

Worm Hotspots: Explaining Non-Uniformity in Worm Targeting Behavior

Evan Cooke, Z. Morley Mao, Farnam Jahanian
University of Michigan
{*emcooke, zmao, farnam*}@*umich.edu*

November 12, 2004

Abstract

Long after the Blaster, Slammer/Sapphire, and CodeRedII worms caused significant worldwide disruptions, a huge number of infected hosts from these worms continue to probe the Internet today. This paper investigates *hotspots* (non-uniformities) in the targeting behavior of these important Internet worms. Recent data collected over the period of a month and a half using a distributed blackhole data collection infrastructure covering 18 networks including ISPs, enterprises, and academic networks show 75K Blaster infected hosts, 180K slammer infected hosts, and 55K CodeRedII hosts. We discover through detailed analysis how critical flaws and side effects in the targeting behavior lead to a significant bias for certain destination address blocks. In particular, we demonstrate three previously unexplored biases: a severely restricted initial random seed forcing infection attempts to certain blocks; flaws in the parameters of a random number generator making certain hosts cycle through limited target addresses; and the widespread use of private address space dramatically changing the targeting distribution of certain worms. A direct consequence of these biases is that certain blocks are subjected to far more infection attempts than others. We discuss the implication of these *hotspots* on worm simulation and modeling, placement of blackhole sensors, worm detection and quarantine.

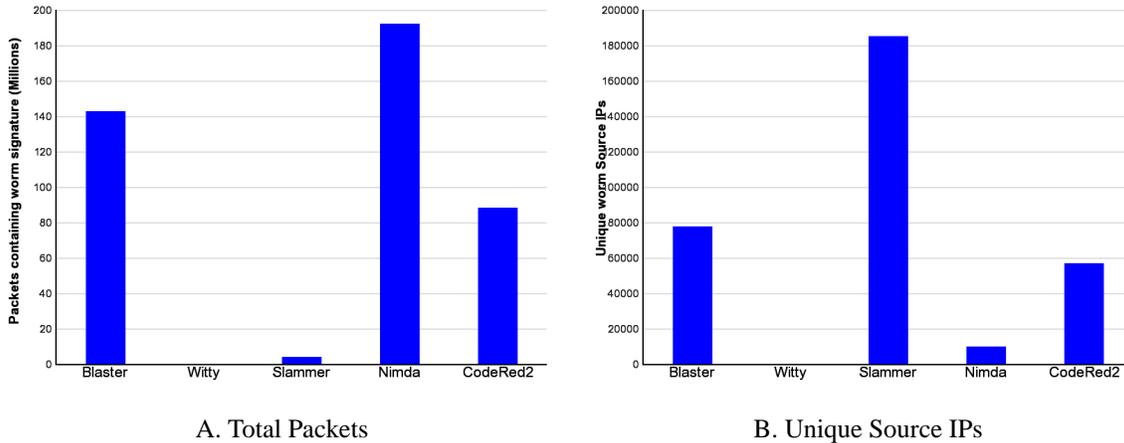


Figure 1: Observations of the Blaster, Witty, Slammer, Nimda, and CodeRedII worms over all monitored blocks

1 Introduction

The Internet today has been compared to the Wild West of the late 19th century United States. While this analogy may go a bit far, it does highlight the serious lack of security. The truth is that systems today are under constant attacks by malicious programs, packet floods, and automated worms. One might expect that as defenses are erected to protect against each new threat, these threats would be eliminated. The reality is quite different, a large number of threats never really die off. Data collected on Internet worms using a distributed blackhole data collection infrastructure over a period of a month and a half show 75K Blaster worm infected hosts, 180K Slammer/Sapphire worm infected hosts, and 55K CodeRedII worm hosts. These numbers are quite surprising considering the Slammer/Sapphire worm and the Blaster worm were released over a year ago and the CodeRedII worm over three years ago. The presence of these worms illustrates that persistent worms are a problem of today’s Internet.

In this paper we use data on persistent worms to explore and validate the root causes of *hotspots* or non-uniformities in the targeting distribution of three Internet worms. We show how a severely restricted initial random seed causes a large bias for certain address blocks in the Blaster worm. We also show how flaws in the parameters of the random number generator in the Slammer/Sapphire worm force certain hosts to repeatedly attempt to infect a few specific target addresses. Finally, we show how local preference in the CodeRedII propagation algorithm causes a dramatic bias for certain address blocks from infected hosts using private address space. We argue that *hotspots* have a significant implication on worm simulation and modeling, placement of blackhole sensors, worm detection, and worm quarantine.

Over the past few year there have been many analyses of these three important worms. The infection mechanism and technical details of the Blaster worm have been published by a number of companies [6, 11, 28]. Nazario studied the life cycle and impact of the Blaster worm [21]. Companies such as Eeye and Microsoft also documented the details of the Slammer/Sapphire worm [12, 5]. The outbreak and impact have been carefully studied by the academic community [19, 18, 12, 5]. Similarly, the CodeRedII worm was also the subject of a number of technical analyses [2, 9] and academic study [26]. These works have provided excellent details on the worms themselves; however, they have overlooked the importance of environmental factors that affect worm propagation. For example, the random number generator in many worms is seeded using the number of milliseconds since a system has been booted. This is a very poor form of entropy and can have a huge impact on randomness. In addition, the widespread adoption of private address space in both home and office settings has had a significant impact on worm propagation both inside and outside private address space. In this paper we attempt to go beyond merely studying the scanning algorithm to provide a more holistic view of worm propagation based on understanding the execution environment.

Persistent worms provide an excellent case study for understanding *hotspots*. First, the vulnerable population and number of infected hosts are relatively stable providing a more controlled environment to make long term measurements. Second, policy deployed at the edge and at upstream providers such as firewalls rules and router ACLs is much less likely to change than during the beginning of a worm outbreak. Finally, the sample size needed to reduce statistical variance and detect small changes in targeting behavior is quite large.

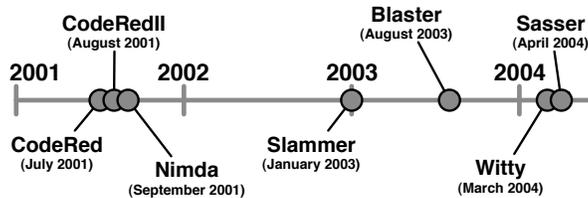


Figure 2: Timeline of major recent worms

The key contributions of this work are the following previously unknown observations and their implications.

- Using data collected with a distributed blackhole deployment we demonstrate significant *hotspots* in the targeting behavior of the Blaster worm, the Slammer/Sapphire worm, and the CodeRedII worm. These biases were not previously discovered or explained.
- Blaster: We demonstrate that a severely restricted initial random seed causes a large bias for certain address blocks. In particular, we show that the first few IPs in many subnets see significantly more Blaster hosts and infection attempts than other IPs.
- Slammer/Sapphire: We show that flaws in the parameters of the random number generator cause certain Slammer/Sapphire hosts to cycle through a small number of target address so that certain large address blocks are targeted significantly more than others.
- CodeRedII: We present evidence showing that the widespread use of private address space can dramatically change the targeting distribution of worms with a local preferences. Blackhole data are used to show that a certain block observes more than 27 times the number of CodeRedII hosts than the other blocks over the same period.
- We explore the implications of non-uniform worm targeting behavior on worm simulation and modeling, placement of blackhole sensors, worm detection and quarantine, and the cost on the infrastructure.

This paper is organized as follows. § 2 presents empirical evidence demonstrating the problem of persistent worms. We proceed to describe our measurement infrastructure in § 3. We illustrate the presence of hotspots in worm targeting behavior through three concrete real worm examples in § 4, § 5, and § 6. Finally, we conclude and discuss the implication of our work in § 7.

2 Worm Persistence

Over the past few years there have been a number of Internet worms that have significantly impacted the entire Internet. Figure 2 shows the timeline for 5 of these worms. A recent study [24] has shown that administrators often do not promptly install patches, even after the patches have long been made available. Consequently, many worms released years ago are still surprisingly active today, continuously making infection attempts. These observations are confirmed in our study using recent data over a period of a month and half from a distributed blackhole data collection infrastructure. Figure 1 B shows the distribution of the number of unique worm source IPs for five recent worms as observed in our data set. The number of infected hosts are staggering, ranging from tens of thousands of hosts for the Nimda worm to a few hundreds of thousands of hosts for the Slammer worm.

Figure 1 A shows the number packets containing worm signatures for these five worms. Interestingly, the Witty worm appears to be completely eradicated. There are two reasons accounting for this behavior. Witty targeted Internet Security Systems products from BlackICE and RealSecure, naturally generating more urgency for applying patches. Secondly, Witty was memory resident and was the first worm that actively damaged infected machines by overwriting random sectors of a random hard disk, thus limiting the lifetime of the worm. In addition to lack of Witty observations, it is also interesting to compare the number of unique sources and packets. Contrasting with Figure 1 A and B, the Slammer worm stands out as having a large number of infected IPs but many fewer worm packets. Because Slammer

Label	Organization	Size
A	ISP	/23
B	Academic Network	/24
C	Academic Network	/24
D,E,F	ISP	/20, /21, /22
G	ISP	/25
H	Large Enterprise	/18
I	National ISP	/17
M	ISP	/22
Z	ISP	/8

Table 1: Distributed Blackhole Deployments

is a UDP-based worm, there is no TCP retransmission for TCP-based worms such as Blaster, Nimda, and CodeRedII. Packet retransmission increases the total number of packets containing worm signatures.

In subsequent analysis we focus on the three most persistent worms based on the number of unique hosts as well as number of infection attempts – Blaster, Slammer, and CodeRedII.

3 Measurement Methodology

The data analyzed in this paper were collected using a network of distributed blackhole sensors that monitor blocks of unused address space. Because there are no legitimate hosts in an unused address block, any observed traffic destined to such addresses must be the result of misconfiguration, backscatter from spoofed source addresses, or scanning from worms and other network probing. This pre-filtering of traffic eliminates many of the false positives in identifying malicious traffic and scaling issues of other monitoring approaches. There exist several names to describe this technique such as network telescopes [17], blackholes [27, 1], and darknets [7]. Using these techniques, researchers have successfully characterized and classified traffic observed at unused blocks [18, 22, 4, 1].

Each blackhole sensor in the distributed infrastructure monitors a dedicated range of unused IP address space. The sensors have an active and passive component. The passive component records all packets sent to the sensor’s address space. The active component responds to TCP SYN packets with a SYN-ACK packet to elicit the first data payload on all TCP streams. This approach collects the necessary payload data to uniquely identify all the Internet worms studied in this paper. The details of this architecture are beyond the scope of this paper and are published elsewhere.

When a packet is received by a blackhole sensor, the passive component computes the MD5 signature of the payload. If the signature doesn’t match any previously observed signatures, then the payload is stored and the signature is added to the signature database. The payload is also passed through a series of filters to isolate specific worms. Take for example the filter for the Blaster worm. The filter monitors TCP port 135 which is the port over which the exploit is transmitted and TCP port 4444 over which commands are sent to download the worm payload. This filter identifies the worm using a specific byte sequence and also by tracking the transaction on the backdoor port. By using carefully chosen byte sequences we are able to differentiate between worms and between worm variants. For example, the filter for CodeRedII is able differentiate CodeRed, CodeRedv2 and CodeRedII.

The results presented in this paper are based on data collected on a deployment of nine blackhole sensors monitoring eleven unique address blocks. These deployments include major service providers, a large enterprise, and academic networks with address blocks that range in size from a /25 to a /8. Table 1 indicates the different deployments and their associated anonymized labels which are used to annotate the plots shown in the rest of the paper. These sensors represent a range of organizations and a diverse sample of the routable IPv4 space including seven of all routable /8 address ranges. Unless otherwise noted, all measurements were made over a period of a month and a half.

4 Blaster Targeting Analysis

The Blaster worm swept onto the Internet on August 11, 2003 and quickly infected a huge number of vulnerable systems. Also known as the Lovsan or MSBlast worm, it exploited a buffer overflow in the Windows Distributed Com-

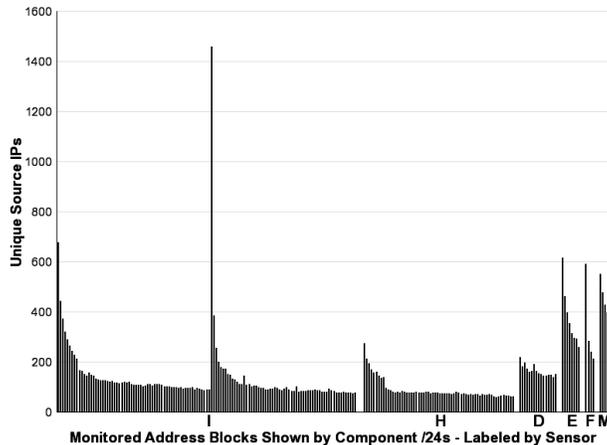


Figure 3: Unique Source IPs of Blaster Infection Attempts by /24

ponent Object Model (DCOM) Remote Procedure Call (RPC) interface. On a default installation of Windows, the RPC service is on by default and accepts requests from any remote system. The prevalence of the vulnerability combined with a large population of vulnerable newly connected broadband users set the conditions for an outbreak. The Blaster worm hit home users, businesses, and government hard with reports of severe outages around the world [16].

The Blaster worm propagates by scanning for vulnerable hosts across the entire IPv4 space. When it finds a potentially vulnerable system, it performs an RPC bind operation and then sends the exploit code in an RPC request packet. The exploit code then sets up a simple TFTP server and sends commands on TCP port 4444 to retrieve the worm payload. When the upload is complete, the exploit code then launches the worm binary. The first operation performed by the worm is to add an entry in the Windows registry directing the worm to be launched each time the system boots. Next, the worm adds a mutex to prevent re-infections and, depending on the system date, launches a SYN flood against *windowsupdate.com*. The Blaster worm then enters the propagation phase during which it attempts to infect other systems.

The propagation phase of Blaster involves two simple steps. Choosing a random starting address and then sequentially scanning from that point. The starting address is not entirely random. The Blaster worm has a preference for addresses in the same local /16 as the infected host. The idea behind this behavior is that nearby addresses are more likely to have live systems and those systems are likely to have a similar configuration (e.g., a computing lab). Thus, nearby addresses are a more likely source of vulnerable hosts than a totally random host. Once the starting address has been selected, then the Blaster worm moves sequentially through IPv4 space attempting to infect 20 hosts at the time.

Given the sequential behavior of Blaster, one might expect a relatively uniform target distribution biased slightly by local preference in addresses with lots of nearby hosts. Figure 3 shows the distribution of unique source IPs attempting a Blaster infection across the individual /24s that make up the monitored address blocks. Each bar in the figure represents the number of unique source IPs seen at each destination /24. Thus, large address blocks like I/17 are composed of many component /24s that are all shown individually. Monitored blocks are labeled on the horizontal axis and the separation between blocks is denoted with a small amount of white space. Smaller blocks and the /8 are not shown for legibility reasons.

Because hosts within the same /16 as Blaster infected hosts might be biased by local preference, Figure 3 only shows sources that are not in the same /16 as the block being plotted. Hence, the data plotted do not include any hosts scanning from within the /16 of any of the blocks shown. Very surprisingly, the target distribution is clearly not uniform. There are distinct differences between /24s and those differences are not randomly distributed. Take for example the distribution of Blaster sources targeting block I. In the middle of block I is a large spike which is significantly different from the previous /24s within the block. In addition, the /24s after the spike show an ordered decrease which cannot be the result of a true random process.

The question is then what could possibly account for these non-uniformities. Previous work [4] has speculated that differences may exist because of the following reasons:

- **Filtering policy** - factors that affect the *reachability* from sources to the blackhole sensors include filtering at the core and at the edge.
- **Propagation Algorithm** - worm target selection algorithms or propagation strategies often have a bias towards *local addresses* and random address generation functions can have flaws.
- **Sensor address visibility** - the lack of global reachability can be accounted for by reasons such as misconfiguration, policies, network failures, or even malicious intent.
- **Resource constraints** - hardware and software problems that can affect the performance and availability of the end-to-end path or monitoring system.
- **Statistical variations** - there may still be variance in the observations due to sampling errors.

Many of these possible explanations can be eliminated because the observations were made over a period longer than a month. In addition, policy measurements involving sending packets on different ports from various locations to each sensor showed that port 135 and 4444 used by Blaster are not blocked. Furthermore, the non uniformities in Figure 3 also occur between /24s in the same block. Thus, the most plausible explanation appears to be related to the Blaster propagation algorithm.

4.1 Blaster Propagation

While a number of organizations have published information about the Blaster worm [6, 11, 28], these overviews have left out important details about how the worm propagates. Using the decompiled Blaster source code [25], the exact list of activities involved in Blaster propagation can be understood.

1. The Blaster executable is launched as the result of a new infection or because an already infected machine has rebooted
2. A local /16 address is chosen in case Blaster selects a local scan. This involves seeding the random number generator using the *GetTickCount()* function
3. The random generator is again seeded using *GetTickCount()* and a random number is chosen using *rand()* to determine if scanning should use a local or completely random address. (60% random, 40% local)
4. Another call to *rand()* is used to set whether the offset in the exploit code will affect Windows 2000 or Windows XP. (80% XP, 20% 2K)
5. If a totally random address was chosen, then the first, second, and third octets of the address are set using three calls to *rand()*
6. On certain system dates, a thread to launch a DoS attack against *windowsupdate.com* is launched
7. The code then uses the previously generated address and exploit offset to attempt infections to 20 sequential addresses using 20 threads. This process is repeated on the next 20 sequential addresses and so on indefinitely. There are no further calls to the random number generator

The Blaster worm is essentially a sequential scanning worm and only utilizes a random number generator to choose the location to begin scanning. The observation here is that the random number generator is seeded and then used only a few times. This behavior places critical importance on the value of the initial seed. If the seed is random, the first few random numbers generated will also be random. However, if the seed is predictable, the first few random numbers will be similarly non-random.

We described earlier how the Blaster worm is launched each time an infected system booted and how Blaster uses the *GetTickCount()* function to seed the random number generator. The *GetTickCount()* function returns the number of milliseconds since boot time. Thus, if a significant number of Blaster hosts were periodically power-cycled, those hosts would choose an initial seed value that was restricted by the time it takes a typical system to boot. In fact, there is strong evidence to suggest a large number of hosts are power cycled every day [21]. Figure 4 shows the circadian pattern in the number of unique Blaster hosts over a 7 day period in July 2004. One could easily imagine an infected laptop fitting profile of a system that might only be on for short periods and rebooted often. Before delving into this hypothesis it is important to understand how initial seeds affect pseudo random number generators.

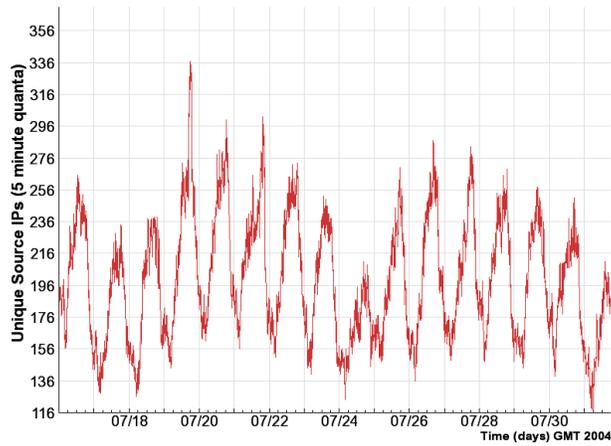


Figure 4: Circadian pattern in unique Blaster hosts over a 7 day period in July 2004

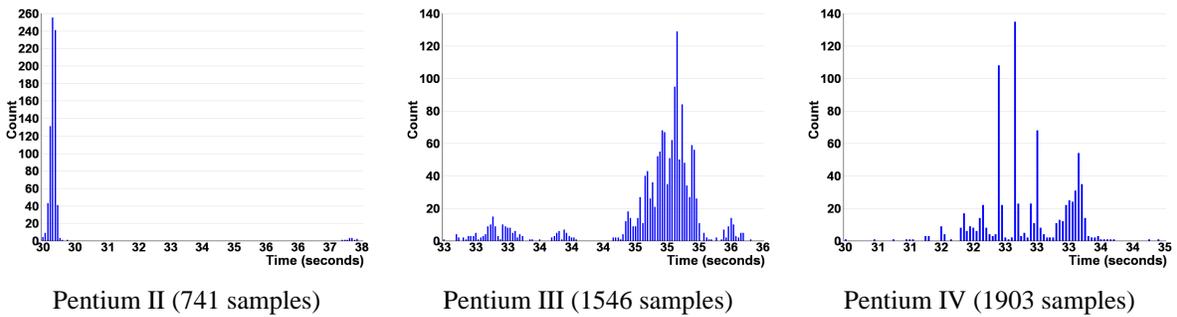


Figure 5: Tickcount distributions after bootup on three generations of systems

Table 2: Mean and Variance in Tickcounts

Configuration	Mean	Standard Deviation
Pentium II	30.03 seconds	1.01
Pentium III	34.98 seconds	0.65
Pentium IV	32.55 seconds	0.66

4.2 Pseudo Random Number Generators

Pseudo-random number generators (PRNG) are often used in worms to choose new targets. To produce high-quality or unpredictable random numbers, the PRNG itself must have a high degree of entropy, a measurement of randomness, so that the generated random numbers are unguessable. The security guarantees are non-existent even in the presence of a PRNG with a very complex algorithm and excellent statistical properties, if it is not properly initialized or seeded. This is known as the Fallacy of Complex Manipulation [8]. Using a bad seed or no seed at all makes the generated random numbers predictable, as the PRNG algorithm is often publicly known. The critical importance of properly seeding PRNG is well illustrated by a successful attack on Version 1.1 of the Netscape web browser by two graduate students in January 1996. They exploited the predictability of the seed to the PRNG used in SSL [15]. Once the seed was guessed successfully, it was trivial for adversaries to decrypt an intercepted message, breaking the security of any browser-based communications, such as e-commerce transactions.

When a PRNG is used to generate random target addresses, attackers aim to use PRNGs with high entropy to reduce redundant probing, a slightly different objective compared to cryptography. PRNG are often used in two ways by worm authors; to choose an initial scanning IP address for sequential scanning worms, e.g., Blaster; or to repeatedly to generate the next IP address for scanning by random scanning worms, e.g., Slammer. In both cases, using a good PRNG but a bad seed can produce highly redundant scans.

4.3 Blaster PRNG Seeding

The Blaster worm uses *GetTickCount()* function as a source of entropy for its PRNG. Given the evidence presented in Figure 4 indicating many Blaster infected systems appear to be rebooted each day, the *GetTickCount()* function appears to be a very poor source of entropy. We can test the randomness of the *GetTickCount()* function by analyzing tick counts from real Windows systems.

When the Blaster worm is first run, it adds an entry into the registry of the infected computer instructing the system to launch the worm after each reboot. Using the same registry-based launching mechanism, we wrote a simple program that called *GetTickCount()* and logged the result to a file. The program then instructed the computer to reboot and the process was repeated. Tick count distributions were gathered from three generations of Intel-based systems; a Pentium II, a Pentium III, and a Pentium IV. All system were running clean installations of Windows 2000 without any service packs or additional software beside the tick count logging program.

Figure 5 shows the distribution of tick counts on the three generations of Windows systems. There is significant clustering in each distribution. Interestingly, each distribution is only remotely Gaussian in shape. There are many hardware factors that might account for the different shapes in these distributions, however, the important consequence of these plots is that the range of tick count values are extremely restricted. Table 2 shows mean and variance for each configuration. The mean is around 30 seconds for all configurations.

The restricted tick count distributions presented in Figure 5 mean that a large number of rebooted Blaster infected systems will end up choosing exactly the same initial seed. If two infected systems pick the same seed then they will proceed to attack exactly the same targets in exactly the same order. This behavior should produces spikes in infection attempts that correlate with probably initial seed values. The next section investigates the correlation of spikes in the observed distribution and the initial seed.

4.4 Correlating Blaster PRNG Seeds with Empirical Evidence

At the beginning of § 4 we observed distinct differences in the distribution of Blaster targets within a single contiguous address block. We have also shown that the initial seed values to a Blaster worm instance are extremely restricted if the worm was started as the result of a power-cycle. Given the limited range of probable seed values, we would expect a biases toward infecting certain address based on the range of possible initial seed values

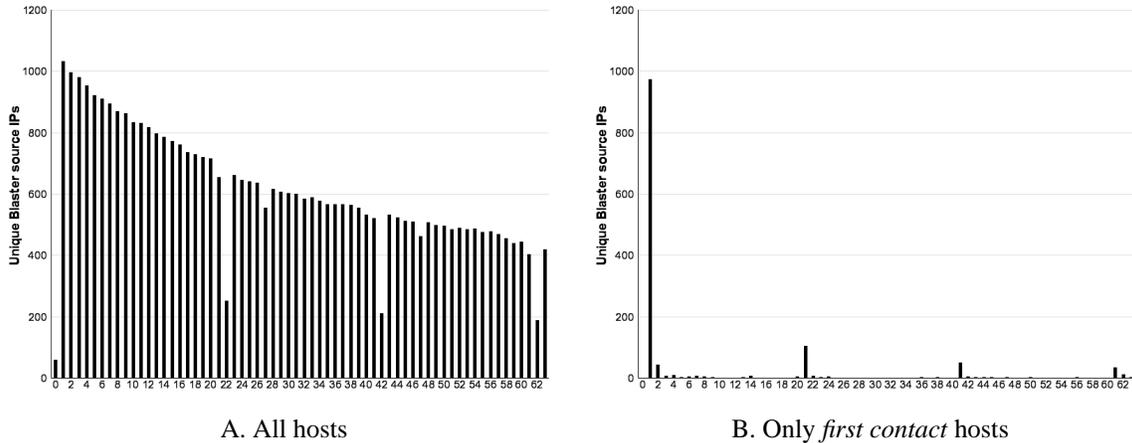


Figure 6: Unique source IPs of Blaster infection attempts to the first 64 IPs of a /24

There is however a problem with directly correlating spikes shown in Figure 3 with possible initial seed values. Because the Blaster worm chooses an address and then sequentially scans from that address onward, the spikes will include both hosts starting to scan and hosts that are sequentially scanning. To make the correlation strong, it is important to isolate only those hosts that are just starting to scan. These *first contact* hosts can be isolated using a simple filter. The Blaster propagation algorithm sequentially scans 20 hosts at a time. Thus, a sequentially scanning host must have attempted to infect at least one of the previous 21 hosts. If we can ensure a Blaster host has never contacted any of the previous 21 hosts, then we can be confident this is the first destination address contacted by a particular source over this invocation of the worm. There have been reports of another Blaster variant with more than 20 threads however we have not observed it.

Figure 6 A shows the number of unique source IPs attempting Blaster infections to the first 64 IPs of a /24 blackhole in the block labeled I. The /24 shown is the large spike in center of the I block (see Figure 3) that observes more unique Blaster hosts than any other monitored /24 block. In order to isolate the Blaster hosts that have just started scanning (*first contact* hosts), we use the method described above and filter out all hosts except those that have not contacted any of the previous 21 hosts. Isolating the *first contact* hosts, produces the plot shown in Figure 6 B. The result is a large spike of Blaster hosts that have just started scanning at the X.X.X.1 address.

The position of the spike can be explained quite readily by an inspection of the Blaster disassembled code [25]. First, when Blaster starts scanning, it chooses the first, second, and third octet, and then sets the fourth octet to zero. That explains why all the sources are bunched at the beginning of the block but not why the sources are seen at X.X.X.1 rather than X.X.X.0. The reason for this behavior is Blaster increments the destination address before it starts scanning. Thus, X.X.X.0 is always skipped when the worm begins scanning. In practice, X.X.X.0 can be a network specific address but there are very rarely valid hosts using this IP address.

The spike in *first contact* hosts in Figure 6 B indicates a large number of Blaster worms instances have either just been rebooted or have just been infected. The theory presented early was that these Blaster instances were rebooted hosts with a bad initial seed. If this theory is correct it should be possible to correlate the spike with an initial seed that corresponds to reasonable boot time.

The first step in finding such a seed is to generate a list of target addresses for a range of tick count values. Using the decompiled Blaster source code [25], and a range of possible tick count values from 1000 to 10,000,000, (i.e., boot times ranging from 1 second to 2.8 hours) a mapping from seed to destination address can be generated. Searching the map for the address of the /24 spike then reveals the seed used to produce the destination address. Using this method, spike was mapped back to a seed value corresponding to a *GetTickCount()* of 2.3 minutes. While it's possible for two seeds to map to the same address, this /24 only had one possible seed in the tested range. Only 36% of the 10 million seeds mapped to two or more destination addresses. The boot time of 2.3 minutes is greater than the approximately 30 second boot times shown in Figure 5. However, unlike the systems tested, real systems have other software that starts at boot time and can significantly slow the time it take for the worm to launch and thus pick a seed.

Using the seed-to-target mapping, the other spikes in Figure 3 were mapped back to possible seeds. Resulting seed values ranged from about approximately 1 minute to 20 minutes with the distribution centered around 4-5 minutes.

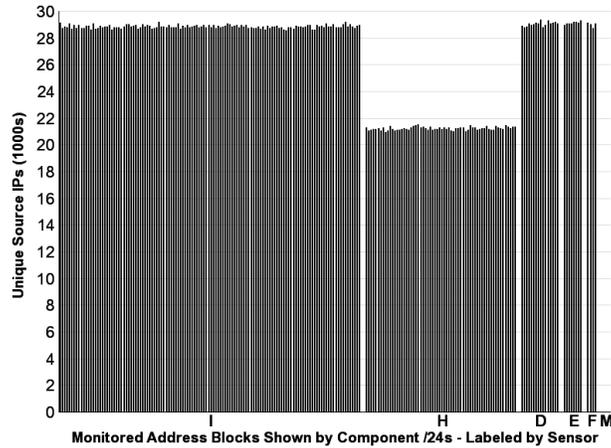


Figure 7: Unique Slammer Infection Sources by destination /24s

In addition, /24s without spikes mapped to seed values in the hours to days range. The majority of spike-to-seed mappings produced viable boot times offering strong evidence that the Blaster worm scanning is severely affected by a poorly seeded PRNG. In addition, the gentle decrease in the hosts to the right of the spike represents hosts at which Blaster has quickly stopped scanning again. This could be due to the host powering down again, loading policy software like a firewall, or losing Internet conductivity.

5 Slammer Targeting Analysis

Over the short history of internet worms there has been no worm with a larger impact on the Internet infrastructure than the Slammer worm. The Slammer worm, also known as Sapphire, infected more than 90 percent of the more than 75,000 vulnerable hosts within 10 minutes [18]. A side effect of the rapidly propagating Slammer worm was a huge amount of worm scan traffic causing wide-spread network congestion and even serious outages. The Slammer worm utilized a buffer overflow vulnerability in Microsoft SQL Server 2000 on the SQL Monitor Port (UDP port 1434).

The Slammer worm itself is a very simple memory resident worm with a tiny footprint (376 byte UDP payload). The small size of the worm combined with the speed of using stateless UDP rather than state-full TCP meant the Slammer worm could attempt new infections at an incredible rate. Compared to previous worms which are latency-limited, Slammer is bandwidth-limited. Scanning rates as high as 26,000 scans/second have been observed [18]. The Slammer targeting algorithm is also very simple. The worm chooses a random IPv4 address by using a pseudo-random number generator and then attempts to infect the targeted address with a single UDP packet. Unlike the Blaster worm, Slammer does not use Windows's `rand()` library function. Instead, it has a simple built-in pseudo-random number generator. The problem is the built in generator contains several flaws that could potentially affect the randomness of the output.

5.1 The Flawed Slammer PRNG

It is well known that the Slammer PRNG contains serious flaws [12, 18]. However, given the huge impact of Slammer when it struck, and its persistence today, it is clear those flaws did not cripple the worm. Figure 7 shows the number of Slammer infection attempts by /24. There are a few important observations about this plot. First, M block did not see any Slammer infection attempts. This is due to policy blocking the worm deployed at an upstream provider. Second, the H block shows almost 8000 fewer Slammer sources than the other blocks. Recall that these observations were made over a period longer than one month so temporal effects (especially considering Slammer's fast propagation) can be discounted.

There is clearly an anomaly in the Slammer targeting distribution located in the H block and no clear explanation why. The analysis of the Blaster worm in the previous section demonstrated that a bad initial seed can have a huge

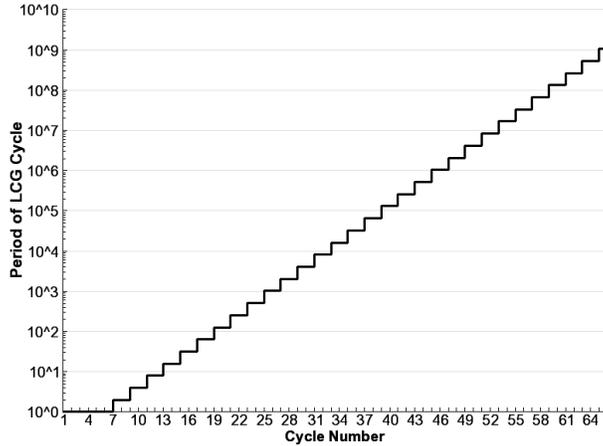


Figure 8: Period of all possible cycles in the Slammer LCG for b of 0x88215000

effect on propagation. It is possible such a seed might be responsible again. Like Blaster, the Slammer PRNG is seeded with number of milliseconds since boot time using the *GetTickCount()* function. There is however an important difference between Blaster and Slammer as to when the *GetTickCount()* function is actually invoked. Slammer is purely a memory resident worm and when a Slammer machine is rebooted the infection is cleared. Therefore, the time when the *GetTickCount()* function is called in Slammer varies relative to the boot time because the time at which the machine is re-infected varies. Thus, in Slammer, the seed is always chosen at infection time rather than boot time. This means the seed could take on almost any value if the machine has been running for an extended period of time (the millisecond counter loops every 49.7 days). This behavior means that the initial seed distribution is potentially much more diverse than was observed with the Blaster worm.

Another reason to place less emphasis on the initial seed is that Slammer generates a new random address for every infection attempt. This is unlike Blaster in which the first few random numbers are key to determining which addresses will be targeted. Instead of focusing on the initial seed, a more effective method of characterizing Slammer's targeting behavior is to look at the random generator itself. Rather than relying on a library function, Slammer has its own PRNG. The Slammer PRNG is a linear congruent generator (LCG) [14] of the form shown in Equation 1.

$$s(i+1) = a * s(i) + b \text{ mod } p \tag{1}$$

In the case of Slammer, the value of a is 214013 and p is 2^{32} because value is stored as a 32 bit integer. The b parameter however is not fixed. Although it may have been the intention of the worm author to fix the value of b at 0xffd9613c (which is a commonly used value of b in many LCGs), it appears the author made an error and used the OR instruction instead of XOR to clear a register. The result is that 0xffd9613c becomes OR'ed with a value leftover in the *ebx* register. This leftover value is the *sqlsort.dll* Import Address Table entry which can vary with the version of the DLL. Three versions have been widely reported (0x77f8313c, 0x77e89b18, and 0x77ea094c) [12, 18].

In order to determine the value of b in the Slammer LCG we take the three possible leftover *ebx* register values and XOR them with 0xffd9613c. Thus, the possible values of b (more if other DLL versions exist) are 0x88215000, 0x8831fa24, 0x88336870. Unlike the value of b that appears to have been originally intended, these b values are not optimal for producing the greatest possible range of random numbers. The result is that certain choices of the initial seed may cause the LCG to loop over a small subset of the possible 32-bit values. The implications of these imperfections are enormous for the Slammer targeting behavior. If the choice of initial seed forces Slammer into one of these small loops, the resulting target distribution will be extremely restricted.

5.2 Slammer Empirical Analysis

Based on the flaws in the Slammer PRNG described above, it is possible to make certain testable predictions about Slammer scanning patterns. For example, if a significant number of Slammer hosts end up choosing a bad initial seed, they will only scan a distinct subset of the entire IPv4 space. To test this hypothesis we tracked individual Slammer

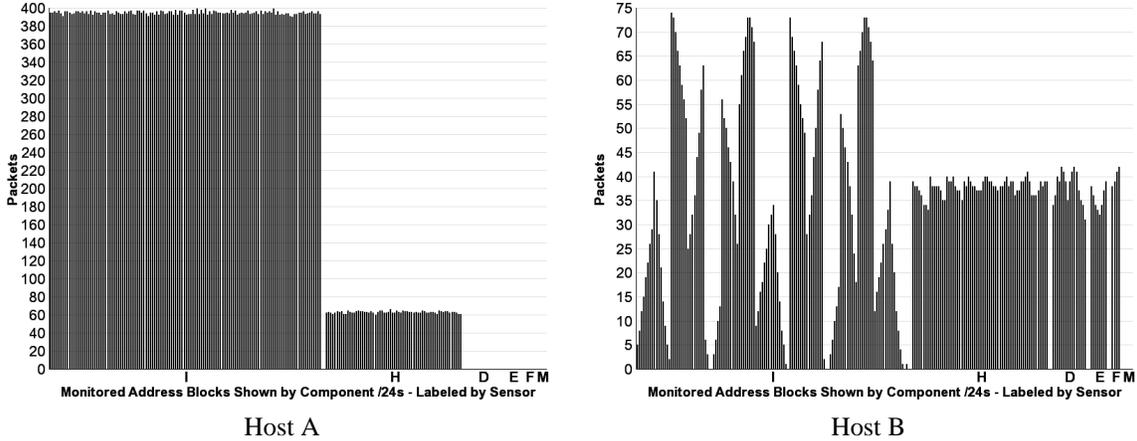


Figure 9: Slammer infection attempts from two unique hosts by destination /24s

Block	Sum (divided by 16^{10})
D	42.67
H	29.33
I	42.67

Table 3: Computed Slammer targeting preference

sources across the distributed blackhole network. Host A in Figure 9 shows the number of Slammer infection attempts by /24 from a particular Slammer source IP. Notice how block D observed no infection attempts from this particular source while block H observed some and block I received the most. Host B in Figure 9 shows another unique Slammer source. In this figure the intra-block variance is quite high, however, there is a distinct pattern in infection attempts. It is possible the two Slammer hosts shown in these two figures could have been rebooted and then subsequently re-infected acquiring different seeds. However, the critical observation is that there exists huge non-uniformity in the scanning patterns of individual hosts.

These results show that because of flaws in the Slammer PRNG, certain infected hosts end up repeatedly scanning only a subset of the IPv4 space. A natural question to ask is how many cycles the flawed PRNG has and how long each cycle is. Using a simple program, we compute for each value of b , the length of all cycles of the LCG described by Equation 1. There were 64 cycles for each b value, and the lengths were very similar in each case. Cycle lengths for b equal to $0x88215000$ are shown in Figure 8. Notice that the log plot shows many small cycles and seven cycles having a period of only one. A Slammer instance stuck in one of those small cycles would be simply hammering the limited set of target addresses.

It has been shown that certain seeds can force an infected Slammer host into a very restricted target distribution. Using information about the probability of getting into a particular cycle it should be possible to derive an aggregate distribution by summing the probabilities for each individual seed. Assuming a uniformly random initial seed (recall the seed is chosen at infection time rather than boot time), then there is a higher probability of getting into a cycle with a longer period than a cycle with a shorter period. This means that a single target address belonging to a long cycle will observe more unique Slammer source hosts than an address having a short cycle. In other words, longer cycles cover more seeds and thus have a higher probability of observing a new hosts than shorter cycles.

If longer cycles observe more unique sources than short cycles, then we can make a statement about the number of unique sources observed in different blocks. If a /16 block called *Alpha* contains more long cycles than the /16 address block *Beta*, then *Alpha* should statistically also observe more unique Slammer source IPs than *Beta*. Because there are longer cycles in *Alpha*, *Alpha* will effectively cover more address space than *Beta*. That is, many cycles contain addresses outside the specific blocks being monitored and so the probability of seeing a unique host depends on the coverage of those cycles rather than the size of the monitored block.

The distribution of unique Slammer hosts depicted in Figure 7 shows a clear bias away from the H block. Using the logic above, the H block should have fewer long cycles than the other blocks. We can show this by checking the

0FFFFFFFh	0FFFFFF0h	0FFFFFF0h	0FFFFFF0h
0FFFFFF0h	0FFF0000h	0FFF0000h	0FFF0000h

Table 4: CodeRedII address masks

length of all cycles that touch each block. This is implemented by traversing all values in all cycles and converting each value into an IPv4 address and checking whether the address lies in the same address space as any of the blocks. To make the comparison fair, the /16 around each monitored block is used, so that equally-sized samples can be compared. If the target address does indeed lie in the /16 around a monitored block, then a 64-bit counter specific to that block is incremented with the length of the cycle. This biases longer cycles over shorter ones. After visiting all possible cycles, the counter values at each block are compared. This process was completed for each value of b and the results averaged. The Slammer preference computed using this method is shown in Table 3 and confirms the observation made from Figure 7. Thus, it appears the H block has fewer long cycles than the other blocks and is less likely to observe as many Slammer hosts.

Understanding the flaws in the Slammer PRNG and the implication of those flaws on worm propagation behavior is important for a number of reasons. First, even tiny errors in the parameters used in the PRNG can have huge effects on the set of target addresses produced. Errors like those found in Slammer produce dramatic *hotspots* in the targeting behavior of individuals hosts and the overall number of unique Slammer hosts observed.

6 CodeRedII Targeting Analysis

On Friday, the 13th of July, 2001, the first traces were detected of a new worm targeting a known vulnerability in Microsoft IIS Web Servers [10]. The worm, christened Code-Red, used a buffer overflow to deface the targeted webserver and launch a distributed denial of server attack against *www.whitehouse.gov*. The worm propagated by randomly choosing target addresses. However, there was a serious flaw in the generator function. The value used to seed the random number generator was static. The consequence was that infected machines would all end up targeting the same sequence of hosts. Despite this flaw, the worm successfully defaced enough systems to be widely reported in the popular press.

Six days later a new variant of Code-Red named Code-Redv2, was released. This version fixed the flaw in the random generator by replacing the fixed seed with a random one. This variant was much more volatile and was reported to have spread to more than 359,000 hosts in less than 14 hours [26].

A few weeks later on August 4, 2001, a highly virulent new worm based on vulnerability used in Code-Red and Code-Redv2 was detected. This worm, dubbed CodeRedII, had a complete different payload from previous two Code-Red's. First, the worm installed a backdoor on the infected system allowing administrator access to the affected computer even after a reboot. Second, CodeRedII added *local preference* to the scanning algorithm. The result was that the worm spent more time trying to infect systems in nearby address space. As a consequence, many of the less protected machines, like cable modems, experienced a huge number of infection attempts from their nearby infected neighbors.

Like Blaster and Slammer, the CodeRedII worm uses a random generator function seeded using the *GetTickCount()* function. While Blaster only generates a single random number from which to begin scanning, Slammer and CodeRedII generate a random address for each new target. Unlike Slammer, CodeRedII generates each octet separately so the random generator is called four times per target address. The target address is also checked to make sure it's not a loopback address (127/8), a multicast address (224/8), or the local address. If any of these checks fails then the address is regenerated.

There is another important feature to the CodeRedII targeting function, a strong local preference. After the four target address octets have been generated, the last three bits of the first octet are used to index into a mask table (Table 4). The resulting mask is then used to logically *and* the address of the currently infected machine with the randomly generated target address. The effect is to create a preference for addresses near the address of the infected machine. Assuming a randomly generated first octet, the local preference will have the distribution shown in Table 5.

6.1 CodeRedII Hotspots

The scanning algorithm used in CodeRedII does not appear to suffer from flaws such as a poor initial seed, or bad PRNG parameters so one might expect a reasonably uniform scanning pattern. Figure 10 A. shows the number of

Target	Fraction
Local /16	3/8
Local /8	4/8
Random	1/8

Table 5: CodeRedII local preference

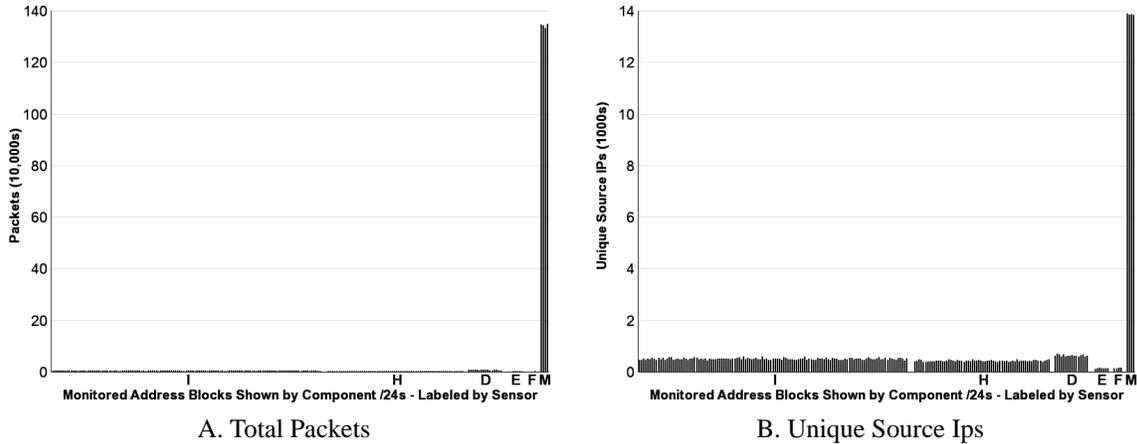


Figure 10: CodeRedII infection attempts by destination /24s

CodeRedII infection attempts by /24. This distribution is clearly not uniform. There is a large spike in infection attempts in the M block that massively dominate the infection attempts seen at any other blocks. Figure 10 B. shows number of unique CodeRedII sources by /24. Thus it's not just that the M block see more infection attempts, it also sees vastly more CodeRedII hosts than other blocks.

There are a few simple explanations for the large spike shown in Figures 10:

- Policy deployed upstream from the blocks other than M is blocking many potential infections.
- There is a temporal distortion in the scanning which causes a large number of CodeRedII hosts to scan the same block at the same time.
- Another variant of the CodeRedII with the same signature has a different targeting function.
- A large number of CodeRedII hosts are located within the same /16 or same /8 as the M blocks and thus the spike is the result of local preference.
- Some other feature of the scanning algorithm is causing a significant targeting bias.

Looking first at the question of upstream policy. A potential problem in the CodeRedII observations is that many of the blocks have upstream policy deployed to blocks CodeRedII. It turns out policy is not an issue for two reasons. First, CodeRedII uses TCP port 80 to propagate. Port 80 traffic is typically less hampered by policy because of the ubiquity of HTTP. Second, we tested the policy by sending connection requests to TCP port 80 on each sensor from multiple locations and the requests were never blocked.

The next important question is temporal bias. We can eliminate temporal effects as a possible explanation with confidence due to the long period over which the CodeRedII observations were conducted. The CodeRedII measurement was taken over a month and a half making such an effect highly unlikely given the magnitude of the observed spike.

Another possible explanation is that the CodeRedII worm being tracked is not the same variant as has been previously analyzed [9]. To test this theory we compared our observed payload with the disassembled code presented in the Eeye analysis [9]. The code was byte-for-byte identical except for the built-in kill value which sets the date when the worm stops propagating. The value had been modified to `0x8888` which corresponds to the year 34,952, effectively removing the kill value. This change has also been observed elsewhere [22]. Most importantly, the code

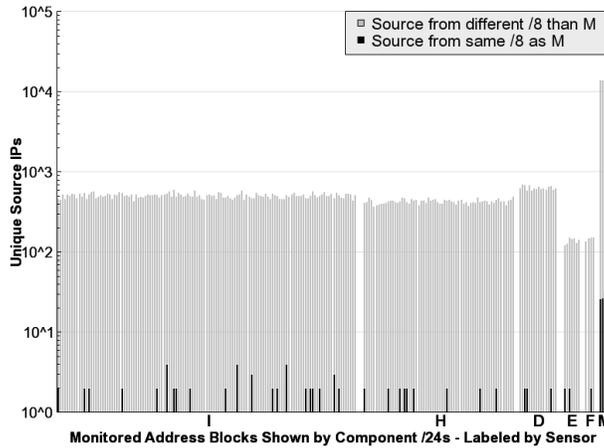


Figure 11: Unique CodeRed2 Infection Sources by destination /24s

that controls CodeRedII propagation was identical. It contained exactly the same propagation algorithm and nothing to indicate targeting at specific blocks. We can therefore conclude that a new CodeRedII variant with irregularities in the scanning algorithm does not appear to be responsible for the observed CodeRedII targeting pattern.

A third explanation for the highly uneven distribution is *local preference*. It is possible a large concentration of CodeRedII hosts in the M blocks is responsible for the observed behavior. To test this hypothesis we looked at the distribution of CodeRedII hosts in a /8 blackhole over the same month and a half period. The /8 revealed an almost perfectly even distribution. Because monitored block is an entire /8, *local preference* is not a factor. Hence, the even distribution produced when *local preference* is removed indicates that the non-uniformities in the other monitored blocks could be related to CodeRedII *local preference* algorithm.

Another way of looking at local preference is to explore the behavior of individual CodeRedII sources. Using a random sample of 100 unique CodeRedII hosts, we plotted the distribution of infection attempts by /24 over the monitored blocks. The results were intriguing, a large fraction of those individual hosts had a distribution almost identical to Figure 10. That is, those hosts appear to randomly target all blocks, however, there was a huge bias toward the M block.

Thus far we have presented evidence that *local preference* could be responsible for the observed CodeRedII distributions. There is however one important piece of contradictory evidence. If one looks at the source addresses of the large number of hosts exhibiting the spike behavior, we find that only a very small fraction have source addresses in the same /8 as the M block. This is a startling observation. If local preference were responsible, then source addresses should be in the same /8 as the spike in the M block. Figure 11 plots the number of unique CodeRedII sources by /24 (same data as Figure 10) with the source hosts that are from the same /8 as the M block highlighted. The graph had to be plotted on the log-scale just to see the small number of source from same /8 as the M block.

6.2 CodeRedII Hotspot Validation

The fact that such a large number of CodeRedII sources have a source address outside the /8 of the block implies some other process beyond a simple *local preference* must be at work. A hint about the nature of this process lies in the information about the M block itself. This block is located in 192/8. 192/8 is also the same /8 as an important chunk of private non-routable address space, 192.168/16 [23]. 192.168/16 is the default address block used by many popular Network Address Translation (NAT) [13] software packages and dedicated home and small office networks. One theory to explain why many sources that have an address outside 192/8 exhibit a local preference for the M block is that these infected hosts are behind a NAT device. That is, these hosts are configured with a 192.168/16 address but have reachability to the wider Internet because of a NAT device. The consequence is that these infected and NAT'ed systems repeatedly attack hosts within 192/8 because of the local preference in the CodeRedII targeting algorithm.

Given the large number of NATs deployed in home and corporate settings today, it is not hard to believe the conjecture

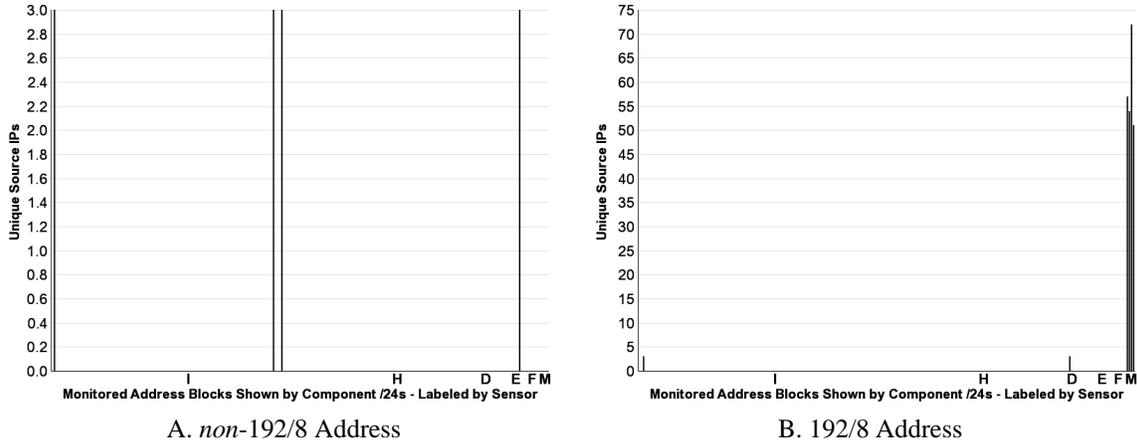


Figure 12: CodeRedII infection attempts by destination /24s from a quarantined host

that the spike in Figure 10 is related to a large number of CodeRedII infections behind NATs. The problem is the theory is very difficult to test without tracking down the infected machines one-by-one. Instead, we attempted to replicate the situation of a CodeRedII host behind a NAT and then monitor the scanning pattern. The first task was to catch a real instance of the CodeRedII worm. Using a honeypot running VMWare, the CodeRedII worm was captured and placed in a controlled environment. Once infected, a CodeRedII host will scan for new targets for the period of one day before rebooting and leaving just the backdoors open. After the worm had been captured, it was allowed to scan in an isolated network until it rebooted itself.

In the first experiment, the vulnerable host running under VMWare was configured with an address located in a /8 which was *not* 192/8. Figure 12 A. shows the scan targets plotted over the same /24s that are shown in Figure 10. Even though a total of 7,567,093 infection attempts were recorded from the quarantined host, only a small number of attempts reach the monitored blocks. Given that a totally random address is only chosen 12.5% of the time, and the number of possible destination addresses is on the order of billions, a small number of attempted infections to these blocks are expected.

In the second experiment, the vulnerable host running under VMWare was configured with the address 192.168.1.50/32. Figure 12 B. shows the scan targets plotted over the same /24s that are shown in Figure 10. During this run, a total of 7,567,361 infection attempts were recorded from the infected host. However, this time, the graph shows a distinct spike at the M block just like the distribution observed on the live blackholes. While the number of samples is not huge, there is clearly a significant effect due to local preference.

These two experiments provide strong evidence that the uneven distribution in CodeRedII targeting is related to *local preference*. In addition, there is a strong correlation between the scanning behavior of a host behind a NAT device and the observed distributions. This result illustrates how the widespread use of private address space can have a profound effect on worm propagation. In case of CodeRedII, there is a huge *hotspot* located in the same /8 as a private address block forcing hosts in that block to endure many more CodeRedII infection attempts than other blocks.

7 Discussion

The results presented in this paper focus on unexpected non-uniformity in worm targeting behavior. The causes for these hotspots are quite subtle and required an understanding of the worm execute environment to fully understand. We presented concrete evidence for these external environment factors through the analysis of three persistent worms: Blaster, Slammer and CodeRedII. In Blaster, we found that a severely restricted initial random seed can force infection attempts to certain blocks. Flaws in the parameters of Slammer’s random number generator were shown to cause certain hosts cycle through limited target addresses. And in CodeRedII, we discovered that the widespread use of private address space can dramatically change the targeting distribution due to *local preference*. We now discuss some of the implications *hotspots* on modeling, monitoring, and containing future worms.

There has been much work on worm modeling [32, 31, 30, 3] mostly focusing on worm scanning algorithms. None of the prior work investigates the impact due to factors from the external environment on the worm scanning behavior. Factors like PRNG seeds, PRNG input parameters, and local addresses as demonstrated by the three popular worms in our analysis create non-uniformities or hotspots in worm targeting behavior. To accurately model and predict how worms spread, hotspots need to be taken into consideration, as they create biases in the simplified well-known scanning algorithms. In general, aside from the worm scanning algorithm, the runtime execution environment can have a significant influence on the worm spreading patterns. As part of our future work, we plan to study other runtime factors that can create hotspots.

Worm monitoring is often accomplished using blackhole sensors. From the perspective of placing such sensors, the results presented in this paper have a slightly negative implication. Given the difficulty of predicting worm hotspots a priori, it is difficult to know where to place sensors to catch new worms early. Even if hotspot behavior is known, it can still be difficult to predict where they are empirically, as they may depend on external unknown input data. This implies that a wide distributed deployment would be most beneficial to increase the probability of observing worms without knowing scanning behavior. Worm detection also becomes harder without knowledge of hotspots as they can create unpredictable irregularity in worm scanning patterns.

Several recent studies [20, 31, 29] address the problem of worm quarantine. Similar to worm modeling, to contain a worm, it is important to predict its targeting behavior and react sufficiently fast to stop the worm from spreading. If a network happens to fall within a hotspot of a worm, there will be more infection attempts and possibly more collateral damage in the case of bandwidth-limited worms. Many worm containment proposals attempt to quickly identify worm scans. Understanding the location of hotspots can improve the accuracy of detection.

In conclusion, we have presented a detailed study of hotspots in worm targeting behavior using three important persistent worms. Our work is an important step towards accurate worm modeling and provides essential new details on worm propagation that are important for worm monitoring, detection, and containment.

References

- [1] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The Internet Motion Sensor: A distributed blackhole monitoring system. In *To appear in Network and Distributed System Security Symposium (NDSS '05)*, San Diego, CA, February 2005.
- [2] CERT Coordination Center. Code Red II: Another Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. Available at http://www.cert.org/incident/_notes/IN-2001-09.html, 2001.
- [3] Z. Chen, L. Gao, and K. Kwiat. Modeling the spread of active worms. In *IEEE INFOCOMM*, 2003.
- [4] Evan Cooke, Michael Bailey, Z. Morley Mao, David Watson, and Farnam Jahanian. Toward understanding distributed blackhole placement. In *To appear in ACM CCS Workshop on Rapid Malcode (WORM'04)*.
- [5] Microsoft Corporation. Finding and fixing slammer vulnerabilities. February 2003.
- [6] Microsoft Corporation. What you should know about the blaster worm. August 2003.
- [7] Team CYMRU. The darknet project, June 2004.
- [8] D. Eastlake, S. Crocker, and J. Schiller. Randomness Recommendations for Security. Request for Comments: 1750, December 1994.
- [9] eEye Digital Security. ANALYSIS: CodeRed II Worm. Available at <http://www.eeye.com/html/research/advisories/AL20010804.html>, 2001.
- [10] eEye Digital Security. ANALYSIS: .ida Code Red Worm. Available at <http://www.eeye.com/html/research/advisories/AL20010717.html>, 2001.
- [11] eEye Digital Security. ANALYSIS: Blaster Worm. August 2003.
- [12] eEye Digital Security. ANALYSIS: Microsoft SQL Server Sapphire Worm. Available at <http://www.eeye.com/html/research/advisories/AL20030124.html>, 2003.
- [13] K. Egevang and P. Francis. RFC 1631: The IP Network Address Translator (NAT). 1994. <http://www.ietf.org/rfc/rfc1631.txt>.
- [14] Paul B. Garrett. *Making, Breaking Codes: an Introduction to Cryptology*. Prentice-Hall, Inc., Upper Saddle River, NJ 07458, USA, 2001.
- [15] Ian Goldberg and David Wagner. Randomness and the Netscape Browser. *Dr. Dobbs Journal*, January 1996.
- [16] Brian Krebs. Internet worm plaguing computers worldwide. 2003.

- [17] David Moore. Network telescopes: Observing small or distant security events. In *11th USENIX Security Symposium, Invited talk*, San Francisco, CA, August 5–9 2002. Unpublished.
- [18] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security & Privacy*, 1(4):33–39, 2003.
- [19] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The Spread of the Sapphire/Slammer Worm. In *Proceedings of the 27th NANOG Meeting*, Phoenix, Arizona, February 2003.
- [20] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *INFOCOM*, December 21 2003.
- [21] Jose Nazario. The blaster worm: The view from 10,000 feet. 29th Meeting of the North American Network Operators Group, October 2003.
- [22] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet Background Radiation. <http://www.cs.princeton.edu/nsg/papers/telescope.pdf>, June 2004.
- [23] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Rfc 1918: Address allocation for private internets. 1996. <http://www.ietf.org/rfc/rfc1918.txt>.
- [24] Eric Rescorla. Security holes... Who cares? In *Proceedings of USENIX Security Symposium*, August 2003.
- [25] Robert Graham. Decompiled Source for MS RPC DCOM Blaster Worm.
- [26] Colleen Shannon, David Moore, and Jeffery Brown. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the Internet Measurement Workshop (IMW)*, December 02 2002.
- [27] Dug Song, Rob Malan, and Robert Stone. A snapshot of global Internet worm activity. FIRST Conference on Computer Security Incident Handling and Response 2002, June 2002.
- [28] Symantec Corporation. W32.Blaster.Worm. August 2003.
- [29] N. Weaver, D. Ellis, S. Staniford, and V. Paxson. Worms vs. Perimeters: The Case for Hard-LANs. In *Proc. Hot Interconnects 12*, August 2004.
- [30] N. Weaver, I. Hamadeh, G. Kesidis, and V. Paxson. Preliminary Results Using ScaleDown to Explore Worm Dynamics. In *To appear in ACM CCS Workshop on Rapid Malcode (WORM'04)*, October 2004.
- [31] Cliff C. Zou, Weibo Gong, and Don Towsley. Worm Propagation Modeling and Analysis under Dynamic Quarantine Defense. In *Proceedings of ACM CCS Workshop on Rapid Malcode (WORM'03)*, October 2003.
- [32] Cliff C. Zou, Don Towsley, and Weibo Gong. On the performance of internet worm scanning strategies. Technical Report Umass ECE Technical Report TR-03-CSE-07, Department of Computer Science Univ. Massachusetts, Amherst, November 2003.