Interested in learning more about security?

SANS Institute
InfoSec Reading Room

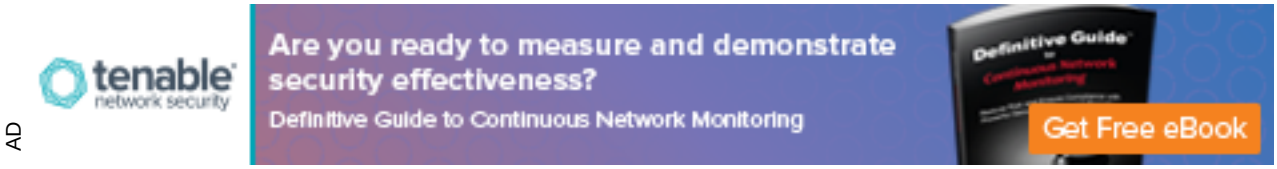
This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

IPFilter: A Unix Host-Based Firewall

With the advent of TCP wrappers and dedicated firewalling hardware, host-based firewall packages for unix operating systems have fallen by the wayside. Daemons such as inetd, xinetd, and tcpd allow hosts to effectively limit outside connections to an out-of-the-box unix distribution, and as such, many users seldom consider using a third party firewall package. IPFilter is one such host-based firewall. It provides several useful security features which are lacking in stock unix installs, such as the ability to filter eg...

Copyright SANS Institute
Author Retains Full Rights

AD



IPFilter: A Unix Host-Based Firewall
GSEC Practical Assignment v.1.4 option 1
Dana Price
June 1st, 2002

Abstract

With the advent of TCP wrappers and dedicated firewalling hardware, host-based firewall packages for unix operating systems have fallen by the wayside. Daemons such as inetd, xinetd, and tcpd allow hosts to effectively limit outside connections to an out-of-the-box unix distribution, and as such, many users seldom consider using a third party firewall package. IPFilter is one such host-based firewall. It provides several useful security features which are lacking in stock unix installs, such as the ability to filter egressing traffic, protocol/packet state filtering, and true stateful firewalling. This paper will explain the benefits of using IPFilter on a unix host by detailing its configuration and implementation on a Solaris 8 SPARC box, and providing examples users can follow to safeguard their machines against some of the more popular remote exploits.

Background

First, a little background. What is a host-based firewall and why would a user need one?

A firewall can be defined as “a system that is designed to prevent unauthorized access to private computers or networks.”(1) It is a hardware or software device which filters network traffic based on a set of rules. The firewall examines each packet received and determines whether there are any characteristics of this data matching the ruleset by which it is disallowed. If there is a match (or lack thereof), the packet can be barred from passing on. In short, users can protect their machines from unwanted (or malicious) network traffic.

The first thing a potential hacker wants is information on the target machine or network. Logic would have it that preventing them from getting this information is a huge step in securing an infrastructure. A firewall does exactly this using the methods described above, disallowing the traffic that represents the 'prying eyes' of a hacker. So where does the 'host-based' part come in?

Where some firewall devices protect specific access points to a network, a host-based firewall protects only a single machine or host on the network. This may seem like a drawback until one realizes that good network security comes in layers, and that the combination of a host-based and network-based firewalling structure can provide a very strong multi-tiered defense. There are an abundance of windows host-based firewalls such as ZoneAlarm, BlackIce, etc. which I will not cover. What about those users with unix workstations as opposed to a windows box? They have a very strong option in IPFilter.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Notes on Installation

IPFilter has been tested on many flavors of unix, including but not limited to Solaris 2.3-9, earlier versions of SunOS, NetBSD, FreeBSD, IRIX, and HP-UX. Since I am discussing a Solaris 8 implementation, I will discuss installation on that platform only (which is fairly straightforward). Pre-compiled binaries can be obtained at
http://www.maraudingpirates.org/ipfilter/archive/ipf-3.4.28-Sol8-sparc-64bit.pkg.gz
Note that IPFilter resides as a kernel driver, and Solaris versions greater than 7 require kernel drivers to be compiled 64-bit. Since gcc only recently became able to generate 64-bit code, users must either a.) compile the IPFilter source using the Sun C compiler, or b.) use the pre-compiled binaries above, which is much easier, and the path I chose.

After un-gzipping the file, you are left with a standard Solaris .pkg package. Use pkgadd –d to install it:

```
% gzip –d ipf-3.4.28-Sol8-sparc-64bit.pkg.gz
% pkgadd –d ipf-3.4.28-Sol8-sparc-64bit.pkg
```

You will be prompted with two sub-packages to install:

```
The following packages are available:
  1  ipf      IP Filter
           (sparc) 3.4.28
  2  ipfx     IP Filter (64-bit)
           (sparc) 3.4.28
```

Since the ipf package contains the post-install script that loads the drivers contained by the ipfx package, ipfx must be installed FIRST followed by ipf. The /usr/local/ipf and /etc/opt/ipf trees are created by the script, as well as an /etc/init.d/ipfboot script to start the daemon at boot time. Link the script into /etc/rc2.d:

```
/etc/rc2.d% ln –s ../init.d/ipfboot S98ipfboot
```

A reboot will ensure the kernel module is loaded and the daemon started. The following messages at boot will certify this:

```
ipf: [ID 920137 kern.notice] IP Filter: attach to [hme0,0] - IPv4
ipf: [ID 989912 kern.notice] IP Filter: v3.4.25, attaching complete.
```

Now that IPFilter has been installed, what can we do with it? We can filter network traffic with it. How do we tell it what and how to filter? With a ruleset,

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

which is.. a set of rules. Each rule gives IPFilter some characteristic of the packet to match on, and an action to take when a match is made. This ruleset is stored as a simple text file, with one rule per line. Since the functionality of a firewall (host based or otherwise) is only as good as its ruleset, a large part of this paper will be dedicated to explaining how to get IPFilter to do what we want as a function of the rules we give it. From this point on, I will refer to the file containing the ruleset as the 'config file'. On different platforms, the location of this file may vary. The Solaris pre-packaged distribution looks for it in /etc/opt/ipf/ipf.conf

Each time a change is made to the config file, the IPFilter daemon must be restarted so that it can grab the updated rules. This can be done by reloading the rc script:

```
% /etc/rc2.d/S98ipfboot reload
```

Basic Operations

Let's start simple by adding a single rule to the file. This will give IPFiler a single directive. It is important to mention at this point that the file is read from top to bottom, and that the firewall does NOT stop parsing after the first match is made. This means that if two rules in the file match the same packet, the action which the second rule directs is the one that is carried out. This can lead to rather confusing situations in lengthy config files. With that said, lets go ahead and add one rule to the file:

```
block in all
```

Upon restarting the daemon, users may notice that their machine has just lost all network connectivity. This is a direct result of IPFilter doing its job. As stated before, the overall function of a firewall is to limit access. It limits access by blocking traffic. As such, the 'block' statement will be used quite a bit from here on out. It is of the format
[block/pass] [INcoming/OUTgoing packets] [packet characteristics to filter on]
With that in mind, we can take a look at the rule we implemented above and see that we blocked access to incoming packets of type 'all' (or all packets). This is why network connectivity was lost.. all incoming traffic was being blocked. Let's add a second rule:

```
block in all
pass in all
```

Once in effect, we can see that network connectivity has been restored. As mentioned earlier, if more than 1 rule matches a given packet, the latter of those rules is applied. In this case, both rules match all traffic, but the action dictated by the latter most one (which is to allow the incoming traffic to pass) is applied.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Some machines may have rulesets that are several hundred lines long. Based on what was stated above, one can imagine how tedious it would be to try to troubleshoot and determine if any blocked traffic is unintentionally being allowed to pass by a latter statement. This problem can be solved by using the 'quick' keyword:

block in quick all
pass in all

Once a match is made on a statement containing the 'quick' keyword, IPFilter stops the comparison step and continues on to the next packet.

So what about the next rule?

pass in all

This rule is never encountered. "It could just as easily not be in the config file at all. The sweeping match of all and the terminal keyword quick from the previous rule make certain that no rules are followed afterward."(3)

Nevertheless, the last line should be included in the config file as we are currently operating under an 'allow unless explicitly denied' premise, which is to say that traffic that has not matched any of our filters should eventually be allowed though.

Methods of filtering

In a real world scenario, it is not likely that we would want to block ALL traffic to our machine. Normally, we want to filter what we deem necessary and let the rest pass on by. Let's say that each night our logs constantly show port scans from a host that resolves to 24.24.208.56. Chances are this is a hacker scanning for open ports to gather information on our host/network. It would be wise to deny him this ability. As I stated before, IPFilter can filter traffic based on a myriad of packet characteristics. One of these characteristics is the originating IP address. We can instantly drop every packet coming from his machine henceforth by adding

block in quick from 24.24.208.25/255.255.255.0 to any

to the config file. Note the usage of the 'from/to' keywords. 'from' being the originating host IP and 'to' denoting the destination IP of the traffic (currently set to 'all', we'll revisit this later). Take note as well that the netmask of the remote network must also be present. This will allow us to block a single IP address as above, or an entire network segment, i.e

block in quick from 24.24.0.0/255.255.0.0 to any

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

will block all traffic from the network 24.24.0.0. One can also use the CIDR shorthand when denoting netmasks. The following serves the same purpose as the above:

block in quick from 24.24.0.0/16 to any

Blocking incoming traffic is an important means in securing our machine, but what about outgoing traffic? Let's say a worst case scenario exists in which a hacker was able to compromise the machine and is now attempting a man-in-the middle attack by spoofing a different IP and grabbing network traffic destined for it. Not only does IPFilter examine incoming traffic, but outgoing as well:

pass out quick from 165.230.10.10/255.255.255.0 to any
block out quick from any to any

This rule combination will forward on only outgoing traffic with a source IP address of 165.230.10.10 (i.e, that came from our machine). Traffic with a spoofed IP address will be dropped. The destination address can be anywhere. Notice that the 'in' keyword has been replaced with 'out' (as we are filtering outbound traffic). The rest of the syntax remains unchanged. The second of the two rules will ensure that NO other traffic save for that which matches the first rule is passed on. This is an example of a 'deny unless explicitly allowed' situation. By "spoof-proofing" our box, we have heightened not only our own network security, but that of others around us.

Now it's time to magnify our network traffic even further. I stated earlier that the first thing a hacker wants is information on a target machine. The ever popular port scan attempts a connection to an entire range of tcp/udp ports to determine if any are open. These open ports can then be used in an attempt to gain access to the machine by any one of a number of exploits, the most common being a buffer overflow. This is where the most notorious functionality of IPFilter, and any firewall for that matter, comes in to play. Filtering of specific protocols and ports.
We can filter based on protocol by using the 'proto' keyword:

block in quick proto tcp from any to any

This will serve to block all incoming tcp traffic. This probably isn't a good idea, since all we really wanted to do was stop that hacker from finding our open telnet port (port 23). Taking it one step further, we will filter only traffic on tcp port 23 using the 'port' keyword:

block in quick proto tcp from any to any port = 23

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

No more telnet traffic, but is that really what we wanted? Maybe we still want people from within our local subnet to have telnet access:

pass in quick proto tcp from 165.230.10.0/24 to any port = 23
block in quick proto tcp from any to any port = 23

The first rule allows in tcp port 23 traffic only from our subnet, while the second ensures that no other telnet traffic is allowed by. Remember again that by using the 'quick' keyword, we need to list passing traffic before that which is blocked since the firewall begins anew after a match is made. Blocking udp as opposed to tcp can be done by substituting 'udp' for 'tcp' in the above, or with 'tcp/udp' to denote both protocols. It is in this manner that one can pick and choose which ports can be accessed, and by whom.

Logging

'By whom' brings us to another very important feature of any good firewall.. logging. We can gain invaluable information by being able to see exactly who has been making connections to our machine, which port they were attempting to connect to, and other major characteristics of the traffic coming from the remote machine. IPFilter provides the ability to log all of this and more to several facilities. IPFilter by default sends its log data to the device /dev/ipl. It can be converted into human readable form and analyzed with the 'ipmon' utility provided with the standard distribution. Ipmon can be set to pipe logged data to screen, a text file, or the syslog service. The precompiled Solaris package, by default, loads an instance of ipmon at startup which logs all ipfilter messages to the syslog local0 facility, so little needs to be done here to begin viewing your logs. The 'log' keyword must be inserted into each rule for which you wish to see log information for. When inserted into a block statement, logs will be generated for each packet the firewall drops, as is the opposite for a pass statement, i.e;

pass in log quick proto tcp from 165.230.10.0/24 to any port = 23

will log connections on port 23 from machines on the designated subnet.

Advanced Features; Securing Other Common Vulnerabilities

The features I've discussed up until this point are not necessarily unique to Ipfilter, or to firewalls in general. Many out-of-the-box unix distributions come with tcp wrappers and/or daemons such as tcpd. These utilities are able to filter and log very efficiently based on IP access lists for different ports/services. I'll now discuss a few common vulnerabilities that are not as easily circumvented in a stock unix distribution, and how to use more advanced features of Ipfilter to secure against them.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

A well known range of exploits which can be easily evaded by use of IPFilter are the denial of service (DoS) attack. A DoS attack is a somewhat generalized term which encompasses several types of attacks, all of which prevent legitimate network users from being able to access that particular networks resources. The CERT Coordination center lists some subsets of DoS attacks in the form of ICMP echo bandwidth consumption, IP fragmenting, and SYN flooding. We will first deal with ICMP echoes.

> "An intruder may also be able to consume all the available bandwidth on your network by generating a large number of packets directed to your network. Typically, these packets are ICMP ECHO packets, but in principle they may be anything. Further, the intruder need not be operating from a single machine; he may be able to coordinate or co-opt several machines on different networks to achieve the same effect." (2)

The IPFilter howto addresses this topic directly by stating that "Denial of Service attacks are as rampant as buffer overflow exploits. Many denial of service attacks rely on glitches in the OS's TCP/IP stack. Frequently, this has come in the form of ICMP packets. Why not block them entirely?"(3)

We can block ICMP packets by again using the protocol specific filtering functionality of IPFilter, this time filtering on ICMP as opposed to tcp:

```
block in quick proto icmp from any to any
```

The use of the 'proto icmp' keyword here should be self explanatory by now. In a real world situation, just as with tcp, one may not want to block ALL ICMP traffic, since some of it can be useful. Utilities such as ping and traceroute can help diagnose network connectivity problems, etc. The ICMP protocol itself has different forms, ping being of type 0 and traceroute of type 11. What if we wanted to allow only these two variants of ICMP through the firewall? Ipfilter can differentiate between them:

```
pass in quick proto icmp from any to any icmp-type 0
pass in quick proto icmp from any to any icmp-type 11
block in quick proto icmp from any to any
```

The use of the 'icmp-type' keywords make sure that ping and traceroute will still function to and from our machine, while the third rule functions as an implicit deny to make sure all other ICMP traffic is blocked (recall that CERT stated coordinated attacks from several different machines were common).

Now let's deal with IP fragments. IP fragments are, as defined by SANS as

> "a certain type of IP packets that are not sent at once but in multiple parts. The destination or target system has to reassemble the pieces into an IP packet. There are legitimate reasons why fragmentation can (and must) occur. One example of the legitimate uses of IP fragments is for a router

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

> that connects networks with different MTU's. It has no choice but to create IP fragments…"(4)

These packets can be just as malicious as they are legitimate, however. Several very popular exploits have evolved around the requirement of the target machine to reconstruct these fragments into a valid IP datagram, the intricacies of which are given by Microsoft:

> "This vulnerability results because of a flaw in the way the affected systems perform IP fragment reassembly. If a stream of IP fragments with a particular type of malformation are directed against an affected machine, the work factor associated with performing IP fragment reassembly can be driven arbitrarily high by varying the data rate at which the fragments are sent. This could allow a malicious user to consume most or all of the machine's CPU availability. "(5)

This sounds like something we would want to prevent. One approach would be to block all IP fragments. Ipfilter can easily do so with the following rule:

block in all with frag

But I stated above that IP fragments are a useful, if not integral, aspect of networking. Luckily, those most *malicious* fragments all possess one common characteristic. They are simply too short to contain even the headers for fragment reassembly. Ipfilter has accounted for this, and we can narrow our rule to accommodate only these short fragments:

block in all with short

This will offer protection within networks utilizing IP fragments.

The last type of DoS attack I would like to discuss is the SYN flood. First, a brief explanation. Each TCP packet that is responsible for creating a new connection with a remote host will have its SYN flag set. The remote host then does two critical things, it answers back with the necessary packet to continue opening the connection, and it creates a data structure within system memory for that particular connection.

> "The potential for abuse arises at the point where the server system has sent an acknowledgment (SYN-ACK) back to client but has not yet received the ACK message. This is what we mean by half-open connection. The server has built in its system memory a data structure describing all pending connections. This data structure is of finite size, and it can be made to overflow by intentionally creating too many partially-open connections."(6)

Like the packet fragmentation exploit, SYN floods are not meant to consume bandwidth, rather to require massive amounts of CPU usage, which eventually

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

causes the target machine to crash. With that being said, how can we safeguard ourselves? TCP flags are yet another packet characteristic that Ipfilter is able to discern and filter. It is a simple measure to never again see another SYN packet:

block in quick proto tcp from any to any flags S/SA

All that needed to be appended to our stock 'block' rule was 'flags S/SA', which filter those packets with only the SYN flag set.

> "The flags after the / represent the TCP flag mask, indicating which bits of the TCP flags you are interested in checking. When using the SYN bit in a check, you SHOULD specify a mask to ensure that your filter CANNOT be defeated by a packet with SYN and URG flags, for example, set (to Unix, this is the same as a plain SYN)"(7)

Aside from protecting against SYN floods, this rule also prevents any machine from ever creating a tcp connection to our machine, period. If our machine is the one *sending* the SYN packet, however, a session can be successfully opened since the rule applies only to incoming packets. This effectively means that connection requests must come from our machine, which may be a very good policy in the case of a client only, or a not-so-very-effective policy should we be discussing a machine running remote services, or a server. It is the only realistic way to prevent something such as a SYN flood with IPfilter, however, and comes at an expense. Discussing server firewalling in this manner provides an interesting segway into my next topic.

In that last example, it was shown that it could be somewhat difficult to secure a host which does in fact need to serve remote services. What solutions exist for those machines that perform such tasks? Creating a ruleset for them can be an extremely tedious and lengthy task when we look at the different ports they listen on and types of traffic that each should receive. The fact is that Ipfilter offers a solution which allows our host to serve data, blocks rogue network traffic, and provides for a simple rule set. It is called *stateful firewalling*.

The stateful firewalling concept is quite simple. Established connections (or those that have completed the TCP handshake) are allowed to transmit data completely free of examination by Ipfilter. In other words, once the start of the conversation has been validated, the firewall no longer examines the traffic comprising the middle or end. It is assumed to be legitimate data, "rather than just arbitrary TCP packets which can be used to perform 'stealth scanning'." (8)

Say for example we have an ftp server we want to secure. The machine should do nothing save for serve ftp. We can ensure this is the case with the following:

Block in proto tcp all
Block out proto tcp all
pass in quick proto tcp from any to 154.230.171.5 port = 21 keep state

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Note the use of the 'keep state' keyword added to the third rule, it makes our lives much easier. The first two rules blocks *all* tcp traffic, ingressing and egressing. We didn't use the 'quick' keyword here, however, which means that Ipfilter will continue down the list and apply other rules that match. Should the packet in question be an ftp request coming in on port 21, a match is made on the third rule and an entry is created in the Ipfilter state table for this connection. From here on out, packets flowing across this connection are ignored entirely by the firewall. This remains in effect until the connection is terminated. What we have done here is make the machine completely invisible to the network, save for a single port. Apply this model to a machine with multiple network services and the benefits are immediately seen. One need not fret over creating rules to block would-be attackers "because there's no need to track down what ports we're listening to, only the ports we want people to be able to get to." (3)

Apply the same logic to securing a client, or other machine that does not run any services:

```
block in quick all
pass out quick proto tcp from 165.230.10.10 to any keep state
```

The first rule blocks *all incoming* traffic, period. The second rule matches *all outgoing* traffic, passes it, and applies the 'keep state' directive.
This means that all tcp connections can only be initiated from our machine, and once they are initiated they are entered into the state table and ignored by the firewall. I'll refer to our machine in this situation as a 'true client' in that absolutely no traffic is allowed past the kernel unless the session was first opened by the machine itself. This host has complete access to the internet, however the internet has virtually no access to the host itself. In a system where access limitations and security go hand in hand, this is an extremely useful capability.

Conclusion

Through discussing the basic and extended features of IPFilter, it becomes apparent that this host-based firewalling package offers benefits above and beyond stock daemons provided with unix operating systems. Its ability to filter outgoing traffic, coupled with its myriad packet characteristic filters allow users to lock down their stations from virtually any form of unwanted network traffic, and avoid several of the most common remote exploits. Using the true stateful firewalling features provides functionality that, until now, was only available from rather expensive netowork firewall devices. Perhaps not to be overlooked as well is the fact that this software and all its features comes absolutely free, which can for any IT budget.

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

References

1. D-Link Web Site, "Frequently Asked Questions" URL: http://www.dlink.com/tech/faq/broadband/di701_3.htm (7/7/02)

2. CERT Coordination Center, "Denial of Service Attacks" June 14, 2001 URL: http://www.cert.org/tech_tips/denial_of_service.html (7/7/02)

3. Conoboy, Brendan and Fichtner, Erik. "IP Filter Based Firewalls HOWTO" March 1, 2002 URL: http://www.obfuscation.org/ipf/ipf-howto.txt (7/9/2002)

4. SANS Institute. "Intrusion Detection FAQ v. 0.91" 1999 URL: http://secinf.net/info/ids/IDFAQ/fragments.htm (7/9/02)

5. Microsoft TechNet. "Microsoft Security Bulletin (MS00-029): Frequently Asked Questions" URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/fq00-029.asp (7/12/2002)

6. CERT Coordination Center, "CERT Advisory CA-1996-21 TCP SYN Flooding and Spoofing Attacks" Nov. 29, 2000 URL: http://www.cert.org/advisories/CA-1996-21.html (7/10/02)

7. Reed, Darren. "IP Filter Examples". URL: http://coombs.anu.edu.au/~avalon/examples.html#packetstate (7/10/02)

8. Reed, Darren. "IP Filter FAQ". URL: http://coombs.anu.edu.au/~avalon/faq/IPFques.html#1 (7/10/02)

© SANS Institute 2002, As part of the Information Security Reading Room. Author retains full rights.

Last Updated: September 2nd, 2015

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Crystal City 2015	Crystal City, VAUS	Sep 08, 2015 - Sep 13, 2015	Live Event
SANS Network Security 2015	Las Vegas, NVUS	Sep 12, 2015 - Sep 21, 2015	Live Event
SANS Seoul 2015	Seoul, KR	Sep 14, 2015 - Sep 19, 2015	Live Event
Data Breach Investigation Summit & Training	Dallas, TXUS	Sep 21, 2015 - Sep 26, 2015	Live Event
SANS Baltimore 2015	Baltimore, MDUS	Sep 21, 2015 - Sep 26, 2015	Live Event
SANS Perth 2015	Perth, AU	Sep 21, 2015 - Sep 26, 2015	Live Event
SANS Tallinn 2015	Tallinn, EE	Sep 21, 2015 - Sep 26, 2015	Live Event
SANS ICS Amsterdam 2015	Amsterdam, NL	Sep 22, 2015 - Sep 28, 2015	Live Event
SANS Bangalore 2015	Bangalore, IN	Sep 28, 2015 - Oct 17, 2015	Live Event
SANS DFIR Prague 2015	Prague, CZ	Oct 05, 2015 - Oct 17, 2015	Live Event
SANS Seattle 2015	Seattle, WAUS	Oct 05, 2015 - Oct 10, 2015	Live Event
SANS Tysons Corner 2015	Tysons Corner, VAUS	Oct 12, 2015 - Oct 17, 2015	Live Event
SOS: SANS October Singapore 2015	Singapore, SG	Oct 12, 2015 - Oct 24, 2015	Live Event
SANS Gulf Region 2015	Dubai, AE	Oct 17, 2015 - Oct 29, 2015	Live Event
SANS Cyber Defense San Diego 2015	San Diego, CAUS	Oct 19, 2015 - Oct 24, 2015	Live Event
SANS Tokyo Autumn 2015	Tokyo, JP	Oct 19, 2015 - Oct 31, 2015	Live Event
SANS South Florida 2015	Fort Lauderdale, FLUS	Nov 09, 2015 - Nov 14, 2015	Live Event
SANS Sydney 2015	Sydney, AU	Nov 09, 2015 - Nov 21, 2015	Live Event
SANS London 2015	London, GB	Nov 14, 2015 - Nov 23, 2015	Live Event
Pen Test Hackfest Summit & Training	Alexandria, VAUS	Nov 16, 2015 - Nov 23, 2015	Live Event
SANS Hyderabad 2015	Hyderabad, IN	Nov 24, 2015 - Dec 04, 2015	Live Event
SANS Cape Town 2015	Cape Town, ZA	Nov 30, 2015 - Dec 05, 2015	Live Event
SANS San Francisco 2015	San Francisco, CAUS	Nov 30, 2015 - Dec 05, 2015	Live Event
SANS Milan 2015	OnlineIT	Sep 07, 2015 - Sep 12, 2015	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced