

Proposals for packet capture changes to capture VLAN tagged packets

1 Overview

Packet capture on Solaris involves use of the snoop command and two kernel modules, bufmod and pfmmod. When administrators want to watch network traffic or troubleshoot networking problems, they execute the snoop command. Customers have reported that snoop is unable to capture VLAN tagged packets, and that snoop has poor performance. The VLAN problems are the primary reasons for this paper.

2 How packet capturing works on Solaris

The packet capturing subsystem on Solaris consists of snoop, a command line user-space tool, and two kernel modules, bufmod and pfmmod. Snoop is the only piece the user will directly interact with. It will take a user specified packet filtering expression and converts it into a packet filtering program, and snoop displays or writes to a file any packet that is captured by the filter. Bufmod is a kernel module responsible for buffering up data for a reader or a writer. Pfmmod is a strict implementation of CMU/Stanford Packet Filter (CSPF) [1] and it is responsible for executing filter programs to filter out unwanted packets. Snoop also has an implementation of CSPF, but snoop's implementation contains features that pfmmod's implementation does not. Snoop interacts with bufmod and pfmmod via the STREAMS interface. Snoop and pfmmod are the two pieces of interest.

CSPF is a language for a stack machine with no storage beyond a stack. The only operations are push, pop, comparison of the top two elements of the stack (pop the top two elements, compare them, and push the result), and some short circuit comparisons (if the comparison evaluates a certain way, push the result on the stack and end the program). When the program ends, its status is on the top of the stack. There is no indirect addressing, addressing is by hard coded constants only, and there are no branch operations. As stated earlier, pfmmod's implementation of CSPF is a strict implementation, there is nothing extra in it. Snoop's implementation of CSPF adds forward branching and a simplistic attempt at indirect addressing. Snoop's indirect addressing is simplistic because it added opcodes to CSPF to find the end of a header (i.e. OFFSET_IP means find the end of the IP header) and then all further operations are based on that address.

When the user runs snoop he gives it options (such as write to a file, or read from a file), an optional device name to read data from, and a filter expression. Example filter expressions are:

```
tcp and port 80
host 1.1.1.1
src 1.1.1.1 and dst 2.2.2.2
arp and host 1.1.1.1
```

These expressions mean:

```
Return only TCP packets going to or from port 80
```

Return any packets going to or from host 1.1.1.1
Return any packets whose source is 1.1.1.1 and destination is 2.2.2.2
Return only ARP packets to or from host 1.1.1.1

Snoop opens the device the user wants to capture packets on, creates a stream, and pushes pfmmod and bufmmod onto the stream.

Snoop takes the user specified expression and converts it to CSPF. If the program is sufficiently simple, then snoop passes the program to pfmmod for which executes it on each packet it receives. Otherwise, snoop will generate a CSPF program taking advantage of snoop's CSPF modifications and snoop will execute the program on each packet received. Snoop does have optimizations where it will try and break filter expressions up into a kernel space and a user space piece.

Several factors are used to determine if a program is not simple enough for pfmmod. The interface to pfmmod imposes a maximum size of a program. Larger programs must be executed in user space by snoop. Certain filter expressions require variable offsets. For example, the filter "tcp and port 80" cannot be expressed in a strict CSPF language because IP headers have variable length.

The packet filtering program is executed, either by snoop or pfmmod, on each packet received by the interface snoop opened. If the packet filter program generates a match, then the packet is captured and displayed to the user. Otherwise, nothing happens. Note that packet capture does not interfere with or modify the packet.

3 Analysis of the problem

3.1 Capturing packets with variable sized headers

Snoop and pfmmod do not correctly handle packets with variable sized headers, such as VLAN tagged packets and 802.3 frames. Some packets with variable sized headers must be filtered entirely in user space, adversely affecting performance.

Snoop does not generate code to check the ethernet header for a VLAN tag. Snoop assumes that the ethernet header is always a fixed size. If this packet has a VLAN tag, the ethernet header is bigger than a normal ethernet header. Therefore when filtering a VLAN tagged packet, then the offsets in the program are pointing to the incorrect position.

Note that we have a similar problem with tunneled packets. Tunneled packets change the packet from the usual structure of link header (usually ethernet), followed by network header (usually IP), followed by transport (usually either TCP or UDP).

Snoop does not handle dynamic packet structure beyond a few simple cases (such as the IP header, which can have a variable size), and those cases must be handled in

user space. It is a long term goal to be able to filter on tunneled packets.

The fact that snoop cannot capture VLAN tagged packets is the primary reason for this paper. The Clearview project has requirements to modify snoop so that it can filter on packets with dynamic headers.

3.2 Displaying VLAN tagged packets

Snoop has no code to display VLAN tagged packets. Work has been done internally to update snoop to print out a VLAN tagged packet if it receives one. There is also work going on now to ensure the network card driver does not strip out a VLAN tag before it passes the packet to snoop or pfmmod.

3.3 No snoop test suite

There is no snoop test suite. There is no requirement to create one for the Clearview project. However, the lack of a test suite will have a negative impact on work done to snoop for Clearview because there is no easy way to verify that changes made during Clearview work do not break snoop.

3.4 Performance

Customers have complained about a large number of dropped packets while using snoop in real network environments. Any changes to snoop or the underlying components cannot make the problem worse. Any change that improves performance would be an extra, however, there are no requirements for Clearview to improve packet filtering performance.

4 Options

Time estimates assume one engineer working without interruptions on the task. The estimates are uncertain. More detailed design work is required before a more accurate estimate can be developed.

4.1 Implement tcpdump [3] style VLAN capture

Current versions of tcpdump have some VLAN capture capability in the form of a vlan filter expression [4]. If vlan is specified as part of a packet filter, then the assumption is that all expressions logically and-ed with the vlan expression are to be applied only on packets that have a VLAN tag. If the tag is not specified, then it is assumed that the user does not want to filter on VLAN tagged packets. For example, if using tcpdump a user specified the following filter expression:

```
tcpdump vlan and port 22
```

Then the output of the filter expression will be all VLAN tagged packets going to or from port 22. Packets that do not have a VLAN tag will not be matched regardless of their source and destination port. The following command:

`tcpdump port 22`

Will capture only packets going to or from port 22 that are not VLAN tagged.

If the user wants to filter on port 22 whether or not the packets are VLAN tagged, then the user must do the following:

`tcpdump port 22 or (vlan and port 22)`

This proposal is the simplest to implement and should take one to two weeks to implement. It requires no changes to `pfmod`, only changes to how snoop generates offsets for its program, and changes to how snoop parses a filter expression. If snoop detects a `vlan` expression, then any expression AND-ed with the `vlan` expression will use a different offset to the network layer header.

Performance when the user does not want to filter on VLAN tags will be unaffected, snoop will work the same way it does today. If the user wants to filter on VLAN tags, then there will be a small performance degradation as snoop must do more work than it does now.

Testing consists of simply verifying that snoop does not generate code to filter VLAN tagged packets when a `vlan` tag is not present in the user specified command line expression. If a tag is specified, then testing consists of verifying that snoop will generate the correct code to detect the VLAN tag and use the correct offset into the packet.

However, this option does not fit in well with the Clearview framework. If an administrator wants to capture traffic going through a device, why should he have to know that there is VLAN tagged traffic going through an interface in order for him to generate a filter expression that will get him the traffic he wants to see? This also does not solve the problem of capturing tunneled traffic or other packets with dynamic structure.

4.2 Add address indirection to `pfmod`

We could add address indirection to `pfmod`. Address indirection would allow `pfmod` to handle packets with variable length headers. Snoop would generate code to check every packet for a VLAN tag, and then generate the appropriate offsets. If the user entered the expression “port 80”, then any packet, whether or not it has a VLAN tag, to or from port 80 will match this expression.

`Pfmod`'s language does not have any storage beyond its stack, so to implement address indirection we will need to add storage. Snoop will change to generate code to determine if a packet is VLAN tagged, and then store an appropriate offset in `pfmod`'s storage. When `pfmod` needs to access data in the packet, it will use the generated offset.

This option will involve changing `pfmod`. The change can be designed in a way

that it should not affect any other users of pfmod (such as the dhcp agent and dhcpd, plus possible third party users). However, extra testing should be done to insure that other users of pfmod are not adversely affected by the change. Performance impact should be minimal, but will not improve performance.

Modifying pfmod to use address indirection and snoop to take advantage of the new pfmod should take one to two months. We can take advantage of the fact that snoop already has an implementation of a modified CSPF with indirection, though the implementation is limited as described earlier. The problem with snoop's implementation is that it is tightly coupled with the code to generate a user space packet filtering program. Ideally, these two pieces should be decoupled and the modified CSPF implementation put in a common area where both pfmod and snoop can share it.

4.3 Integrate Berkeley Packet Filter

The Berkeley Packet Filter (BPF) [2] was developed because of deficiencies in CSPF, such as a lack of flexibility and performance problems. BPF added storage, address indirection, forward branches, and a new machine model. CSPF is built around a stack machine, and BPF emulates a stripped down RISC processor.

This will require a new kernel module. The current pfmod could be re-written to implement BPF, but then current users of pfmod will have to change, increasing the amount of work required to implement the change. There are components (dhcpagent is an example) that use pfmod, so we will not remove pfmod. Writing the new kernel module might not take a large amount of time or resources because the source code for BPF is available on the web from the Lawrence-Berkeley Laboratory. It is possible we will have to decouple code for generating a BPF program and code to execute a BPF program. BPF is available under a BSD license, which should allow us to use the code and modify it. However, if there are any legal issues that would prevent us from using that code, then we would have to do a clean-room implementation of BPF. Research shows that this is how BPF found its way into Linux [5].

Once we have the new kernel module, then snoop must change to generate programs in the new code. Snoop should also change to use BPF in user space for when snoop must filter the packets, for example, when the packet filtering program is too large for the kernel to process, when processing a packet capture file, or when dealing with tunneled packets. Since BPF is much more flexible than CSPF, some filters that cannot be expressed in strict CSPF (such as filtering on a TCP or a UDP port) can be expressed in BPF and executed in the kernel, improving packet filtering performance.

To support VLAN filtering in snoop once BPF is integrated, snoop will generate code to check if a packet is VLAN tagged, and if so use a different offset.

Assuming that we can use the BPF code available on the web and that we write a new kernel module, this route should take a few months. The most difficult parts would be changing snoop's parsing code to support the new language and testing. The major component of testing will be to verify that each command line expression is correctly parsed and translated into BPF code. Since there is no test suite for snoop, this will be a difficult and time consuming task. Snoop's user space filtering implementation should switch to BPF, ideally using code shared between the kernel and user space.

Theoretically, performance will greatly improve with BPF. BPF uses smaller programs than CSPF for the same filtering expression. More filter expressions can be expressed in a program that can be executed in the kernel. Because BPF instructions are more closely mapped to real machine instructions, BPF instructions are usually executed faster than CSPF instructions. Without having profiling data for the current packet filtering implementation, it is difficult to say whether or not BPF will result in a noticeable empirical performance improvement.

4.4 Develop an entirely new language

Alternatively, we could develop an entirely new language for packet filtering. This is the most time consuming of all the options. We will have to develop the new language and then implement it. This gives us the most flexibility, but at the cost of time. We will avoid any possible legal issues from using the BPF source code.

Given the amount of work already spent on developing BPF, if we were to decide to get rid of CSPF and pfm for use by snoop, then I would advise that we do not spend the time developing an entirely new language but instead use BPF.

4.5 Do more filtering in user space

We could do more filtering in user space where we already have a modified CSPF that can handle variable size packet headers. With a simple code change, snoop could do all of its filtering in user space and the user space code can be changed to check for a VLAN tag. If the tag is present, then it will use different offsets.

The disadvantage of this approach is performance. User space filtering is generally slower than kernel space filtering. We stand to increase the number of dropped packets. Before we get involved with this solution, we should run timing trials to see how well user space filtering can keep up with data. This proposal has the advantage of not requiring a change in pfm.

We could mitigate the performance problem by generating two filters. One would go into kernel space, and would only match those packets that could be packets that the user is interested in. Then in user space we narrow that set of packets down to only those the user is interested in. For example, the user wants to see packets that match "tcp and port 80". The first set of filters would only match IP packets or any

other packet that could encapsulate TCP. Then the second set of filters would find the TCP header, if one exists, otherwise the filter would drop the packet. Then it would check the port field. The first set of filters would run in the kernel space and would eliminate packets such as ARP packets. This would be very difficult to implement in the general case.

5 Recommendations

I recommend that we create a BPF module called `bpfmod` and change `snoop` to use this module instead of `pfmod`. BPF was designed to be more flexible and faster than CSPF. BPF's source code is available, reducing the amount of work.

We could further improve CSPF. Someone started that by adding address indirection and branching to `snoop`'s user level packet filtering code. While this solution would fix the immediate problem, I do not think that would be a good long term solution to all of the packet filtering problems. As stated in the Usenix paper [2], CSPF is much slower and less efficient than BPF. CSPF programs are larger, and adding the necessary features to support filtering on more modern packets will cause `snoop` to generate even larger programs. This would increase the chances that a program will be too big for `pfmod`, requiring it to be executed in user space, which means that `snoop`'s performance will degrade further. CSPF is also not an easy language to work with. It was designed for PDP-11 type machines, when modern machines are not stack based. Modern developers will have more experience with the machines that formed the basis for BPF, and will find BPF easier to work with. I believe the long term health of the packet capture subsystem of Solaris is best served by converting it to use BPF.

A short term solution if the recommended solution takes too long would be to add indirection to `pfmod`, and then in parallel work on the longer term solution of moving to BPF.

6 References

1. Paper describing CSPF
<http://www.hp.com/techreports/Compaq-DEC/WRL-87-2.pdf>
2. Usenix paper on BPF
<http://www.tcpdump.org/papers/bpf-usenix93.pdf>
3. Tcpdump
<http://www.tcpdump.org>
4. Information on how to filter on VLANs with tcpdump
<http://www.tcpdump.org/lists/workers/2003/12/msg00009.html>
5. An analysis of SCO's Las Vegas slide show presentation that mentions the heritage of Linux's packet filtering mechanism

<http://www.perens.com/SCO/SCOSlideShow.html>