tuned for throughput via 3 parameters, resources of CPU/memory and via environment variable of TAP_PROFILE.

CPU	Memory	TAP_PROFILE	Throughput (API request per second)
0.5	256M	low	250 req/sec
1	512M	medium	500 req/sec
2	2G	default	1000 req/sec

While TAP_PROFILE isnt a mandatory parameter, it helps to fine-tune the buffer resources inside Sniffer.

Please note that this is only relevant to sniffer sensor only and doesn't apply to integrated controller and sniffer based deployments.

Architecture Support

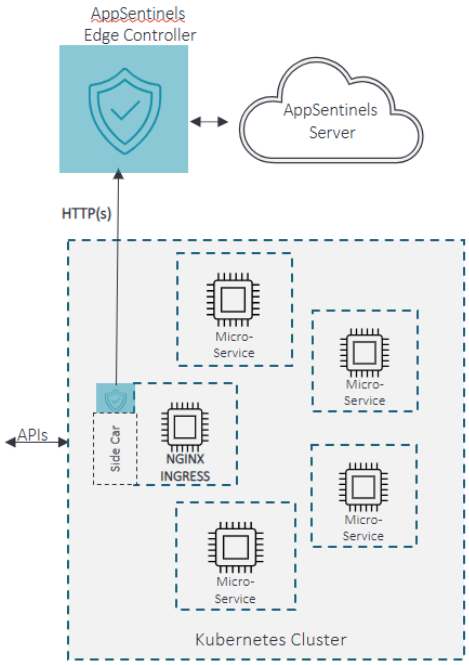
Sniffer sensor is supported on x84-64 and aarch64/arm architectures. The container images used vary based on this.

Architecture	Image
x86-64	appsentinels/ng-controller:latest
aarch64/arm	appsentinels/ng-controller:aarch-latest

Rest of the document assumes the architecture being used is x86-64. Please change the specs according if otherwise.

Sensor deployment as Ingress sidecar in Kubernetes

Provided API traffic is clear text, sniffer sensor can be deployed as a sidecar to the existing ingress controller. This provides a single point of integration for a cluster.



Reference sidecar spec:
https://sample-config.appsentinels.ai/appsentinels-deployment/sniffer/sidecar/deployment-nginx-tap-sample.yaml

Procedure:

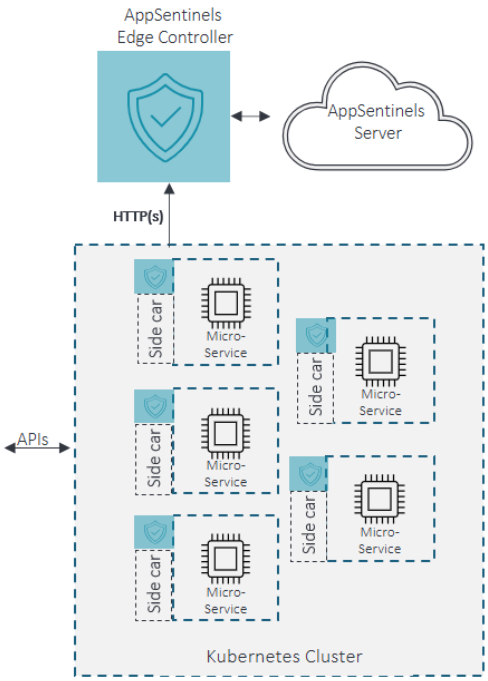
1. Patch the ingress controller spec as in the above reference spec (look for start and end of appsentinels block)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
spec:
  replicas: 1
  .....
    spec:
      serviceAccountName: nginx-ingress
      containers:
        # ######################################################
        # Start of properties of appsentinels sniffer sidecar container
        - image: appsentinels/ng-controller:latest
          imagePullPolicy: Always
          name: appsentinels-controller
          env:
          - name: APPLICATION_INFO
            value: <your application>
          - name: ENVIRONMENT
            value: <your environment>
          - name: SNIFFER_SENSOR_INSTANCE
            value: <sniffer sensor instance, eg: service-1>
          - name: REMOTE_CONTROLLER_SERVER_NAME
            value: <dns mapped hostname or IP of controller, eg: remote-controller>
          - name: REMOTE_CONTROLLER_SERVER_PORT
            value: "9004"
```

```
          - name: TAP_INTERFACE
            value: default
          - name: TAP_FILTER
             value: tcp-and-port-not-22-and-port-not-443-and-port-not-9004
          - name: RELAY_PROTOCOL
            value: http
          resources:
            requests:
              cpu: 0.5
              memory: 512Mi
            limits:
              cpu: 0.5
              memory: 512Mi
        # end of properties of appsentinels sniffer sidecar container
        # ######################################################
      - image: nginx/nginx-ingress:2.1.0
        imagePullPolicy: IfNotPresent
..... <existing spec of ingress>
```

2. Define relevant resource limits for the AppSentinels sniffer container (refer here)
3. Configure the container's environmental variables like,
    a. REMOTE_CONTROLLER_SERVER_NAME
    b. REMOTE_CONTROLLER_SERVER_PORT
    c. TAP_INTERFACE =default
    d. TAP_FILTER=tcp-and-port-not-22-and-port-not-443-and-port-not-9004 # Modify based on application service listening port
    e. SNIFFER_SENSOR_INSTANCE=<instance name> # Provide user identifiable instance name for visibility, eg: sniffer-sensor-dev-app-1
4. Insert the sniffer into the ingress pod
   > kubectl –f deployment-nginx-tap-sample.yaml

Sensor deployment as Pod sidecar in Kubernetes



Reference sidecar spec:
https://sample-config.appsentinels.ai/appsentinels-deployment/sniffer/sidecar/deployment-sniffer-sensor-tap-sample.yaml

Procedure:

1. Patch the sidecar spec as in the above reference spec (look for start and end of appsentinels block)

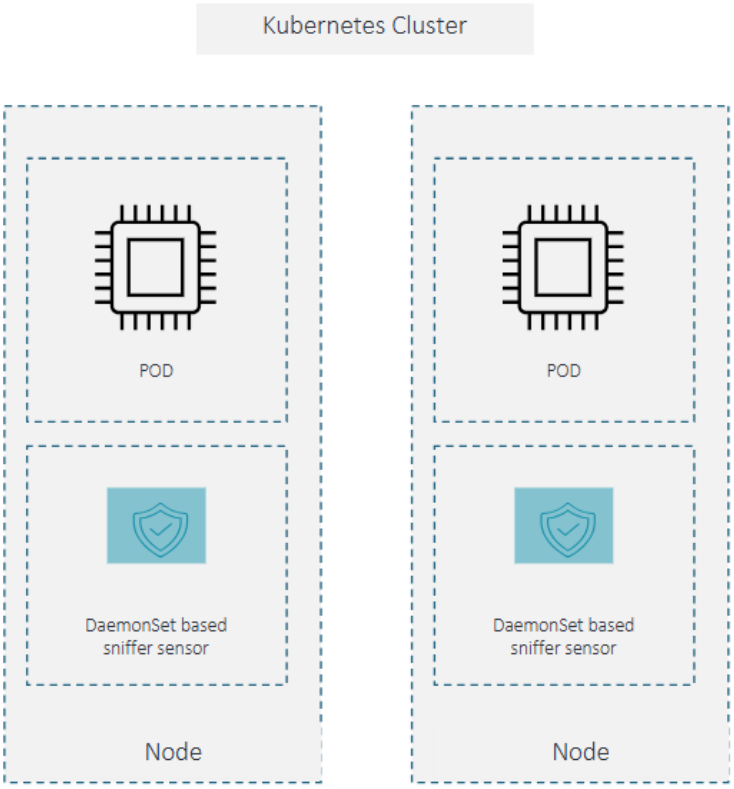
```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: app1
  namespace: http-service
  labels:
    app: app1
spec:
    .....
    spec:
      containers:
        # ######################################################
        # Start of properties of appsentinels sniffer sidecar container
        - image: appsentinels/ng-controller:latest
          imagePullPolicy: Always
          name: appsentinels-controller
          env:
          - name: APPLICATION_INFO
            value: <your application>
          - name: ENVIRONMENT
            value: <your environment>
          - name: SNIFFER_SENSOR_INSTANCE
            value: <sniffer sensor instance, eg: service-1>
          - name: REMOTE_CONTROLLER_SERVER_NAME
            value: <dns mapped hostname or IP of controller, eg: remote-controller>
          - name: REMOTE_CONTROLLER_SERVER_PORT
            value: "9004"
          - name: TAP_INTERFACE
            value: default
          - name: TAP_FILTER
            value: tcp-and-port-not-22-and-port-not-443-and-port-not-9004
          - name: RELAY_PROTOCOL
            value: http
          resources:
            requests:
              cpu: 0.5
              memory: 512Mi
            limits:
              cpu: 0.5
              memory: 512Mi
        # end of properties of appsentinels sniffer sidecar container
        # ######################################################
        - name: http-service
          image: appsentinels/http-server:latest
          imagePullPolicy: IfNotPresent
          ports:
            - name: getextractport
              containerPort: 9000
```

```
        env:
         - name: HTTP_SERVICE_PORT
           value: "9000"
```

2. Define relevant resource limits for the AppSentinels sniffer container (refer here)
3. Configure the container's environmental variables like,
    a. REMOTE_CONTROLLER_SERVER_NAME
    b. REMOTE_CONTROLLER_SERVER_PORT
    c. TAP_INTERFACE =default
    d. TAP_FILTER=tcp-and-port-not-22-and-port-not-443-and-port-not-9004 # Modify based on application service listening port
    e. SNIFFER_SENSOR_INSTANCE=<instance name> # Provide user identifiable instance name for visibility, eg: sniffer-sensor-dev-app-1
4. Insert the sniffer into the side car pod
   > kubectl –f deployment-sniffer-sensor-tap-sample.yaml

Sensor deployment as DaemonSet in Kubernetes

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. This provides an opportunity to deploy sniffer sensor without having to make any changes to existing application pods.



The sniffer will be deployed as a DaemonSet container with host network access and privileges to sniff on the node's network traffic. By configuring the correct eBPF filters, application traffic can be learnt.

Reference daemonset spec:

https://sample-config.appsentinels.ai/appsentinels-deployment/sniffer/k8-daemonset/appsentinels-sniffer-daemonset.yaml

Procedure:

1. Create a namespace called “appsentinels”
2. Define relevant resource limits for the AppSentinels sniffer container (refer here)
3. Configure the container’s environmental variables like,
    a. REMOTE_CONTROLLER_SERVER_NAME
    b. REMOTE_CONTROLLER_SERVER_PORT
    c. TAP_INTERFACE =default*
    d. TAP_FILTER=tcp-and-port-not-22-and-port-not-443-and-port-not-9004 # Modify based on application service listening port
    e. SNIFFER_SENSOR_INSTANCE=<instance name> # Provide user identifiable instance name for visibility, eg: sniffer-sensor-dev-app-1
4. Insert the daemonset into the kubernetes cluster
   > kubectl –f appsentinels-sniffer-daemonset.yaml

Please note that the daemonset need to be inserted only on worker nodes where the application API traffic is seen.

Sensor deployment in AWS EC2

Sniffer sensor can also be deployed in a zero touch manner on EC2 instances. AWS provides for user data configuration during AMI launch. This allows for installation of required tools or packages during the first boot of an AMI.

AppSentinels provides an user data script that can be inserted into the AMI during its launch. This script installs the AppSentinels Sniffer service on Ubuntu, Amazon Linux or RHEL/CentOS along with necessary utils like docker (if not present).

If inserting as a user data script isn’t feasible, this script can run in a standalone fashion.

Script Configuration:

Depending on the deployment and the application running, the following basic configurations will need to be done.

```
REMOTE_CONTROLLER_SERVER_NAME="<edge controller hostname>"      #hostname of deployed edge controller
TAP_INTERFACE="default"                                        #Interface over which API traffic traverses.
                                                               #If unknown, keep as default
TAP_FILTER="<ebpf filter with hyphens for spaces>"              #Filter as described in earlier section of
                                                               #the document
TAP_PROFILE="low"                                              #(low|medium|high) as per resource
                                                               #availability
INSTANCE_NAME=”default”                                         # Provide user identifiable instance name for
                                                                # visibility, eg: sniffer-sensor-dev-app-1
RELAY_PROTOCOL="http"                                          #input needed here for logging
protocol
                                                              #(http|https)
HTTPS_INSECURE_SKIP_VERIFY="true"                              #input needed here to skip verify for https
                                                            #(set to true if using self-signed certs)
```

Script Location:

https://sample-config.appsentinels.ai/appsentinels-deployment/sniffer/install-appsentinels-sniffer.sh

Sensor deployment in AWS ECS

This describes different methods to deploy AppSentinels sidecar sensor in the AWS ECS cluster. In this case the controller is deployed separately

How it works

- Sensor sidecar deployment requires following inputs:
    - Cluster name
    - Service name
    - Task definition family or ARN
    - Sensor name
    - Sensor image complete URL
    - Controller name or URL
    - Controller port number
    - Type of environment (Dev/QA/Prod)
    - TAP interface name
        - This can be specified as *default*, sensor picks the one having default route defined.
        - Otherwise, please specify interface name as *eth0* and *eth1* for launch type *EC2* and *FARGATE* respectively.
    - eBPF filter
    - Relay protocol
    - AWS region
- Fetch the existing task definition, identified by the task definition family or ARN.
- Check if the AppSentinels sensor sidecar spec is already present in the task definition.
- If AppSentinels sidecar spec is not present, append AppSentinels sensor spec in container definitions in task definition.
- Include different environment variables in the AppSentinels container spec based on the values passed as parameters.
- Update the task definition in the ECS cluster.
- Update the service with the new task definition, so that all the containers are recreated including AppSentinels sensor container.

Methods supported

AppSentinels supports following methods of sensor sidecar deployment in AWS ECS cluster.

- AWS CLI script
- CloudFormation

TaskRole requirement

AppSentinels sensor generates logs and requires permissions to forward those logs to AWS CloudWatch. This requires *TaskRoleArn* defined the in the task definition attached with the ECS service. If the *TaskRoleArn* does not exist in the task definition, please follow the below steps to create a *TaskRole* and attach the required policy with that role, before running the steps to deploy AppSentinels sensor.

- Create a file *task-execution-assume-role.json* with following contents.

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Sid": "",
             "Effect": "Allow",
             "Principal": {
                 "Service": "ecs-tasks.amazonaws.com"
             },
             "Action": "sts:AssumeRole"
         }
     ]
 }
```

- Create the task role using the command:
  aws iam create-role --role-name <ECS Task Role Name> --assume-role-policy-document file://task-execution-assume-role.json
- Execute following command:
  aws iam attach-role-policy --role-name <ECS Task Role Name> --policy-arn arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy
- Create a file *ecs-task-logging-policy.json* with following contents.

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [
                 "logs:CreateLogGroup",
                 "logs:CreateLogStream",
                 "logs:PutLogEvents",
                 "logs:DescribeLogStreams"
             ],
             "Resource": "*""Resource": [
                 "arn:aws:logs:*:*:log-group:/ecs/*",
                 "arn:aws:logs:*:*:log-group:/ecs/*:log-stream:*"
             ]
         }
     ]
 }
```

- Create policy for enabling task logging. Please note the ARN of the policy displayed in the output of the command.
  aws iam create-policy --policy-name <ECS Task Logging Policy Name> --policy-document file://ecs-task-logging-policy.json
- Attach the policy with the role.
  aws iam attach-role-policy --role-name <ECS Task Role Name> --policy-arn <Policy ARN>
- Update the *TaskRoleArn* in the task-definition of the ECS service in the cluster.
- Update the service with modified task definition.

AWS CLI script

- Please use script *appsentinels_sensor_install.sh* to deploy the AppSentinels sniffer sensor in AWS ECS cluster.

- This script uses *aws ecs* CLI commands to get task-definition, register task-definition and update the service.
- This script requires parameters listed above.
- Prerequisites:
  - AWS CLI
  - jq
- IAM role permissions required
  - ecs:DescribeTaskDefinition
  - ecs:RegisterTaskDefinition
  - ecs:UpdateService
  - logs:CreateLogGroup
  - Doesn't create or modify IAM roles, but it uses existing Task role and Execution role
  - Task and execution roles should have permissions for task to run, pull images and send logs
  - Permissions to rule AWS CLI
- Example: https://sample-config.appsentinels.ai/appsentinels-deployment/aws-ecs/appsentinels_sensor_install.sh

  ./appsentinels_sensor_install.sh test-cluster test-service test-app appsentinels-sensor docker.io/appsentinels/ng-controller:latest appsentinels-controller 9004 dev eth0 port-80-and-tcp http ap-south-1
  Where,
  test-cluster is the name of the ECS cluster,
  test-service is the name of the ECS service,
  test-app is the task definition family,
  appsentinels-sensor is the name given to AppSentinels sensor,
  docker.io/appsentinels/ng-controller:latest is the AppSentinels sensor docker image,
  appsentinels-controller is the name of the AppSentinels controller,
  Controller port is 9004,
  Environment is dev,
  eth0 is the tap interface,
  The eBPF filter is port-80-and-tcp and
  The relay protocol is http.
  ap-south-1 is the AWS region.

AWS CloudFormation

- It is possible to use AWS CloudFormation to deploy the AppSentinels sensor sidecar container in ECS cluster.
- The CloudFormation manifest is define in the *appsentinels_sensor_deployment.yaml*.
- The commands used to create and update the stack are documented in *cloud_formation_cmds.txt* along with the example command.
- Prerequisites
  - AWS CLI

    - jq
- IAM role permissions
    - Need CloudFormation permissions, as well as Lambda permissions
    - CloudFormation permissions
        - cloudformation:CreateStack
        - cloudformation:UpdateStack
        - cloudformation:DeleteStack
        - lambda:CreateFunction
        - lambda:DeleteFunction
        - lambda:AddPermission
        - lambda:RemovePermission
        - iam:CreateRole
        - iam:DeleteRole
        - iam:PutRolePolicy
        - iam:DeleteRolePolicy
        - iam:AttachRolePolicy
        - iam:DetachRolePolicy
        - AWSLambdaBasicExecutionRole
    - Other permissions
        - ecs:DescribeTaskDefinition
        - ecs:RegisterTaskDefinition
        - ecs:UpdateService
        - iam:PassRole (for TaskRole and TaskExecutionRole)
        - logs:CreateLogGroup
- Example: https://sample-config.appsentinels.ai/appsentinels-deployment/aws-ecs/cloud_formation_cmds.txt

  aws cloudformation create-stack --stack-name add-appsentinels-sensor-stack --template-body file://appsentinels_sensor_deployment.yaml --parameters ParameterKey=ExistingTaskDefinitionArn,ParameterValue=arn:aws:ecs:ap-south-1:488922454646:task-definition/test-app:13 ParameterKey=SidecarContainerName,ParameterValue=appsentinels-sensor ParameterKey=SidecarContainerImage,ParameterValue=docker.io/appsentinels/ng-controller:latest ParameterKey=EcsClusterName,ParameterValue=test-cluster ParameterKey=EcsServiceName,ParameterValue=test-service ParameterKey=ControllerServerName,ParameterValue=appsentinels-controller ParameterKey=ControllerPort,ParameterValue=9004 ParameterKey=EnvironmentType,ParameterValue=test ParameterKey=TapInterface,ParameterValue=eth0 ParameterKey=TapFilter,ParameterValue=port-80-and-tcp ParameterKey=RelayProtocol,ParameterValue=http ParameterKey=AwsRegion,ParameterValue=ap-south-1 --capabilities CAPABILITY_IAM CAPABILITY_AUTO_EXPAND

Sensor deployment on host (container-less)

AppSentinels Sniffer can be deployed as a systemd service. This allows for deployment on platforms which do not support containers.

Procedure

1. Download the provided deb package

   ```
   sudo dpkg -i appsentinels-sniffer-<version>-<ubuntu-version>.deb
   ```

2. Resolve any missing dependencies

Please note that this package requires net-tools package to be present on the system

3. Verify installation: You can check that the package is installed with.

   ```
   dpkg -l | grep appsentinels-sniffer
   ```

Configuring and Managing Installed Services

After installing the DEB package, the following systemd services will be available:

- **appsentinels-tapper.service**
  Runs Suricata in tap/sniffer mode to capture and process network traffic.
- **appsentinels-data-path-handler.service**
  Handles data path processing and communication with the controller.

Service Configuration

The environment file for the sniffer is installed at: /usr/local/appsentinels-onprem/env/env.txt

Memory Usage Configuration (MEMORY_MAX)

Both systemd services (appsentinels-tapper.service and appsentinels-data-path-handler.service) support memory usage limits via the MemoryMax and MemoryLimit (older kernels) directives.
By default, these are set in the service files generated by the packaging script:

/etc/systemd/system/appsentinels-tapper.service:
MemoryMax=700M
 # older kernels don't support MemoryMax
MemoryLimit=700M

/etc/systemd/system/appsentinels-data-path-handler.service:

MemoryMax=300M
 # older kernels don't support MemoryMax
MemoryLimit=300M

Note:

- MemoryMax is supported on most modern Linux distributions. If your kernel does not support it, MemoryLimit acts as a fallback.
- Setting these values too low may cause the service to be killed if it exceeds the limit
- In the above example, both the tapper and data path services together use max of 1G of memory

CPU Usage Configuration (CPUQuota)

Both systemd services (appsentinels-tapper.service and appsentinels-data-path-handler.service) support CPU usage limits via the CPUQuota directive. By default, these are set in the service files generated by the packaging script:

/etc/systemd/system/appsentinels-tapper.service:

CPUQuota=70%

/etc/systemd/system/appsentinels-data-path-handler.service:

CPUQuota=30%

Note:

- CPUQuota specifies the maximum CPU time that the service can use, expressed as a percentage of a single CPU core.
- A value of 100% means the service can use up to one full CPU core (all threads included).
- A value of 200% means the service can use up to two full CPU cores (all threads included)
- A value of 50% means the service can use up to half a CPU core (all threads included)
- Setting these values too low may cause the service to be throttled or perform poorly under high load.
- You can adjust these values based on your system's available CPU resources and performance requirements.

Post-Install: Reload systemd and Start Services

After installing the deb package, perform config as above. You should reload the systemd daemon and start the services. Optionally, you can enable them to start automatically on boot.

Run the following commands as root (or with sudo):

```
sudo systemctl daemon-reload
sudo systemctl enable appsentinels-tapper  (enable on boot)
```

```
sudo systemctl enable appsentinels-data-path-handler (enable
on boot)
sudo systemctl start appsentinels-tapper
sudo systemctl start appsentinels-data-path-handler
```

Modifying environmental variables

If you modify environment variables or configuration files, remember to reload and restart the services:

```
sudo systemctl daemon-reload
sudo systemctl restart appsentinels-tapper
sudo systemctl restart appsentinels-data-path-handler
```

Debugging and Logging

Data Path Handler Logs

The data-path-handler service logs to the system journal.
To view recent logs:

```
sudo journalctl -u appsentinels-data-path-handler –f
```

Tapper (Suricata) Service Logs

The tapper service (appsentinels-tapper) also logs to the system journal.
To view logs:

```
sudo journalctl -u appsentinels-tapper –f
```

Troubleshooting Tips

If a service fails to start, check its status and logs:

```
sudo systemctl status appsentinels-tapper
sudo systemctl status appsentinels-data-path-handler
```

Verify Deployment

AppSentinels Edge Controllers deployed in your environment will be listed on the System Health page on AppSentinels Dashboard.

Generate application traffic in the monitored application and check the API catalogue on AppSentinels Dashboard for the discovered APIs.

Advanced Configuration

Performance Mode Configuration

Performance mode for sniffer provides for an advanced tuning to improve logging and resource efficiency. This configuration is advisable when the system properties are known and well defined. Here is the procedure to configure the performance mode,

1. Set env **TAP_PROFILE** to performance. If specific interface is provided (like default or eth0 or lo), sniffer will create as many tapper threads as defined by **TAP_WORKER_THREADS**
2. **TAP_WORKER_THREADS** should be set to as many CPUs one can afford. However, in non-container based deployments, the CPU will be capped via CPUQuota and in container based deployments, POD/docker resource limits apply
3. If **TAP_WORKER_THREADS** isn't defined a single worker thread is created
4. It is recommended that a single interface sniffing is performed to conserve resources in performance mode

Filtering for URI Extension, Hostname, URI and Extension Filtering

These environment variables should be set in the container environment for your sniffer sensor.

SKIP_HOSTNAME_REGEX

Purpose:
Skip (exclude) HTTP transactions where the hostname matches this regular expression.

How it works:

- If the regex matches the hostname, the transaction is **SKIPPED** (not logged).
- If not set or empty, no hostnames are excluded by this filter.
- **Do not include port numbers** in the hostname for this filter.

Example values:

- SKIP_HOSTNAME_REGEX="^www\.example\.com$"
  Skips any transaction where the hostname is exactly `www.example.com`.
- SKIP_HOSTNAME_REGEX="(test|dev)\.mydomain\.org$"
  Skips hostnames ending with `test.mydomain.org` or `dev.mydomain.org`.

HOSTNAME_INCLUDE_REGEX

Purpose:
ONLY process (include) HTTP transactions where the hostname matches this regular expression.

How it works:

- If set, ONLY hostnames matching this regex are processed.
- If the hostname does NOT match, the transaction is skipped.

- If not set or empty, all hostnames are included (unless excluded by SKIP_HOSTNAME_REGEX).
- **Do not include port numbers** in the hostname for this filter.

Example values:

- HOSTNAME_INCLUDE_REGEX="^api\."
  Only process hostnames starting with `api.`
- HOSTNAME_INCLUDE_REGEX="(prod|main)\.mydomain\.org$"
  Only process hostnames ending with `prod.mydomain.org` or `main.mydomain.org`.

SKIP_EXTENSIONS

Purpose:
Skip (exclude) HTTP transactions where the URI path ends with one of the specified file extensions.

How it works:

- Value should be a | (pipe)-separated list of file extensions (e.g., `svg|png|jpg`).
- If the URI path ends with one of these extensions, the transaction is skipped.
- If not set or empty, a default list of common static file extensions is used.

Default Extensions Skipped:

`svg|txt|png|jpg|jpeg|io|tff|woff|woff2|ico|css|pdf|html|mp4|php|ajax|js|gif|tiff|ttf|docx|wasm|aspx`

Example values:

- SKIP_EXTENSIONS="svg|png|jpg|ico"
  Skips URIs ending with `.svg`, `.png`, `.jpg`, or `.ico`.
- SKIP_EXTENSIONS="js|css|woff2"
  Skips URIs ending with `.js`, `.css`, or `.woff2`.

SKIP_URI_REGEX

Purpose:
Skip (exclude) HTTP transactions where the URI path contains the specified string.

How it works:

- Value should be a | (pipe)-separated list of path string (e.g., `tracking|getprice|getval`).

- If the URI path contains with any of this string, the transaction is skipped.
- If not set or empty, nothing will be skipped.

Example values:

- `SKIP_URI_REGEX=jsonloopback|timeout|headerloopback`

Order of Evaluation

1. If `HOSTNAME_INCLUDE_REGEX` is set, the hostname **must** match this regex to be processed.
2. If `SKIP_HOSTNAME_REGEX` is set, the hostname **must not** match this regex to be processed.
3. If `SKIP_EXTENSIONS` is set, the URI path **must not** end with one of the specified extensions.
4. If both hostname regexes are set, the hostname must:
    a. Match the **include regex**, and
    b. **Not** match the **skip regex**.
5. If `SKIP_URI_REGEX` is set, the URI **must not** match this regex to be processed.

Regex Syntax and Escaping

- Write regexes as you would in C or Python. E.g., `^foo\.bar$`, `.*\.test\.local$`
- Use a single backslash (`\.`) to escape metacharacters.
- In **YAML (Kubernetes, Docker Compose)**, you may need to **double the backslash** (`\\.`) to ensure proper escaping.

Example for Kubernetes (helm chart variables)
values.yaml:

```
skipHostnameRegex: "^www\\.example\\.com$"
includeHostnameRegex: "^api\\."
uriExtentionFilter: "svg|png|jpg"
```

Notes

- All regexes are **case-insensitive**.
- If **neither** hostname regex is set, **all** hostnames are processed.
- If **both** hostname regexes are set, both conditions must be satisfied.
- If `uriExtentionFilter (or SKIP_EXTENSION)` is not set, a default list of static file extensions is used.

Traffic Sampling

In case, sampling is preferred rather than complete logging, use env **HTTP_SAMPLE_PERCENT.** This env takes a values between 1 to 99 (percent).

Note: If HTTP_SAMPLE_PERCENT value of 0 or 100 are considered to be invalid values and full logging will be performed in that case.

Deployment/Debugging

Deploying sniffer sensor/integrated sensor controller

1. Confirm the listening ports of your application server. If this isn't known a prior, it can be confirmed by running a tcpdump on the application server.

tcpdump –i <ingress device, eg: eth0, ens3 etc> port <application listening port> -n

Please set the <application listening port> to suspected listening port and hit your application server and some activity should be seen with the above command

Example usage: tcpdump –i eth0 port 8080 -n

2. Set the TAP_FILTER in the YAML for the sniffer as described in sections 3.2 or 4.3

3. Bring up the sniffer sensor using docker-compose

docker-compose -f <YAML spec> up –d

Also confirm if the container is running fine using "docker ps"

4. If application traffic is flowing, then sniffer sensor should be able to pickup HTTP traffic. This can be confirmed via debugging interface on the sniffer/controller. We should be able to see the counter *TapMsgs* increment

docker exec -it <container_name> /usr/local/appsentinels-onprem/utils/dp-commands.sh dp-debugstats

The output of this command will be of the format,

```
{
        ...
        "TrafficStats":{"TapMsgs":18,"TapUsablePacket":18},
        ...
}
```

5. Deploy edge controller using the YAML specs as explained in section 4.2. Once the appropriate environment values are filled, you can start the edge controller service,

docker-compose -f <YAML spec> up –d

You can confirm if the container is running using “docker ps” command.

6. Sniffer sensor will ship logs to edge controller’s (default port 9004) usually running on a different VM. To check if there is any connectivity issue between sniffer sensor and edge controller, run the below tcpdump on the edge controller VM.

```
tcpdump -i <ingress device eg: eth0, ens3> -A  'port 9004 and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0)' | egrep --line-buffered "^........(GET |HTTP\/|POST |HEAD )|^[A-Za-z0-9-]+: " | sed -r 's/^........(GET |HTTP\/|POST |HEAD )/\n\1/g'
```

The above command should output API requests called **POST /mergedlogs** which signals, that connectivity is good between sensor and edge controller’s VM.
The above command assumes that sniffer is sending logs to port 9004, however, if you have setup sensor to ship logs to a different port (via REMOTE_CONTROLLER_SERVER_PORT environment variable in YAML), please change the port according in tcpdump

7. If edge controller is connected to AppSentinels cloud, you should be able to see your APIs on the dashboard.

