CERIAS Tech Report 2000-10

Subliminal Traceroute in TCP/IP

Thomas E. Daniels, Eugene H. Spafford
Center for Education and Research in
Information Assurance and Security
Purdue University, West Lafayette, IN 47907

Subliminal Traceroute in TCP/IP

Thomas E. Daniels, Eugene H. Spafford
{daniels,spaf}@cerias.purdue.edu
Center for Education and Research in Information
Assurance and Security — CERIAS
Purdue University
West Lafayette, IN 47907

Abstract

We introduce a technique for tracing a class of "man in the middle" TCP spoofing attacks. The technique works by embedding a traceroute-like mechanism, which we call subliminal traceroute (ST), in the acknowledgment stream of an active TCP connection. We consider the design considerations of ST and show that the attacker can take an active role to defeat our method. We conclude by suggesting future work on ST that may make it more difficult to defeat.

1 Introduction

In this Section, we discuss the problem known as TCP spoofing and past work that addresses the problem. We also attempt to motivate our approach to tracing spoofed connections as opposed to preventing them as in the past work.

1.1 The Problem

TCP spoofing attacks have been widely discussed by others in the literature.[Mor85, Bel89] Most of these discuss attacks that hinge upon the guess-ability of initial TCP sequence numbers (ISN) so that an arbitrary host can exploit an address-based trust relationship to establish a client write-only TCP session. The session is client write-only because the server will respond to the claimed IP address of the client which will not be routed to the attacking client, and therefore the client will not receive any of the server's responses unless it is on the route from server to client. Many operating systems have made this attack much more difficult by using randomly generated initial sequence numbers thereby requiring the attacker to receive at least one packet (SYN-ACK) from the server to carry out the attack.

We consider a less general TCP spoofing attack where guess-ability of ISN's is not an issue. The attacker, Mallory, sits between the client, Alice, who he wishes to impersonate and the target server, Bob. We assume that Mallory is able to read packets from Bob bound to Alice and also cause either the network to drop the packet or Alice to ignore the packet. As shown in Figure 1, Mallory can then create a TCP connection to Bob while masquerading as Alice. In this case, Mallory may not be exploiting trust relationships as in the past work[Mor85], but instead Mallory may just be trying to hide his true identity (IP address) from Bob. Physically, Mallory may accomplish this attack by controlling a host on the route between Bob and Alice or by using one of the routing elements between Bob and Alice to redirect TCP packets from a spoofed stream to him.

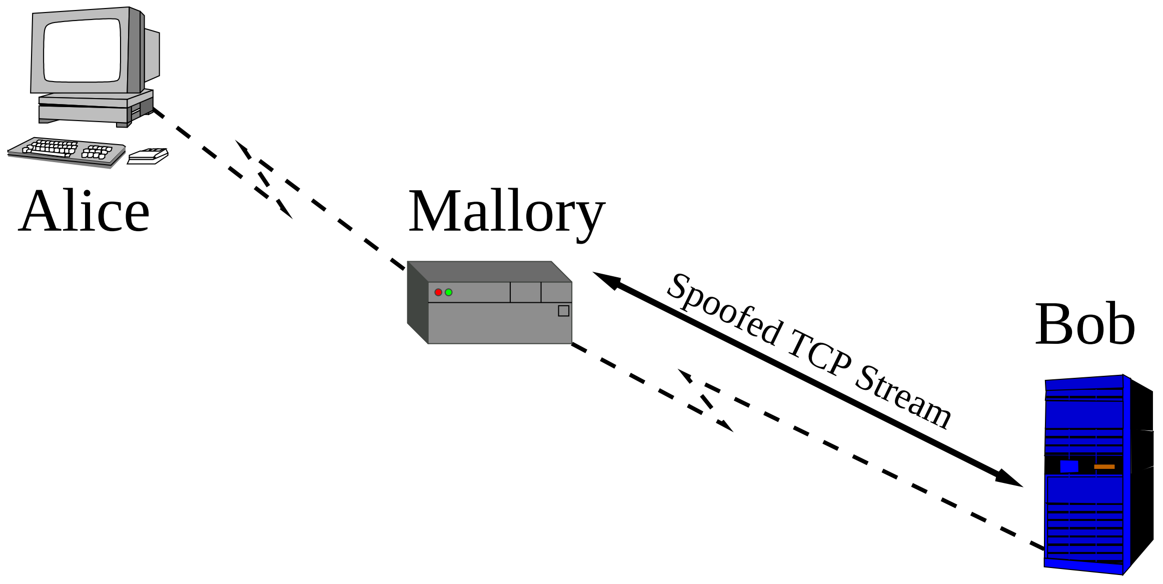

Figure 1. A TCP spoofing man in the middle attack allows Mallory to create a bidirectional TCP stream with Bob while masquerading as Alice.

1.2 Past work

1.2.1 Source Authentication

The most common work in the area of determining the source of network traffic is source address authentication such as in IPSec[Atk95]. The IPSec protocol allows for a digital signature to authenticate the source address (and integrity) of each packet. The problem with such schemes is that it isn't clear that each packet deserves such a high expense procedure. Furthermore, if a host's key material is somehow covertly compromised then spoofing attacks are still possible leaving us with all of the old spoofing problems. Another point is that although the signature authenticates the source address, the scheme does not provide any other information such as a trace that might help trace an attack in a forensic situation.

1.2.2 Tracing packets

Three past works exist that are devoted to tracing specified network packets to their point of origin. They are DoSTracker[CD97], the intrusion detection and isolation protocol (IDIP)[Row99], and Mansfield's tracking protocol[MOT $^+$99]. In the remainder of this section, we will discuss how each of these work and some of their limitations.

DoSTracker[CD97], written by MCI, does a directed search across a network of routers looking for packets bound for a target network and claiming to have a certain source address. DoSTracker is used by MCI to trace denial of service attacks such as smurf attacks[CA-98]. Unfortunately, DoSTracker requires that the network infrastructure be homogeneous (all Cisco routers) and that the tracing person have full control of each router.

DoSTracker does its directed search by starting with the edge router of the target network and setting a "trap" for a packet that represents the denial of service attack in progress. When the trap goes off, the source interface that the packet arrived on is reported, and DoSTracker moves on to all routers that can directly communicate to that interface. The traps are set again and the search continues. Obviously, this approach only works while an attack is active and requires trapping at least as many packets as there are hops to the source of the attack.

IDIP and Mansfield's tracking protocol are very similar so I will refer to both of them as IDIP. The main difference between the two approaches is that IDIP looks for packets that match a specific criteria passing a router whereas Mansfield's approach tries to correlate flows of packets based on RMON-like counts of types of packets seen by the router. IDIP uses a router modification to trace flows of packets. It does this in a way similar to DoSTracker in that the trace begins at the IDIP routers nearest (by network topology) to the attacked network. The router that has seen the packets reports it, and the trace continues with that router's neighboring IDIP routers. Another difference between IDIP and Mansfield's work is that IDIP attempts active response by allowing IDIP routers nearest the source of the attack to filter the malicious packets.

Each of these approaches may be capable of tracing a stream of spoofed TCP packets as in our problem statement, but each has certain limitations in this regard. DoSTracker with trivial modification could trace the stream, but since we are dealing with a "man in the middle" attack, the attacker may have already subverted one or more routers. In this case, the router may be configured to "lie" to DoSTracker and never set the requested traps. IDIP and Mansfield's work might work as well, but both require significant modification to the network infrastructure and are therefore not immediately applicable. Also, these techniques require knowledge that an attack is occurring and are too costly to run for any significant number of packets.

Another similar past work is the Firewalk tool[SG98]. Firewalk can perform traditional traceroute functionality using ICMP, UDP, or TCP. The tool can be used to determine network traces behind a traditional packet filtering firewall. While Firewalk can do a traceroute using TCP packets, our work is different in that it embeds the tracing into an active TCP stream.

1.3 Why Trace?

In most networks, an address is usually considered to be the one identifier that uniquely refers to an entity. Unfortunately, in IP addressing this is not necessarily true. For instance, there are private address spaces such as 10.0.0.0 where many disparate networks use those addresses to refer to their hosts and then use network address translation (NAT) to connect to other networks. In this case, even an authenticated address of 10.0.0.1 would be of very little use because we would not know the network the host was on. A trace may take us only back to the NAT gateway, but this is still much better than the internal address! Furthermore, NAT may be used in several layers thereby requiring several hops of a trace to uniquely identify a host. We understand that conventional tracing does not penetrate NAT borders, but we use this an example of how a trace may be preferable to a single address. In our work, we do not claim to address the problem of tracing through NAT gateways, but we include this as an example of the insufficiency of the IP address to uniquely identify a host.

There is an even better reason for considering a trace useful. A trace can be wrong but not completely so. For instance, the last few hops of a trace may be wrong or unobtainable, but we may be able to trust the trace only as far back as the attacker's Internet service provider, A trace therefore provides levels of accuracy whereas a wrong address is likely to be completely wrong. The reason for this is if the attacker can lie about his IP address why not lie in a big way so as to avert attention elsewhere? On the other hand, it may be very difficult to subvert enough routers to make them lie about substantial portions of the trace.

2 Our Approach

Our approach is to build a traceroute-like[KS94] mechanism into an active TCP stream so that a server can collect traces by modifying an active TCP stream. This approach requires no modifications to the network infrastructure but has weaknesses of its own as we shall see.

Traceroute works as shown in Figure 2. By sending UDP packets to a host with successively increasing time to live (TTL) values, we cause the routers along the path to reply with ICMP time exceeded messages when the TTL expires on a packet. When the TTL becomes great enough that the host receives the UDP packet, the assumption is that the host is not listening on the packet's destination port and therefore the system replies with an ICMP port unreachable message. This signifies the end of the trace.

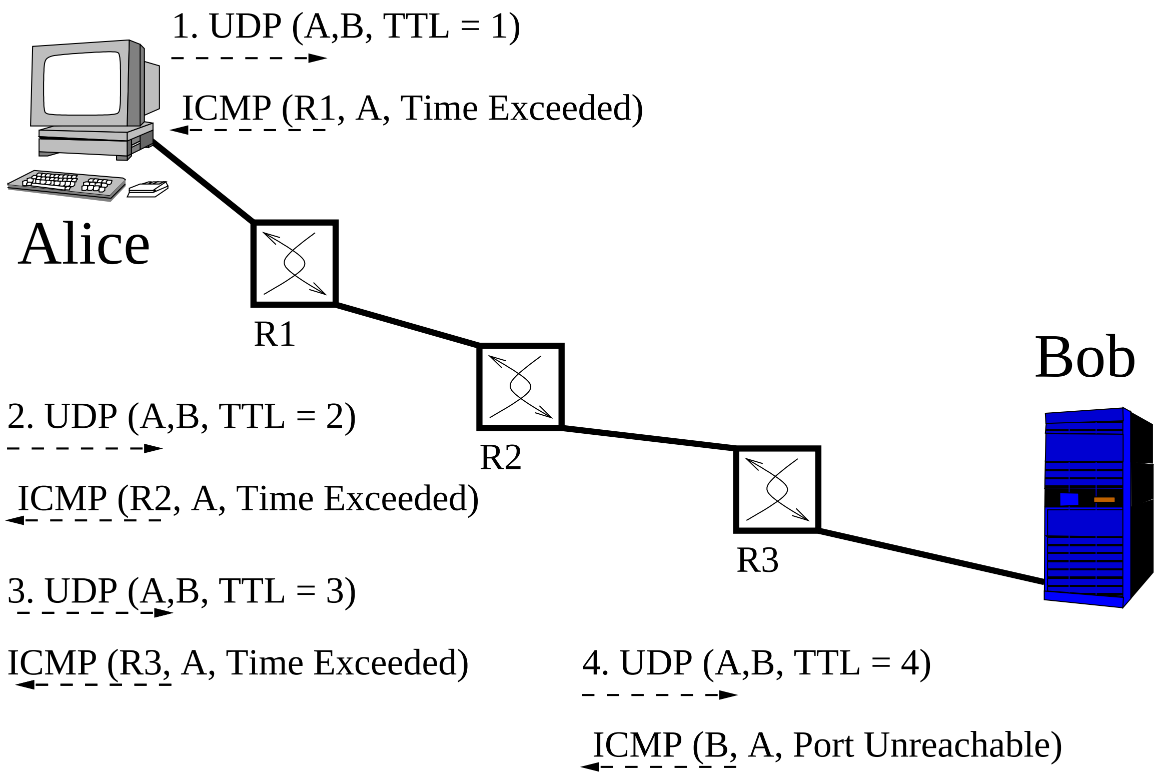
1. UDP (A,B, TTL = 1)
ICMP (R1, A, Time Exceeded)
Alice
R1
R2
R3
Bob
2. UDP (A,B, TTL = 2)
ICMP (R2, A, Time Exceeded)
3. UDP (A,B, TTL = 3)
ICMP (R3, A, Time Exceeded)
4. UDP (A,B, TTL = 4)
ICMP (B, A, Port Unreachable)

Figure 2. The traceroute protocol works by using successively larger time to live'd UDP packets sent to the target host.

An obvious approach to tracing a man in the middle spoofed connection is to just use traceroute to attempt a trace. Unfortunately, this easily defeated as the attacker can choose to allow the UDP packets to continue on to their original destination. In this case, the route will lead to the host that is being imitated, not our attacker.

Our approach is to perform a traceroute inside a TCP stream so that if the attacker is redirecting TCP packets headed for a spoofed client or consuming the TCP packets prematurely, the trace will indicate it and presumably lead us to the attacker. We do this by setting the TTL low on every other ACK sent by the TCP state machine. When a low TTL ACK times out, the remote router responds with a ICMP time exceeded packet that is then captured by the host. We allow TCP to compensate for and resend the lost ACK as part of its reliable service.

Note that it is more difficult for our attacker to simply ignore the packets in our mechanism because they are actually part of the TCP stream in which we wants to participate. The only distinguishing characteristic of the packets used in ST is an occasional TCP acknowledgment with a low TTL.

2.1 Design and Implementation

The ST system was designed around the socket API in the Linux 2.2.10 kernel. Our additions to the socket API allow user mode processes to configure which TCP sockets to trace, and the kernel socket structures provide a convenient abstraction for storing state about each trace in progress.

To allow processes to specify which TCP sockets to trace, we added a new socket option, SOCK_SUBLIM,to the `setsockopt` system call. By setting SOCK_SUBLIM to 0 (the default), no trace is done on the socket. Setting SOCK_SUBLIM to 1 will enable the tracing functionality which we will soon describe. Other values are allowed so that different trace strategies may be added. For instance, if the TCP stream is expected to be short-lived, a more aggressive trace style may be activated. We have modified a version of SSH[Ylo95] to set this socket option.

To implement ST, we modified the portion of the TCP implementation that sends acknowledgments. To this part of the code we added a simple state machine (shown in Figure 3) that dictates the TTL set on each acknowledgment sent by the system. Each edge has a two lines associated with it—the top line is the firing condition for the edge and the bottom line is the action taken when the edge is traversed. Note that each edge has an implicit firing condition: a TCP ACK is ready to be sent.

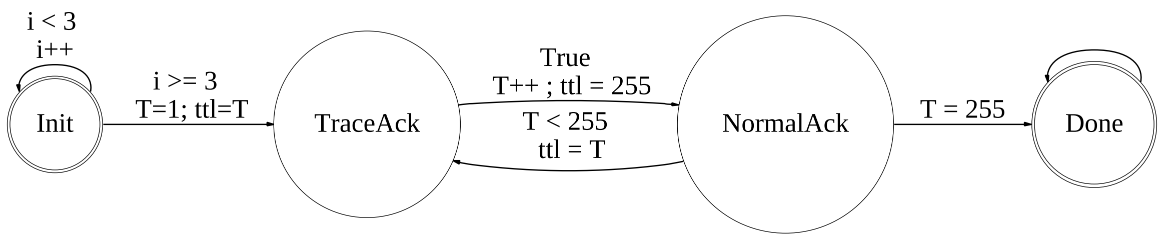

Figure 3. The finite state machine that determines the behavior of the subliminal traceroute. The top line of the edge label represents the firing condition while the bottom line represents the action taken during the transition. i and T are per socket variables and initialized to 0 at time of socket creation.

The values of most of the constants in the state machine were chosen somewhat arbitrarily. We let 3 ACKs go by unmodified so that there is plenty of traffic for the TCP handshake to proceed as normal, and the TCP stream will get established. After this initial stage, the first ACK is sent with a TTL of 1, and every other ACK is then sent with the next higher TTL value. It is important to note that the variables referenced in the state machine are kept separate for each socket in its own data structures. The value of 255 was used as it is the maximum value for a TTL in IP.

The resulting ICMP time exceeded messages are then collected with Tcpdump[Lab]. The source addresses of the ICMP messages are the trace between the server and the client. Packets from multiple simultaneous streams can be distinguished by looking at the message's contents to find the triggering TCP packet's addresses and ports.

2.2 Tradeoffs

There are many tradeoffs in the design and configuration of ST. The state machine controlling the trace can be modified to do a more aggressive trace by reducing the initial waiting period and sending multiple low-TTL packets at once instead of for every other ACK. Also, the state machine could be modified to do many low-TTL ACKs at first so that the trace is done very aggressively.

I chose not to send ACK's in parallel with the TCP stream because it might have interfered with the TCP state machine by causing side effects in the kernel code. However, doing so would allow for a fast trace instead of the current implementation which requires $numberofhops + 3$ acknowledgment cycles to occur to complete the trace. This might come into play if one was tracing HTTP requests where there may be few acknowledgments sent back and forth, and hence too few ACKs may be exchanged for the trace to complete. Our system works fine in most interactive login sessions because each character sent by the client calls for an acknowledgment.

Other design tradeoffs involve implementation details such as our choice to modify the kernel itself. It should be possible but more difficult to create a kernel module that sits between the driver module and the IP stack that performs a subliminal trace. In this case, it might be more difficult to implement control over which TCP streams are traced.

Another problem with ST is that there is no guarantee of correctness of the trace. For each TTL value, we only send one ACK, and therefore if any ICMP time exceeded packet is lost, the trace will be missing a host. Also, time exceeded messages may return out of order causing the trace to have certain hops listed in the incorrect order.

3 Results

We have implemented the ST system and demonstrated that it works in an SSH daemon modified to enable tracing. After a login, each packet sent by the client causes the next hop of the trace to be captured by script running Tcpdump. By sorting the ICMP time exceeded messages by the destination address found in the packet payloads, we can reconstruct a trace for a given peer even if multiple traces occurred simultaneously.

3.1 Defeating the Trace

Unfortunately, we also discovered a mechanism by which the attacker can defeat the ST mechanism. To do this, we capture all IP packets with TTL equal to one, drop the packet, and reply with an ICMP time exceeded message with source address spoofed to appear to come from some other router. Similarly, we can do this for all packets with TTL $\leq n$ to simulate the last n hops of a faked path.

As a proof of concept, we modified the same kernel used to implement ST to simulate fake routers that immediately precede the host. We changed the IP layer of the kernel so that IP packets with TTL below some threshold were dropped and an ICMP time exceeded message was sent in response. One instance of this technique replied to $TTL = 1$ packets with ICMP messages apparently from a host at the National Security Agency. Similarly, $TTL = 2$ packets appeared to time out at a Central Intelligence Agency host. This made the last two traceroute hops preceding the modified host to appear rather comical. Of course, a real hijacker could use this technique to fake the trace between himself and the host he is impersonating.

If implemented naively, it is possible to detect faked route. Since the same host is simulating a number of routers to send false time outs, the delay times observed in a traceroute command should be nearly the same. In order to overcome this, a robust implementation would simulate the increasing latency of successive routers along the path. This could be done by adding variable delays based on path measurements to the implementation, but it would greatly complicate a kernel implementation as it would require scheduling the sending of ICMP time exceeded messages.

4 Conclusions and Future Work

We have demonstrated a subliminal traceback technique in TCP streams for tracing “man in the middle” attacks to their source. Although we have also found a way for the attacker to defeat the trace, we still believe it raises the bar for a would–be attacker as he would have to either drastically modify his host by modifying the kernel or drivers to reject low-TTL’d TCP packets or by somehow intercepting low TTL’d packets before they arrive at the host.

For future work, we believe that by measuring hop times on a route using conventional traceroute and then comparing them with times for a subliminal traceroute, we can make it more difficult to fake a trace as described above. Another possibility for future work involves continually doing a subliminal traceroute in a TCP stream with the goal of detecting TCP hijacking attacks.

References

[Atk95] R. Atkinson. RFC 1825: Security architecture for the internet protocol. `http://www.cs.purdue.edu/homes/clay/papers/ipsec/rfc1825.txt`, Aug 1995.

[Bel89] S.M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, APR 1989.

[CA-98] CERT Advisory CA-98.01. ’Smurf’ IP Denial-of-Service Attacks. http://www.cert.org/advisories/CA-98.01.smurf.html, January 1998.

[CD97] H. Chang and D.Drew. DoSTracker. This was a publically available PERL script that attempted to trace a denial-of-service attack through a series of Cisco routers. It was released into the public domain, but later withdrawn. Copies are still available on some websites. http://www.artsci.net/ jlinux/security/dostrack/, June 1997.

[KS94] G. Kessler and S. Shepard. RFC 1739: A primer on internet and tcp/ip tools. `ftp://ftp.isi.edu/in-notes/rfc1739.txt`, Dec 1994.

[Lab] Lawrence Berkeley National Laboratory. Tcpdump. `ftp://ftp.ee.lbl.gov/tcpdump-3.4.tar.Z`.

[Mor85] Robert T. Morris. A weakness in the 4.2BSD Unix TCP/IP software. Technical report, AT and T Bell Laboratories, Murray Hill, New Jersey 07974, Feb 1985.

[MOT+99] Glenn Mansfield, Kohei Ohta, Y. Takei, N. Kato, and Y. Nemoto. Towards trapping wily intruders in the large. In *Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection*, West Lafayette, IN, USA, Sep 1999.

[Row99] Jeff Rowe. Intrusion detection and isolation protocol: Automated response to attacks. Talk presented at Recent Advances in Intrusion Detection Workshop, 1999, September 1999.

[SG98] Mike Schiffman and David Goldsmith. Firewalk. A publically available tool for doing traceroutes through firewalls. Available at `http://www.packetfactory.net/firewalk/`., Oct 1998.

[Ylo95] Tatu Ylonen. Secure shell man page. Available at `http://www.cs.hut.fi/ssh`, Nov 1995.