

PortSentry for Attack Detection, Part One

Ido Dubrawsky 2002-05-15

PortSentry for Attack Detection, Part One

by *Ido Dubrawsky*

last updated May 15, 2002

PortSentry by Psionic Technologies (now a part of [Cisco](#)) is a component of their TriSentry suite of attack detection tools: portsentry, hostsentry, and logsentry. This article is the first of a two-part series that will describe in detail how Portsentry works from both a theoretical and a technical point of view. The second article will discuss installing, configuring, and tailoring PortSentry for individual systems.

Port Sentry - The Basics Theory

The basic theory behind PortSentry operation revolves around the detection of scans on a host and the response that should be implemented in response to those scans. This is one reason why PortSentry should be classified as an "attack detection" tool. A network or system scan serves as a precursor to an attack and possible intrusion. Unless the attacker knows beforehand which ports are open or available on a system, they will initiate a scan to determine what services the host is running. This is where PortSentry comes into play. This tool monitors the TCP and UDP ports on a system and, depending on how the system is configured, will respond appropriately to an identified scan.

PortSentry monitors for both TCP as well as UDP scans and, as of version 2.0, can detect stealth scans such as those produced by [Nmap](#). Some of the scans which it can detect include:

- **Connect scans** - These are full connection scans. The entire TCP three way handshake is completed before being torn down. These types of scans also are the most obvious since the target host may record the fact that a connection from the scanning IP address was made.
- **SYN scans** - Also known as "half-open" scans are one way an attacker can try to enumerate ports on a system in a stealthy manner. These scans only execute the first two steps of the TCP 3-way handshake. The initiating system sends a TCP SYN packets as though it were requesting to open a full connection. The target system responds with a SYN-ACK packet. The initiator then sends a TCP RST (reset) packet back to the target,

thereby closing the connection. The idea here is to prevent the full connection from being established since it may possibly be logged.

- **FIN scans** - These scans use packets with the TCP FIN flag set. Typically FIN packets are only seen during the closing sequence of a connection. Unsolicited FIN packets sent to a closed TCP port should illicit a RST packet from the other side.
- **NULL scans** - NULL scans use packet without any of the TCP flags set. Again, as per RFC 793, this should illicit a RST packet in return.
- **XMAS scans** - XMAS scans have the FIN, URG, and PUSH TCP flags set in the TCP header. Again, these are not technically "normal" packets seen across the internet (or even on a local LAN) and should illicit a RST from a closed port.
- **FULL-XMAS scan** - This scan has all of the TCP flags set (SYN,ACK,RST, FIN,URG,PSH). This type of packet should never be seen on a LAN much less the Internet.
- **UDP scan** - This scan is detected by the presence of multiple UDP packets originating from a single IP address.

If PortSentry cannot identify a scan as one of the above, it has a default case in which it will still trigger on the scan. Many other tools may actually ignore "unidentifiable" scans and let them through: not so with PortSentry, which turns out to be a very nice feature.

One obstacle that PortSentry faces is how to determine whether a packet is part of a scan or whether it is part of normal traffic for a given port. It does this using two methods. First, PortSentry ignores ports that are in use (that is ports with a service bound to them). Second, the user specifies a list of ports for PortSentry to monitor in the configuration file. If a port in the configuration file list also happens to have a service bound, then PortSentry ignores that particular port.

One nice feature in PortSentry is that it is intelligent about how it monitors ports. For protocols like FTP, for which the client opens up ports in the ephemeral range (TCP ports greater than 1024) and the server then connects *back* to the host, PortSentry is able to examine the incoming connection and, if it is destined for one of the ephemeral ports, ignores it. As soon as the connection is terminated, PortSentry monitors that port as before.

Checking for TCP or UDP scans are not the only methods PortSentry uses to detect a scan. If the IP header length is less than 5 (the minimum length specified in [RFC 791](#)), PortSentry will log a message noting this fact. Similarly, if any IP options are set, PortSentry notes that as well. While at first this may seem problematic, the use of options such as source-routing,

record-route, or timestamp may be used by an attacker for reconnaissance information.

Internal Workings

In a very basic way, PortSentry operates by monitoring ports for potentially suspicious packets and then responding if it has determined that a scan has been detected. To do this, PortSentry must maintain state with all of the IP addresses connecting into a host.

PortSentry utilizes an internal state engine in the form of an array of IP addresses that determines whether a host has previously connected to the system. The array is a two-dimensional table consisting of an IP address and a counter. As an attacker scans a target (whether the scan is a sequential port scan or a random port scan) PortSentry checks the array to determine if the attacker's IP has been seen before and, if so, increments the counter. It does this for every port it monitors and detects packets coming from the attacker's IP address. Once the trigger value has been reached, PortSentry reacts to the scan.

The method by which PortSentry reacts to a scan varies depending on what the user selects in the configuration file. PortSentry provides for three ways of preventing an attacker from completing their scan, which shall be discussed in greater detail below. All three are equally valid methods of blocking a host and each method has its risks and its benefits.

Scanning Methods

First, it can insert a route into the host's routing table so that all traffic from the scanning IP is sent to some "bit-bucket". By using the host route tables to re-route traffic from a scanning host to a non-existent IP address or network PortSentry makes the system look like a black hole...traffic goes in, but nothing comes out. One problem with this method is that it increases the routing table size on a host. Normally, these days, hosts are not actively participating in routing network traffic (unless, of course, it has more than one interface and is being used as a router). As the routing table grows more memory is needed to remember the routes.

Theoretically an attacker can forge the source address of packets used in a scan which would cause PortSentry to indiscriminantly add route table entries in response. This would then fill up the system memory and may create performance issues.

Second, PortSentry can be tied into various firewall software packages (as of version 2.0 the firewalls supported by PortSentry are ipfw, ipfilter, ipfwadm, ipchains, and iptables) so that a

firewall rule can be used to block traffic from the scanning IP. With firewalls PortSentry would detect a scan and then add an appropriate rule to the firewall configuration. This would then block the IP address of the scanning host. Again, as with using the route tables to block traffic, this can be abused. A skillful attacker using forged packets can cause PortSentry to firewall off hosts indiscriminately. This could prevent legitimate hosts from reaching available services on the scanned host.

Finally, a TCP wrapper rule for the attacking IP can be added to the system `/etc/hosts.deny` file. Using TCP wrappers PortSentry simply adds the scanning host into the `/etc/hosts.deny` file. This doesn't block a scan, but it does prevent the attacker from connecting from the scanning host to services protected by TCP wrappers on the scanned host. This is not as strong a protection scheme as the use of route table entries or firewall re-configuration; however, it is the least likely method to cause a denial of service by the scanned host on itself.

While it may appear that manipulating the route tables or using the firewall rules to block scans can easily lead to a host blocking all traffic from reaching it, it is not that simple. Yes, there are dangers in using these techniques; however, by properly configuring PortSentry and monitoring its log output, that risk is significantly reduced. As the authors of PortSentry state in their documentation:

"It is our experience though that spoofed scans are not an issue and we recommend people use auto-blocking knowing that %99.9 of the time it will block a scan.

Again though, we strongly feel that the benefits of auto-blocking hosts *far outweighs* the limited risk you take by having auto-blocking turned on." (Craig Rowland, Portsentry-2.0b1)

Once a scan is detected and a response initiated PortSentry keeps state on previously blocked hosts by writing the IP address of the blocked host to a `Portsentry-blocked.<:protocol>:` file (where is either TCP or UDP). These files are consulted when PortSentry detects scans before it consults its own internal state array. If the host was previously blocked then PortSentry will ignore it. The blocked file gets recreated whenever PortSentry restarts. PortSentry also writes blocked hosts to the `portsentry.history` file. This file is not erased upon startup and is only appended to. It provides a complete history of all hosts blocked.

Conclusion

This article has provided a theoretical and technical overview of the operation of PortSentry. This useful tool can be used to complement other security measures to prevent an attacker from scanning a host and possibly using such information to launch an attack against the target. The second article in this two-part series will cover the installation, configuration, and operation of PortSentry on a host.

[Privacy Statement](#)

Copyright 2006, SecurityFocus