USENIX Association

Proceedings of the 9th USENIX Security Symposium

Denver, Colorado, USA
August 14–17, 2000

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Defeating TCP/IP Stack Fingerprinting

Matthew Smart G. Robert Malan Farnam Jahanian
Department of Electrical Engineering and Computer Science
University of Michigan
1301 Beal Ave.
Ann Arbor, Mich. 48109-2122
{mcsmart,rmalan,farnam}@eecs.umich.edu

Abstract

This paper describes the design and implementation of a TCP/IP stack fingerprint scrubber. The fingerprint scrubber is a new tool to restrict a remote user's ability to determine the operating system of another host on the network. Allowing entire subnetworks to be remotely scanned and characterized opens up security vulnerabilities. Specifically, operating system exploits can be efficiently run against a pre-scanned network because exploits will usually only work against a specific operating system or software running on that platform. The fingerprint scrubber works at both the network and transport layers to convert ambiguous traffic from a heterogeneous group of hosts into sanitized packets that do not reveal clues about the hosts' operating systems. This paper evaluates the performance of a fingerprint scrubber implemented in the FreeBSD kernel and looks at the limitations of this approach.

1 Description

TCP/IP stack fingerprinting is the process of determining the identity of a remote host's operating system by analyzing packets from that host. Freely available tools (such as `nmap` [3] and `queso` [15]) exist to scan TCP/IP stacks efficiently by quickly matching query results against a database of known operating systems. The reason this is called "fingerprinting" is therefore obvious; this process is similar to identifying an unknown person by taking his or her unique fingerprints and finding a match in a database of known fingerprints. The difference is that in real fingerprinting, law enforcement agencies use fingerprinting to track down suspected criminals; in computer networking potential attackers can use fingerprinting to quickly create a list of targets.

We argue that fingerprinting tools can be used to aid unscrupulous users in their attempts to break into or disrupt computer systems. A user can build up a profile of IP addresses and corresponding operating systems for later attacks. `Nmap` can scan a subnetwork of 254 hosts in only a few seconds, or it can be set up to scan very slowly, i.e. over days. These reports can be compiled over weeks or months and cover large portions of a network. When someone discovers a new exploit for a specific operating system, it is simple for an attacker to generate a script to run the exploit against each corresponding host matching that operating system. An example might be an exploit that installs code on a machine to take part in a distributed denial of service attack. Fingerprinting scans can also potentially use non-trivial amounts of network resources including bandwidth and processing time by intrusion detection systems and routers.

Fingerprinting provides fine-grained determination of an operating system. For example, `nmap` has knowledge of 21 different versions of Linux. Other methods of determining an operating system are generally coarse-grained because they use application-level methods. An example is the banner message a user receives when he or she uses telnet to connect to a machine. Many systems freely advertise their operating system in this way. This paper does not deal with blocking application-level fingerprinting because it must be dealt with on an application by application basis.

Almost every system connected to the Internet is vulnerable to fingerprinting. The major operating systems are not the only TCP/IP stacks identified by fingerprinting tools. Routers, switches, hubs, bridges, embedded systems, printers, firewalls, web cameras, and even game consoles are identifiable. Many of these systems, like routers, are important parts of the Internet infrastructure, and compromising infrastructure is a more serious problem than compromising end hosts. Therefore a general mechanism to protect any system is needed.

Some people may consider stack fingerprinting a nuisance rather than a security attack. As with most tools, fingerprinting has both good and bad uses. Network administrators should be able to fingerprint machines under their control to find known vulnerabilities. Stack fingerprinting is not necessarily illegal or an indication of malicious behavior, but we believe the number of scans will grow in frequency as more people access the Internet and discover easy to use tools such as `nmap`. As such, network administrators may not be willing to spend time or money tracking down what they consider petty abuses each time they occur. Instead they may choose to reserve their resources for full-blown intrusions. Also, there may be networks that no single authority has administrative control over, such as a university residence hall. A tool that detects fingerprinting scans but turns them away would allow administrators to track attempts while keeping them from penetrating into local networks.

This paper presents the design and implementation of a tool to defeat TCP/IP stack fingerprinting. We call this new tool a *fingerprint scrubber*. The fingerprint scrubber is transparently interposed between the Internet and the network under protection. The intended use of the scrubber is for it to be placed in front of a set of end hosts or a set of network infrastructure components. The goal of the tool is to block the majority of stack fingerprinting techniques in a general, fast, scalable, and transparent manner.

We describe an experimental evaluation of the tool and show that our implementation blocks known fingerprint scan attempts and is prepared to block future scans. We also show that our fingerprint scrubber can match the performance of a plain IP forwarding gateway on the same hardware and is an order of magnitude more scalable than a transport-level firewall.

The remaining sections are organized as follows. We describe TCP/IP stack fingerprinting in more detail in Section 2. In Section 3 we describe the design and implementation of our fingerprint scrubber. In Section 4 we evaluate the validity and performance of the scrubber. In Section 5 we cover related work and in Section 6 we cover future directions. Finally, in Section 7 we summarize our work.

2 TCP/IP Stack Fingerprinting

The most complete and widely used TCP/IP fingerprinting tool today is `nmap`. It uses a database of over 450 fingerprints to match TCP/IP stacks to a specific operating system or hardware platform. This database includes commercial operating systems, routers, switches, firewalls, and many other systems. Any system that speaks TCP/IP is potentially in the database, which is updated frequently. `Nmap` is free to download and is easy to use. For these reasons, we are going to restrict our talk of existing fingerprinting tools to `nmap`.

`Nmap` fingerprints a system in three steps. First, it performs a port scan to find a set of open and closed TCP and UDP ports. Second, it generates specially formed packets, sends them to the remote host, and listens for responses. Third, it uses the results from the tests to find a matching entry in its database of fingerprints.

`Nmap` uses a set of nine tests to make its choice of operating system. A test consists of one or more packets and the responses received. Eight of `nmap`'s tests are targeted at the TCP layer and one is targeted at the UDP layer. The TCP tests are the most important because TCP has a lot of options and variability in implementations. `Nmap` looks at the order of TCP options, the pattern of initial sequence numbers, IP-level flags such as the don't fragment bit, the TCP flags such as RST, the advertised window size, and a few more things. For more details, including the specific options set in the test packets, refer to the home page for `nmap` [3].

Figure 1 is an example of the output of `nmap` when scanning our EECS department's web server, www.eecs.umich.edu, and one of our department's printers. The TCP sequence prediction result comes from `nmap`'s determination of how a host increments its initial sequence number for each TCP connection. Many commercial operating systems use a random, positive increment, but simpler systems tend to use fixed increments or increments based on the time between connection attempts.

(a)

```
TCP Sequence Prediction:
            Class=truly random
            Difficulty=9999999 (Good luck!)
Remote operating system guess:
            Linux 2.0.35-37
```

(b)

```
TCP Sequence Prediction:
            Class=trivial time dependency
            Difficulty=1 (Trivial joke)
Remote operating system guess:
            Xerox DocuPrint N40
```

Figure 1: Output of an nmap scan against (a) a web server running Linux and (b) a shared printer.

While nmap contains a lot of functionality and does a good job of performing fine-grained fingerprinting, it does not implement all of the techniques that could be used. Various timing-related scans could be performed. For example, determining whether a host implements TCP Tahoe or TCP Reno by imitating packet loss and watching recovery behavior. We discuss this threat and potential solutions in Section 3.2.4. Also, a persistent person could also use methods such as social engineering or application-level techniques to determine a host's operating system. Such techniques are outside the scope of this work. However, there will still be a need to block TCP/IP fingerprinting scans even if an application-level fingerprinting tool is developed. Currently, TCP/IP fingerprinting is the fastest and easiest method for identifying remote hosts' operating systems, and introducing techniques that target applications will not make it obsolete.

3 Fingerprint Scrubber

We developed a tool called a fingerprint scrubber to remove ambiguities from TCP/IP traffic that give clues to a host's operating system. In this section we discuss the goals and intended use of the scrubber and its design and implementation. We demonstrate the validity of the scrubber in the face of known fingerprinting scans and give performance results in the next section.

3.1 Goals and Intended Use of Fingerprint Scrubber

The goal of the fingerprint scrubber is to block known stack fingerprinting techniques in a general, fast, scalable, and transparent manner. The tool should be general enough to block classes of scans, not just specific scans by known fingerprinting tools. The scrubber must not introduce much latency and must be able to handle many concurrent TCP connections. Also, the fingerprint scrubber must not cause any noticeable performance or behavioral differences in end hosts. For example, it is desirable to have a minimal effect on TCP's congestion control mechanisms by not delaying or dropping packets unnecessarily.

We intend for the fingerprint scrubber to be placed in front of a set of systems with only one connection to a larger network. We expect that a fingerprint scrubber would be most appropriately implemented in a gateway machine from a LAN of heterogeneous systems (i.e. Windows, Solaris, MacOS, printers, switches) to a larger corporate or campus network. A logical place for such a system would be as part of an existing firewall. Another use would be to put a scrubber in front of the control connections of routers. The network under protection must be restricted to having one connection to the outside world because all packets traveling to and from a host must travel through the scrubber.

Because the scrubber affects only traffic moving through it, an administrator on the trusted side of the network will still be able to scan the network. Alternatively, an IP access list or some other authentication mechanism could be added to the fingerprint scrubber to allow authorized hosts to bypass scrubbing.

3.2 Fingerprint Scrubber Design and Implementation

We designed the fingerprint scrubber to be placed between a trusted network of heterogeneous systems and an untrusted connection (i.e. the Internet). The scrubber has two interfaces; one interface is designated as *trusted*, and the other is designated as *untrusted*. A packet coming from the untrusted interface is forwarded out the trusted interface and vice versa. The basic design principle is that data

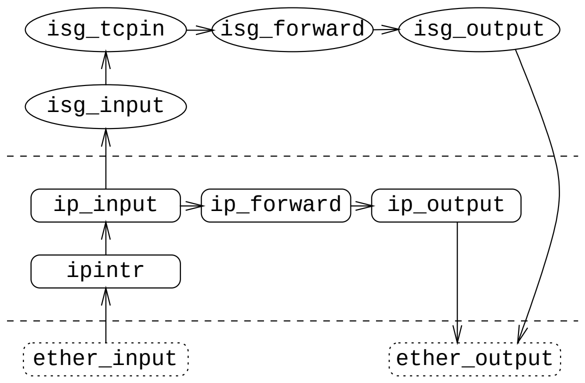

Figure 2: Data flow through modified FreeBSD kernel.

coming in from the untrusted interface is handled differently than data traveling out to the untrusted interface.

The fingerprint scrubber operates at the IP and TCP layers to cover a wide range of known and potential fingerprinting scans. We could have simply implemented a few of the techniques discussed in the following sections to defeat `nmap`. However, the goal of this work is to stay ahead of those developing fingerprinting tools. By making the scrubber operate at a generic level for both IP and TCP, we feel we have raised the bar sufficiently high.

The fingerprint scrubber is based off the protocol scrubber by Malan, et al. [7]. The protocol scrubber operates at the IP and TCP layers of the protocol stack. It is a set of kernel modifications to allow fast TCP flow reassembly to avoid TCP insertion and deletion attacks as described by Ptacek and Newsham [13]. The protocol scrubber follows TCP state transitions by maintaining a small amount of state for each connection, but it leaves the bulk of the TCP processing and state maintenance to the end hosts. This allows a tradeoff between the performance of a stateless solution with the control of a full transport-layer proxy. The protocol scrubber is implemented under FreeBSD, and we continued under FreeBSD 2.2.8 for our development.

Figure 2 shows the data flow through the kernel for the fingerprint scrubber. Packets come in from either the trusted or untrusted interface through an Ethernet driver. Incoming IP packets are handed to `ip_input` through a software interrupt, just as would be done normally. A filter in `ip_input` determines if the packet should be forwarded to the TCP scrubbing code. If not, then it follows the normal IP forwarding path to `ip_output`. If it is, then `isg_input` (ISG stands for Internet Scrubbing Gateway) performs IP fragment reassembly if necessary and passes the packet to `isg_tcpin`. Inside `isg_tcpin` the scrubber keeps track of the TCP connection's state. The packet is passed to `isg_forward` to perform TCP-level processing. Finally, `isg_output` modifies the next-hop link level address and `isg_output` or `ip_output` hands the packet straight to the correct device driver interface for the trusted or untrusted link.

We must also make sure that differences in the packets sent by the trusted hosts to the untrusted hosts don't reveal clues. These checks and modifications are done in `isg_forward` for TCP modifications, `isg_output` for IP modifications to TCP segments, and `ip_output` for IP modifications to non-TCP packets.

3.2.1 IP scrubbing

IP-level ambiguities arise mainly in IP header flags and fragment reassembly algorithms. Modifying flags requires no state but requires adjustment of the header checksum. Reassembly, however, requires fragments to be stored at the scrubber. Once a completed IP datagram is formed, it may need to be re-fragmented on the way out the interface.

The fingerprint scrubber uses the code in Figure 3 to normalize IP type-of-service and fragment bits in the header. This occurs for all ICMP, IGMP, UDP, TCP, and other packets for protocols built on top of IP. Uncommon and generally unused combinations of TOS bits are removed. In the case that these bits need to be used (i.e. an experimental modification to IP) this functionality could be removed. Most TCP/IP implementations we have tested ignore the reserved fragment bit and reset it to 0 if it is set, but we wanted to be safe so we mask it out explicitly. The don't fragment bit is reset if the MTU of the next link is large enough for the packet. This check is not shown in the figure.

Modifying the don't fragment bit could break MTU discovery through the scrubber. One could argue that the reason you would put the fingerprint scrubber in place is to hide information about the systems behind it. This might include topology and bandwidth information. However, such a modification is controversial. We leave the decision on whether or

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.