Garland Prisms Documentation

Release 1.1

Product

Jun 24, 2020

Garland Prisms User Guide

CHAPTER 1

Welcome to Garland Prisms

Traffic visibility is a crucial component in securing the business and keeping systems operational. However, network monitoring has been blinded in the cloud. Organizations have made significant investments in specialized tools that ingest and analyze packet-level data. With compute resources, application development and core business systems moving to the cloud, IT teams are no longer able to acquire, process and distribute packet-level cloud traffic to their selected tools. Consequently, the move to the cloud creates significant blind-spots and loss of ROI on vital tools that are powerless without access to packet-level cloud data. Garland Prisms is a Software as a Service (SaaS) offering that provides complete packet visibility into any public cloud. Garland Prisms mirrors packets within a cloud instance and forwards them to security and analysis tools. Garland Prisms has a split SaaS architecture comprised of central control: `Prisms Cloud Console` and `Cloud Agents` (also referred to as `Prisms`). The control plane is split between the `Prisms Cloud Console` and `Cloud Agents`. The architecture is designed to be secure and robust.

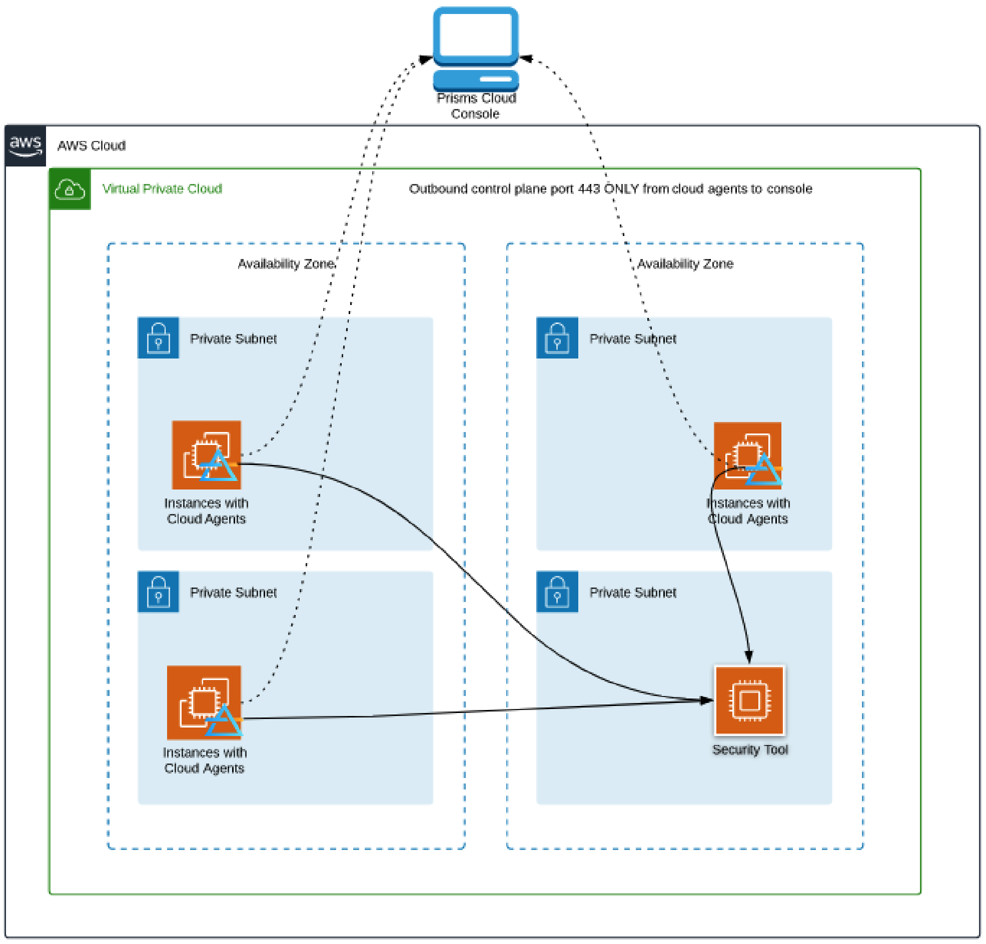


Figure 1: Prisms services architecture

Figure 1 depicts a sample deployment in an AWS could. Cloud Agents filter and mirror traffic based on mirroring policies. Policies are comprised of source groups, connections, and destinations which users define using the Cloud Console.

When any instance containing a Cloud Agent launches, the agent will automatically connect to the Prisms Cloud Console and register itself, obtain configuration updates and automatically install software updates when upgrades are available. Prism Cloud Agents use HTTPS to make REST API calls to the Cloud Console. Control traffic always originates at the agent. Data plane traffic (mirrored filtered traffic) is routed based on the users' network configurations. Mirrored packets are never sent to the Cloud Console. The control plane does not directly modify, nor does it require the user to modify networks or security setting, save for allowing outbound HTTPS (TCP port 443) from subnets containing Cloud Agents.

The following URLs and IP addresses should be accessible for the agents to connect:

```
https://garland.nuos.io/api/1.1/wf
https://rvs.nuos.io
13.248.140.181
52.183.93.152
```

CHAPTER 2

Getting Started

2.1 Logging in

To get started with Garland Prisms, start by creating an account and log in on the *Garland Prisms Cloud Console*:

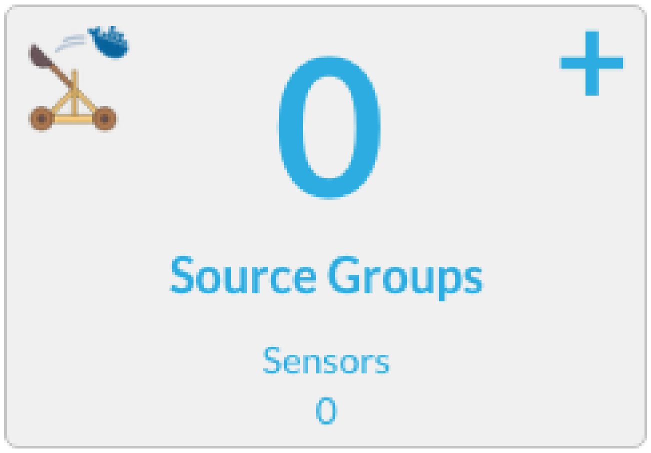
1. Navigate to and select 'Login' from the main menu.
2. First-time users will be prompted to create an account. Go ahead and use one of the OAuth partners to log in.
3. When you log in to Garland Prisms for the first time, you will see a Prisms Overview. This overview will help you through the process of setting up your first Prism as well as a destination. You can skip this if you want and dive right in. You can always revisit this on the Help page.

When you log in a 'Default Project' is created automatically for you. You can now install Cloud Agents.

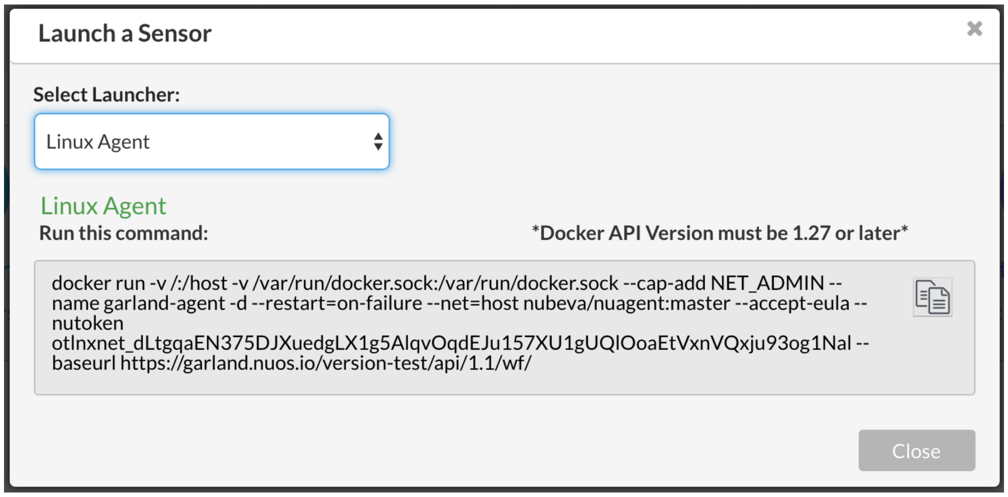
2.2 Creating Cloud Agents - Prisms

`Cloud Agents` or `Prisms` are available for Linux and Windows. The Linux agent is supported on Centos, Ubuntu, RHEL, Amazon AMI. Windows agents support Windows 10 and Server 2016 or higher.

Adding a `Cloud Agent` or `Prism` to an existing cloud instance is simple. The Cloud Agent for Linux is a container that can be easily installed by following the steps below: 2. Click on the "Docker Catapult" icon on the top left of the Source Group box.



3. This will pop up a box similar to the figure below. Select 'Linux Agent' from the dropdown menu. Click the button on the right to copy the installation command.

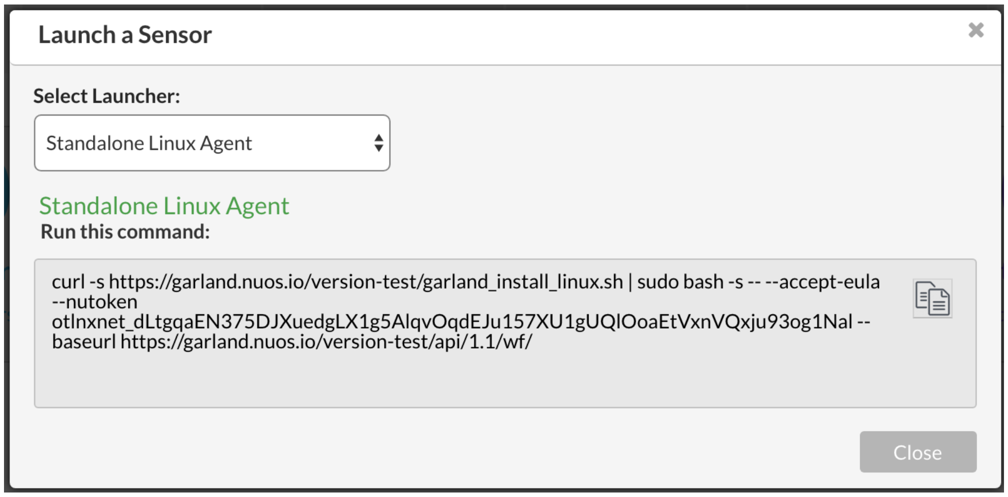


Linux Agent

Run this command: *Docker API Version must be 1.27 or later*

```
docker run -v /:/host -v /var/run/docker.sock:/var/run/docker.sock --cap-add NET_ADMIN --
name garland-agent -d --restart=on-failure --net=host nubeva/nuagent:master --accept-eula --
nutoken
otlnxnet_dLtgqaEN375DJXuedgLX1g5AlqvOqdEJu157XU1gUQlOoaEtVxnVQxju93og1Nal --
baseurl https://garland.nuos.io/version-test/api/1.1/wf/
```

4. Paste this command into a command shell on the cloud instance. The Prism container will automatically download and install.
5. About 10-20 seconds after installation, the agent counter in the Source Groups box will increase by 1 indicating that the agent is active.
6. To launch a standalone Linux installer select the option shown in the figure below:



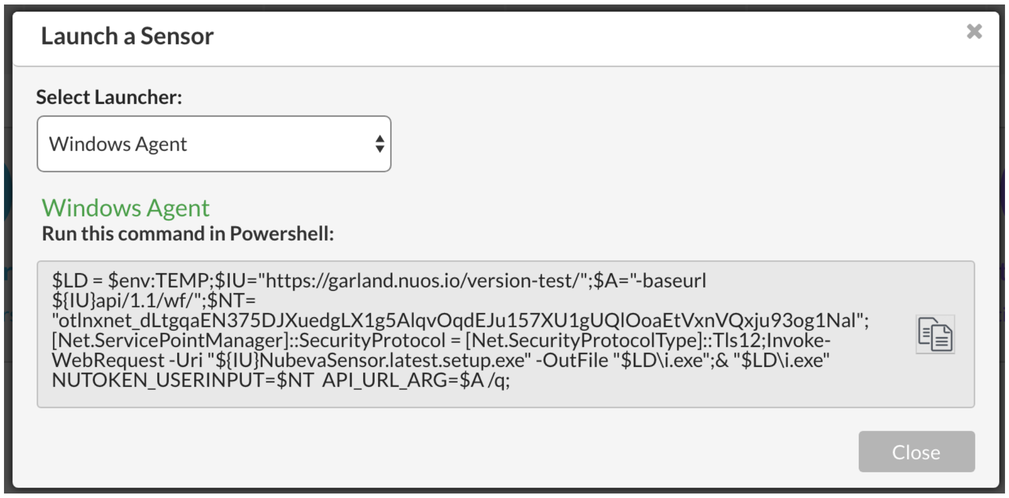
Standalone Linux Agent

Run this command:

```
curl -s https://garland.nuos.io/version-test/garland_install_linux.sh | sudo bash -s -- --accept-eula
--nutoken
otlnxnet_dLtgqaEN375DJXuedgLX1g5AlqvOqdEJu157XU1gUQlOoaEtVxnVQxju93og1Nal --
baseurl https://garland.nuos.io/version-test/api/1.1/wf/
```

Note: 'Standalone' means that the agent is installed without docker and runs as a process rather than within a container.

7. To launch a Windows agent, select 'Windows Agent' option from the dropdown.



Windows Agent

Run this command in Powershell:

```
$LD = $env:TEMP;$IU="https://garland.nuos.io/version-test/";$A="-baseurl
${IU}api/1.1/wf/";$NT=
"otlnxnet_dLtgqaEN375DJXuedgLX1g5AlqvOqdEJu157XU1gUQlOoaEtVxnVQxju93og1Nal";
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12;Invoke-
WebRequest -Uri "${IU}NubevaSensor.latest.setup.exe" -OutFile "$LD\i.exe";& "$LD\i.exe"
NUTOKEN_USERINPUT=$NT API_URL_ARG=$A /q;
```

9. Paste this command into a command shell on the cloud instance. The Prism container will automatically download and install.

Note: The Windows Agent uses the Winpcap API with the Npcap driver to read packets from of the network interface card.

To uninstall the Windows Agent run:

```
wmic product where "description='Nubeva Sensor' " uninstall
```

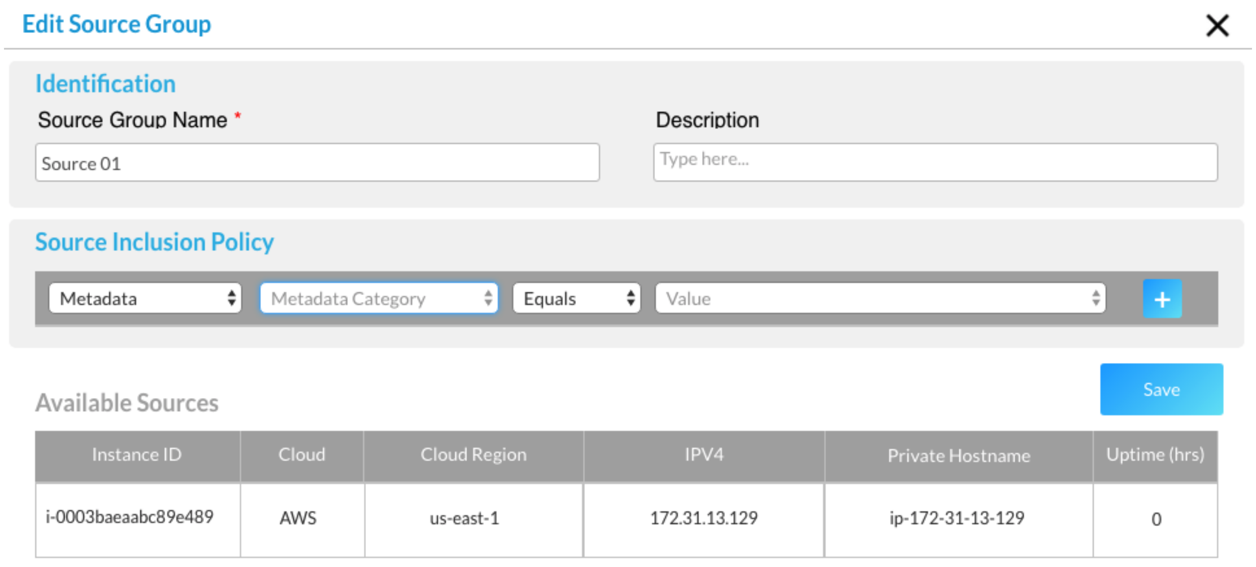
The next step is to configure source groups.

2.3 Creating and Configuring Source Groups

Source groups are policies for grouping Cloud Agents based on their *CIDR address ranges*.

These are the steps to create a source group:

1. Click on the "+" icon at the top right of the "Source Group" box. This will load the Source Group editing window.



Source 01

Available Sources

Instance ID	Cloud	Cloud Region	IPV4	Private Hostname	Uptime (hrs)
i-0003baeaabc89e489	AWS	us-east-1	172.31.13.129	ip-172-31-13-129	0

2. Name the new group.
3. Click on the 'Filter Type' (leftmost) drop down to select 'CIDR block'.
4. Click on the Metadata Category in the "Source Inclusion Policy" and choose something. Select the condition and select the values. Then click the '+' button.
5. You can add multiple conditions.
6. Select save. This will return you to the dashboard.

As new agents with matching metadata and/or custom tags appear, they are automatically added to the source group. This is a very powerful adaptation to the elastic nature of cloud environments. For instance, if you routinely spin up/down instances with web scaling events, selecting filters such as AMI type, VPC, subnet, or custom tag values, will ensure that any new Cloud Agents that appear will be immediately added to the source groups and start mirroring traffic to tools based on the existing connections defined.

Next, you'll create destinations to consume the data.

2.4 Creating Destinations

There are many packet inspection and processing tools in the open source community as well as many vendor offerings. Before we create a destination in the UI, we need to have an actual tool. If you already have something running, skip ahead to the next section. Otherwise, here are some steps for setting up a few simple tools.

1. One of the simplest tools you can use is tcpdump. This is a simple Unix command that takes all data received on an interface and displays it on the screen. You also use it to write this traffic to a file.
2. Create a Unix instance in your cloud provider. Connect to it and issue the command:

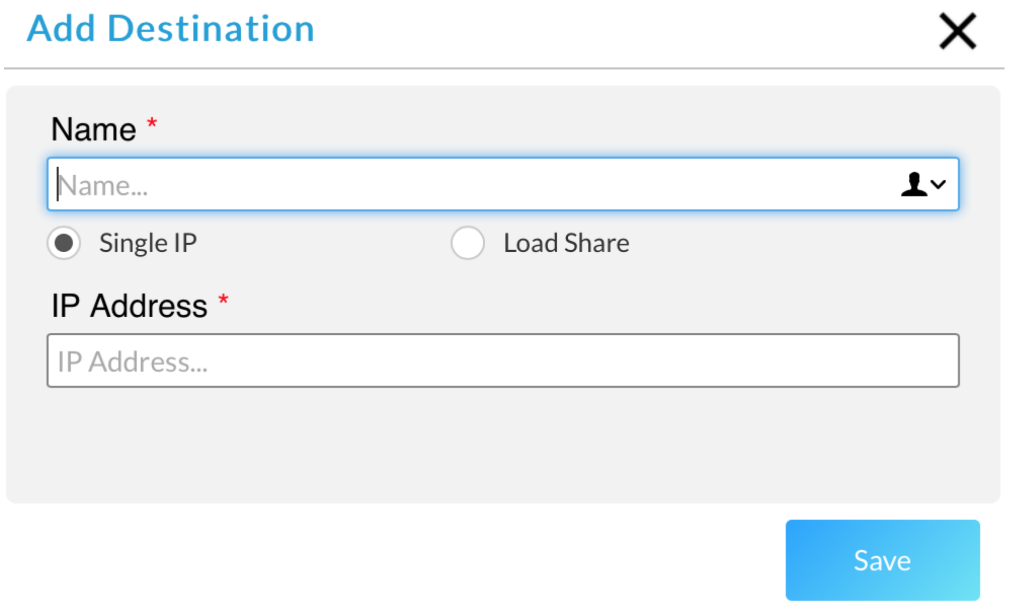
```
tcpdump -na -i eth0 port not 22
```

3. This will start a tcpdump session which will display all traffic on your default interface but it will not show your SSH session traffic.

Note: If you want a tool with a little more sizzle, look at . NTOP is a network traffic analysis solution that can be deployed in many forms.

These are the steps to create a destination that points to a tool:

1. Click on the "+" icon at the top right of the "Destinations" box.
2. The next screen allows you to define the properties of this new destination group.



3. Name the destination.
4. Choose between a single tool or a set of tools that you want to load share against. For a load share environment, insert multiple comma-separated IP addresses into the field and the traffic will be load-shared among the defined tools.

Note: If you need more advanced load-balancing, you can use the front-end IP address of a cloud load balancer as the IP and configure this LB via the cloud portals.

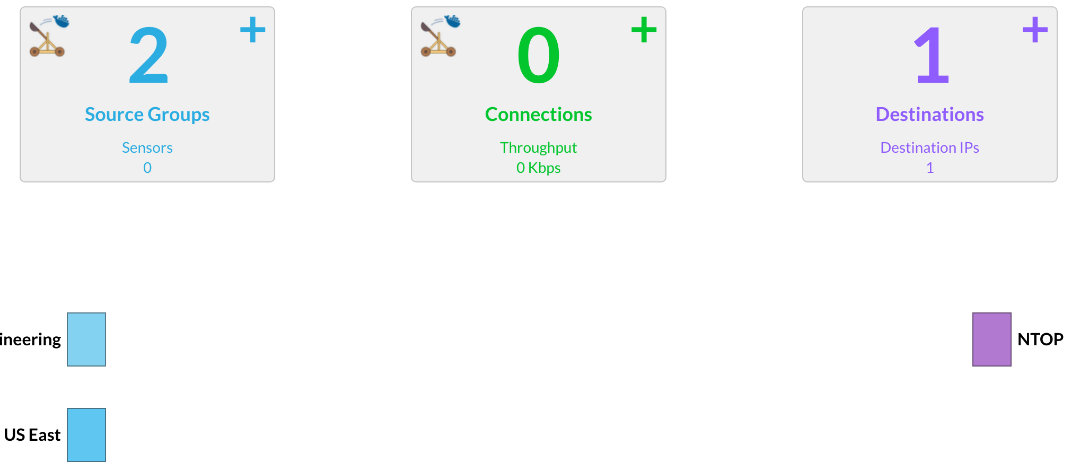
---

5. Click 'save' when finished. This will return you to the dashboard.

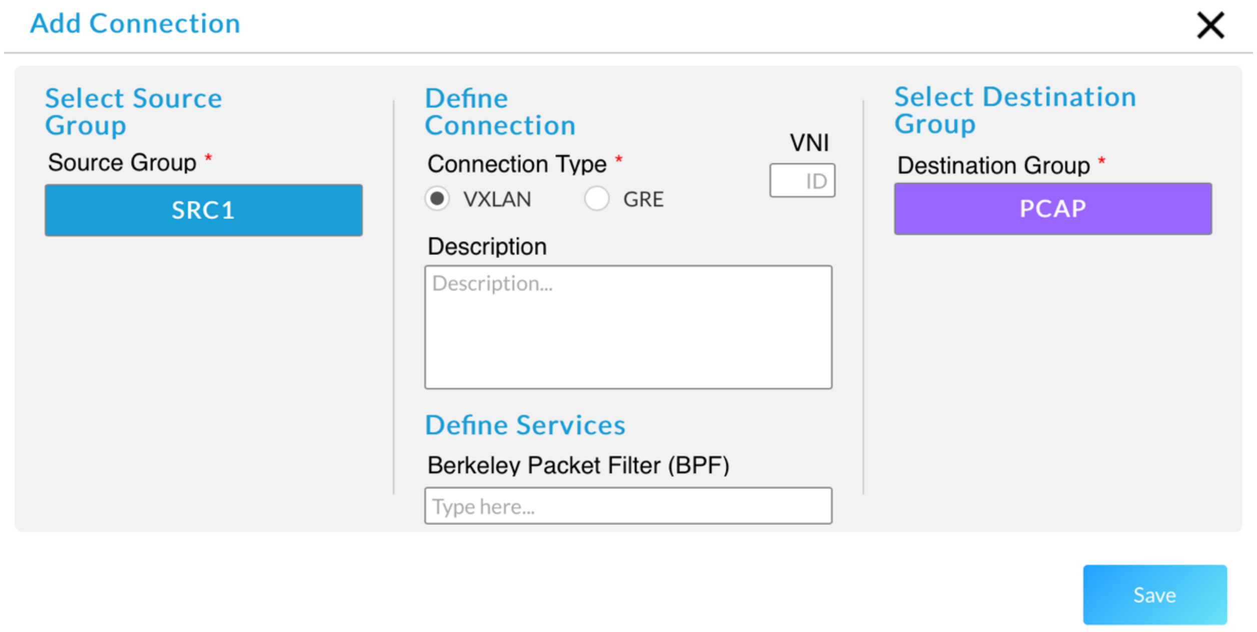
The final step is to define a connection between your source group and your destination.

2.5 Creating Direct Connections

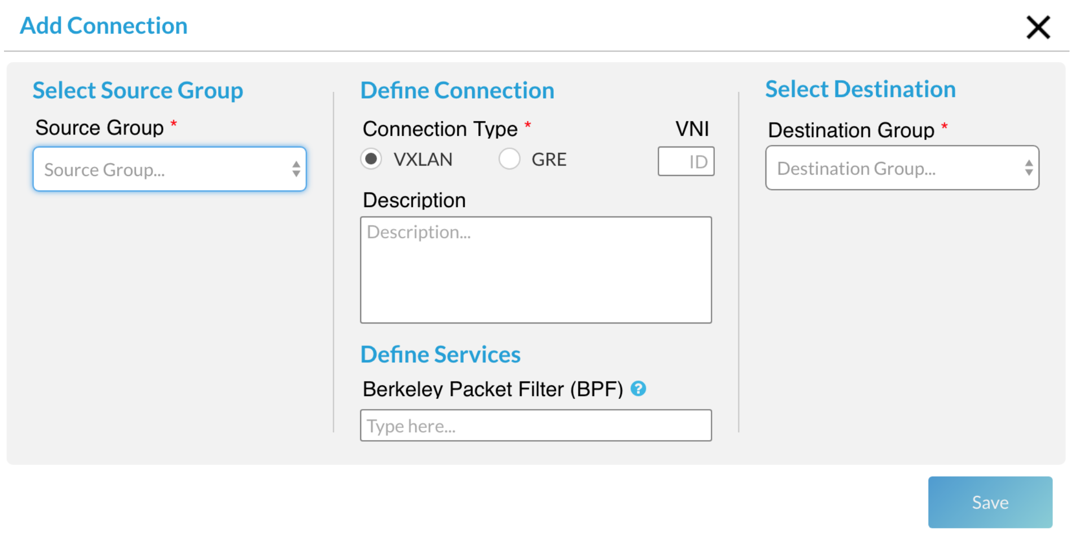
At this point, you can have one or more source groups and one or more tool destinations. Your screen could resemble this.



1. To connect a source group to a destination/tool, simply click on the source and drag the connection line to the required destination and drop it. This will popup the following connection profile window, with the 'Source Group' and 'Destination Group' values preset.



2. You can also click on the "+" icon on the Connections box and it will take you to the same screen, but you will have to choose the Source Group and Destination Group from their dropdown.



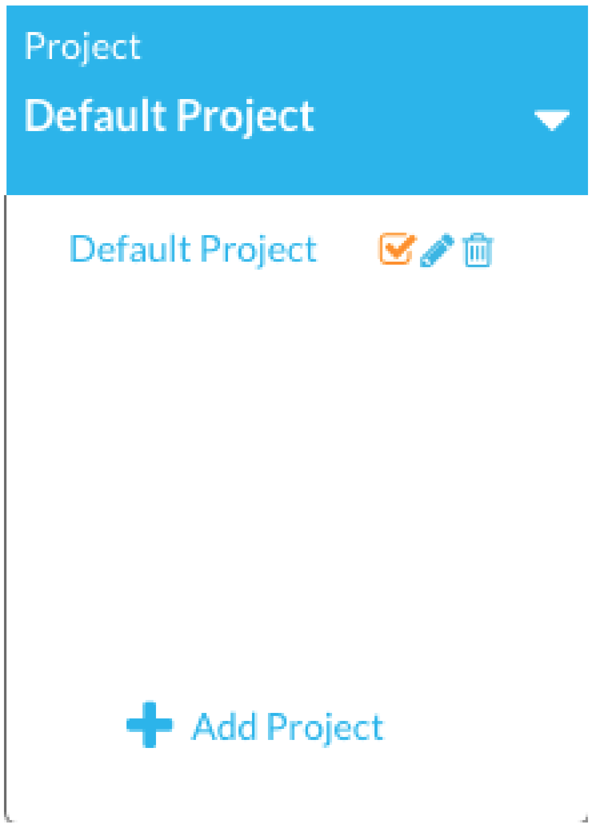
3. You can use either GRE or VXLAN tunnel encapsulation to send traffic to the destination. The destination should be able to terminate the tunnels of the specified type. The VNI/GRE ID should be a unique number for the source group.
4. Berkeley Packet Filter (*BPF*) is where you enter the data filters you want to apply to this connection.
5. Click save to return to the dashboard.

CHAPTER 3

Accounts and Projects

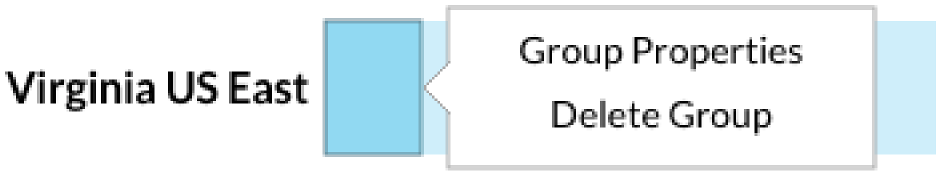
3.1 Working with Projects

Projects are logical units of mirroring workflows. An agent can belong to only one project at a time. Projects allow you to easily manage deployments as they grow. You can add/remove/select projects from the 'Project' option in the main menu.



You can modify or delete any element by clicking an element and selecting an option from the popup menu:

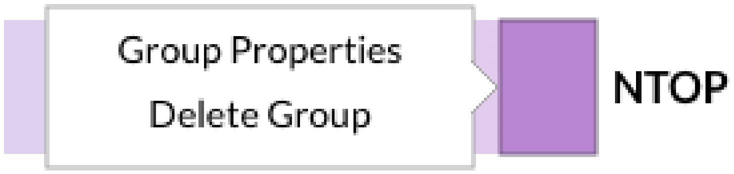
1. Source Groups:



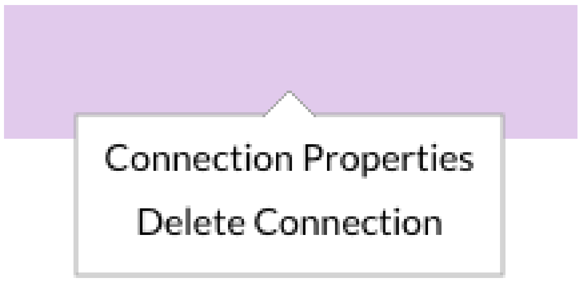
View/Modify Source Groups

Select `Group Properties` to view and modify a source group, `Delete Group` to remove a source group and its connections.

2. Destinations:



3. Connections:



View/Modify Connections

Select `Connection Properties` to view and modify a connetion, `Delete Connection` to remove it.

3.2 Monitoring

The Insights page shows agent status and traffic histories.

3.3 Help and Support

To report issues or if there is anything else you would like to see in the product, please let us know. Click on the Help page and put it in the form. That will tell us directly what you think we should do next.

CHAPTER 4

Installing Docker

`Key Agents` require version 18.09 of Docker which might not be the default version in your cloud. The instructions below show how to install versions of docker which our agents support.

4.1 Ubuntu 18.04

```
apt update && apt install -y docker.io
```

4.2 AWS Linux

AWS Linux AMI version 1 cannot install 18.09 the following commands will install 18.06.1-ce which works correctly.

```
On a completely new instance:

Amazon Linux AMI

sudo yum update -y
sudo yum upgrade
sudo yum install -y docker
sudo usermod -aG docker $USER
exit
# ssh back to instance
sudo service docker start
```

4.3 AWS Linux 2

```
On a completely new instance:

Amazon Linux 2 AMI (HVM), SSD Volume Type
amzn2-ami-hvm-2.0.20190508-x86_64-gp2 (ami-0cb72367e98845d43)


sudo yum update -y
sudo amazon-linux-extras install docker
sudo service docker start
sudo usermod -a -G docker ec2-user
exit
```

4.4 Cent OS 7

```
yum install -y docker
yum remove -y docker docker-common docker-selinux docker-engine
yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
yum install -y docker-ce
systemctl enable docker.service
systemctl start docker.service
```

4.5 Red Hat Enterprise Linux 7.5 (RHEL)

```
yum update -y
yum-config-manager --enable rhui-REGION-rhel-server-extras
yum install -y container-selinux yum-utils
yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo
yum install -y docker-ce
systemctl enable docker.service
systemctl start docker.service
```

4.6 Cloud Provider Instructions

Note: The cloud provider default version of Docker might not be compatible with our agents

These are the links to install cloud provider versions of docker on AWS and Azure instances.

CHAPTER 5

Berkeley Packet Filters

Berkeley Packet Filter (BPF) Syntax

The *expression* consists of one or more *primitives.* Primitives usually consist of an *id* (name or number) preceded by one or more qualifiers. There are three different kinds of qualifier:

type qualifiers say what kind of thing the id name or number refers to. Possible types are **host**, **net** **,** **port** and **portrange**. E.g., ‘host foo’, ‘net 128.3’, ‘port 20’, ‘portrange 6000-6008’. If there is no type qualifier, **host** is assumed.

dir qualifiers specify a particular transfer direction to and/or from *id*. Possible directions are **src**, **dst**, **src or dst** and **src and dst**. E.g., ‘src foo’, ‘dst net 128.3’, ‘src or dst port ftp-data’. If there is no dir qualifier, **src or dst** is assumed. For some link layers, such as SLIP and the ‘‘cooked” Linux capture mode used for the ‘‘any” device and for some other device types, the **inbound** and **outbound** qualifiers can be used to specify a desired direction.

proto qualifiers restrict the match to a particular protocol. Possible protos are: **ether**, **fddi**, **tr**, **wlan**, **ip**, **ip6**, **arp**, **rarp**, **decnet**, **tcp** and **udp**. E.g., ‘ether src foo’, ‘arp net 128.3’, ‘tcp port 21’, ‘udp portrange 7000-7009’. If there is no proto qualifier, all protocols consistent with the type are assumed. E.g., ‘src foo’ means ‘(ip or arp or rarp) src foo’ (except the latter is not legal syntax), ‘net bar’ means ‘(ip or arp or rarp) net bar’ and ‘port 53’ means ‘(tcp or udp) port 53’.

‘fddi’ is actually an alias for ‘ether’; the parser treats them identically as meaning ‘‘the data link level used on the specified network interface.” FDDI headers contain Ethernet-like source and destination addresses, and often contain Ethernet-like packet types, so you can filter on these FDDI fields just as with the analogous Ethernet fields. FDDI headers also contain other fields, but you cannot name them explicitly in a filter expression.

Similarly, ‘tr’ and ‘wlan’ are aliases for ‘ether’; the previous paragraph’s statements about FDDI headers also apply to Token Ring and 802.11 wireless LAN headers. For 802.11 headers, the destination address is the DA field and the source address is the SA field; the BSSID, RA, and TA fields aren’t tested.

In addition to the above, there are some special ‘primitive’ keywords that don’t follow the pattern: **gateway**, **broadcast**, **less**, **greater** and arithmetic expressions. All of these are described below.

More complex filter expressions are built up by using the words **and**, **or** and **not** to combine primitives. E.g., ‘host foo and not port ftp and not port ftp-data’. To save typing, identical qualifier lists can be omitted. E.g., ‘tcp dst port ftp or ftp-data or domain’ is exactly the same as ‘tcp dst port ftp or tcp dst port ftp-data or tcp dst port domain’.

Allowable primitives are:

dst host *host* True if the IPv4/v6 destination field of the packet is *host*, which may be either an address or a name.

src host *host* True if the IPv4/v6 source field of the packet is *host*.

host *host* True if either the IPv4/v6 source or destination of the packet is *host*. Any of the above host expressions can be prepended with the keywords, **ip**, **arp**, **rarp**, or **ip6** as in:

```
ip host host
```

which is equivalent to:

```
ether proto \ip and host host
```

If *host* is a name with multiple IP addresses, each address will be checked for a match.

ether dst *ehost* True if the Ethernet destination address is *ehost*. *Ehost* may be either a name from /etc/ethers or a number (see *'ethers </cgi-bin/man/man2html?5+ethers>'*__(5) for numeric format).

ether src *ehost* True if the Ethernet source address is *ehost*.

ether host *ehost* True if either the Ethernet source or destination address is *ehost*.

gateway *host* True if the packet used *host* as a gateway. I.e., the Ethernet source or destination address was *host* but neither the IP source nor the IP destination was *host*. *Host* must be a name and must be found both by the machine's host-name-to-IP-address resolution mechanisms (host name file, DNS, NIS, etc.) and by the machine's host-name-to-Ethernet-address resolution mechanism (/etc/ethers, etc.). (An equivalent expression is

```
ether host ehost and not host host
```

which can be used with either names or numbers for *host / ehost*.) This syntax does not work in IPv6-enabled configuration at this moment.

dst net *net* True if the IPv4/v6 destination address of the packet has a network number of *net*. *Net* may be either a name from the networks database (/etc/networks, etc.) or a network number. An IPv4 network number can be written as a dotted quad (e.g., 192.168.1.0), dotted triple (e.g., 192.168.1), dotted pair (e.g, 172.16), or single number (e.g., 10); the netmask is 255.255.255.255 for a dotted quad (which means that it's really a host match), 255.255.255.0 for a dotted triple, 255.255.0.0 for a dotted pair, or 255.0.0.0 for a single number. An IPv6 network number must be written out fully; the netmask is ff:ff:ff:ff:ff:ff:ff:ff, so IPv6 "network" matches are really always host matches, and a network match requires a netmask length.

src net *net* True if the IPv4/v6 source address of the packet has a network number of *net*.

net *net* True if either the IPv4/v6 source or destination address of the packet has a network number of *net*.

net *net* mask *netmask* True if the IPv4 address matches *net* with the specific *netmask*. May be qualified with **src** or **dst**. Note that this syntax is not valid for IPv6 *net*.

net *net/len* True if the IPv4/v6 address matches *net* with a netmask *len* bits wide. May be qualified with **src** or **dst**.

dst port *port* True if the packet is ip/tcp, ip/udp, ip6/tcp or ip6/udp and has a destination port value of *port*. The *port* can be a number or a name used in /etc/services (see *tcp*(7) and *udp*(7)). If a name is used, both the port number and protocol are checked. If a number or ambiguous name is used, only the port number is checked (e.g., **dst port 513** will print both tcp/login traffic and udp/who traffic, and **port domain** will print both tcp/domain and udp/domain traffic).

src port *port* True if the packet has a source port value of *port*.

port *port* True if either the source or destination port of the packet is *port*.

dst portrange *port1-port2* True if the packet is ip/tcp, ip/udp, ip6/tcp or ip6/udp and has a destination port value between *port1* and *port2*. *port1* and *port2* are interpreted in the same fashion as the *port* parameter for **port**.

src portrange ***port1-port2*** True if the packet has a source port value between *port1* and *port2*.

portrange ***port1-port2*** True if either the source or destination port of the packet is between *port1* and *port2*. Any of the above port or port range expressions can be prepended with the keywords, **tcp** or **udp**, as in:

```
tcp src port port
```

which matches only tcp packets whose source port is *port*.

less ***length*** True if the packet has a length less than or equal to *length*. This is equivalent to:

```
len <= length.
```

greater ***length*** True if the packet has a length greater than or equal to *length*. This is equivalent to:

```
len >= length.
```

ip proto ***protocol*** True if the packet is an IPv4 packet (see *ip*(4P)) of protocol type *protocol*. *Protocol* can be a number or one of the names **icmp**, **icmp6**, **igmp**, **igrp**, **pim**, **ah**, **esp**, **vrrp**, **udp**, or **tcp**. Note that the identifiers **tcp**, **udp**, and **icmp** are also keywords and must be escaped via backslash (\), which is \ in the C-shell. Note that this primitive does not chase the protocol header chain.

ip6 proto ***protocol*** True if the packet is an IPv6 packet of protocol type *protocol*. Note that this primitive does not chase the protocol header chain.

ip6 protochain ***protocol*** True if the packet is IPv6 packet, and contains protocol header with type *protocol* in its protocol header chain. For example,

```
ip6 protochain 6
```

matches any IPv6 packet with TCP protocol header in the protocol header chain. The packet may contain, for example, authentication header, routing header, or hop-by-hop option header, between IPv6 header and TCP header. The BPF code emitted by this primitive is complex and cannot be optimized by BPF optimizer code in *tcpdump*, so this can be somewhat slow.

ip protochain ***protocol*** Equivalent to **ip6 protochain** *protocol*, but this is for IPv4.

ether broadcast True if the packet is an Ethernet broadcast packet. The *ether* keyword is optional.

ip broadcast True if the packet is an IPv4 broadcast packet. It checks for both the all-zeroes and all-ones broadcast conventions, and looks up the subnet mask on the interface on which the capture is being done. If the subnet mask of the interface on which the capture is being done is not available, either because the interface on which capture is being done has no netmask or because the capture is being done on the Linux "any" interface, which can capture on more than one interface, this check will not work correctly.

ether multicast True if the packet is an Ethernet multicast packet. The **ether** keyword is optional. This is shorthand for '**ether[0] & 1 != 0**'.

ip multicast True if the packet is an IPv4 multicast packet.

ip6 multicast True if the packet is an IPv6 multicast packet.

ether proto ***protocol*** True if the packet is of ether type *protocol*. *Protocol* can be a number or one of the names **ip**, **ip6**, **arp**, **rarp**, **atalk**, **aarp**, **decnet**, **sca**, **lat**, **mopdl**, **moprc**, **iso**, **stp**, **ipx**, or **netbeui**. Note these identifiers are also keywords and must be escaped via backslash (\). [In the case of FDDI (e.g., '**fddi protocol arp**'), Token Ring (e.g., '**tr protocol arp**'), and IEEE 802.11 wireless LANS (e.g., '**wlan protocol arp**'), for most of those protocols, the protocol identification comes from the 802.2 Logical Link Control (LLC) header, which is usually layered on top of the FDDI, Token Ring, or 802.11 header. When filtering for most protocol identifiers on FDDI, Token Ring, or 802.11, *tcpdump* checks only the protocol ID field of an LLC header in so-called SNAP format with an Organizational Unit Identifier (OUI) of 0x000000, for encapsulated Ethernet; it doesn't check whether the packet is in SNAP format with an OUI of 0x000000. The exceptions are:

iso *tcpdump* checks the DSAP (Destination Service Access Point) and SSAP (Source Service Access Point) fields of the LLC header;

stp and netbeui *tcpdump* checks the DSAP of the LLC header;

atalk *tcpdump* checks for a SNAP-format packet with an OUI of 0x080007 and the AppleTalk etype.

In the case of Ethernet, *tcpdump* checks the Ethernet type field for most of those protocols. The exceptions are:

iso, stp, and netbeui *tcpdump* checks for an 802.3 frame and then checks the LLC header as it does for FDDI, Token Ring, and 802.11;

atalk *tcpdump* checks both for the AppleTalk etype in an Ethernet frame and for a SNAP-format packet as it does for FDDI, Token Ring, and 802.11;

aarp *tcpdump* checks for the AppleTalk ARP etype in either an Ethernet frame or an 802.2 SNAP frame with an OUI of 0x000000;

ipx *tcpdump* checks for the IPX etype in an Ethernet frame, the IPX DSAP in the LLC header, the 802.3-with-no-LLC-header encapsulation of IPX, and the IPX etype in a SNAP frame.

decnet src *host* True if the DECNET source address is *host*, which may be an address of the form ‘‘10.123’‘, or a DECNET host name. [DECNET host name support is only available on ULTRIX systems that are configured to run DECNET.]

decnet dst *host* True if the DECNET destination address is *host*.

decnet host *host* True if either the DECNET source or destination address is *host*.

ifname *interface* True if the packet was logged as coming from the specified interface (applies only to packets logged by OpenBSD’s **pf**(4)).

on *interface* Synonymous with the **ifname** modifier.

rnr *num* True if the packet was logged as matching the specified PF rule number (applies only to packets logged by OpenBSD’s **pf**(4)).

rulenum *num* Synonymous with the **rnr** modifier.

reason *code* True if the packet was logged with the specified PF reason code. The known codes are: **match**, **bad-offset**, **fragment**, **short**, **normalize**, and **memory** (applies only to packets logged by OpenBSD’s **pf**(4)).

rset *name* True if the packet was logged as matching the specified PF ruleset name of an anchored ruleset (applies only to packets logged by **pf**(4)).

ruleset *name* Synonymous with the **rset** modifier.

srnr *num* True if the packet was logged as matching the specified PF rule number of an anchored ruleset (applies only to packets logged by **pf**(4)).

subrulenum *num* Synonymous with the **srnr** modifier.

action *act* True if PF took the specified action when the packet was logged. Known actions are: **pass** and **block** (applies only to packets logged by OpenBSD’s **pf**(4)).

ip, ip6, arp, rarp, atalk, aarp, decnet,iso**, stp, ipx, *netbeui*** Abbreviations for:

```
ether proto p
```

where *p* is one of the above protocols.

lat, moprc, mopdl Abbreviations for:

```
ether proto p
```

where *p* is one of the above protocols. Note that *tcpdump* does not currently know how to parse these protocols.

vlan ***[vlan_id]*** True if the packet is an IEEE 802.1Q VLAN packet. If *[vlan_id]* is specified, only true if the packet has the specified *vlan_id*. Note that the first **vlan** keyword encountered in *expression* changes the decoding offsets for the remainder of *expression* on the assumption that the packet is a VLAN packet. The **vlan** *[vlan_id]* expression may be used more than once, to filter on VLAN hierarchies. Each use of that expression increments the filter offsets by 4. For example:

```
vlan 100 && vlan 200
```

filters on VLAN 200 encapsulated within VLAN 100, and

```
vlan && vlan 300 && ip
```

filters IPv4 protocols encapsulated in VLAN 300 encapsulated within any higher order VLAN.

mpls ***[label_num]*** True if the packet is an MPLS packet. If *[label_num]* is specified, only true is the packet has the specified *label_num*. Note that the first **mpls** keyword encountered in *expression* changes the decoding offsets for the remainder of *expression* on the assumption that the packet is a MPLS-encapsulated IP packet. The **mpls** *[label_num]* expression may be used more than once, to filter on MPLS hierarchies. Each use of that expression increments the filter offsets by 4. For example:

```
mpls 100000 && mpls 1024
```

filters packets with an outer label of 100000 and an inner label of 1024, and

```
mpls && mpls 1024 && host 192.9.200.1
```

filters packets to or from 192.9.200.1 with an inner label of 1024 and any outer label.

pppoed True if the packet is a PPP-over-Ethernet Discovery packet (Ethernet type 0x8863).

pppoes True if the packet is a PPP-over-Ethernet Session packet (Ethernet type 0x8864). Note that the first **pppoes** keyword encountered in *expression* changes the decoding offsets for the remainder of *expression* on the assumption that the packet is a PPPoE session packet. For example:

```
pppoes && ip
```

filters IPv4 protocols encapsulated in PPPoE.

tcp, udp, icmp Abbreviations for:

```
ip proto p or ip6 proto p
```

where *p* is one of the above protocols.

iso proto ***protocol*** True if the packet is an OSI packet of protocol type *protocol*. *Protocol* can be a number or one of the names **clnp**, **esis**, or **isis**.

clnp, esis, isis Abbreviations for:

```
iso proto p
```

where *p* is one of the above protocols.

l1, l2, iih, lsp, snp, csnp, psnp Abbreviations for IS-IS PDU types.

vpi ***n*** True if the packet is an ATM packet, for SunATM on Solaris, with a virtual path identifier of *n*.

vci ***n*** True if the packet is an ATM packet, for SunATM on Solaris, with a virtual channel identifier of *n*.

lane True if the packet is an ATM packet, for SunATM on Solaris, and is an ATM LANE packet. Note that the first **lane** keyword encountered in *expression* changes the tests done in the remainder of *expression* on the assumption that the packet is either a LANE emulated Ethernet packet or a LANE LE Control packet. If **lane** isn't specified, the tests are done under the assumption that the packet is an LLC-encapsulated packet.

llc True if the packet is an ATM packet, for SunATM on Solaris, and is an LLC-encapsulated packet.

oamf4s True if the packet is an ATM packet, for SunATM on Solaris, and is a segment OAM F4 flow cell (VPI=0 & VCI=3).

oamf4e True if the packet is an ATM packet, for SunATM on Solaris, and is an end-to-end OAM F4 flow cell (VPI=0 & VCI=4).

oamf4 True if the packet is an ATM packet, for SunATM on Solaris, and is a segment or end-to-end OAM F4 flow cell (VPI=0 & (VCI=3 | VCI=4)).

oam True if the packet is an ATM packet, for SunATM on Solaris, and is a segment or end-to-end OAM F4 flow cell (VPI=0 & (VCI=3 | VCI=4)).

metac True if the packet is an ATM packet, for SunATM on Solaris, and is on a meta signaling circuit (VPI=0 & VCI=1).

bcc True if the packet is an ATM packet, for SunATM on Solaris, and is on a broadcast signaling circuit (VPI=0 & VCI=2).

sc True if the packet is an ATM packet, for SunATM on Solaris, and is on a signaling circuit (VPI=0 & VCI=5).

ilmic True if the packet is an ATM packet, for SunATM on Solaris, and is on an ILMI circuit (VPI=0 & VCI=16).

connectmsg True if the packet is an ATM packet, for SunATM on Solaris, and is on a signaling circuit and is a Q.2931 Setup, Call Proceeding, Connect, Connect Ack, Release, or Release Done message.

metaconnect True if the packet is an ATM packet, for SunATM on Solaris, and is on a meta signaling circuit and is a Q.2931 Setup, Call Proceeding, Connect, Release, or Release Done message.

expr relop expr True if the relation holds, where *relop* is one of >, <, >=, <=, =, !=, and *expr* is an arithmetic expression composed of integer constants (expressed in standard C syntax), the normal binary operators [+, -, *, /, &, |, <<, >>], a length operator, and special packet data accessors. Note that all comparisons are unsigned, so that, for example, 0x80000000 and 0xffffffff are > 0. To access data inside the packet, use the following syntax:

```
proto [ expr : size ]
```

Proto is one of **ether, fddi, tr, wlan, ppp, slip, link, ip, arp, rarp, tcp, udp, icmp, ip6** or **radio**, and indicates the protocol layer for the index operation. (**ether, fddi, wlan, tr, ppp, slip** and **link** all refer to the link layer. **radio** refers to the "radio header" added to some 802.11 captures.) Note that *tcp, udp* and other upper-layer protocol types only apply to IPv4, not IPv6 (this will be fixed in the future). The byte offset, relative to the indicated protocol layer, is given by *expr*. *Size* is optional and indicates the number of bytes in the field of interest; it can be either one, two, or four, and defaults to one. The length operator, indicated by the keyword **len**, gives the length of the packet.

For example, '**ether[0] & 1 != 0**' catches all multicast traffic. The expression '**ip[0] & 0xf != 5**' catches all IPv4 packets with options. The expression '**ip[6:2] & 0x1fff = 0**' catches only unfragmented IPv4 datagrams and frag zero of fragmented IPv4 datagrams. This check is implicitly applied to the **tcp** and **udp** index operations. For instance, **tcp[0]** always means the first byte of the TCP *header*, and never means the first byte of an intervening fragment.

Some offsets and field values may be expressed as names rather than as numeric values. The following protocol header field offsets are available: **icmptype** (ICMP type field), **icmpcode** (ICMP code field), and **tcpflags** (TCP flags field).

The following ICMP type field values are available: **icmp-echoreply**, **icmp-unreach**, **icmp-sourcequench**, **icmp-redirect**, **icmp-echo**, **icmp-routeradvert**, **icmp-routersolicit**, **icmp-timxceed**, **icmp-paramprob**, **icmp-tstamp**, **icmp-tstampreply**, **icmp-ireq**, **icmp-ireqreply**, **icmp-maskreq**, **icmp-maskreply**.

The following TCP flags field values are available: **tcp-fin**, **tcp-syn**, **tcp-rst**, **tcp-push**, **tcp-ack**, **tcp-urg**.

Primitives may be combined using:

A parenthesized group of primitives and operators (parentheses are special to the Shell and must be escaped). Negation ('**!**' or '**not**'). Concatenation ('**&&**' or '**and**'). Alternation ('**||**' or '**or**').

Negation has highest precedence. Alternation and concatenation have equal precedence and associate left to right. Note that explicit **and** tokens, not juxtaposition, are now required for concatenation.

If an identifier is given without a keyword, the most recent keyword is assumed. For example,

```
not host vs and ace
```

is short for

```
not host vs and host ace
```

which should not be confused with

```
not ( host vs or ace )
```

Expression arguments can be passed to *tcpdump* as either a single argument or as multiple arguments, whichever is more convenient. Generally, if the expression contains Shell metacharacters, it is easier to pass it as a single, quoted argument. Multiple arguments are concatenated with spaces before being parsed.

EXAMPLES

To capture all packets arriving at or departing from *sundown*:

```
host sundown
```

To capture traffic between *helios* and either *hot* or *ace*:

```
host helios and \( hot or ace \)
```

To capture all IP packets between *ace* and any host except *helios*:

```
ip host ace and not helios
```

To capture all traffic between local hosts and hosts at Berkeley:

```
net ucb-ether
```

To capture all ftp traffic through internet gateway *snup*: (note that the expression is quoted to prevent the shell from (mis-)interpreting the parentheses):

```
gateway snup and (port ftp or ftp-data)
```

To capture traffic neither sourced from nor destined for local hosts (if you gateway to one other net, this stuff should never make it onto your local net).

```
ip and not net localnet
```

To capture the start and end packets (the SYN and FIN packets) of each TCP conversation that involves a non-local host.

```
tcp[tcpflags] & (tcp-syn|tcp-fin) != 0 and not src and dst net localnet
```

To capture all IPv4 HTTP packets to and from port 80, i.e. print only packets that contain data, not, for example, SYN and FIN packets and ACK-only packets. (IPv6 is left as an exercise for the reader.)

```
tcp port 80 and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0)
```

To capture IP packets longer than 576 bytes sent through gateway *snup*:

```
gateway snup and ip[2:2] > 576
```

To capture IP broadcast or multicast packets that were *not* sent via Ethernet broadcast or multicast:

```
ether[0] & 1 = 0 and ip[16] >= 224
```

To capture all ICMP packets that are not echo requests/replies (i.e., not ping packets):

```
icmp[icmptype] != icmp-echo and icmp[icmptype] != icmp-echoreply
```

*This was taken from the man page of ‘tcpdump <http://www.tcpdump.org/>‘*__.