Invisible Trojan: An Architecture, Implementation and Detection Method

Raheem A. Beyah, *Michael C. Holloway, and John A. Copeland

Communications Systems Center
Georgia Institute Of Technology
Atlanta, Georgia 30339

*College of Computing
Georgia Institute Of Technology
Atlanta, Georgia 30339

ABSTRACT - **In this paper we give an overview of different system-security tools, including several types of intrusion detection systems (IDSs) and host based detection tools. We also discuss, in detail, port scanning and the primary algorithm used in current port-scanning devices. In addition, we discuss the limitations in the current algorithms used in port-scanning devices and exploit these limitations by implementing an Invisible Trojan that can elude today's port scanners. Finally, we discuss defenses against this type of Trojan. This includes: a proposed method that port-scanning devices can implement, as well as general system-security recommendations.**

I. Introduction

Three major components of "good" security systems are IDSs, host-based detection tools and some sort of port-scanning device. Each component has its strengths and weaknesses, but used collectively they can provide a "reasonably secure network".

Security tools vary in complexity from simple open-source programs, like nmap, to multi-computer client-server commercial security solutions, like ISS RealSecure. The level of investment, in terms of both dollars and training, generally increase with the system's complexity. Though port scanners exist in a sea of many more sophisticated security tools, their importance in system administrators' security arsenals remains high. They provide a good first look into a system for possible problems.

Port scanners are often the first line of defense (or detection) against break-ins. What happens when a port scanner cannot be trusted? We respond to such a query with detailed discussion of port scanning methods and address specific weaknesses. We will illustrate that a Trojan can be created to listen on a "backdoor" port without revealing itself to any port-scanning devices currently available.

II. Intrusion Detection Systems

The main purpose of IDSs is to detect attacks against computer systems and networks. Specifically, IDSs look for malicious attempts in users' abuses of privileges or the improper exploitation of security flaws. A block diagram of a basic knowledge-based IDS is shown in Figure 1.

The IDS receives audit information from the system it is protecting. Several other types of data are also used in a system with this particular architecture. These inputs include: a database full of currently known attacks, the current configuration of the system and audit information that describes the events as they are happening to the system. Once the detector has access to all the required data, it then must decide which information is important and evaluate the likelihood that normal user actions can be considered symptoms of intrusions (to reduce false positives) [1].

The most widely used IDSs are signature-based. Signature-based IDSs assemble the packets in a TCP connection to create a byte stream [2]. This byte steam is examined closely to determine if

certain string patterns (signatures) in the data exist. These signatures are text strings that have been discovered from known past attacks. The signature database must constantly be updated with newly discovered attack strings to remain a viable resource. However, the more signatures the IDS has, the longer it will take to perform pattern matching. This can greatly inhibit network performance and degrade the effectiveness of the IDS. In addition to the afore-mentioned drawback, this type of IDS cannot detect new attacks that are not resident in its signature database.

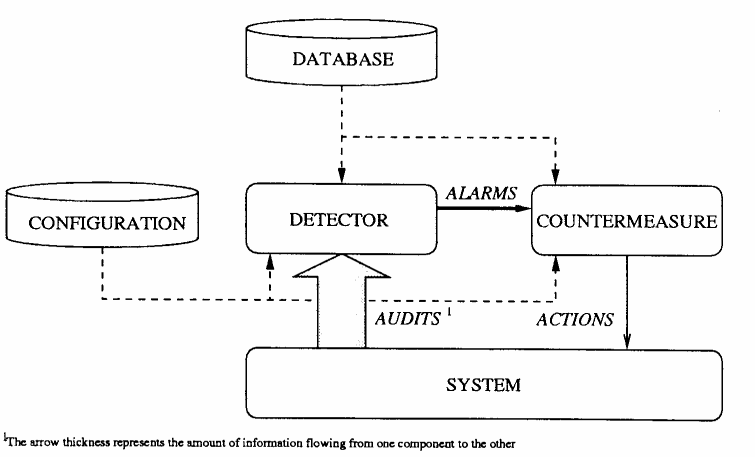

Figure 1. Block diagram of basic intrusion detection system [3]

Anomaly-based IDSs overcome the drawbacks of signature-based IDSs. It identifies unusual activity associated with attacks as opposed to pattern matching, using known attack strings. One method used to determine abnormal activity is data flow analysis [4]. Flows represent all the packets exchanged between the hosts with a "single" service. Certain statistical data is updated in the Flow data record. This included number of bytes, packets, etc. The Flow is examined to determine if it has characteristics of a possible attack. When the Flow ends, the statistical data is examined and recorded. One Flow alone may not be sufficient to detect an attack, so the recorded statistical data is correlated with other events to potentially discover an attack. By collecting data on the complete transaction better decisions can be made (less false positives). Thus, it is able determine the normal and anomalous behavior of a system. This system is not without flaws. Although it can catch new attacks, it would fail to recognize specific attacks that can only be detected by signature matching [2] [5].

Other types of IDSs deserving mention here include petri nets (graphical representation of signatures), state transition analysis (based on attack descriptions as a set of goals and transitions) and soft computing based [1].

III. Port Scanning

The general concept of scanning has been around for decades. The basic idea is to probe as many listeners as possible and keep track of the ones who are receptive. This concept is very similar to the "to current resident" brute-force style of bulk mail that the postal service delivers - put a letter in each mailbox and wait for the responses to trickle back. This method is readily applicable to computers and computer networks. Scanning computer ports as a method of discovering potentially exploitable communication channels is called port scanning.

Port scanning is a technique to determine what ports of a particular host are listening for connections. Open ports represent potential communication channels. Port scanning can be used to assist in securing unknowingly vulnerable machines or by hackers to locate potential vulnerabilities.

Over time, several different techniques have been constructed to see what ports on a machine, using what protocols, are open and listening for a connection. Two commonly used techniques for determining if a TCP port is listening are TCP connect() scanning and TCP SYN scanning. TCP connect() scanning is said to be the most basic form of TCP scanning. The connect() system call is aimed at each questionable port. If the port is open and listening, the command will succeed. Otherwise, it will fail because the port is not reachable.

Another approach is the TCP SYN scanning technique. It is often referred to as "half-open" scanning because one does not open a full TCP connection. This technique is slightly more involved than TCP connect() scanning. Here, the probing host sends a SYN packet, as if it were initiating a real connection. If a SYN/ACK is received, the port is listening and you can tear it down by sending a RST. If a RST is received, the port is not listening. For a more exhaustive list and detailed description on TCP port-scanning techniques see [6].

Above we discussed several TCP port scanning techniques. UDP scanning also comes in several variations. UDP write() scanning is a popular method of UDP scanning. With this technique the write() function is called for desired ports. The kernel will acknowledge the error if the port is closed. If the write() function is called a second time to the desired port, the user is usually notified that the call failed.

Another form of UDP port scanning is UDP ICMP port unreachable scanning. In this paper we exploit this method. The target machine is sent UDP packets to suspected ports. If the port is open, no response is received at the polling host. If the port is closed, the target host responds with an ICMP port unreachable packet. There are potential problems with this method. Neither UDP packets, nor the ICMP errors are guaranteed to arrive. This problem is exacerbated if the target is flooded with more packets than it can handle and packets discarded. If packets destined for closed ports are dropped, the port scanner will not get a response from the port, which will falsely indicate that the port is open. For this reason, UDP scanners of this sort must also implement retransmission of packets that appear to be lost.

IV. Limitations of Port Scanners

Port scanners that use the UDP ICMP port unreachable technique have an obvious weakness that can be exploited by hackers. These scanners rely on an ICMP response or on a lack thereof from the noted ports on the target machine to determine whether the port is listening. The format of this packet is given in Figure 2.

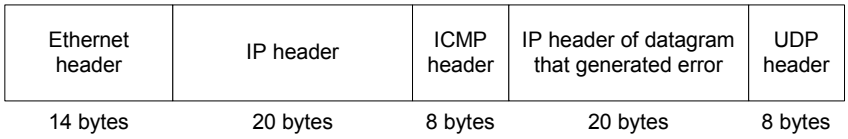

Figure 2. ICMP Response [7]

The weakness of this method is that this packet can be constructed by anyone. There is no authentication performed on the ICMP response packet. To construct the packet, we first consider the ICMP packet header (Figure 3).

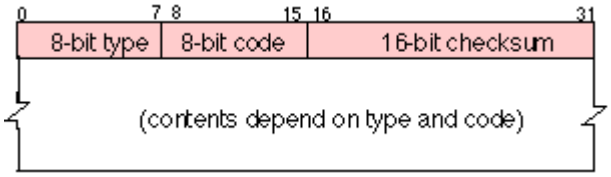

Figure 3. ICMP packet header [7]

Specifically, the 8-bit type field should have the value 3, and the 8-bit code should contain the value 3. This labels the packet as a port-unreachable, destination-unreachable packet. The 16-bit checksum is calculated using the exact same algorithm used to calculate the IP checksum and is easily accessible [7].

The next field of concern is the payload of the ICMP packet. The first 20 bytes of the payload is the IP header of the datagram from the port scanner that generated the error. The general format of the IP header is given in Figure 4.

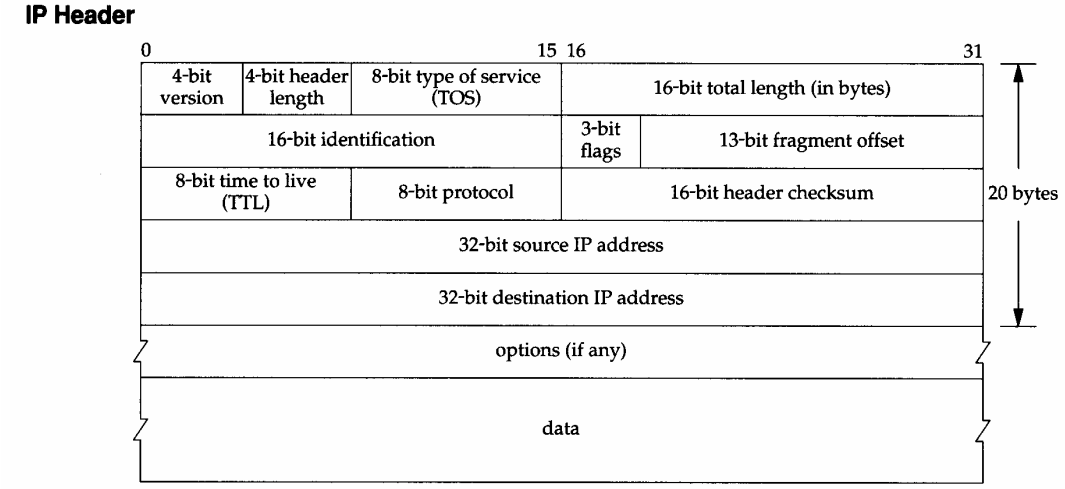

Figure 4. IP packet header [7]

The only fields that are obtainable from a regular user are the source and destination IP addresses. The rest are consumed by the kernel. This causes a problem based on the need for the entire 20-byte header for the payload of the ICMP packet. Otherwise, the port scanner will discard the packet. Root privileges are necessary to access the remainder of the header. The final 8 bytes of the packet is a portion of the IP payload that generated the error, the UDP header (Figure 5).

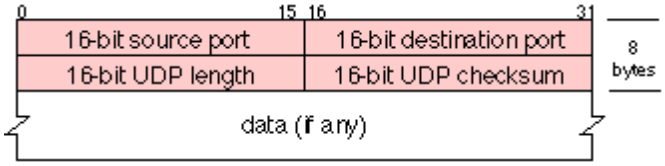

Figure 5. UDP packet header [7]

Similarly, the only data available from the UDP header to a regular user is the source and destination ports. Root privileges are again needed to obtain the remainder of the packet. Once the appropriate information is captured from the polling packet, the ICMP packet can be properly constructed and transmitted to the port-scanning device. The port that is open will be reported as closed by the port-scanning device.

V. Invisible Trojan Design

A. Scope

An Invisible Trojan is simply a process that can lay dormant and undetected in "stealth mode". This intelligent process evades port scanners, but can be awakened on command by the client process. This is done by examining packets destined to the port and selectively responding. The focus of this paper is not the method of delivery of the Trojan but rather the ability, once delivered, to be completely undetected from external system probing (i.e. port scanners). The delivery issue has been exhaustively examined. Nor is this paper's focus how root privileges are obtained. The familiar concept of "root kits" or buffer overflow techniques to obtain a root shell [8], or other familiar techniques may be used. Thus, the Trojan is assumed to be installed in conjunction with a "root kit" and the delivery method and root access will not be discussed further. Also, we do not seek to address host-based tools (netstat, etc.) that can by evaded by established techniques, like modifying log files [8] once root privileges are acquired.

B. Architecture Description

The architecture is given below (Figure 6). Hosts A and B are running Red Hat Linux 7.1. Host C is running Windows NT 4.0. To verify the integrity of our proposed technique, two different port scanners are used. The widely known port scanner nmap was installed on Host A. Internet Scanner from ISS was installed on Host C. To observe the traffic on the network, Ethereal [9] (network sniffer) was loaded on Host B. Host B was also considered the target host, and thus, the Trojan server process was installed there.

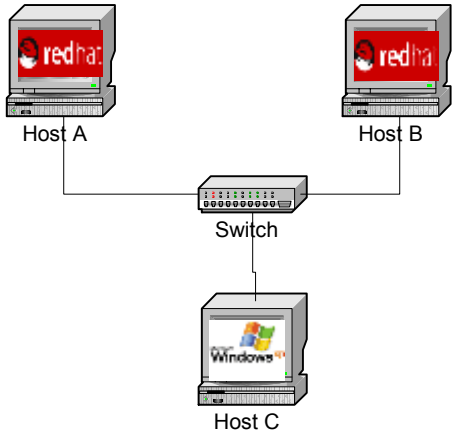

Figure 6. Architecture

The software needed to implement the Invisible Trojan is broken into two separate programs - a client and server. The client generates command-UDP packets. The server program contains a kernel-level filter that intercepts packets destined for the port where the Trojan is listening. The second function of the server program is to examine filtered packets to determine if the packets come from its master, the client program, or from another source, possibly a port scanner. If the UDP packet is deemed "familiar" (i.e. the payload of the UDP packet contains the "magic" string), we know the remote program is our Trojan client, so we communicate with the Trojan. In

our example, we simply send a reply of "WHO'S THERE" to the client program. If the packet is deemed "unfamiliar", an ICMP port unreachable message is returned.

VI. Invisible Trojan Implementation

A. Software Details

Having an overview of the architecture, it is now appropriate to consider the details of implementation. The client program is fairly trivial and can be implemented in several different ways. We choose to open a datagram socket on the client machine. The code populates the UDP payload with random garbage as well as the "magic" string and generates the UDP wake-up packet when invoked.

The server program opens a UDP datagram socket (SOCK_DGRAM) and listens on port 5000 for incoming UDP packets. The idea is to capture packets destined for port 5000. The problem that arises with this approach is the need to selectively send a message back to the client machine or ICMP port unreachable packet to all others. To properly construct the port unreachable packet, access to the entire incoming UDP packet is needed. This allows retransmittal of the IP and UDP headers of this packet in the payload of the newly constructed ICMP port unreachable packet. When listening using a datagram socket, the kernel strips the header information from incoming packets and only passes up the payload and a small portion of the header (the source and destination ports and addresses).

To overcome shortcomings of using the datagram socket, we attempted to open a raw socket (SOCK_RAW). The problem with this approach is that only IP datagrams with a protocol field that the kernel does not understand are passed to a raw socket [10]. This is obviously not the case with probing UDP packets. Consequently, to read the entire IP datagram containing UDP or TCP packets, the packets must be captured at the datalink layer [10].

B. Accessing the Datalink Layer

There are several different methods for providing access to the datalink layer. Some include the Data Link Provider Interface (DLPI), the SOCK_PACKET method in Linux, using libcap and BPF:BSD Packet Filter method. These methods are discussed in detail in [10]. We choose the BPF:BSD Packet Filter technique. BPF is a kernel-level filter, which limits the amount of data copied to the application. Using this method, each datalink driver calls BPF right before a packet is transmitted and right after a packet is received. The packets are passed to the BPF engine and then to the filter. The remaining filtered packets are passed to the buffer, and finally to the application. The BPF filter program can be written as either ASCII strings (i.e. tcp and port and tcp[13:1] & 0x7 – [10]) or in a virtual machine language. The virtual machine-language code we implemented to filter packets to port 5000 is shown below (Figure 7).

```
struct sock_filter BPF_code[]= {
{ 0x28, 0, 0, 0x0000000c },
{ 0x15, 0, 8, 0x00000800 },
{ 0x30, 0, 0, 0x00000017 },
{ 0x15, 0, 6, 0x00000011 },
{ 0x28, 0, 0, 0x00000014 },
{ 0x45, 4, 0, 0x00001fff },
{ 0xb1, 0, 0, 0x0000000e },
{ 0x48, 0, 0, 0x00000010 },
{ 0x15, 0, 1, LISTEN_PORT },
{ 0x6, 0, 0, 0x0000ffff },
{ 0x6, 0, 0, 0x00000000 },
};
```

Figure 7. BPF filter virtual machine-language code used to filter on port 5000

C. The "Magic" String

At this point, the only visible packets (to our Trojan server) are the packets with destination port 5000. Each packet is closely examined to see if the "magic" string of "KNOCK KNOCK" is present. If the packet has this value in the 21st – 31st byte in the payload (the first 20 bytes are superfluous and ignored and modified at each transmission) a message of "WHO'S THERE" is sent back to the client program. If the "magic" string is not present, we construct the appropriate ICMP port unreachable packet and respond accordingly. This is accomplished by opening a UDP raw socket and byte-wise constructing the ICMP packet.

When attempting to communicate with the Trojan we use the client program. It opens a UDP datagram socket and transmits a UDP packet with the "magic" string in the payload to the target host. It then opens another socket to listen on, where it perpetually listens for a response from the Trojan.

D. Illustration of Stealth Trojan

First, we start the server process without "stealth mode" to see if the port scanners detect the listening ports correctly. We start nmap first, and it sends several UDP probes to each port and awaits responses. After the nmap scan has completed, we start the scan with Internet Scanner, which takes the same approach, send several UDP packets and wait for a response. The packets sent by Internet Scanner are shown below (Figure 8) in the Ethereal traffic capture. The basic packet structure for both scanners is the same, a standard UDP datagram. However, Internet Scanner appends 20 bytes of data in the payload. This is the string "UDP Scan by ISS (11)". Since no ICMP destination unreachable packet is returned for port 5000 (Figure 8), both nmap (Figure 9) and Internet Scanner (Figure 10) show port 5000 as open.

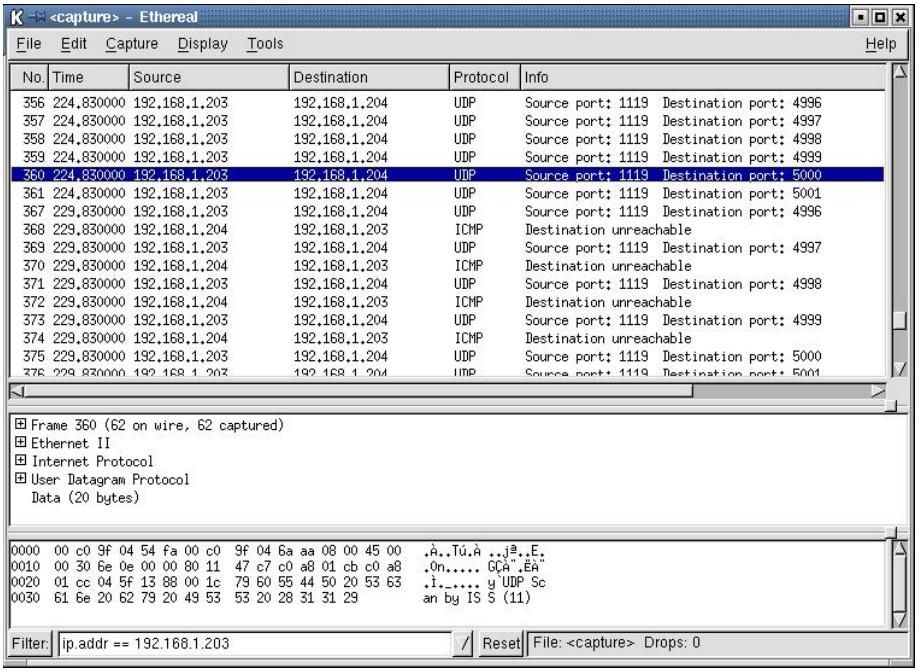

Figure 8. UDP polling packets from port scanner

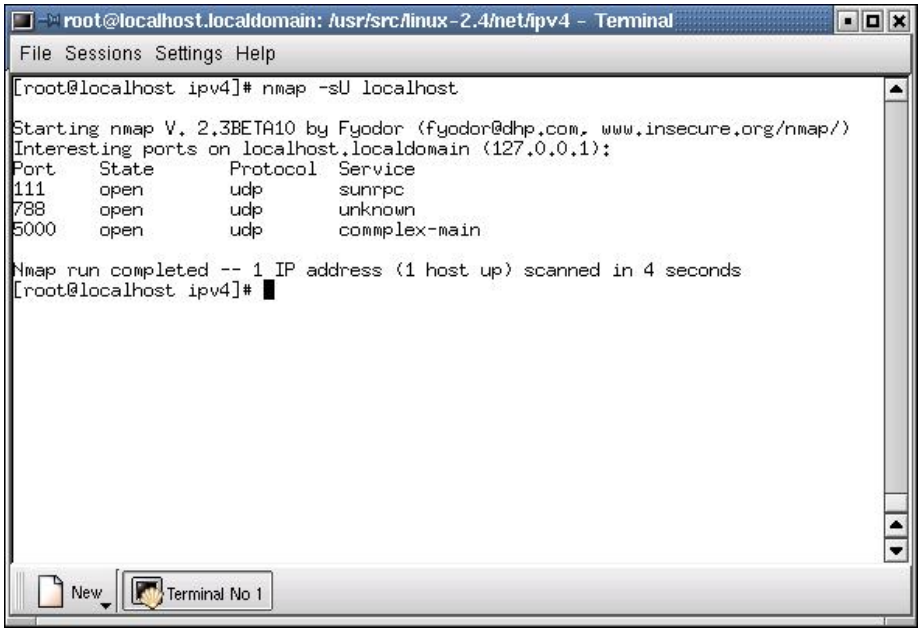

Figure 9. nmap's open port list – "stealth mode" off

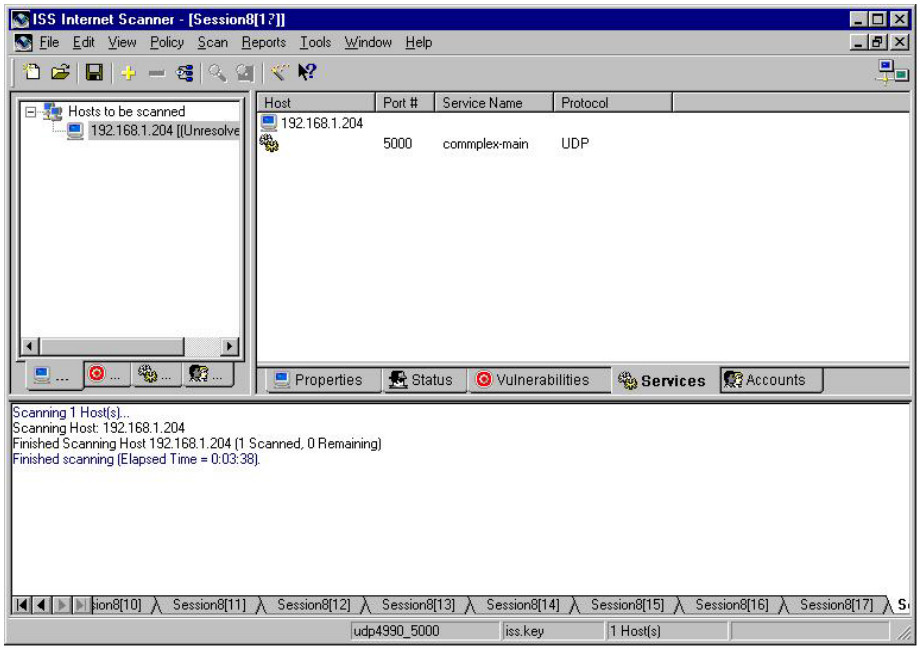

Figure 10. Internet Scanner's open port list – "stealth mode" off

Next, we place the Trojan in stealth mode and begin the same process. As expected, both nmap (Figure 12) and Internet Scanner (Figure 13) were unable to detect the Invisible Trojan. This was accomplished by constructing the appropriate ICMP response packet using the above-mentioned method. The packet is shown captured in Ethereal (Figure 11).

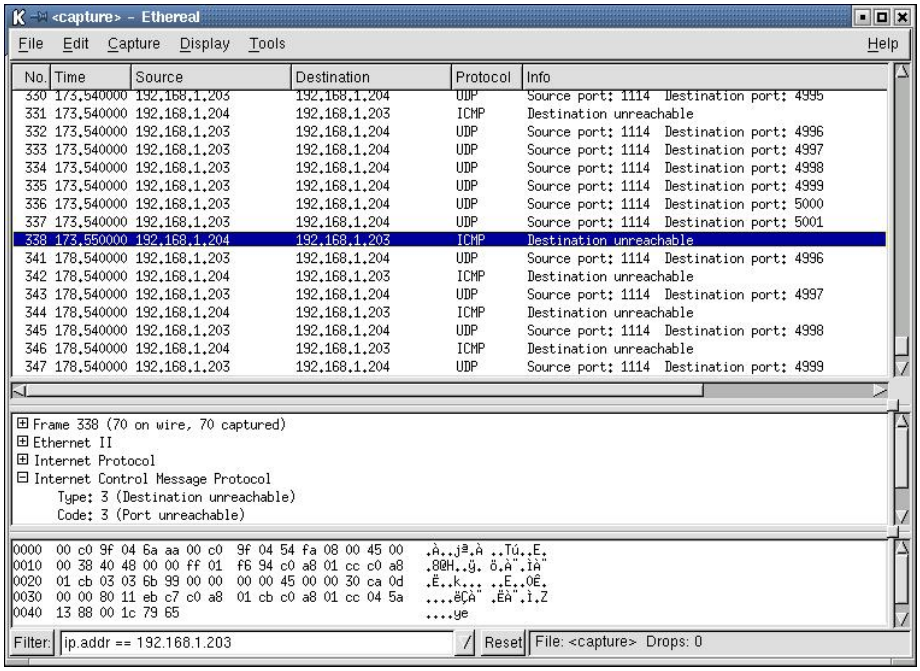

Figure 11. Maliciously constructed ICMP response

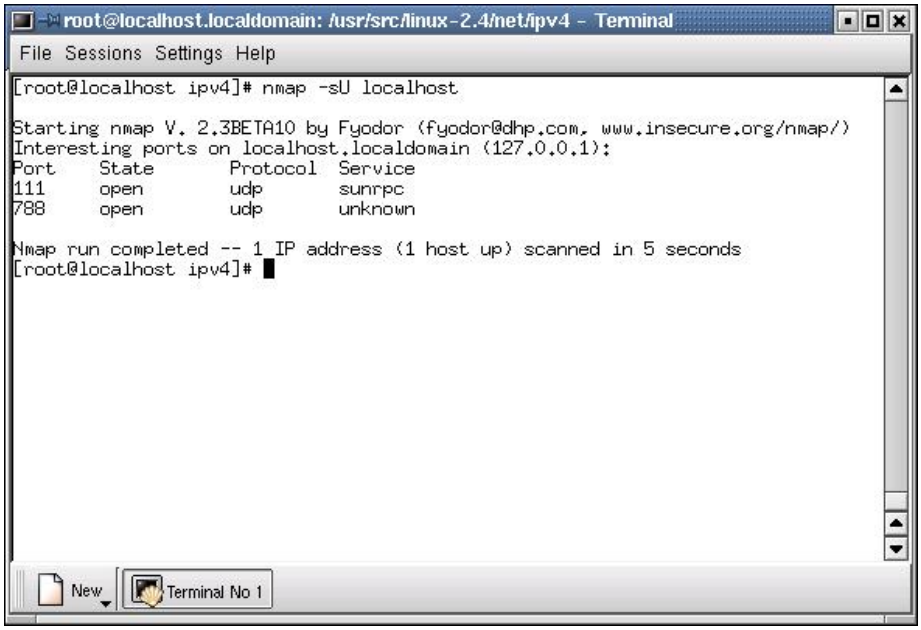

Figure 12. nmap's open port list – "stealth mode" on

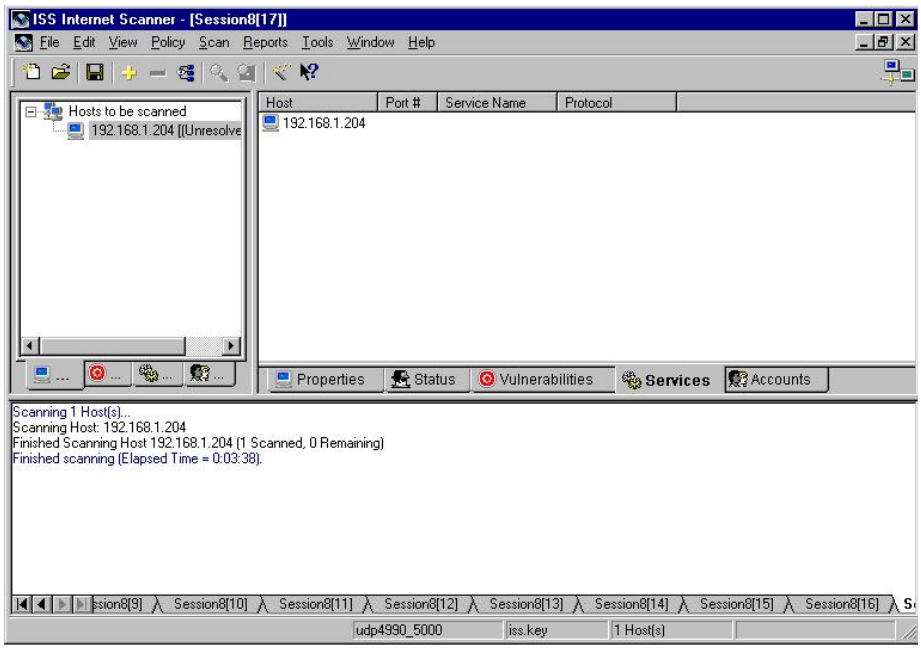

Figure 13. Internet Scanner's open port list – "stealth mode" on

Above, we saw the Trojan can effectively hide from port scanning devices when placed in "stealth mode". Now, we illustrate how the Trojan can be "awakened" selectively by the client process. We send a packet with the "magic" string contained in the payload. This packet is seen captured using Ethereal in Figure 14.

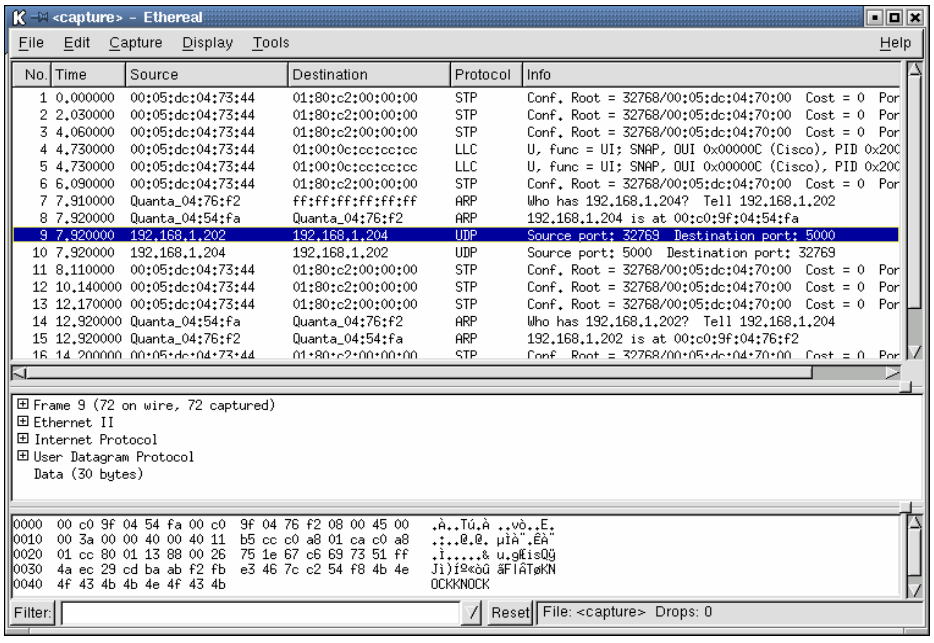

Figure 14. "KNOCK KNOCK" message from client to Trojan server on host

The packet is filtered appropriately by our BSD filter and examined to see if the "magic" string of "KNOCK KNOCK" is present in the 21st –32nd byte of the UDP payload. If present we send back

a string to the client of "WHO'S THERE" to let the client program know that the Trojan is in place and still undetected. This return packet is seen below in Figure 15.

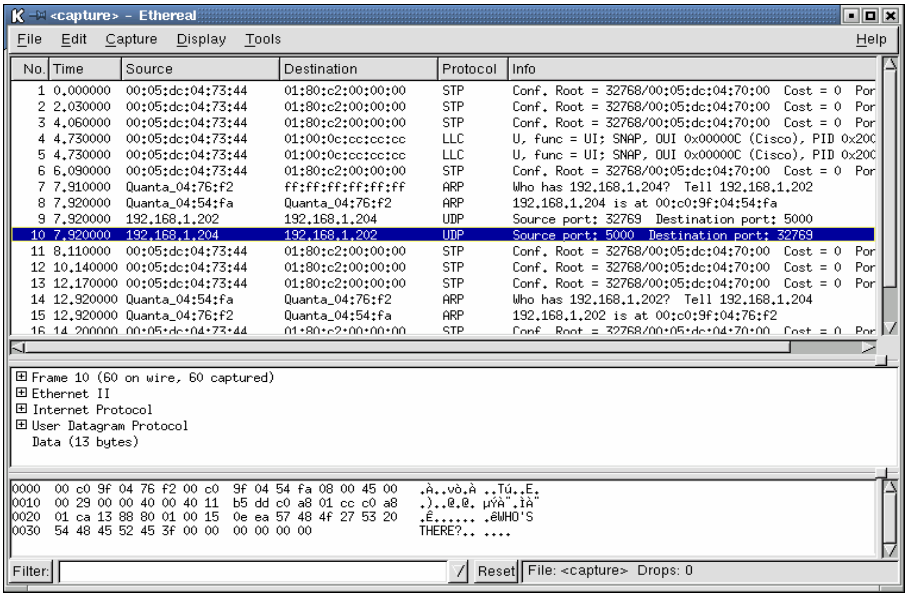

Figure 15. "WHO'S THERE" response from Trojan

VII. Protecting Against Trojan

A. Processing Time Anomaly

The Trojan that we have discussed can cause an immeasurable amount of damage depending on its specific implementation. We have determined a method to identify if this sort of Invisible Trojan exists on a system. This method has been proven for a very lightly loaded network, but requires further research. Also, the case of a congested network is due consideration.

The first arrow (packet #321) identifies a UDP polling packet (to a closed port) from the port scanner. The second arrow (packet #326) shows the ICMP response packet that was constructed by the kernel. This illustrates the appropriate maximum delay (approx. 150ms without our current network topology) between the polling device sending the UDP message and the target responding with ICMP port unreachable. If the kernel receives the message, it will immediately send a response. This is a result of very little processing latency due to the packet being quickly consumed by the kernel. The highlighted packet pair (packet no. 314 and 316) in Figure 15 identifies the UDP polling packet and the ICMP response packet constructed by the Trojan. The delay (approx. 300ms) is on average, at minimum twice as much as the former case.

This delay is incurred for two reasons. First, the Trojan is a user process. Therefore, the kernel accepts and then passes the packet to the Trojan. The more variable delay occurs when the Trojan scans the packet for the "magic" string. The latter delay can increase depending on the processing complexity of the Trojan. Therefore, on average there will be at least one packet or a noticeable delay between the UDP polling packet and the maliciously constructed ICMP response. After careful analysis, the Invisible Trojan can quickly be identified and removed.

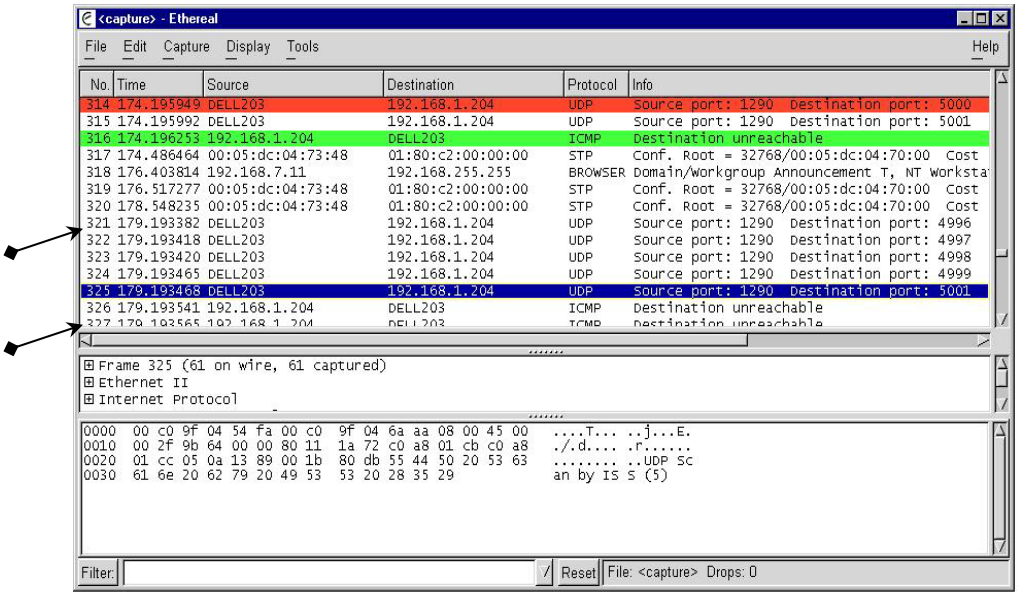

Figure 16. Delay variance in kernel-generated and Trojan-generated ICMP response packets

B. Other Detection Methods

i. Host Based

We have presented a method for evading port scanners by forging ICMP port not available messages. The desired goal was to have a Trojan run on some specified port and interact only with its "master", the Trojan client. However, our Trojan is still susceptible to some host-based tools like netstat or lsof, that show lists of opened sockets. The bulk of our discussion has been on external port scanning, and that is the sole concern of our Trojan design. However, a slightly savvy sysadmin may use simple host based tools like netstat, or lsof to get a list of open sockets on any linux machine.

However, host-based tools are not the complete answer. Many hackers will replace these host-based tools with Trojanized versions that show (or neglect to show) the information they desire. Also, a slightly modified Trojan (though not more difficult to code), might include its "selective behavior" as part of another "trusted" program [11]. For example, root kits often include Trojanized versions of identd. We could embed our simple Trojan code in the (open-source) identd program that runs on port 113. Basically, we would examine all incoming packets for "magic". If the packet is "normal" we would pass the control to the normal identd function. However, if the packet is our "magic" packet, we could communicate with our Trojan client appropriately. Because our Trojan is running on a well-known port we decrease the likelihood that our Trojan shows up on the sysadmin's "radar"

It has been shown that through Trojanizing any number of host-based tools; the attacker can essentially "cover his tracks". The only completely reliable way to detect system compromise is to bit compare all executables on your system with "known good" ones. A similar solution is to run checksums on executables. One tool that does this is Tripwire [12].

ii. IDS Detection

Specific implementations of IDSs can also play a key role in detecting the communication between the Trojan server and client process. However, this method is not without drawbacks. Our Trojan also attempts to evade current IDS techniques. The vast majority of commercial IDS tools use a string-matching algorithm to match network traffic against a known database of attack "signatures". Also most of these tools simply match characters at the beginning of a packet. We exploit this weakness by encoding a number of random bytes at the beginning of our "magic" packet. The client generates 20 bytes of random data, followed by the "magic string" and the Trojan server ignores the first 20 (garbage) bytes. Because of this simple fact, it would be very difficult for any existing signature-based IDS to detect our Trojan.

Some signature-based IDSs use partial string matching. This implementation could detect communication to and from our Invisible Trojan. However, though improved, the implementation continues to rely on matching "static" strings that do not change. Even with partial string matching, we can devise a Trojan to defeat signature based IDSs as follows. The Trojan client simply needs to construct a packet that can change each time he connects, but still is detectable to the Trojan server as "magic". One simple way to accomplish this is to embed some "changing" information in the packet that the server can verify. For example, the Trojan client can replicate the source port of its magic up packet in the up payload. This number changes each time the Trojan connects, but can be implicitly verified at the server. If the source port in the payload matches the source port in the up header, the server knows this packet is "magic".

A rule-based IDS might be able to detect our modified Trojan, because it follows some defined rule (up source port == first 4 bytes of payload). However, with a little more effort, we can evade just about any rule-based IDS. We simply agree on a password or key that we will use to encrypt/decrypt traffic between our Trojan client and server. Simply use the agreed upon key, and concatenate the source port number of the client packet to the key to form a modified (dynamic) key. Only the Trojan server knows about the agreed upon key, so only the server can decrypt the packet. Further, the key (and hence encrypted traffic bytes) changes with each connection, because the key changes with the source port. Although this method is certainly complex, it should evade any rule or signature-based IDS.

The only IDS that would easily detect the communication between the Trojan client and Trojan server program are the anomaly-based systems. Once this system has determined the "normal" behavior of a host, any "irregular" communication would be noticed and placed in a host's profile and eventually will trigger an alarm to a sysadmin.

VIII. Conclusion

We have seen that evading port scanners is a fairly simple feat. It would be extremely difficult for any port scanner to reliably detect our invisible Trojan that simply forges ICMP port not available messages, except under "ideal" network conditions, which may not be available. It should be clear that relying exclusively on port scanning for detecting open ports is a bad idea.

We have also discussed several other security measures for detecting Trojans, including host based tools and several types of Intrusion Detection Systems. We have also presented novel ways to evade the majority of these tools. Certainly it is difficult to rely on any one method for detecting attacks.

One of the strongest (and possibly simplest) methods we presented is bit comparison of executables with "known good" versions. However, this obviously does you no good if the offending program does not Trojanize any well-known programs (it would not detect our first proposed Trojan). An effective security solution should combine a variety of methods, including host-based tools, network probing, and possibly Intrusion Detection Systems.

References

[1] R. C. Garcia. "A soft Computing Approach to Anomaly Detection with Real-Time Applicablility," Ph.D. Thesis, Georgia Institute Of Technology, April 2001.

[2] LANcope, "StealthWatch vs. Existing IDS Technology," 2001; http://www.lancope.com.

[3] Debar, H., et al, "Towards a Taxonomy of Intrusion-Detection Systems," IBM Research Report 93076, 1998.

[4] LANcope, "The Use of "Flows" to Analyze Data Network Traffic," 2001; http://www.lancope.com.

[5] LANcope, "Data Flow Analysis for Traffic Characterization and Network Security," 2001; http://www.lancope.com.

[6] "The Art of Port Scanning," 2002; http://www.insecure.org/nmap/nmap_doc.html.

[7] W.R. Stevens, TCP/IP Illustrated Volume 1: the protocols. Boston, MA: Addison Wesley, 1994.

[8] M. Elkins, " Anatomy of a Breakin," May 1, 2001. Linux Journal. http://www2.linuxjournal.com/cgi-bin/frames.pl/index.html

[9] www.ethereal.com

[10] W.R. Stevens, UNIX Network Programming Volume 1 – Networking APIs: Sockets and XTI. Upper Saddle River, NJ: Prentice Hall, 1998.

[11] C. P. Pfleeger, Security In Computing. Upper Saddle River, NJ: Prentice Hall, 1997.

[12] K. Fenzi, "Tripping up Intruders with TripWire," Feb 17, 2001. Linux Journal. http://www2.linusjournal.com/lj-issues/issue40/2160.html