GIAC
CERTIFICATIONS

Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Security Essentials: Network, Endpoint, and Cloud (Security 401)"
at http://www.giac.org/registration/gsec

Worm Propagation and Countermeasures

GSEC Practical v1.4b Option 1

© SANS Institute 2004, Author retains full rights.

Prepared by: Glenn Gebhart
2/24/2004

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Abstract

Recent history has amply demonstrated the threat that worms pose to the Internet and those who rely on its correct functioning. Most of the damage done by worms can be traced to the burden they place on networks due to their characteristic exponential growth as they seek to propagate themselves. As such, if security professionals can develop a framework for preventing worm propagation then they can significantly reduce the risk that worms pose to the Internet.

This paper is an attempt to approach the problem of worm control in a systematic fashion. Beginning with a motivating discussion of the current threat posed by worms, it moves on to examine a selection of the most notorious worms both old and new. Highlighting the commonalities of these worms allows for the synthesis of a general model of worm propagation. Analysis of this model shows that the process of worm propagation has a number of steps, each one of which can potentially be disrupted through the deployment of the appropriate security technology. A discussion then follows of the technologies that can be deployed at each step to prevent, contain, or slow the spread of worms.

© SANS Institute 2004, Author retains full rights.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

1. Introduction

Network worms are an especially virulent form of abuse that have become an increasing threat to the Internet over the past several years. Unlike their viral predecessors, worms are network-aware and self-propagating, greatly increasing their potential reach and impact. What's worse, worms are not just a problem for those who become infected; they regularly cause problems for the uninfected as well through secondary effects such as increased network load. As such, it is important for security professionals to study their behavior and deploy prophylactic measures to reduce the impact of the worms that will inevitably arise in the future. In an effort to further this goal this paper will analyze a sample of classic and recent worms, resulting in a list of common propagation mechanisms. Each mechanism will then be examined to determine what countermeasures, if any, exist to reduce the efficacy of that mechanism.

2. What is a "worm"?

So what, exactly, is a worm? Parties within and outside of the security community have bandied about this term with great imprecision. For the sake of clarity and ease of analysis it is important to come up with a concise definition that accurately captures the behavior to be examined.

To a first approximation a worm is a virus. The Webopdeida defines a "virus" as "[a] program or piece of code that is loaded onto your computer without your knowledge and runs against your wishes". This captures the basic nature of worms as malicious code that users don't want running on their system. It does not, however, capture the behavior of worms in terms of propagation, so further refinement is in order.

The Webopedia defines a worm as "[a] program or algorithm that replicates itself over a computer network and usually performs malicious actions, such as using up the computer's resources and possibly shutting the system down". This definition, however, captures a broad range of phenomena ranging from the annoying (but relatively low impact) programs that require users to execute an attachment all the way up to high impact, rapidly replicating programs that propagate via UDP. The critical distinction amongst programs in this continuum appears to be the degree of user interaction required to trigger propagation; the lower the degree of interactivity the more quickly a program can spread. That being the case it seems prudent to add an additional qualifier to the operational definition of the term "worm": a worm is self-propagating, where "self-propagating" will be defined as "capable of spreading between computer systems without user intervention".

Thus the final definition of the term "worm" is "a self-propagating virus that spreads via computer networks". The usefulness of this definition is that it serves to separate high-impact phenomena from low-impact phenomena (which might be more correctly classified by terms such as "trojan" or "email-borne virus"). Admittedly there are edge cases with this definition; does it count as "user

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

interaction" if the act of rendering an email proves sufficient to trigger propagation? These edge cases don't materially affect the overall analysis and so will be included/excluded as best serves the discussion.

3. History and Motivation

Having defined the term "worm", it still remains to be seen why worms are a phenomenon with which the security community should concern itself. At this point a visual is probably in order:

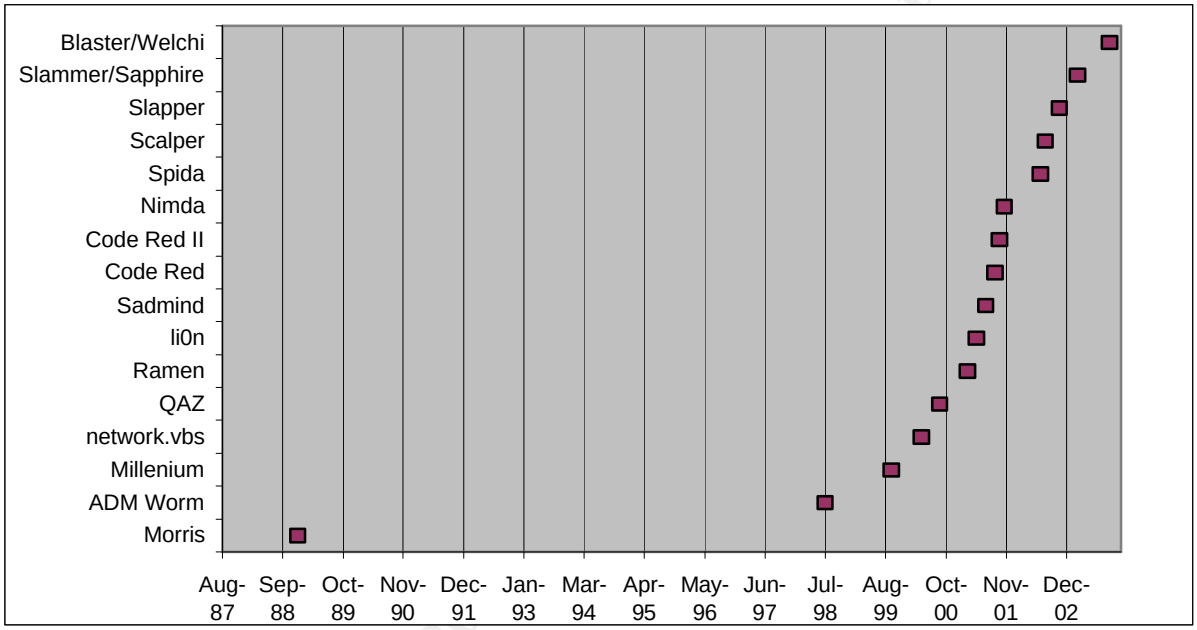

Figure 1: First Reported Occurrence of Select Worms[1]

As can be seen in Figure 1 above, prior to 1998 worms were essentially a non-issue. Since that time, however, the release and discovery of new worms has become a regular occurrence. Bear in mind also that the above represent the most notorious of the bunch; there are a host of minor worms and major worm variants as well.

The increasing prevalence of worms is not, by itself, necessarily a cause for concern. However, there is good evidence to support the contention that worms are becoming more virulent as well. Consider the following chronology:

- network.vbs, February 2000: This worm had no payload and spread via unprotected Windows shares (Singer). While there were enough reports of infection for CERT to issue an advisory on this worm, it does not appear to have been associated with any notable disruptions to Internet traffic.
- Ramen, January 2001: This worm targeted RedHat Linux systems via exploits that were 4 – 7 months old and, aside from defacing web pages,

[1] Dating of first appearance of worms is approximate and is based on the reported dates contained in the source material for each worm. Please see the List of References for source material on each worm.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

did not appear to be particularly malicious (Coolbaugh). However, as noted by the Linux Weekly News, multicast traffic was affected as a byproduct of the worm's scanning mechanism, resulting in degraded service over the MBONE for both unicast and multicast traffic.

- Code Red II, August 2001: CNN This worm caused major disruptions due to both infection and the concomitant increase in bandwidth consumption. Major players such as Qwest, Microsoft, and AT&T were affected; the overall cost of Code Red II and its predecessor, Code Red, was estimated to be at least $2 billion (Sieberg).
- Blaster, August 2003: Blaster caused network outages and financial damage of similar scope and magnitude to Code Red/Code Red II (Varghese). Additionally, there is good reason to suspect that it may have played a contributory role in the August 14, 2003 blackout on the East Coast (Verton).

This chronology is intended to demonstrate that worms are evolving from a nuisance to a genuine threat to business, the Internet, and possibly the public at large. More importantly, it takes no great stretch of the imagination to project that this trend will continue as worm writing evolves from a "black art" to a known science: The more worms are released, the more they will be studied. The more they are studied, the more information about worms and worm writing is disseminated to the technical community at large. The more information is available about worm writing, the easier it is to write one. The easier it is to write one, the more worms will be released, and the cycle goes on to repeat itself.

Most of the worms to date, including all of the major ones mentioned above, were primarily concerned with propagating themselves and installing trojans. The havoc that they caused, rather than being intentional, has largely been a byproduct of their propagation mechanisms. There hasn't been a major worm outbreak with a truly destructive payload. It doesn't take much to imagine a worm that deliberately targets routing on backbone networks combined with destructive actions on the local host or some other, equally crippling payload. Some of the countermeasures suggested below may appear drastic to some but, when the potential impact of a truly destructive worm is taken into account, are probably long overdue.

4. Worm Propagation Methods

The following section contains an analysis of the propagation methods of various famous worms. As will become evident there are several propagation mechanisms that turn up again and again. By deploying countermeasures tailored to these specific mechanisms it is possible to prevent, contain, and/or slow the propagation of current and future worms.

4.1. Morris

The Morris worm, a.k.a. "the Internet Worm", is named after its creator, Robert T. Morris. Morris is/was the first worm, produced by Robert Morris when he was a

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

grad student just to see if it could be done. It could, and quite successfully; about 4000 of the Internet's approximately 60,000 (at that time) hosts were infected within 16 hours of the worm's deployment (Spafford).
Morris is also a great example of the adage "the more things change, the more they stay the same"; variations on its propagation mechanisms are used by worms to this day. Morris propagates via the following mechanisms (Page):

- The "debug" option in Sendmail: The worm connects to a Sendmail daemon on port 25 and issues the "debug" command to enter into debug mode. Once in debug mode the worm can pipe data through a shell, making it possible for the worm to compile itself from C code using the local C compiler and linker.
- A buffer overflow in fingerd: The buffer overflow was used to invoke /bin/sh, from which point the worm could compile itself as with the Sendmail exploit.
- Rsh/rlogin: The worm performs a dictionary attack against the local /etc/passwd file to obtain passwords for local accounts. Once a password is obtained the worm logs in to that account (presumably via 'su', 'login', or similar mechanism) and accesses the account's .rhosts file if such a file exists. Using the contents of .rhosts Morris can identify systems that are likely to trust the local host; Morris can then propagate itself to these hosts via rsh or rexec without having to supply a password.

4.2. ADM Worm

The ADM worm propagates via a buffer overflow in Unix systems running DNS server daemons derived from v 4.9.6 of the ISC BIND code. The worm performs an incremental IP scan, starting from a random IP address, looking for DNS servers which support the IQUERY command. When such a server is encountered the worm attempts to exploit a buffer overflow in IQUERY response processing which, if successful, allows the worm to create an account for itself on the exploited host along with a setuid root shell. This account and shell are used to transfer the worm's tarball to the targeted host via ftp, at which point the tarball is untar'd and the worm is executed on the target host, beginning the propagation process all over again (Vision).

The ADM worm is an example of the early genesis of network worms. ADM and other early worms (Millenium, Ramen, li0n, and Sadmind specifically) are composed of the following components:

- IP Scanner: A mechanism for selecting IPs to target.
- One or more exploits: Pre-existing, programmatic-attack type exploit used by the worm to escalate its privilege level on the targeted system.
- Propagation mechanism: Provides the logic necessary to move the worm archive from system to system, usually via the use of ftp or tftp.
- Glue/misc scripts: These scripts tie the other components together and provide worm-specific functionality.

As can be seen from the above, the earliest worms do not appear to have been written from the ground up. Rather, they took pre-existing, stand-alone exploits and glued them together to produce a self-propagating system. By the standards of today's worms they are large, cumbersome, and not terribly virulent. The basic

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

architecture described above, however, is still popular today and can be found in more recent worms such as Slapper and Scalper (Arce) and Blaster (Miller).

4.3. Network.vbs

The Network.vbs worm propagates via unprotected Windows shares. The process as described in CERT Incident Note IN-2002-02 is as follows:

1. Perform a pseudo-random IP scan, looking for hosts with Windows filesharing enabled.
2. Attempt to mount the share named “C” as local drive J.
3. If mount is successful copy network.vbs script into the “Startup” program group.

Provided that the above is successful, the worm will be executed the next time someone logs into the system. It should be noted that the QAZ worm uses a similar mechanism, enumerating hosts within the “Network Neighborhood” and replacing notepad.exe with the worm binary (Yamamura).

4.4. Code Red

The Code Red worm spreads via a buffer overflow in IIS’ Indexing Services (“the .ida vulnerability”). Infection begins when the worm issues an HTTP GET command to a vulnerable IIS system. The GET command contains just enough code to exploit the .ida vulnerability, which is used to cause the system to execute the actual worm code contained in the body of the GET request. This code contains the logic for scanning and infecting additional hosts using the same mechanism, allowing the worm to propagate without the need for file transfer or the use of bundled scripts/exploits. The Code Red II worm uses the same infection vector and a similar mechanism for propagation (eEye Digital Security).

4.5. Nimda

Nimda is one of the first worms (with the exception of Morris) to use multiple mechanisms for propagation. It is this robustness that allowed Nimda to become widespread so quickly. As reported by Incidents.org, Nimda uses four distinct mechanisms to propagate itself:

1. **Web Server Attack:** The worm scans IP addresses looking for IIS servers that may be exploitable. Upon finding an IIS server, the worm attempts to obtain a local directory listing using various directory traversal exploits or the backdoors left by infection with the Code Red II virus. A successful directory listing indicates that the server is vulnerable, and the worm proceeds to TFTP a DLL from the infecting host to the targeted system using the same mechanism. This DLL contains the Nimda code and is executed on the local system via directory traversal exploit or Code Red II backdoor.
2. **Email:** Nimda harvests email addresses on the infected system via the Windows Messaging Application Programming Interface (MAPI), allowing it to extract addresses from the most widely used Windows email clients.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

It also harvests addresses from certain HTML files stored on the local system. Once the list of addresses is compiled the worm mails itself to each address as a MIME-encoded executable. The format of the message is such that rendering the message with Windows mail clients that use an unpatched version of Internet Explorer (IE) will cause the attachment to execute. Systems without a vulnerable mail client can still be infected if the user executes the attachment.

3. **Browser-Based Attack:** Nimda modifies content on infected web servers, attaching a small piece of JavaScript code to a variety of HTML and ASP files. This code causes browsers to attempt to open a MIME-encoded copy of the worm similar to that used for email propagation. While most browsers won't do anything with the file, certain versions of IE will automatically execute it, infecting the local system. Nimda is the first worm known to propagate itself in this fashion; this mechanism is especially nasty because it allows the worm to propagate through firewalls
4. **File Infection:** Nimda leaves copies of itself throughout an infected system using various innocuous- or important-sounding names and attaches itself to existing executables. This mechanism is applied to both local and network drives, allowing the worm to propagate to additional systems. Executing any of these files will cause the system to become infected with the worm. In certain instances the worm will also modify the system so that a copy of itself is executed on system startup.

As can be seen from the above, Nimda used a variety of mechanisms, some of them quite clever and sophisticated, to propagate itself. Using multiple, independent mechanisms increases the chances that the worm will be able to propagate itself, even in a well-maintained and secured network.

4.6. Spida

The Spida worm is similar to the early Unix worms discussed above in that it consists of a collection of scripts and binaries that are moved from system to system through the used a remote exploit combined with network file transfer. In this case the worm takes advantage of Microsoft SQL Server installations that have the default blank password for the 'SA' account. The worm scans for MSSQL servers, attempting to log in as user 'sa'; a successful login to the server using this account gives the worm privileged access to the database, which includes the 'xp_cmdshell' stored procedure for executing shell commands. The worm uses the stored procedure to escalate its privileges, transfer the worm code to the target system, and execute that code, completing the propagation process (eEye Digital Security).

4.7. Slammer

The Slammer worm is a very efficient execution of the basic worm architecture discussed above. It consists of an IP scanner combined with an exploit for MS SQL Server, written in 376 bytes of code (Moore). And the fact that it transmits via UDP, a connectionless protocol, eliminates much of the overhead (and some

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

of the safeguards) associated with using connection-based protocols such as TCP. The propagation mechanism for Slammer is essentially the same as a number of others worms which have been discussed. An infected system creates a UDP packet that contains the worm payload. This packet is sent to randomly generated IP addresses; if a vulnerable version of MSSQL is listening at the destination it will be infected with the worm via buffer overflow and the whole process beings again (eEye Digital Security).

4.8. General Model of Worm Propagation

Having identified the propagation mechanisms for a number of notable worms, it is now possible to synthesize a generalized model for worm propagation and discuss how specific countermeasures can be used to disrupt various steps in this process. The proposed model is as follows:

© SANS Institute 2004, Author retains full rights.

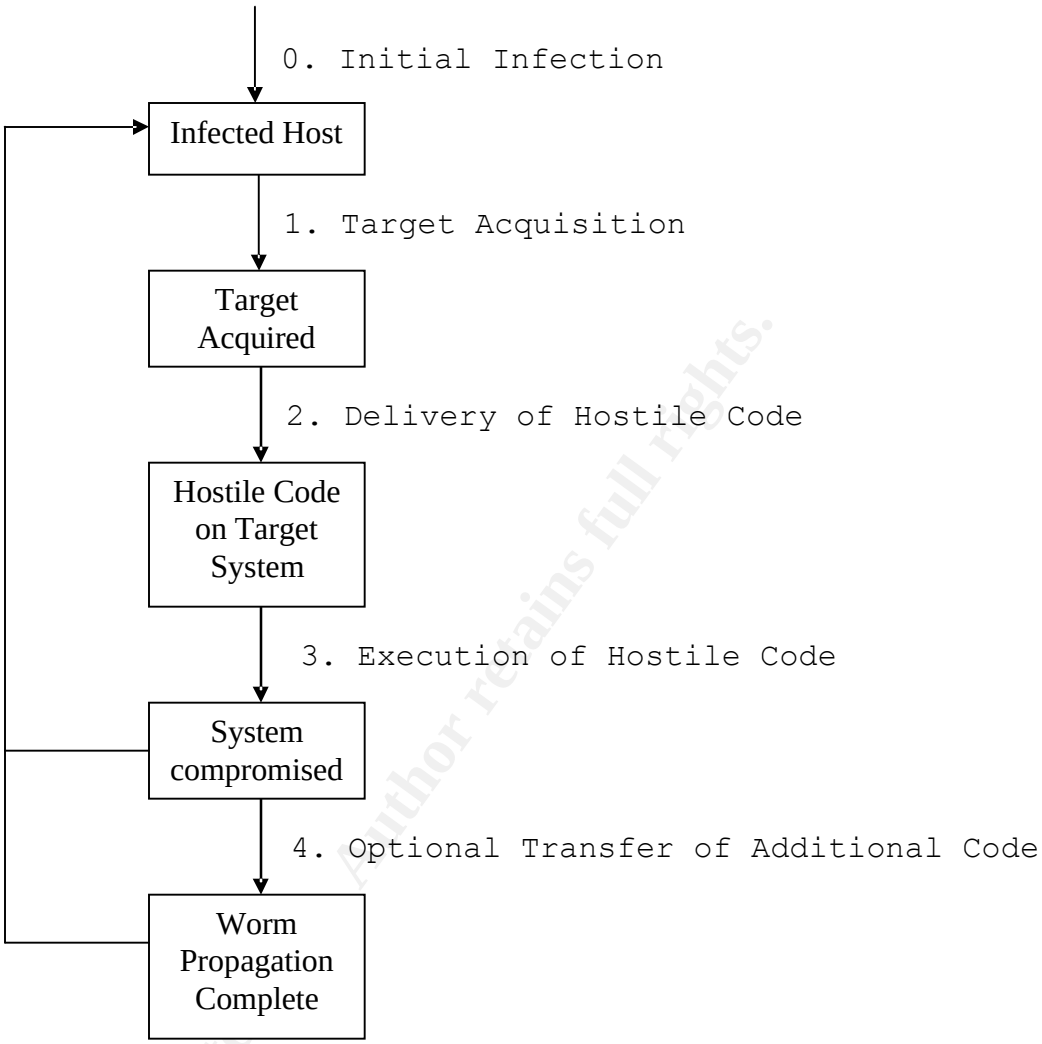

Figure 2: Generalized Model for Worm Propagation

Based on the worm descriptions in section 4 it appears that worm propagation can be broadly described by a 3 (or 4) step process illustrated in Figure 2:

0. Initial Infection: The model begins with the presumption that there exists a system that is already infected by the worm and that the worm is active on this system.
1. Target Acquisition: In order for the worm to propagate itself it must find additional systems to infect. Worms may actively target systems using:
    a. IP addresses
    b. Email addresses
    c. File system traversal

    It should also be noted that worms may passively target client system i.e. the trojaned web content delivered by web servers infected with the Nimda worm.
2. Delivery of Hostile Code: Once a system has been targeted, it is necessary to transfer the worm (or some portion thereof) to the targeted

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

system in preparation for infection. Code delivery has been observed to take place via the following:
    a. Network filesystems
    b. Email
    c. Web clients
    d. Remote command shell (or equivalent)
    e. As part of packet payload associated with buffer overflows and similar programmatic exploits.
3. Execution of Hostile Code: The presence of hostile code on a system is not sufficient for worm propagation; execution of the code must be triggered in some fashion. Code may be executed via:
    a. Direct invocation from the command line (or equivalent)
    b. Buffer overflow or other programmatic attack
    c. Email clients
    d. Web clients
    e. User intervention
    f. Automatic execution by target system.
4. Some worms may only transfer a portion of their code in step 3. In that case it is necessary for them to transfer the remaining code once the target system has been compromised. This can be achieved via
    a. FTP/TFTP
    b. Network file systems

5. Propagation Countermeasures

With a generalized propagation model in hand it becomes much easier to develop and deploy effective countermeasures. For the purposes of the following discussion an “effective countermeasure” is any mechanism that prevents, contains, or slows the spread of a worm. The analysis below examines each step in the propagation model in detail to determine what countermeasures, if any, prove effective.

5.1. Target Acquisition

During the target acquisition phase an infected system seeks out other systems that may be susceptible to infection. The specific targeting mechanism varies based on the means by which the hostile code will be delivered to the target system.

5.1.1. IP Scanning

The most popular method for targeting systems to date seems to be IP scanning. This should come as no surprise since the vast majority of services offered on the Internet are designed to be accessed, directly or indirectly, via an IP address combined with a TCP port number. The most basic scanning algorithm is as follows:

1. Generate an IP address.
2. Perform local setup for network communication.

© SANS Institute 2004, Author retains full rights.

3. Attempt to connect to the targeted system by sending a TCP SYN packet to <Targeted IP Address>:<Port of Targeted Service>.
    a. If a TCP SYN-ACK packet is received then the remote system at <Target IP> is listening on <Port of targeted service>. Send an ACK packet and proceed with transfer of hostile code.
    b. Receipt of any other type of packet from <Target IP>, or failure to receive any packet after a certain number of tries, indicates that the targeted service is not available for some reason. Return to step 1.

The above describes a *connect scan* as described by Fyodor in the Nmap documentation. Since the scanning system is completely compromised there's nothing that can be done locally to prevent or disrupt the scan. However, step 3 involves interaction with the outside world, so external countermeasures can potentially be deployed to disrupt those steps.

The simplest countermeasure to deploy is also the most effective; unneeded services should be turned off. In this situation, the infected host sends a SYN packet that is received by the target host as usual. However, since the service is turned off, there is no process listening on the destination port on the target host. The proper response in this situation is for the target host to send back an RST packet, the receipt of which tells the infected host that the targeted service is unavailable, causing the infected host to move on to the next target (Loop). Services that use other IP-based protocols (such as UDP) can be scanned for and treated in a similar fashion.

The efficacy of the "turn it off" approach should not be underestimated. Very often services are left on even when they aren't needed, resulting in preventable security breaches. For example, the Blaster worm took advantage of a buffer overflow in Windows RPC DCOM, a service that allows programs to perform remote procedure calls over the network (Bautista). However, most applications and, consequently, most users, have no need for this functionality so DCOM can be turned off without affecting system functionality (Gibson). This approach suffers from two shortcomings: 1) it is necessary to leave services on if they are to be used and 2) the SYN and RST packets can still cause congestion on the network path between the infected host and the hosts which are being targeted. Both of these issues can be solved by appropriately deploying some kind of firewall to protect services that must be left on.

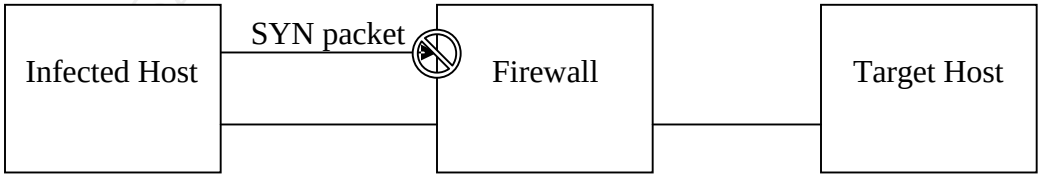

Figure 3: Basic Firewall Operation

In a typical network configuration a firewall is deployed somewhere on the network path between the infected host and the target host as show in Figure 3 above. When the infected host sends a SYN packet to the target host the packet is first intercepted by the firewall. The firewall is configured to prevent most

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

systems (presumably including the infected host) from accessing services on the target host, which is (usually) achieved by silently discarding the SYN packet. The infected system will generally send several more SYN packets that will be treated in the same manner, after which the infected system will assume that the targeted service is unavailable and move on to the next target (Curtin). It should be noted that most other protocols (including UDP) can be firewalled in a similar manner with similar results.
Using a firewall in this situation has many benefits. Of primary importance is that, when deployed appropriately, it prevents infected systems from targeting active services, stopping the propagation process at an early stage. It also reduces the impact of the infected host on the network. Preventing the SYN packet from reaching the target host prevents the generation and transmission of reply traffic. Furthermore, the closer the firewall is place to the infected host the earlier the SYN packet is dropped, preventing it from causing congestion closer to the target host. A final benefit, at least in the case of TCP communication, is that silently dropping the SYN packet causes the infected system to timeout and resend the SYN packet several times. This slows down the rate at which the infected system can scan for targets which, in turn, further reduces network congestion and slows the rate of propagation.

The discussion above covers the essentials of how to prevent propagation during the IP scanning phase. Recall that in addition to preventing propagation a countermeasure is also considered effective if it contains or slows propagation. To this end there are two other technologies to consider, intrusion detection systems and honeypots.

An intrusion detection system (IDS) can aid in the containment of an infection by identifying infected hosts shortly after they begin scanning for targets. Infected hosts generate a unique traffic signature, attempting to initiate contact the same service (or set of services) on a wide variety of hosts in a short period of time. Modern IDSs typically include a portscan detection component that can raise an alarm when this traffic pattern is observed (Farshchi). Alarms of this nature can alert system administrators that their systems are infected so that appropriate steps can be taken to deactivate and/or disinfect the system in a timely fashion.

Honeypots are another technology that can help contain and slow the spread of worms. The application of honeypot technology to is a relatively new development in the security community's ongoing efforts to combat worms. The essential idea behind the deployment of honeypots in a worm-fighting context is to force worms to spend time and effort attacking hosts that aren't actually vulnerable to the worm, thereby depriving the worm of the opportunity to attack hosts that are vulnerable. Most of the work in this area hasn't progressed beyond the theory stage, but there are a couple of implementations of honeypot technology that have proven successful.

One of these is *honeyd*, a daemon that can be configured to respond to attempts to connect to unused IPs on a network. Under normal conditions an infected host will scan a network, but only a certain percentage of IPs on that network will respond, the remaining IPs being unallocated. With *honeyd* installed it can

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

appear as if every IP that the infected host tries to connect to is live, even though a single host running *honeyd* is spoofing most of those IPs. The infected host will attempt to infect each of the spoofed IPs, a futile activity since the *honeyd* host isn't vulnerable to the worm. This has the overall effect of reducing the rate at which hosts are infected, slowing the spread of the worm (Oudot).

The other widely spread tool in this area is the Labrea daemon. This tool functions in a manner similar to honeyd, responding to connection attempts for unused IPs on a network. In this case the objective is to slow down the worm by slowing down the speed of the TCP session established between the infected host and the Labrea daemon. This is achieved by permanently setting the TCP window size to 0, which has the effect of allowing a TCP conversation to take place between the infected host and the Labrea daemon without any actual data being exchanged. In theory this approach can cause the infected host to spend an arbitrarily large amount of time trying to infect the Labrea host, thus slowing the rate at which the worm can infect other hosts (ibid.).

5.1.2. Targeting via Email Address

Worms can find additional systems for infection by scouring infected systems for email addresses in preparation for code delivery via email. This can be done via a host-provided API or by mining the contents of any accessible files. Assuming that the infected host has been completely compromised, however, there's very little that can be done to prevent this since this targeting activity is confined to the infected host.

5.1.3. Targeting via Network Filesystems

Worms may also target additional systems via simple network traversal. In this scenario the infected system has mounted one or more remote filesystems prior to becoming infected. In modern operating systems the details of local vs. remote are generally hidden from applications, so a worm that delivers hostile code via file infection can deliver this hostile code to a remote system without even known it. As with email addresses there is little that can be done to prevent this type of targeting. The infected host had the appropriate access to mount the remote filesystems prior to infection; revoking this access would require that the remote host be informed that the infected host had become infected. There does not appear to be, at this time, any intrusion detection/prevention system or other security framework with that level of sophistication available to the public.

5.2. Hostile Code Delivery

Once an infected host has targeted a system the next step in the propagation process is finding some way to transfer the worm's code to the target system, since the code must reside on the target system before it can be executed. What follows is a discussion of the most common methods for transferring hostile code to a target system and the ways in which this transfer may be disrupted.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

5.2.1. Network Filesystems

As discussed above, worms may target remote systems via network filesystems that were mounted on the infected host prior to infection. Worms may also try to mount network filesystems themselves. In either case the ultimate goal is to be able to write hostile code directly to the remote filesystem in the hopes that a means will be found to execute it at a future date.

This being the case, the first step in stopping the delivery of hostile code is to prevent the filesystem from being mounted. It's too late to do this in the case of pre-mounted filesystems, but worms can still be prevented from actively mounting filesystems themselves. Worms such as Network.vbs and QAZ take advantage of weak (or non-existent) authentication requirements for mounting volumes, so this type of worm can be stopped by requiring that clients authenticate themselves in some fashion to the file server prior to mounting a volume.

A simple and effective authentication mechanism is IP-based access control i.e. allowing or denying access based on the requesting system's IP address. For example, according to Sun's documentation, the Unix Network File System allows the administrator to set up access control lists to determine which systems have read and read/write access, and to disallow access from any system not explicitly listed. This kind of access control will help prevent worms from spreading between administrative domains, but will not, as a general rule, prevent a worm from spreading within an administrative domain, since an infected system in this context is likely to be an authorized client of the file server.

Even if the infected system is an authorized client of the file server it is still possible to prevent it from mounting network volumes by increasing the authentication requirements. In addition to requiring that the infected system be on an access list it is also feasible to require the system to supply credentials of some kind prior to allowing it to mount a volume. For example, Windows Server Messaging Block (SMB) file sharing protocol can be configured to require a password or username/password to access a shared volume (Microsoft Corporation). Requiring a username and/or password should be sufficient to prevent most worms from mounting the share. Even in the case where the infected system launches a dictionary attack against the share (Shevchenko) choosing a strong password should generally be sufficient to thwart this type of attack (University of Northern Iowa).

Even if the worm manages to successfully mount a remote filesystem it still has to deliver the hostile code. Delivery can be stopped at this point by the simple expedient of making the filesystem read-only, but this approach is only feasible in situations where it makes sense for the volume to be read-only, such as sharing program binaries or data files (Linux Documentation Project). Very often the whole purpose of a network volume is to provide a centralized place for users to store their files, requiring that the volume must be available for writing as well as reading.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Even if the worm manages to mount a network share in read/write mode, it still faces the challenge of actually delivering the hostile code via the network filesystem. Any technology that has the ability to monitor the flow of data over the network can, in theory, be applied to this end. The only requirement is that the technology in question must understand the network filesystem protocol that is being used to deliver the code. This is problematic, since active interception technologies such as Symantec, Immunix Corp., and Fortinet Corp. focus their analysis on a few common protocols such as HTTP, SMTP, and FTP. This leads to the conclusion that there is not, at this point in time, any commonly available technology for preventing the transmission of hostile code via network filesystems.

The last step in transmitting hostile code to a remote server via network filesystem is the reception of the code by the file server and the subsequent writing of that code to the storage media. To prevent the successful delivery of hostile code, the file server must inspect the data being written and determine that it is of a malicious nature. Once this determination is made the server can deny the write request or take other action as appropriate. In essence this describes the operation of antivirus programs (de la Cuadra), the caveat being that the program must examine the data on the fly rather than during a scheduled scan. At least one antivirus program, the Corporate Edition of Symantec Antivirus, supports this kind of realtime identification of malicious code (Carnegie Mellon Computing Services), indicating that this approach is technically feasible. It should be noted that this approach to prevention will, in general, only work with known worms. Antivirus software generally uses a database of known virus signatures to identify virus-bearing files, so a new worm will not be detected until a suitable signature for it has been developed.

In addition to preventative measures there also exist several technologies that allow system administrators to detect the delivery of hostile code, the hope being that if the delivery is detected early enough the hostile code can be removed before it is executed. As mentioned above, antivirus programs are particularly suited to this task. Scheduling frequent scans of file servers is a good way of catching hostile code before it can be executed. Two other technologies that can also prove useful in this endeavor, intrusion detection systems and file integrity checkers, are described below.

Intrusion detection systems were mentioned previously in conjunction with the detection of IP portscans. In addition to this capability, most IDSs use signature-based intrusion detection, a mechanism for detecting hostile activity that relies on a database of signatures in much the same way that an antivirus program does. As such, an IDS allows for the timely detection of the transfer of hostile code between the infected host and the target system, provided that a suitable signature can be devised to characterize that transfer. As with anti-virus programs, signature-based IDSs will not, in general, catch newly released worms, limiting their usefulness in combating the initial spread of any particular worm (Symantec Corporation).

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

The other detective technology that can be deployed to useful effect in this situation is a class of software designed to protect file integrity, the most well know example of which is the Tripwire package. Unlike anti-virus programs and intrusion-detection systems, Tripwire does not rely on a database of signatures to detect hostile code. Rather, it takes a baseline snapshot of the protected system at a time when it is not infected. The current state of the system can then be compared against the baseline at any time in the future, making it easy to detect the delivery of hostile code due to the deviations that it causes from the baseline snapshot. Because this type of system doesn't rely on a signature database, but rather on monitoring changes from the baseline, it can provide an effective warning system even against unknown worms.

5.2.2. Email

Code delivery via email is a favorite mechanism of worms and worm-like viruses. The process begins with the worm composing a message containing hostile code and attempting to send that message to the targeted email address. The path that the message follows will vary depending on the characteristics of the worm and of the system that the message is being sent from. The worm may use a built-in SMTP engine for sending the message, in which case it is likely to try to deliver the message directly to the mail exchange for domain being addressed. Alternately, the worm may use the infected host's messaging mechanisms (a messaging API, /usr/bin/mail, etc.), in which case the message may be sent to an outgoing SMTP relay, passing through a number of intermediaries on the way, or it may be sent directly to the mail exchange for the addressed domain. Regardless of which path the message takes it eventually ends up on the system that will be used to read the message, at which point the delivery of hostile code is complete (Brain).

SMTP is a (notoriously) promiscuous protocol; it was originally designed in an era when it was desirable to allow any system to send email to any other system. Because this behavior is still widely desired, much of this promiscuous nature remains today. It is generally possible to send a message via SMTP between two arbitrary systems without requiring much in the way of authentication. Although mechanisms such as SMTP AUTH (Meyers) have been devised to try to limit this behavior, the unavoidable fact remains that, for SMTP to retain its widest possible utility, it must be possible to receive a message from a previously unknown party without needing to set up an authentication mechanism offline. As such, the discussion that follows proceeds on the assumption that the use of SMTP authentication mechanisms is not suitable for preventing worm propagation.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

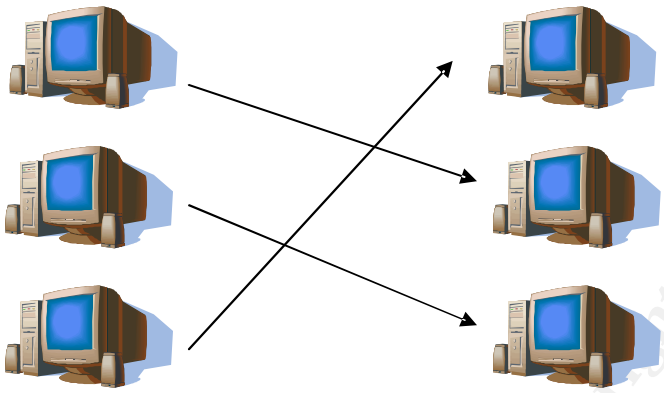

Figure 4: Email Delivery in Standard Configuration

With this in mind the model for hostile code delivery via email becomes one of multiple, unrestricted delivery paths between the sending system and the recipient as shown in Figure 4 above. In order to prevent worm propagation it is first necessary to reduce a multiplicity of delivery paths to a small, finite number of paths that can be more closely monitored. Once again the judicious application of firewall technology can achieve this effect. System administrators can configure their firewalls to block outgoing connections to port 25 originating from most of the systems that they control, and force clients to send email via a designated mail relay. With this block in place, it is no longer possible for an arbitrary, infected system to send email directly to a target system. If the worm provides its own SMTP engine rather than making use of the messaging mechanisms provided by the host then this may very well stop code delivery entirely, since the worm will not know to send the email via the mail relay. System administrators can also apply this logic to incoming email, blocking most incoming connections to port 25 at the firewall, ensuring that email can only be received via a designated mail exchange.

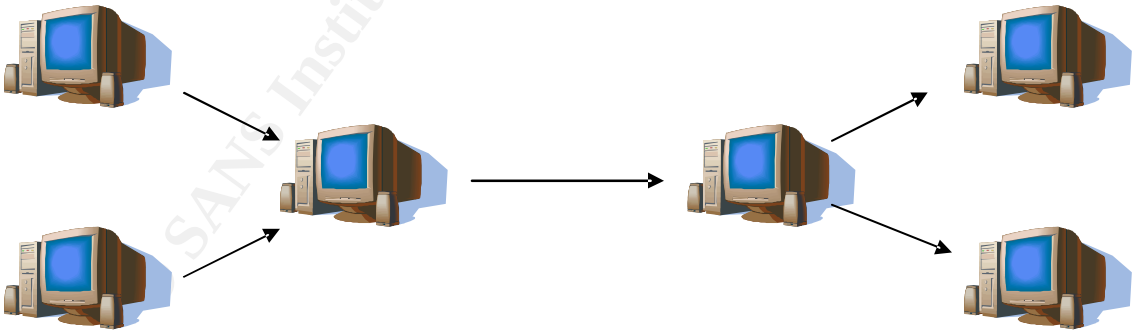

Figure 5: Mail Delivery Using a Designated Relay and Mail Exchange

With these controls put in place mail delivery takes on the form shown in Figure 5 above. This configuration forces the infected system to deliver the email via the designated relay and, furthermore, forces that email to be received by the designated mail exchange, significantly reducing the number of potential delivery paths that the system administrator must monitor.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Having reduced the number of delivery paths to a manageable number, the challenge now is to detect hostile code traversing one of these delivery paths via SMTP and prevent it from being delivered. As discussed previously, the infected host is assumed to be completely compromised, eliminating from consideration any countermeasures that might be deployed on the infected system itself. This being the case, the next potential location for preventing the delivery of the hostile code is at the designated mail relay.

The designated mail relay is an ideal place to detect and stop the delivery of hostile code via email, since this system sees all email leaving the administrative domain. This is the appropriate place to deploy one of the anti-virus gateway/firewall systems mentioned above in the discussion of network filesystems. These systems are designed specifically for the task of scanning a large volume of email for the presence of viruses, worms, etc. Email that is found to contain a virus can be discarded at the gateway, thus preventing it from reaching the target system. This type of technology can, and should, be deployed on the designated mail exchange as well, which will catch those emails that weren't subjected to scanning when leaving their administrative domain.

As with other forms of anti-virus technology, these gateways make use of a signature database that may fail to catch new worms. It is interesting to note, however, that the emails generated by worms share many characteristics in common with another form of malicious email, namely spam (Leavell). To this end it is also beneficial to deploy anti-spam software such as Spam Assassin in conjunction with anti-virus software. The added value here is that anti-spam software doesn't rely on a database of signatures to detect spam. Rather, most anti-spam packages use a number of heuristic tests (forged or non-existent address, non-RFC compliant headers, suspicious formatting/content, etc.) in combination with a scoring system to determine whether or not a message is spam. This means that anti-spam software has a significant chance of catching email sent by a worm, even if that worm doesn't yet appear in any anti-virus databases.

5.2.3. Web Clients

Preventing the delivery of hostile code to web clients is the same, in many respects, to preventing the delivery of hostile code via email as describe above, Web content is delivered via HTTP, a protocol that was designed to allow arbitrary systems to deliver content to any client that might request it. As with SMTP, the authentication and authorization requirements of this protocol are minimal or non-existent, once again in an effort to ensure maximum utility of the protocol. In fact, the situation looks remarkably like that shown in Figure 4 above, with multiple systems transmitting data to other systems in a fairly promiscuous manner. In this case the data is "pulled" by the client rather than being "pushed" by the server but, as demonstrated by the Nimda worm, even trusted servers can be booby-trapped, so the fact that the data is pulled is of minor help at best.

Since the problem of code delivery via HTTP looks so similar to delivery via SMTP, it is not unreasonable that a solution to the problem might take on a

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

similar form as well. It turns out this is exactly the case. As with SMTP, the first step in preventing delivery of hostile code is to reduce the number of delivery paths that must be monitored. Once again, this can be done with firewalls by preventing most systems in an administrative domain from making outgoing connection requests to ports 80 (HTTP) and 443 (HTTPS). Rather than being allowed to establish HTTP connections directly, these systems must use a designated HTTP proxy.

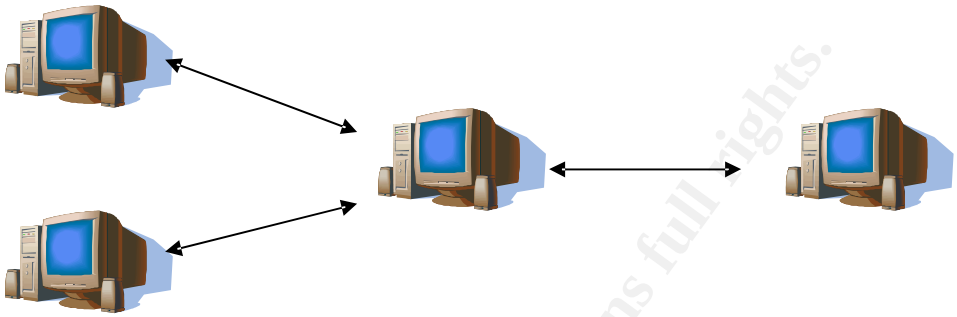

Figure 6: Proxy-based Web Browsing

Forcing clients to use a designated proxy for web communication causes web content delivery to take on the form shown in Figure 6 above. Clients send requests for web content to the proxy, which then forwards the request on to the appropriate web server. The web server, in turn, provides the proxy with the requested content, which the proxy sends back to the requesting client.

Once again, the number of potential paths for the delivery of malicious code have been reduced through the use of a proxy system, making active monitoring of the paths feasible. As with the discussion of email above, it is assumed that the server delivering malicious content has been completely compromised, suggesting that the first place where malicious content can be intercepted is the designated web proxy. What is needed here is some kind of technology that can monitor the HTTP data flowing from the server to the client and screen out hostile code. Not surprisingly, many of the anti-virus gateway/firewall systems discussed above also have an HTTP filtering component that performs just this function, allowing for the interception of malicious content prior to it reaching the web browser.

5.2.4. Remote Command Shell

There are multiple instances of worms (i.e. Morris and Spida) using command shells or command shell equivalents to deliver malicious code to target systems in the form of command sequences to be executed on the target system. In this scenario delivery begins when the infected system makes contact with a network service that provides access to a command shell in some fashion. This access may be provided directly through a command line service such as Telnet or RSH, or it may be provided indirectly as an extension of another service, such as the 'xp_cmdshell' stored procedure in MS SQL Server or the debug feature in older versions of Sendmail.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

The first line of defense is to prevent random systems from being able to access the services in the first place. Due to the nature of these services, however, the solution is often more complicated than simply firewalling the services off from the majority of the outside world. Command shell services are frequently used by system administrators to perform remote administration of the systems under their control. This is especially true in 'lights out' data center operations, where the administrator may be hundreds or thousands of miles from the systems being administered. To complicate matters further, administrators may come from arbitrary IP addresses, especially if they are working from home or are on the road.

What is required here is some means by which one set of arbitrary IP addresses can connect to specific services while other arbitrary IP addresses are denied access. This kind of technology exists in the form of Virtual Private Networks (VPNs). To achieve the desired end, the system administrator would start by deploying a VPN access device of some type at the edge of the network and providing mobile and/or home systems with the appropriate VPN client software. The client software establishes an encrypted connection with the VPN access device that can be used to tunnel traffic to the rest of the internal network (Tyson). When deployed in conjunction with a firewall this setup allows authorized individuals to access command shell services while preventing anyone else from accessing them.

There is still the problem, however, of attacks from within the administrative domain. In this case both the infected system and the target system are on the same network, so there is no firewall to prevent communication between the two systems. In this situation the delivery of hostile code must be prevented by the command shell service itself. This can be achieved by requiring sufficiently strong authentication, where 'sufficiently strong' is defined as 'good enough to stymie a worm'. This standard appears to be fairly low; Morris propagates via command line mechanisms which don't require a password (the Sendmail 'debug' option and RSH/rexec .rhosts authentication) while Spida takes advantage of a blank default password for a privileged account. As mentioned above, worms have been observed launching dictionary attacks against remote systems, but it appears that the desire to limit the size of worms have prevented these attacks from becoming to thorough. Enforcing good password practices as described by the University of Northern Iowa should be sufficient to prevent worms from gaining access to command line services.

5.2.5. Programmatic Attacks

A 'programmatic attack' is any attack that takes advantage of a flaw/problem with one or more programs running on the targeted system. Many of the worms described above (Ramen, li0n, ADM, etc.) rely solely on programmatic attacks, making this propagation vector particularly important in the grand scheme of things.

Unfortunately, it's also hard to make any sort of generalized statement about how to prevent the delivery of hostile code in this type of situation. In a typical

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

scenario, an infected host will contact a service on the target host and send it data/code that is specifically designed to make the program malfunction in a certain way. In order to prevent this code from being delivered it would be necessary to have an intermediary examining the code in a fashion similar to what is described above for SMTP and HTTP. The problem is that there are no filtering proxies available for most of the services that may show up on a network, so this approach won't work for an arbitrary service.

This does suggest that an interesting and fruitful direction for security research would be the development of a generalized proxy framework. A system running this hypothetical technology would be situated between the outside word and the internal network and act as a proxy for arbitrary services. In most instances this proxying would be transparent to the systems conducting the conversation but, if the proxy were to detect malicious code, it would interrupt the conversation and prevent the delivery of the malicious code. Since detecting hostile code is a well-understood problem, the main technical hurdle to clear would be developing a system that can gracefully interrupt a wide variety of protocols.

Nevertheless, the above is only a hypothetical and is of no use in the here and now. Since no magic bullet technology exists to solve this problem in a general fashion, the most that can be said about preventing code delivery in this scenario is that system administrators should follow best security practices: turn it off if it isn't being used, require authentication if possible, install an IDS system and keep it up to date.

5.3. Execution of Hostile Code

Having come this far, the worm has managed to deliver hostile code to the target system. However, this code doesn't serve the worm's purpose unless the target system can then be induced to execute that code. What follows is a discussion of the various means worms have been observed using to achieve this end and the steps that can be taken to try to prevent hostile code from being executed.

5.3.1. Email Clients

There are a number of mechanisms by which email clients can be induced to execute hostile code. None of these methods, however, are unique to email clients, so only a brief mention of each method will be made here. An email client may be induced to execute code in one of three ways:

- **Programmatic attack**: Email clients are vulnerable to programmatic attack just like any other program. Email messages may be formatted to take advantage of any programmatic weakness that the client may have. Successfully exploiting such a flaw can lead to execution by the email client. See below for a full discussion of countermeasures for programmatic attacks.
- **Rendering By-Product:** Modern email is a many-splendored beast, being possessed of multi-part MIME attachments, HTML formatting, any other miscellaneous goodies intended to make the email more expressive and/or readable. Sophisticated email clients automatically render this

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

content to make it readable/available to the user. This rendering process can include the execution of various types of content, making it possible to execute hostile code just by rendering the email. Such execution is usually controlled by a security subsystem, but this subsystem can be tricked into thinking that hostile code is safe to execute. See below for a full description of this type of attack.

- **User Intervention:** In the case where the email client was smart enough not to execute unknown code, it still may be the case that the user isn't. Being generally subservient programs, email clients will allow users to execute any attachment that an email message may contain, though they may warn the user against such action. Thus hostile code may be executed through direct user intervention; see below for a complete discussion of countermeasures.

5.3.2. Web clients

The web is a rich and interesting place, due in large part to the variety of content that is available. Content such as sound and video files must be decoded and played, either through the use of browser plug-ins or through the use of external helper applications. Furthermore, various technologies such as Java, client-side scripting languages, Active X, etc. require the web client to execute foreign code in order to present the full functionality of a web page to the user. In a nutshell, the dynamic nature of modern web content present many avenues by which a web browser can be induced to execute code.

Modern web browsers have dealt with this by dividing content into categories based on how much the content can be trusted. For example, Internet Explorer divides content into five trust categories, from content that originates on the local system (the "Local Machine Zone") to material that is found at an arbitrary site on the Internet (the "Internet Zone"). Few security restrictions are placed on content in the Local Machine Zone, while many restrictions are placed on content from the Internet Zone (Schnoll).

The problem is that web browsers, as a class, have a poor track record when it comes to enforcing these restrictions, due in large part to the inherent complexity of setting up and enforcing a restricted environment. For example, Nimda took advantage of a flaw in Explorer's security model that allowed it to induce web browsers to automatically execute the worm code. Problems such as this make it abundantly clear that browsers' restricted execution environments, by themselves, are insufficient to prevent worm propagation.

Fortunately, most browsers can provide the user with additional security, though this security comes at the price of functionality. Browsers can be made more secure by disabling as many extended features as possible. If a feature is turned off, there's significantly less risk that it can be used to subvert the local system. CERT recommends disabling all scripting languages and, for the particularly security sensitive, disabling other forms of content such as Java and ActiveX. Such restrictions will significantly increase browser security, but will limit the functionality of many web sites. As such, systems administrators should perform

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

a cost/benefit analysis before deciding to make deactivation of extended features a global policy.

5.3.3. Command Line Invocation

There is little to be said about preventing execution from the command line. The command line is, by its very nature, designed to facilitate the execution of code. If a worm has managed to gain a command shell on the local system it probably already has enough of a foothold to start propagating itself. The one saving grace in this situation is that a worm may require elevated access on the local system in order to carry out its propagation tasks; if it can be prevented from gaining this access then it can still be prevented from propagating.

The chances of preventing a worm from gaining a sufficient level of access can be significantly improved by systematically applying the principle of least access. Any network-accessible command shells on the local system should be configured to run at the lowest possible privilege level. This stance should be combined with a systematic tightening of any file control mechanisms with a goal of raising the required access levels to read, write, and execute files as much as is feasible. Successful application of these two techniques may result in a gap between the access level that a worm is granted via a command shell and the access level that it needs to carry out its tasks.

Once this gap is established, the worm needs to be prevented from escalating its privileges to bridge this gap. There are two main ways that a worm may escalate its privileges, through OS provided escalation mechanisms or through flaws in the OS security system. Many operating systems provide some means to temporarily increase the effective access level of a user, such as the 'su' command available in most flavors of Unix. Such mechanisms are often used by system administrators who occasionally need increased access to carry out their tasks. If such mechanisms are available they should be protected as much as is feasible through the used of password, administrator groups, etc. Worms may also gain access through flaws in the security system; keeping the local system updated with the latest patches can reduce the likelihood of this occurring.

5.3.4. Automatic Execution

Many operating systems provide facilities to automatically execute code in response to various events. Code can be run periodically, as with the Unix *cron* system (Raithel), or it can be triggered by asynchronous events such as a system boot or when a user logs into their account (Microsoft Corp.). The downside of such useful facilities is that worms such as Network.vbs can make use of them to trigger execution of hostile code.

This problem is closely related to the command line execution problem discussed above. If a worm can modify the appropriate files it can cause hostile code to be executed with system-level privileges. To prevent such an event from occurring it is once again necessary to create a gap between the privileges allowed to the worm and the privileges required to modify files that trigger automatic execution. For example, if the worm is accessing files by anonymously mounting a network

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

drive then it should be unable to modify any files that can trigger code execution in the context of a user or the system. Similarly, if the worm is accessing a filesystem locally with the privileges of a user account it should be unable to modify any files that can trigger code execution with system level privileges.

5.3.5. User Intervention

Despite the ongoing efforts of system administrators to educate their user base, users still remain a primary cause of infection by worms/viruses. Admittedly, some of these occurrences are not their fault; users can hardly be blamed for running a normally safe program such as notepad.exe which has, unbeknownst to the user, been booby trapped by a worm. Nevertheless, many users still persist in running programs that they have no reason to trust, as illustrated by the recent outbreak of the MyDoom virus (Sophos Corp.).

So what, if anything, can be done to stop this problem? Unfortunately, there seem to be few solutions, at least from a technological standpoint. The primary purpose of a computer is to execute code at the users request; any constraints that are put on this functionality directly impact the utility of the system. Systems can be locked down to the point where users cannot run unknown binaries, but doing so often has the side effect of reducing worker productivity and inducing an adversarial relationship between system administrators and their user base. Creating such a relationship hurts security in the long run as users are less likely to comply with security recommendations in such a situation.

There have also been attempts to create “trusted code” through various schemes for cryptographically signing binaries. There’s nothing wrong with the approach from a technical standpoint; it is quite feasible to set up the infrastructure needed to sign and verify code. The problem is that signed code can unequivocally identify it author, but is fundamentally unable to answer the question of “Is the author trustworthy?” (McGraw). The burden of deciding who is trustworthy is placed on the user, resulting in the MyDoom outbreak and similar incidents.

As such, it seems that it is best to attack this problem socially through user education. Users need to be reminded early and often to think twice about executing programs obtained from the web or attachments that arrive with email messages. In the end this seems to be a fight against human nature, which means that system administrators should accept that users will behave unwisely and should have contingency plans ready for this eventuality.

5.3.6. Programmatic attacks

Lastly, and probably most importantly, worms can induce target systems to execute code through the use of various types of programmatic attacks. This method of inducing code execution is particularly important because it has been used by many, if not most, of the modern worms. It is for this reason that finding a mechanism to prevent programmatic attacks is one of the holy grails of modern security research. Entire papers have been written on this subject, so this section will only hit the highlights of this field of study.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

There are, at this point, two widely exploited types of programmatic attacks. The first of these, the “buffer overflow”, is the oldest and most widely used. In this attack, a static program buffer (usually allocated on the stack for a local variable) “overflows” i.e. the program attempts to write more data into the buffer than the buffer can hold. In this situation the program keeps writing data past the allocated buffer space, overwriting other data items on the stack, possibly including the return pointer for the local stack frame. Attackers can take advantage of this flaw to cause the program to return to the address of their choosing, allowing the attacker to execute arbitrary code. This description glosses over many details of the procedure; interested readers should see “Smashing the Stack” by Aleph One for a complete description of how the buffer overflow process works.
A second type of attack, more recently developed, is the “integer overflow” attack. This attack makes use of the fact that an X-bit integer can represent, at most, the number 2^X-1. For example, suppose that a program is working with 8-bit integer values. The maximum value that 8 bits can represent is 255. So, under standard 8-bit integer arithmetic, 255 + 1 = 1, which is generally not the anticipated result. The effect of such errors varies depending on the context in which they occur; in certain conditions they can lead to buffer overflows or other exploitable anomalies (Howard).

Research into how to deal with programmatic attacks is, at this point, focused on three main areas: safe coding, exploit prevention, and exploit containment. The first of these areas, safe coding, focuses on modifications to programming tools and methods to reduce the risks presented by programmatic attacks. Efforts in this field focus first and foremost on programmer education. Many programmatic attacks result from mistakes on the part of the programmer, so training programmers to recognize and correct these mistakes goes a long way towards solving the problem (Graff). Safe coding research also looks at changes that can be made to programming languages to make them safer. The changes can be small, as in the introduction of safe substitutes for commonly exploited functions and constructs (LeBlanc), or they can be large, as in the implementation of entire languages designed to be safe from the ground up such as Java. However, unless they are writing or auditing code, most of this is largely academic to system administrators. What is of immediate help to them in combating the spread of worms are the results of the other two areas of study.

Much research has gone into ways to prevent successful exploitation of programmatic attacks. Bugs will slip through, even with the use of safe coding practices, so the next step is to make it harder to exploit the bugs at the OS level. There is a lot of promising ongoing research in this area, but most of the research has yet to bear fruit. One technology that has become viable is making various areas of system memory non-executable. This technology is available in two flavors, non-executing stack and non-executing heap, the availability of which is determined by hardware and OS. Non-executing stack, available on a number of operating systems, marks information held on the stack as non-executable. This is especially useful in defending against a subtype of buffer overflow in which hostile code is placed on the stack by directly preventing this hostile code from being executed (Wheeler). But this is only a partial solution; it

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

doesn't handle the case where hostile code is placed on the heap. Enter non-executing heaps, an analogous technology prevents the execution of data placed on the heap, thus making heap-based buffer overflows less likely. Non-executable heaps require special hardware support, so the chances are good that the option may not be available for a given hardware/OS combination (OpenBSD Journal). A potential downside of these technologies is that they may break programs that compose and execute code dynamically. While there seem to be few programs that rely on stack execution, interpreted languages such as Perl make extensive use of execution of data on the heap. Nevertheless, system administrators worried about worms should strongly consider enabling non-executable stacks/heaps on the systems they administer.

A final area of research looks at ways to *a priori* restrict the damage hostile code can do to the target system if it is executed. The problem with programmatic attacks is that the most successful ones allow the attacker to execute arbitrary code. Which is, in many cases, a violation of the principle of least privilege. There's no reason for BIND to be able to edit the system password file, so why should it be able to do so?

To this end people have developed sandboxing technologies such as Systrace to constrain the activities of running programs. Systrace allows system administrators to define ahead of time the set of actions that a particular binary is allowed to take. Programs are then monitored as they execute and prevented from taking unauthorized actions, which can prevent common exploit techniques such as causing the compromised program to execute a shell (Lucas). This approach is very effective; the major downside is that it can be very time consuming to configure this type of system.

In addition to sandboxing there are several additional measures that can be taken to contain a programmatic attack. Programs should be run with the least privilege needed to achieve their end; there's no reason to make an exception for network-enabled daemons. For better or worse, however, a long-standing tradition on the Internet it to treat port <1024 as especially privileged, which means that on many operating systems daemons need escalated privileges to bind to these ports. These daemons should be configured, where possible, to drop their privileges to an appropriate level once they have bound to the desired port. Reducing the daemon's privilege level makes it more difficult for the daemon to do damage if it is hijacked by a programmatic attack.

In a similar vein, most daemons don't require the full run of the system to achieve their tasks. As such, it makes sense to restrict daemons to a particular section of the filesystem, thus depriving them of access to system data files and executables in the event of a successful exploit. This can be achieved through the use of "chroot'd jails", a feature of many operating systems which allows them to "change the root" of the filesystem for a particular process, thus "jailing" the process in a secured subdirectory (Dreyfus)7.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

5.4. Additional Code Transfer

Some worms transfer additional code from the infected system to the target system once the initial exploit of the targeted system is completed. Unfortunately, if the worm gets this far there is likely little that can be done to prevent its spread. At this point both the infected host and the targeted host are completely compromised, so any preventative measures must be deployed between these two systems. Once again, an appropriately configured firewall may prevent the complete propagation of the worm. This underlies the importance of having a well-configured policy regarding outgoing connections in addition to incoming connections.

6. Summary of Findings

As has been amply demonstrated above, worms pose a serious and increasing threat to Internet security and stability. This being the case, system administrators must study the worm phenomenon and devise methods by which the spread of worms can be stopped. This may appear a daunting task at first, given the many and varied nature of worms and similar malware that have been unleashed against the network-using public. Security professionals should not be daunted; there is nothing new under the sun, and worms are no exception.

By examining historical and contemporary worms it becomes clear that, from a security standpoint, they merely represent variations on a few central themes. Highlighting these commonalities allows for the synthesis of a platform-agnostic model of worm propagation that can then be systematically analyzed to determine where security technologies can be deployed to prevent such propagation from occurring. This analysis has shown that all is not lost in the fight against worms and that there are many commonly available security technologies that can be deployed to help prevent the spread of worms.

More importantly, it demonstrates that global worm epidemics do not result from the misconfiguration of a few obscure options somewhere deep in the system. As demonstrated above, worm propagation is a multi-stage process in which a number of security technologies can be successfully deployed at each stage to help prevent, slow, or contain the spread of worms. This leads to the conclusion that mass worm outbreaks must be the result of generally lax security policies on a global scale rather than to a deficit in security technology.

An area for future research, then, must be the question of why there exists this appalling lack of security across the board? Some explanations to this problem are tendered on a perennial basis: not enough time, not enough money, not enough interest. These explanations have ceased to be acceptable in the face of the rising importance of the Internet to global commerce and, potentially, to the safety and well being of any number of individuals. If the Internet really is a valuable piece of infrastructure then its time to start treating it like one.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

List of References

Aleph One. “Smashing the Stack for Fun and Profit.” Phrack Magazine. Volume Seven, Issue Forty-Nine. URL: http://www.shmoo.com/phrack/Phrack49/p49-14 (24 February 2004).

Arce, Ivan and Elias Levy. “An Analysis of the Slapper Worm.” URL: http://www.coresecurity.com/files/files/12/AttackTrends.pdf (27 October 2003).

Bautista, Ron. “RPC DCOM Buffer Overflow – Technical Details.” 27 July 2003. URL: http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=RPC%20DCOM%20BUFFER%20OVERFLOW&VSect=T (9 January 2004).

Brain, Marshal. “How E-mail Works.” URL: http://computer.howstuffworks.com/email.htm (22 January 2004).

Carnegie Mellon Computing Services. “Symantec Antivirus 8.1 for Windows Desktops.” 16 October 2003. URL: http://www.cmu.edu/computing/documentation/virus_pc/NortonAV.html#realtime (20 January 2004).

CERT Coordination Center. “CERT Advisory CA-2001-11: sadmind/IIS Worm.” 10 May 2001. URL: http://www.cert.org/advisories/CA-2001-11.html (27 October 2003).

CERT Coordination Center. “CERT Incident Note IN-2002-02: Exploitation of Unprotected Windows Networking Shares”. 3 March 2000. URL: http://www.cert.org/incident_notes/IN-2000-02.html (27 October 2003).

CERT Coordination Center. “Frequently Asked Questions About Malicious Web Scripts.” 3 February 2000. URL: http://www.cert.org/tech_tips/malicious_code_FAQ.html (24 February 2004).

Coolbaugh, Liz. “Multicast impacts from the Ramen Worm.” Linux Weekly News. 25 January 2001. URL: http://lwn.net/2001/0125/security.php3 (28 October 2003).

de la Cuadra, Fernando. “How an Antivirus Program Works.” 7 May 2003. URL: http://net-security.org/article.php?id=485 (21 January 2004).

Curtin, Matt and Marcus J. Ranum. “Internet Firewalls: Frequently Asked Questions.” Revision 10.0. URL: http://www.interhack.net/pubs/fwfaq/ (9 January 2004).

© SANS Institute 2004, Author retains full rights.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Dreyfus, Emmanuel. “Securing Systems with chroot.” 30 January 2003. URL: http://www.onlamp.com/pub/a/bsd/2003/01/23/chroot.html (24 February 2004).

eEye Digital Security. “.ida ‘Code Red’ Worm.” 17 July 2001. URL: http://www.eeye.com/html/Research/Advisories/AL20010717.html (27 October 2003).

eEye Digital Security. “CodeRedII Worm Analysis.” 4 August 2001. URL: http://www.eeye.com/html/Research/Advisories/AL20010804.html (27 October 2003).

eEye Digital Security. “Microsoft SQL Sapphire Worm Analysis.” 25 January 2003. URL: http://www.eeye.com/html/Research/Flash/AL20030125.html (19 December 2003).

eEye Digital Security. “Spida or Digispid.B.Worm SQL Worm Analysis.” 22 May 2002. URL: http://www.eeye.com/html/Research/Advisories/AL20020522.html (27 October 2003).

Farshchi, Jamil. “SANS Intrusion Detection FAQ: Statistical based approach to Intrusion Detection.” URL: http://www.sans.org/resources/idfaq/statistic_ids.php (9 January 2004).

Fortinet Corporation. “FortiGate 3600: Best-Of-Breed Network Antivirus Protection.” January 2003. URL: http://www.fortinet.com/doc/FGT3600DS.pdf (20 January 2004).

Fyodor. “Nmap network security scanner man page.” URL: http://www.insecure.org/nmap/data/nmap_manpage.html (9 January 2004).

Gibson, Steve. “DCOMbobulator: Effortlessly Tame Windows Dangerous DCOM Facility.” 6 October 2003. URL: http://www.grc.com/dcom/intro.htm (9 January 2004).

Graff, Mark G. “Elements of Secure Coding.” URL: http://www.securecoding.org/authors/articles/may202003/section4.php (24 February 2004).

Howard, Michael. “Review Code for Integer Manipulation Vulnerabilities.” MSDN Library. 28 April 2003. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure04102003.asp (24 February 2004).

Immunix Corporation. “Immunix Antivirus Gateway Software Appliance.” Revision 1103. URL: http://www.wirex.com/pdfs/brochures/immunix-antivirus-gateway.pdf (20 January 2004).

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Incidents.org. “NIMDA Worm/Virus Report – Final.” 3 October 2001. URL: http://www.incidents.org/react/nimda.pdf (27 October 2003).

Leavell, L., et. al. “How Can I Block Latest Worm-Generated SPAM?”. 19 September 2003. URL: http://unix.derkeiler.com/Newsgroups/comp.unix.sco.misc/2003-09/0551.html (24 February 2004).

LeBlanc, David. “Integer Handling with the C++ SafeInt Class.” MSDN Library. 7 January 2004. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure09112003.asp (24 February 2004).

Linux Documentation Project. “Network File System (NFS) Services.” Linux Administration Made Easy. URL: http://www.tldp.org/LDP/lame/LAME/linux-admin-made-easy/nfs-services.html (14 January 2004).

Linux Weekly News. “’Ramen Worm’ attacks Red Hat-based systems”. Linux Weekly News. 18 January 2001. URL: http://lwn.net/2001/0118/ (28 October 2003).

Loop, John D. “Three Way Handshake.” 25 November 2003. URL: http://www.pccitizen.com/threewayhandshake.htm (9 January 2004).

Lucas, Michael. “Systrace Policies.” 30 January 2003. URL: http://www.onlamp.com/pub/a/bsd/2003/01/30/Big_Scary_Daemons.html (24 February 2004).

McGraw, Gary and Ed Felten. “Beyond the Sandbox: Signed Code and Java 2”. Securing Java. URL: http://www.securingjava.com/chapter-three/chapter-three-3.html (24 February 2004).

Meyers, J. “RFC 2554 – SMTP Service Extension for Authentication.” March 1999. URL: http://www.faqs.org/rfcs/rfc2554.html (22 January 2004).

Microsoft Corporation. “CIFS: A Common Internet File System.” November 1996. URL: http://www.microsoft.com/mind/1196/cifs.asp (14 January 2004).

Microsoft Corporation. “Creating an Automated Login Script.” MSDN Library. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/wsconcreatingautomatingloginscrip.asp (24 February 2004)

Miller, Jason, et al. “Microsoft DCOM RPC Worm Alert.” 18 August 2003. URL: https://tms.symantec.com/members/AnalystReports/030811-Alert-DCOMworm.pdf (19 December 2003).

© SANS Institute 2004, Author retains full rights.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Moore, David, et al. “The Spread of the Sapphire/Slammer Worm.” URL: http://www.cs.berkeley.edu/~nweaver/sapphire/ (27 October 2003).

OpenBSD Journal. “Non-executable Heap, too.” 26 August 2002. URL: http://www.deadly.org/article.php3?sid=20020826013453 (24 February 2004).

Oudot, Laurent. “Fighting Internet Worms With Honeypots”. 23 October 2003. URL: http://www.securityfocus.com/infocus/1740 (9 January 2004).

Page, Bob. “A Report on the Internet Worm.” 7 November 1988. URL: http://www.ee.ryerson.ca:8080/~elf/hack/iworm.html (4 November 20030.

Raithel, John. “Scheduled Activity: cron and at.” URL: http://www.rahul.net/raithel/MyBackPages/crontab.html (24 February 2004).

Schnoll, Scott. “Internet Explorer Security Zones.” URL: http://www.nwnetworks.com/iezones.htm (24 February 2004)

Shevchenko, Sergei. “W32.Mumu.B.Worm.” 25 July 2003. URL: http://securityresponse.symantec.com/avcenter/venc/data/w32.mumu.b.worm.html (14 January 2003).

Sieberg, Daniel, et. al. “’Code Red’ impact felt at major companies.” 9 August 2001. URL: http://www.cnn.com/2001/TECH/internet/08/09/code.red/ (29 October 2003).

Singer, Abe. “Analysis of network.vbs.” 6 March 2000. URL: http://security.sdsc.edu/publications/network.vbs.shtml (28 October 2003).

Sophos Corporation. “Sophos virus analysis: W32/MyDoom-A.” URL: http://www.sophos.com/virusinfo/analyses/w32mydooma.html (24 February 2004).

Spafford, Eugene H. “The Internet Worm + 10 Years: Lessons Learned and Not Learned.” 2 November 1998. URL: http://www.cerias.purdue.edu/homes/spaf/presents/Andersen.pdf (4 November 2003).

Spam Assassin. “Welcome to Spam Assassin.” URL: http://spamassassin.rediris.es/index.html (24 February 2004).

Sun Microsystems. “NFS Administration Guide.” URL: http://docs.sun.com/db/doc/802-5754/6i9g806l2?a=view#ch3concepts-42456 (12 January 2004).

© SANS Institute 2004, Author retains full rights.

© SANS Institute 2004, As part of GIAC practical repository. Author retains full rights.

Symantec Corporation. “Intrusion detection systems: Reducing network security risk.” 3 April 2003. URL: http://www.zdnetindia.com/biztech/ebusiness/whitepapers/stories/79198.html (21 January 2003).

Symantec Corporation. “Symantec Antivirus Gateway Solution.” URL: http://enterprisesecurity.symantec.com/products/products.cfm?ProductID=142&EID=0 (20 January 2004).

Tripwire, Inc. “Tripwire for Servers: Frequently Asked Questions.” URL: http://www.tripwire.com/products/servers/faqs.cfm (21 January 2004).

Tyson, Jeff. “How Virtual Private Networks Work.” URL: http://www.howstuffworks.com/vpn.htm (24 February 2004).

University of Northern Iowa. “Creating Strong Passwords.” URL: http://www.uni.edu/its/us/faqs/security/PasswordsStrong.htm (14 January 2004).

Varghese, Sam. “Blaster worm took heavy toll: survey.” 23 September 2003. URL: http://www.smh.com.au/articles/2003/09/23/1064082983214.html?from=storyrhs (29 October 2003).

Verton, Dan. “Blaster Worm Linked to Severity of Blackout.” Computerworld. 11 September 2003. URL: http://enterprisesecurity.symantec.com/content.cfm?articleid=2607&EID=0 (29 October 2003).

Vision, Max. “A Brief Analysis of the ADM Internet Worm.” URL: http://whitehats.com/library/worms/adm/index.html (10 November 2003).

Vision, Max. “Lion Internet Worm Analysis.” URL: http://www.whitehats.com/library/worms/lion/ (19 November 2003).

Vision, Max. “Origin and Brief Analysis of the Millennium Worm.” URL: http://www.whitehats.com/library/worms/mworm/ (10 November 2003).

Vision, Max. “Ramen Internet Worm Analysis.” URL: http://www.whitehats.com/library/worms/ramen/ (17 November 2003).

Webopedia. “Virus.” Webopedia. URL: http://www.webopedia.com/TERM/v/virus.html (27 October 2003).

Webopedia. “Worm.” Webopedia. URL: http://www.webopedia.com/TERM/W/worm.html (27 October 2003).

© SANS Institute 2004, Author retains full rights.

Wheeler, David. “Secure Programmer: Countering buffer overflows.” 27 January 2004. URL: http://www-106.ibm.com/developerworks/library/l-sp4.html?ca=drs-l0504 (24 February 2004).

Yamamura, Motoaki. “W32.HLLW.Qaz.A.” URL: http://www.symantec.com/avcenter/venc/data/w32.hllw.qaz.a.html (4 November 20030.

© SANS Institute 2004, Author retains full rights.