Finding Diversity in Remote Code Injection Exploits

Justin Ma* John Dunagan† Helen J. Wang†
Stefan Savage* Geoffrey M. Voelker*
*University of California, San Diego †Microsoft Research

ABSTRACT

Remote code injection exploits inflict a significant societal cost, and an active underground economy has grown up around these continually evolving attacks. We present a methodology for inferring the phylogeny, or evolutionary tree, of such exploits. We have applied this methodology to traffic captured at several vantage points, and we demonstrate that our methodology is robust to the observed polymorphism. Our techniques revealed non-trivial code sharing among different exploit families, and the resulting phylogenies accurately captured the subtle variations among exploits within each family. Thus, we believe our methodology and results are a helpful step to better understanding the evolution of remote code injection exploits on the Internet.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Algorithms, Measurement, Security

Keywords

worms, bots, malware classification, binary emulation, phylogeny

1. INTRODUCTION

Internet users are increasingly victimized by an online criminal enterprise that spans denial-of-service extortion, identity theft, wire fraud, piracy and unsolicited bulk email. At the core of these activities is network-borne malware—software used to remotely compromise and harness the resources of millions of hosts. While considerable effort has focused on methods for defending against such attacks, there is comparatively little research describing the malware ecosystem itself. That is, how does one piece of malware relate to another, what pressures drive its structural and functional evolution, and what may this teach us about malware authors and the state of our defenses? This paper does not conclusively answer these questions, but we develop a measurement methodology that is a first step in this direction. In particular, we focus on how to identify and measure the diversity among *remote code injection* exploits used to compromise Internet hosts.

Typically, a host is compromised via a software vulnerability (e.g., buffer overflow, format string, integer overflow) that allows network-based input to be "injected" into a running program and executed. Subsequently, the exploit payload may download additional software, join a centralized "botnet," or reconfigure the operating system to evade detection (i.e., a rootkit). While there is significant variation among these "applications," here we focus specifically on the exploit and its initial payload—the so-called shellcode—that first executes on a newly compromised machine. Because shellcodes operate in a constrained environment, they are typically small, simple, hand-coded machine programs and thus are particularly well-suited to automated analysis.

Given a corpus of such shellcodes there are a number of obvious questions that arise. An immediate motivation for understanding how much variation exists among the shellcodes for an exploit is to inform the state of our defenses. In particular, polymorphic shellcodes are considered to be an Achilles heel for exploit-signature-based approaches [9, 10, 22]. This consideration has motivated research in vulnerability-signature-based approaches [2, 3, 18, 29]. However, to date there is little empirical evidence about the presence of such threats in the wild, let alone an attempt to characterize its impact on existing defenses.

At the same time, measuring shellcode diversity may also allow us to better understand how malware is created, potentially inferring the paternity of different samples or, more generally, constructing a shellcode *phylogeny*. Such a malware family-tree would simplify the categorization, naming and analysis of malware performed by security vendors and could also provide insight into the dynamics influencing malware development, evolution and sharing. Moreover, as malware "potency" becomes a powerful market advantage for organized cyber-criminals, a phylogenic analysis may prove useful for estimating the market-share and vigor of different organizations. However, there is a pointed lack of such empirical analysis in the literature today.

In summary, the purpose of this work is to enhance the understanding of today's malware, their diversity and relationship to one another. From our study, we have gained some evidence that today's malware diversity is simple and does not employ sophisticated evasive techniques.

This paper develops and demonstrates a methodology for automatically identifying and quantifying shellcode similarity, and for using this data to create a shellcode phylogeny for a given vulnerability. The remainder of this paper is organized as follows. In

IMC'06, October 25–27, 2006, Rio de Janeiro, Brazil.

Section 2, we provide a brief background on remote code injection exploits and a discussion of related work in the areas of automated malware analysis. Then, we provide an overview of our methodology in Section 3. In Section 4, we analyze the amount and causes of diversity for exploits both within individual vulnerabilities and across multiple vulnerabilities. Section 5 discusses implications of our findings for current network IDS systems, and we conclude in Section 6.

2. BACKGROUND AND RELATED WORK

Remote code injection attacks are a combination of vulnerability, exploit and shellcode. The vulnerability is the particular software structure that allows data provided over the network to subvert and redirect execution control flow. For example, if network input overruns beyond the end of an unchecked buffer, it can overwrite the return address of the calling stack frame and divert control into the buffer itself. An exploit is a particular formulation of an attack against a vulnerability. Typically an attacker can use many different instruction sequences to exploit a particular vulnerability, and as a result there may be multiple exploits for that vulnerability. We refer to the variation in the exploits for a particular vulnerability as the "diversity" of those exploits. Finally, the shellcode is the payload carried by the exploit—it is the first code to execute as a result of control flow being subverted. Figure 1 shows a simple representation of a stack-based buffer overflow, an instance of a remote code injection attack. In this case the shellcode is injected into the vulnerable buffer that is being overrun.

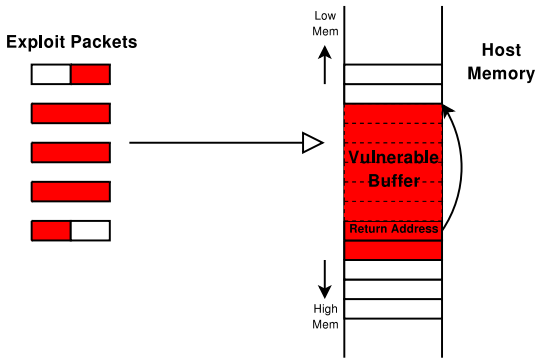


Figure 1: Simple example of a remote stack-based buffer overflow. The shaded regions represent the shellcode of the exploit as sent over network packets, then as injected into the vulnerable buffer of the target host. The return address has been overwritten with injected data, thereby redirecting the execution flow to the shellcode residing in the vulnerable buffer.

Using the terminology from Crandall et al. [4], our study examines polymorphism and diversity in the π portion of the an exploit. The π region of an exploit refers to the injected shellcode that is executed after diverting the flow of execution on a victim machine. We study the shellcode because there are considerable degrees of freedom in their construction. However, they are frequently limited by the size of the buffer being processed and the need for the buffer to contain "NOP sleds," or long regions of consecutive "do nothing" instructions prepended to shellcode that allow for some imprecision in guessing the address of the shellcode itself.

Shellcodes can be quite sophisticated in their construction, including the creation of pseudo-random NOP sleds and "polymorphic" payloads that are encrypted (and potentially compressed) in transit and only decrypted just before execution [23]. One motivation for these techniques is that they can greatly frustrate the ability to find a textual signature for a given shellcode. Indeed, some polymorphic shellcode generators also create random decryptors, further challenging signature-based defenses. Finally, some shellcodes include anti-debugging code (including self-modifying code and breakpoint detection) to complicate disassembly and analysis.

Thus even decoding shellcodes can be a challenging problem in itself. Early attempts to defeat polymorphism used so-called "x-ray analysis" to heuristically decode polymorphic codes based on the assumption of an XOR-based cipher [27]. In x-ray analysis, a portion of a known, decoded instance of the shellcode is XOR-ed against an encoded shellcode to recover the encryption key. A more general approach, called generic decryption, has been to emulate execution while the shellcode decrypts itself (typically using a heuristic to guess when this process terminates). However, the overhead of this technique can be substantial and there are a variety of counter-measures that undermine its efficacy [27]. Thus, researchers have developed techniques that combine emulation, dynamic translation, and static analysis [21, 24]. We use a similar approach in our work.

Having decoded a malware shellcode, comparing it to other shellcodes is another key problem. One approach is to model each shellcode as a binary string and use traditional lexical distance measures to approximate shellcode variation. An alternative approach is to use structural distance measures that capture variation in the control flow and values at the instruction, basic block, or function call levels [5, 6, 11]. We experiment with both techniques in this paper, and then propose another metric that we find better suited to our problem domain.

3. METHODOLOGY

In this section we describe our analysis techniques, including their motivation, potentials, and limitations. Overall, our analysis proceeded with the following major steps. First, we extracted shellcodes directly from a network trace using Shield [29]. We then decoded the shellcodes through restricted binary emulation. Finally, we examined the similarity among exploits by performing clustering on the shellcodes.

3.1 Exploit Collection with Active Responders

Our definition of "remote code injection exploits" refers specifically to exploits that require the attacker to *actively* compromise a victim's machine over the network—no initiating action from the victim is required. Hence, our primary means of collecting exploits is by examining network traces of traffic sent to and from *active responders*, hosts that respond to unsolicited probes (e.g., exploit attempts) for measurement purposes. As Pang et al. observed in their study of exploits destined for unused address space, emulating end-host behavior allows us to collect more session data from an attacking host [20]. In particular, completing the infection handshake will suffice to cause the attacker to transmit the shellcode.

To further illustrate the utility of using active responders for measurement, consider the ISystemActivator and RemoteActivation exploits. These exploits target different vulnerabilities using the same network port. If we only listen for exploit traffic passively (e.g., the network telescope approach [16]), the exploits would simply appear as TCP SYNs destined for port 135 in the trace. Thus, we require active responders to capture the RPC Bind and Request portions of the exploit handshake to differentiate the exploits, and we use network traces of live honeypots and stateless responders [20] to obtain these exploits.

3.2 Extracting Shellcodes

Analyzing the *shellcode* portion of an exploit is promising because it is the first place the malware author is executing new code on the system. In this new code, malware authors can use a variety of different techniques to accomplish a number of goals, e.g., finding needed host subroutines, or downloading a larger executable. By comparison, we did not focus on protocol framing (e.g., application layer headers) because such framing is a compulsory element. Although the message sequence for exploiting the vulnerability can also vary, malware authors likely benefit less from altering such handshake messages.

We used Shield [29] to extract the shellcode for each exploit session from the traces. Shield was originally designed for vulnerability-driven filtering for known vulnerabilities. To apply Shield in our setting, we modified it to collect the data that is beyond the buffer boundary. However, not all of the collected data corresponds to executable code. For example, execution will start at an offset within the buffer that depends on the exact vulnerability being exploited. Indeed, there are cases where the first several bytes of the vulnerable buffer contain the beginning of a URL that is prepended to the executable shellcode proper. As a second example, the buffer may contain random padding that is not part of the exploit code (nor is its data read by the exploit code). We addressed these and other issues using the following exploit emulation technique.

3.3 Exploit Emulation

In many of the exploits we encountered during our study, the main executable payload had been encoded. One encoding technique we commonly observed was a byte-for-byte XOR of the payload with a decoding key, though we also encountered variations such as encoding NULL-bytes with an escape character and multi-byte XOR. A variety of reasons might explain this observed use of encoding: eliminating NULL bytes to allow `strcpy()` buffer overruns to complete, hiding tell-tale strings (Web site names, API names such as "Kernel32", etc.), and finally obfuscating executable code, thereby making network-based intrusion detection more difficult. Regardless, we found that decoding the exploits was often necessary to reveal most of the actual executable code.

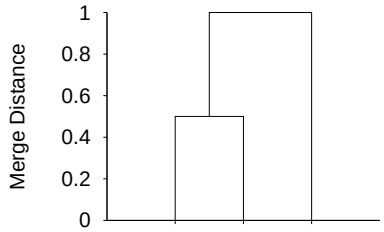
To enable our analysis to scale to many shellcode samples, we needed to automate the process of decoding the exploit payloads, all of which had clear pre-images and post-images. We found that the easiest way to deal with the variety of decoding routines was to use binary emulation, allowing the different exploit decoding routines to execute, and thus reveal the underlying code. Emulating the exploit was also necessary to reveal the actual instruction bytes executed by the exploit. In contrast, we found that even static disassembly over the decoded payload had trouble with the vagaries of x86 instruction alignment and the occasional instance of random filler text decoding to valid x86 instructions.

We implement the emulator using Intel's Pin [13] on Linux. Given an encoded shellcode, we first declare it as a statically allocated buffer in C source code that treats the buffer as a function. Then, we compile the source into a binary that marks the shellcode buffer with read, write and execute permissions to allow the self-modifying shellcode to run properly. We addressed the issue of buffers beginning with non-executable prefixes (mentioned in Section 3.2) by iteratively retrying failed emulations at subsequent offsets. We found this simple retry policy was enough to overcome any issues with non-executable prefixes.

As Pin successfully emulates the binary, we mark the executed instruction bytes for later analysis. Because we are emulating exploits for Windows on a Linux platform, we use heuristics to prolong shellcode execution. During emulation, function calls to addresses and memory accesses that would cause segfaults are intercepted, and the flow of execution restarts at the fall-through instruction. The emulation stops when the control flow makes an absolute jump to a location outside the buffer.

	B	C
A	0.50	0.75
B	—	1.00

(a) Distances



(b) Dendrogram

Figure 2: An example of (a) a distance matrix and (b) its corresponding dendrogram.

Emulating these short shellcodes has the advantage of a relatively low overhead compared to emulating an entire binary executable. Although we did not pursue this direction, we could conceivably also use this methodology to emulate and look for relationships among larger code bases such as rootkits, spyware, or other executable malware files. However, it is unclear whether the overhead of this approach would be prohibitive for larger executables.

3.4 Clustering

To understand the relationship of the shellcodes to each other, we introduced a distance metric on the shellcode instruction bytes generated by binary emulation. We then performed clustering on the shellcode instruction bytes using this metric. In all the cases in our study, we evaluated this clustering manually to confirm that it was intuitively sensible, and that the tree constructed by agglomerative clustering had an intuitive interpretation as the phylogeny of the exploit families.

Agglomerative clustering is a form of hierarchical clustering in which the algorithm *builds up* a hierarchy of similarity among exploit samples by iteratively merging the closest pair of clusters at each step. The algorithm begins with each *unique* shellcode belonging to its own cluster, and performs merging on the closest pair of clusters. We define the *distance between clusters* as the distance between the furthest samples in the two respective clusters (also known as complete linkage distance). In addition, the algorithm records the distance between two clusters before merging them, and this information allows us to construct a dendrogram to visualize the clustering results.

Figure 2b illustrates a simple dendrogram constructed in this way using the distances from the matrix shown in Figure 2a. The dendrogram's x-axis shows each individual sample, and its y-axis shows merge distance. Initially, agglomerative clustering starts with three clusters: A, B and C. Because A and B are the closest clusters with a distance of 0.50, they constitute the first merge, and there is a horizontal bar between A and B at the y-axis value

0.50. At this point, the remaining clusters are AB and C. Since the distance between AB and C is 1.00, the dendrogram has a horizontal bar between the AB cluster and the C cluster at a merge distance of 1.00.

We chose agglomerative clustering as our clustering technique for two reasons. First, the results of agglomerative clustering are typically easy for a human to connect back to features of the input data. (By way of contrast, many researchers do not find that eigenvalue-based clustering [19] techniques have similarly human-explainable results.) Second, when the data possesses a well-defined clustering, agglomerative clustering usually finds it automatically. As we show in Section 4, the data we found during our study did possess such a well-defined clustering.

In this study, our primary metric was exedit distance, which we now define. We also experimented with two other metrics, edit distance over decoded shellcodes, and a metric based on control flow graph coloring.

3.4.1 Exedit Distance

Our methodology focuses on using relative edit distance over the executed bytes of the shellcode, abbreviated as *exedit distance*, to compare two shellcodes. Specifically, for each exploit sample we mark the executed instruction bytes, and concatenate the marked bytes *in the order they appear in the payload* (i.e., memory order) to construct a canonical string representation. As a final step, we compress each consecutive run of the NOP instruction `0x90` into a single `0x90` byte. Then we compute the relative edit distance over all exploits using these canonical strings. Relative edit distance is the number of edit operations (insertion, deletion, substitution) used to transform one string to another, normalized over the length of the longer string.

Using executed instruction bytes is different than using instruction streams, where the instructions in a stream representation appear in execution order (and not memory order) potentially multiple times.

3.4.2 Edit Distance

We also experimented with relative edit distance over the entire decoded shellcode without trying to distinguish between code and data, but encountered limitations. First, because this metric does not distinguish code from data, it is sensitive to changes in embedded string constants. Second, edit distance could not help us infer similarities between exploits across vulnerabilities because of differing vulnerable buffer sizes. Third, random padding that was neither code nor data in some cases generated further noise.

3.4.3 Structural Distance

During our investigation, we implemented a version of the control flow graph (CFG) coloring approach by Kruegel et al. [11]. We used it to calculate structural distance as the percentage of "mismatched" basic blocks between two samples. A block was considered mismatched if it was not part of a subgraph shared between the two samples.

The primary limitation of this technique was that it did not capture subtle variation between related exploits because entire basic blocks were summarized by the presence or absence of certain categories of opcodes. This lost information rendered this metric inappropriate for our purpose – it often grouped together distinct families of exploits.

4. EXPLOIT DIVERSITY

In this section we analyze the diversity of exploit shellcodes in a trace capturing live exploit attempts from hosts on the Internet to four well-known vulnerabilities: SQL Name Resolution, LSASS, MS RPC ISystemActivator, and MS RPC RemoteActivation. We chose the vulnerabilities because various infamous Internet worms used exploits for these vulnerabilities over the past few years to infect hundreds of thousands to millions of hosts, and exploit traffic to these vulnerabilities is still prevalent today.

The trace we use captures exploit attempts on a residential DSL network for two days starting on September 6, 2005 at 5pm. We captured traffic using a DSL-connected machine that ran Windows XP SP2 (fully-patched), listened on 29 IP addresses, and responded to incoming requests. We then used the Shield framework described in Section 3.2 to identify and capture exploits for known vulnerabilities.

We discuss the vulnerabilities in the order of the extent of diversity of their exploits, from negligible (SQL Name Resolution) to most extensive (RemoteActivation). For each vulnerability with substantial diversity, we first examine the number and frequency of shellcodes for that vulnerability. We then use our methodology from Section 3 to (1) cluster the shellcodes according to their variability and thus identify shellcode families, (2) provide a detailed characterization of each family to both convey the structure of shellcode families as well as the subtle functional variations among them, and (3) show the prevalence of each shellcode family in the trace. By manually examining the shellcode families identified using our clustering methodology, we show that the exedit distance metric is capable of correctly identifying exploit shellcode families, as well as capturing subtle functional variations among closely related families.

We then end the section by characterizing exploit shellcode diversity in a second trace of traffic from a honeyfarm at the Lawrence Berkeley National Laboratory. With this trace, we examine shellcode diversity from a different vantage point in the Internet, and we use our clustering methodology across all vulnerabilities in the trace at the same time. As a result, we show that our methodology is also able to identify the most prevalent shellcode exploit in this trace as a multi-vector exploit targeting three different vulnerabilities at once.

4.1 SQL Name Resolution

The Slammer worm exploited the SQL Name Resolution vulnerability (Microsoft Security Bulletin MS02-039) when launched in January 2003. Slammer was particularly virulent, infecting 90% of vulnerable hosts within 10 minutes, and the resulting congestion from probes by infected machines caused disruption on the Internet on a global scale [15]. Despite the small population of vulnerable hosts (roughly 75,000 infected hosts during the initial outbreak in 2003), Slammer continues to propagate in the wild to this day.

Shellcode Instance	# Occurrences
0	766
1	1

Table 1: Exploit frequency per shellcode for the Slammer vulnerability (MS02-039).

Table 1 shows the exploit frequency per shellcode for the SQL resolution vulnerability. We observed two apparent variations of Slammer: 766 exploits with the exact same payload, and one outlier.

Examining the outlier in more detail, we suspect that its payload was likely corrupted on the network before being captured in our trace. The outlier was identical to all the other payloads except for the last 91 out of 376 bytes. Whereas the trailing bytes of all other

exploits contained valid executable code, the tail for the outlier contained (1) an unidentified 22 bytes (possibly a link-layer protocol header), followed by (2) a 20-byte IP header containing the same source address as the sending host but a destination address in an ISP from a different geographic region, (3) a UDP header destined for port 1434 (the SQL resolution port), and (4) the first 41 bytes of the Slammer exploit. Given these trailing bytes, it appears as if the tail of one Slammer packet was overwritten with the head of another Slammer packet. Because of portion (2) of the trailing bytes, we believe the outlier was not the result of incorrect reassembly at the local capture machine.

Other than the corrupted outlier, we conclude that there was only one Slammer exploit in the residential DSL trace and, therefore, no exploit diversity.

4.2 LSASS

The Local Security Authority Subsystem Service (LSASS) validates user logins on Microsoft Windows systems. The original Sasser worm released in April 2004 exploited a buffer vulnerability in LSASS (MS04-011), and as it spread it disrupted operations for airlines, banks, hospitals, news agencies, military, and government offices globally [8]. Indeed, Sasser's significant damage made the trial and conviction of its author a media spectacle [1]. LSASS exploits persist to this day as a regular element of a malware program's repertoire: hundreds of malware variants (mostly bots) incorporate an LSASS exploit according to Trend Micro [28].

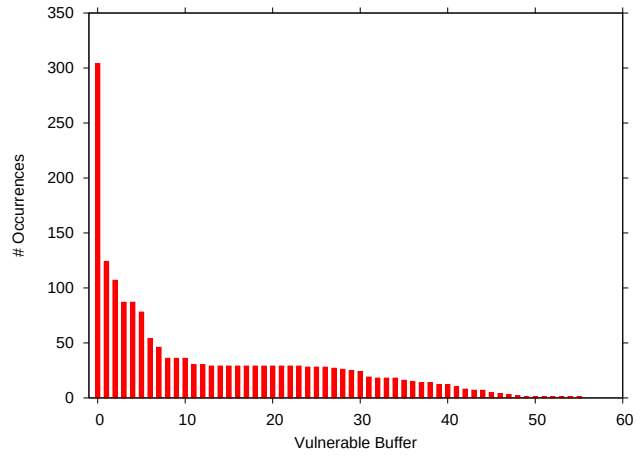


Figure 3: Exploit frequency per shellcode for the LSASS vulnerability (MS04-011) in the residential trace. There are 56 distinct payloads out of 1769 exploits total.

Number of distinct shellcodes: We captured 56 distinct payloads from 1,769 LSASS exploit attempts in the residential DSL trace. As shown in Figure 3, the exploit attempts were skewed across the payloads: a handful of variants were responsible for a large number of occurrences, with a long tail to the distribution.

With 56 different payloads, LSASS exploits clearly have abundant diversity. To what extent do they vary, and what is the nature of the variation? To answer these questions, we analyzed the shellcodes in the exploit payloads according to the methodology described in Section 3 and clustered the exploits first into families and then into phylogenies.

Evaluation of distance metrics and resulting clusters: Figure 4a shows a dendrogram visualizing clustering with the exedit metric (from Section 3.4.1). Each x-axis position represents one of the unique shellcode payloads from Figure 3 (although they are not ordered by prevalence here), and the y-axis shows relative edit distance. A horizontal line segment in the graph at y-axis value y indicates that two sub-clusters had cluster distance y when they were merged into one cluster; when merging clusters, we use complete linkage distance as discussed in Section 3. The vertical line segments extending downward from the endpoints of a horizontal segment are the roots of the two subtrees representing the two sub-clusters.

For example, the 9th and 10th x-axis entries in Figure 4a are joined together by a horizontal segment at a y-axis value of 0%. While constructing the hierarchy, a cluster containing an LSASS exploit and another containing an LSASS exploit were merged at a cluster distance of 0%, indicating that the executed instruction bytes for the samples within each cluster were identical. At the other extreme, the dendrogram shows that the maximum exedit distance between any two exploit samples was 98% because all samples were contained in sub-clusters rooted at the segment at 98%.

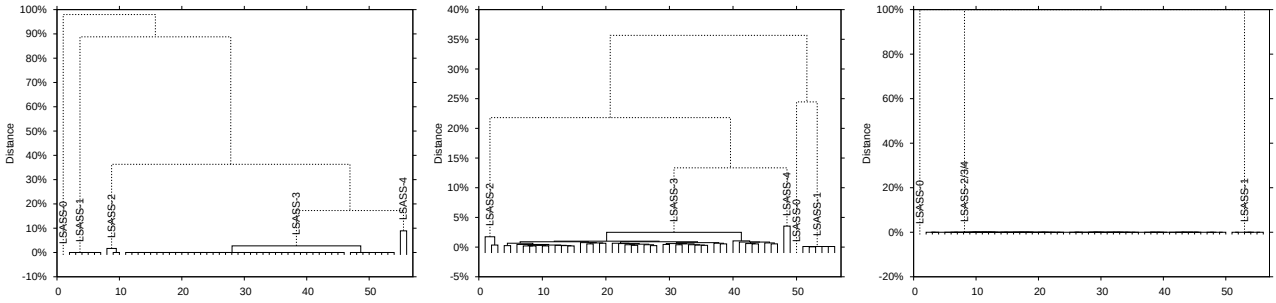
For comparison, Figures 4b and 4c show results using two other metrics: (1) edit distance over decoded payloads (*without* distinguishing code vs. data), and (2) the metric based on Kruegel et al.'s control flow graph fingerprinting technique described in Section 3.4.3. Our experience with decoded-payload edit distance (over LSASS and other vulnerabilities) was that instance-specific data strings and random padding embedded within the payload created substantial noise. As a result, the metric could not capture similarities among exploits due to variations that were not fundamental to the code. On the other hand, we found the CFG metric based on basic blocks to be too general because summarizing basic blocks by simply the presence of certain opcodes ignores subtle differences between shellcodes. Consequently, this metric clustered functionally distinct exploits as identical, as shown with families LSASS-2, 3 and 4 grouped together in Figure 4c.

We found that exedit distance provided a useful balance between these two approaches: it is general enough that it ignores instance-specific variations due to constants and other non-executable bytes, but detailed enough that it can capture the subtle functional variations. Thus, we used exedit distance as our primary means of comparing exploits during clustering throughout this study.

Constructing phylogenies: Because most of the cluster merges occurred at a small exedit distance of 10%, we used 10% as the threshold for defining families among the exploits. With this threshold, we find five families of shellcodes exploiting the LSASS vulnerability. To visualize this process, every vertical line segment intersected by a horizontal line in Figure 4a at the y-axis value of 10% defines an exploit family according to the cluster represented by the tree rooted at the vertical line. We number each cluster in the dendrogram from left to right, and henceforth refer to each of the five families as LSASS-0, LSASS-1, and so on.

Manual analysis of phylogenies: To validate our clustering results, we manually examined the decoded payloads of every shellcode for every family. From this examination, we conclude that the suggested families indeed were five separate code bases. However, LSASS-2, 3 and 4 had sufficient similarity among them, as suggested by the clustering in the dendrogram in Figure 4, that we conclude that they were branches of the same code base. We summarize our inter-family analysis with the evolution diagram in Figure 5.

Each family exhibits a small amount of variation among shellcodes *within* the family. The differences in variations are 2–20 bytes, and correspond to phone-home/connect-back IP addresses, hostnames, and ports encoded in the payload. In a *phone-home* or *connect-back* attempt, a newly-infected victim connects to a specified host from which the victim downloads additional code or exe-



(a) Exedit (b) Edit (c) Structural

Figure 4: Dendrograms for LSASS (MS04-011) exploits using (a) edit distance over executed code, (b) edit distance over decoded shellcodes and (c) structural distance over executed code.

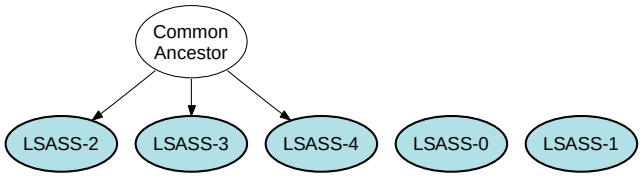


Figure 5: Evolution diagram for the LSASS families.

cutable files. The typical distinction between the two terms is that connect-back refers to connecting to the victim's immediate parent in the infection chain, and phone-home refers to connecting to a central location.

Next we give a detailed characterization of each LSASS family to provide intuition on the structure of the shellcodes, convey the substance of shellcode variation within and among LSASS families, and provide examples of how the exedit distance metric can differentiate subtle functional variations among shellcodes.

LSASS-1's shellcode was straightforward. Function names such as GetProcAddress, ExitThread, and so on formed a miniature data section at the end of the payload, and the main body of the exploit followed immediately after the decoding loop. The main body and the data section were XOR-ed one byte at a time with `0x99`.

LSASS-0's payload consisted of an unencoded main body followed by an encoded data section. The encoding scheme was also a byte-wise XOR, but with the key `0xff`. We found embedded strings like "http://atlantcommerce.com/stuff.exe" and "zoxter.exe" in the decoded data section. A search for these strings suggested that they belonged to malware previously classified by Symantec as Trojan.Netdepix [25].

LSASS-2, LSASS-3 and LSASS-4 shared the same encoding scheme and roughly the same flow of execution. Whenever a `0x99` byte was read from the encoding payload, the decoding loop read the next byte `B` and wrote `((B - 0x30) XOR 0x99)` as the decoded byte. Otherwise, it XOR-ed the byte with `0x99`. All three families had dedicated function blocks at the end of their payloads for library loading and function finding. The portion of the main body after the decoding loop was a series of calls to these function blocks, and there were only minor differences among these families.

Both LSASS-2 and LSASS-3 contained the strings ".\ftpupd.exe" and some variation of "u8," "u13," etc. in the middle of the payload. A search for these strings suggests that they belong to malware classified by various commercial security firms as Korgo [26].

Only LSASS-3 and LSASS-4, however, contained an identifier string that precedes the decoding routine. Although both had a URL containing the attacking machine's IP and port number, the padding after the URL differed. In particular, LSASS-0 identifiers had taken the following form "http://<addr>:<port>/x.exe\xdf\xdf ... \xdfMozilla/4.0" and the `0xdf` characters padded the identifier's length to a fixed value over all exploits (37 bytes). By contrast, LSASS-4 exploits appended a string containing "lsd" to the URL.[1]

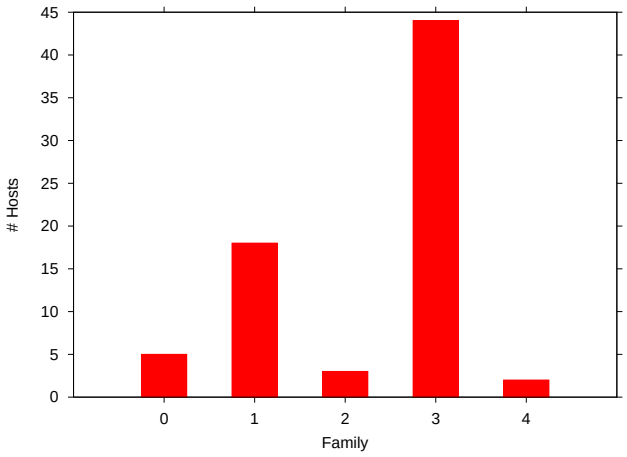


Figure 6: The distribution of hosts for each family exploiting the LSASS vulnerability. We clustered using a similarity threshold of 10%. Most hosts belonged to families LSASS-1 and LSASS-3, which are represented by labels 1 and 3 in the graph.

Prevalence of exploit families: Finally, we complete the analysis of LSASS with a discussion of exploit prevalence at the family granularity (as opposed to the shellcode granularity in Figure 3). Figure 6 shows the relative prevalence of each exploit family, expressed as the distribution of hosts contributing exploits to each family. For each exploit family, we tallied the number of hosts that attempted to exploit the LSASS vulnerability; each host made an

[1]The string "lsd" could be a reference to the group Last Stage of Delirium, famous for discovering the RemoteActivation vulnerability (MS03-026).

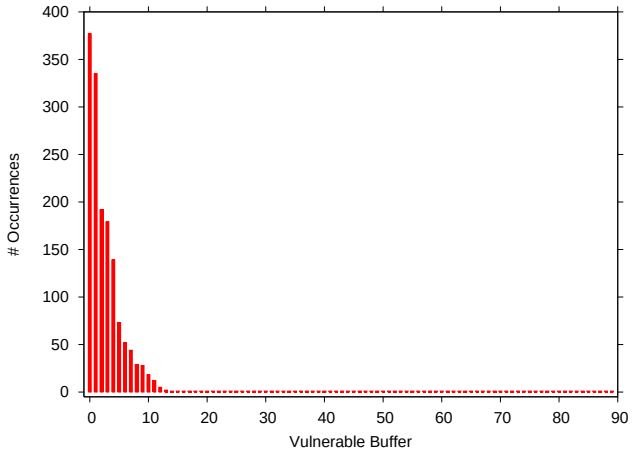


Figure 7: Exploit frequency per shellcode for the ISystemActivator vulnerability (MS03-039). There are 90 distinct payloads out of 1,561 exploits total. There is a long tail of nearly 80 exploit payloads that appear only once.

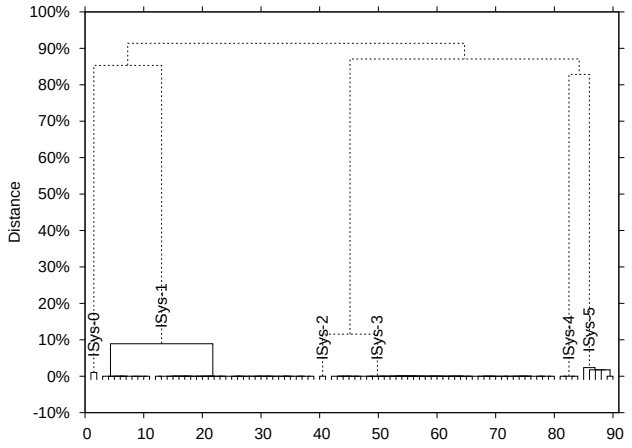


Figure 8: Dendrogram for ISystemActivator (MS03-039) exploits using exedit distance.

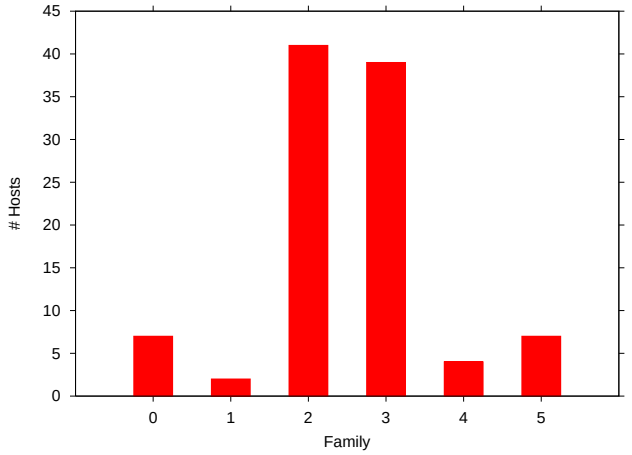


Figure 9: Distribution of hosts for each ISystemActivator family. We clustered using an intra-cluster exedit distance threshold of 10%. Most hosts belonged to families ISys-2 and ISys-3 (represented by labels 2 and 3 in the graph), which were bind and connect-back versions of the same exploit.

exploit attempt using only one distinct payload, so the sets of hosts contributing to each family were disjoint. Most of the hosts (44) used the LSASS-3 exploit, followed by LSASS-1 (18 hosts).

Summary: The LSASS exploits we captured had a well-defined phylogeny. Within this phylogeny, we attributed sources of variation to differing instance-specific constants such as URLs and IP addresses. Moreover, our technique of using exedit distance (Section 3.4.1) made further manual analysis much easier because the clustering resulted in appropriate families. Moreover, the distance metric was a strong predictor of the type of similarity we would find among exploits.

4.3 ISystemActivator

The MS Remote Procedure Call (MS RPC) service is a fundamental Windows service and was exploited by one of the widest spreading worms. In August 2003, the Blaster worm infected hundreds of thousands of computers within the first 24 hours of its propagation, and its variants were responsible for infecting anywhere between half a million to 8 million hosts several months thereafter [12]. Blaster originally exploited the RemoteActivation vulnerability in MS RPC, which we cover in Section 4.4, and later variants exploited a vulnerability in MS RPC's ISystemActivator interface (MS03-039). Since then, exploits of the ISystemActivator vulnerability have also become a mainstay of the botnet exploit arsenal.

Number of distinct shellcodes: We captured 90 distinct payloads from 1,561 exploit attempts in the residential DSL trace. As shown in Figure 7, the exploit attempts were also skewed across payloads: 10 variations were responsible for most of the observed exploits. However, the long tail of almost 80 distinct shellcodes appearing only once raised the question of whether this was the result of polymorphism or other trivial differences. As with the LSASS vulnerability, we performed clustering of the exploit payloads to determine the extent and nature of this variation.

Constructing phylogenies: Figure 8 shows the dendrogram for clustering decoded ISystemActivator payloads using the exedit distance metric. Again, most of the cluster merges happened below a distance of 10%. We use this distance value as the threshold to define families among the exploits, resulting in six families for exploits for the ISystemActivator vulnerability. The low initial threshold and large gap between cluster merges at distances of 10% and 85% indicate that exploits within a family are similar, but that ISys families differ more substantially from each other than the LSASS exploit families.

Manual analysis of phylogenies: Through manual analysis, we confirmed that the clusters shown in the dendrogram reflect six different code bases among the ISystemActivator shellcodes (summarized in Figure 10). Within each family, though, there was no code polymorphism in the exploit shellcodes. The several-byte differences among exploits were due to variations in data constants, such as encodings of phone-home addresses and hostnames, as well as names of executables. We now discuss the structure of the ISystemActivator shellcode families, and the nature of the differences within and among them.

ISys-0 used a 4-byte, non-overlapping XOR to encode its payload, whereas all other exploits we examined used some variant of a byte-by-byte XOR. Also, the decoding loop used subtraction instructions to set register values. In other respects, the organization of its code was straightforward—the main body consisted

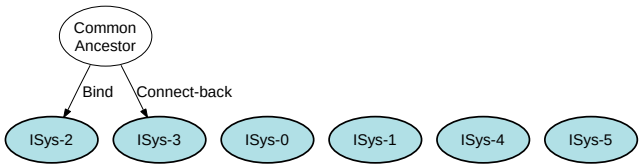


Figure 10: Relationship of the ISystemActivator families.

of a series of calls to a kernel-base loading function block and a function-finding block, both present at the end of the payload.

ISys-4 had a relatively simple decoding loop, but it had the largest payload length and its flow of execution was the most complicated among the ISystemActivator exploits. The data section containing function names was in the middle of the payload, and several function blocks were appended after that. The flow of execution started after the decoding loop, invoking many calls to the function blocks. When the execution approached the data section, however, it jumped past the data section and the function blocks to another set of instructions that performed other function calls.

The moderate exedit distance within the ISys-1 family (9%) was due to the differing decoding routine offsets and constants in its exploits, but the shellcodes were otherwise identical. This difference in decoding routines gives us confidence in the robustness of the clustering algorithm to particular choices of the clustering threshold. A different threshold that split this family in two would have also been reasonable. We could have chosen a lower clustering threshold (e.g., 5%) to split ISys-1 into two small families. This new smaller threshold would not have had a large impact on the results, though, because the differences between shellcodes within ISys-1 were minor compared to the differences between families.

ISys-5 exploits had a characteristic execution flow that not only performed the standard loading of memory addresses and function finding, but also performed consecutive jumps over two text sections to obtain data offsets in the exploit, and executed the remaining code thereafter. The first data section contained the string "tftp.exe -i <address> get <executable name>", and the second data section simply contained the same "<executable name>". Variation in the address (e.g., taking values such as 0.0.0.0 or a connect-back IP) and executable name (such as "updr32.exe" and "system32.exe") accounted for the 6.5% average distance among samples in this family.

The relatively low 10% exedit distance between ISys-2 and ISys-3 suggested that they would be similar in many respects. Further manual analysis revealed that both shared the same decoding loop using `0x99` as the key (this is the same as LSASS-3's loop, although we were not aware of any multi-vector malware responsible for this), and both used a miniature data section containing the same function names near the end of the payload. However, ISys-2 used a shorter exploit payload length and a shorter NOP-sled than ISys-3, but had a longer main body. The basic blocks within the main body were similar except for some block reordering, register renaming, and instruction substitution. Interestingly, the number of iterations in ISys-3's loop overshoots the exploit payload (past the end of the data section). Thus, it seems reasonable to hypothesize that either ISys-2 was a refinement of ISys-3, or that ISys-3 was a poor imitation of ISys-2.

A closer look reveals further similarities between ISys-2 and 3. Both exploits contained the following strings in the same order near the end of the shellcode: `GetProcAddress`, `CreateProcessA`, `ExitThread`, `LoadLibraryA`, `ws2_32`, `WSASocketA`, with `closesocket` at the end of the payload. The only strings that differed were the last few—ISys-3 contained a `connect`, but ISys-2 contained a `bind`, `listen`, and `accept`. As a result, we believe that these two families were derived from the same code base except that ISys-3 required the newly-infected host to *connect back* to the infecting host, while ISys-2 required the newly-infected host to *bind* on a socket and wait for a connection attempt from the infecting host. Malware typically executes connect-back shellcodes when the newly infected host resides within a NAT/firewall, and executes a bind shellcode if the infector is residing within a NAT/firewall. Indeed, more investigation revealed that the 0.5% average intra-cluster edit distance (which is 4 bytes out of ISys-3's 776-byte shellcode) exactly corresponded to the integer form of the connect-back address—further support that ISys-3 was the connect-back version of ISys-2's bind shellcode.

Prevalence of exploit families: Figures 9 shows the distribution of hosts over the families. Most of the exploits belonged to families ISys-2 and ISys-3. Because ISys-2 and ISys-3 only differed in their connect-back functionality and have similar prevalence, we hypothesize that they could have belonged to a botnet that was able to traverse NAT/firewalls by using a combination of bind and connect-back shellcodes.

Summary: The ISystemActivator exploits formed another well-defined phylogeny, and our methodology allowed us to capture the subtle distinction between bind and connect-back versions of a shellcode. Specifically, the moderate distance between families ISys-2 (bind) and ISys-3 (connect-back) provided a strong hint that the two families were related, a hypothesis we confirmed manually.

4.4 RemoteActivation

As discussed at the beginning of Section 4.3, RemoteActivation was the original MS RPC vulnerability (MS03-026) that Blaster and its variants exploited before also targeting ISystemActivator.

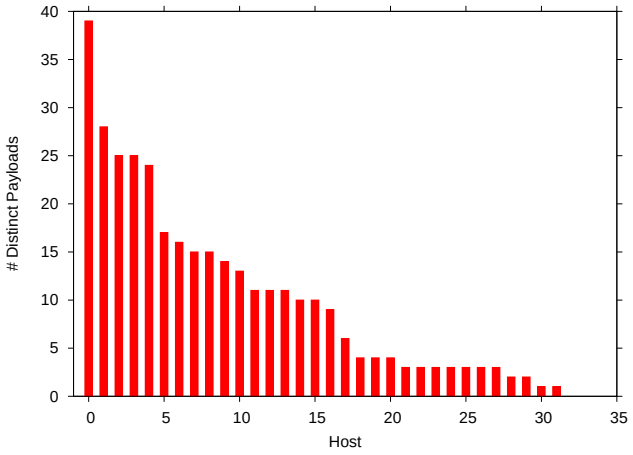


Figure 11: Exploit diversity per sending host for the RemoteActivation vulnerability (MS03-026). Each of the 338 exploit payloads was unique.

Exploit frequency: We captured 338 distinct exploit payloads for the RemoteActivation vulnerability. The RemoteActivation exploits represent the opposite end of the spectrum to the SQL vulnerability: each RemoteActivation exploit attempt used a unique payload. Figure 11 shows the distribution of exploit attempts (and, therefore, distinct payloads) among the 31 remote hosts trying to exploit the RemoteActivation vulnerability. Moreover, unlike the other exploits we analyzed, RemoteActivation exploits exhibited a high amount of exploit diversity per host.

Constructing phylogenies: Figure 12 shows the dendrogram resulting from clustering the payloads using the exedit distance metric. Compared to the LSASS and ISystemActivator shellcodes, the exedit distance among RemoteActivation shellcodes was very small. Judging from the dendrogram, most cluster merges occur below a distance of 1%. Using this threshold results in two families for the RemoteActivation shellcodes, although the 1.3% inter-family exedit distance indicates that the families are closely related.

Manual analysis of phylogenies: Manually inspecting the exploits reveals that the last third (roughly 300 bytes) of the payload contained randomly generated characters selected from lower-case letters, except for several fixed characters embedded within the random text. There were also small 2- and 4-byte differences in the main body as well. These random characters accounted for the variation within each family.

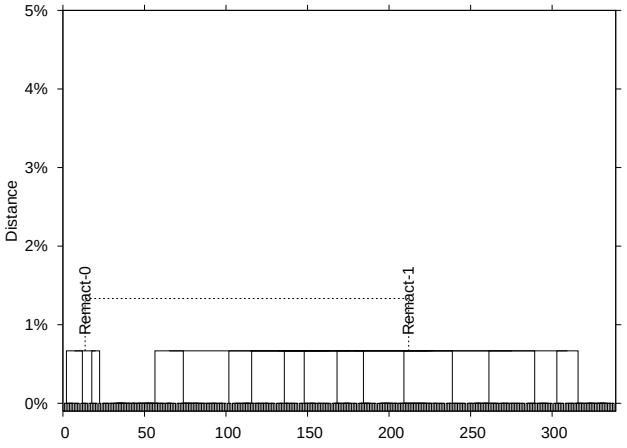


Figure 12: Dendrogram for the RemoteActivation (MS03-026) exploits using exedit distance. Setting a maximum intra-family distance of 1% yielded two closely-related families.

A more thorough inspection led us to conclude that there were two very similar types of RemoteActivation exploits in our trace. Figure 13 shows an evolution diagram that reflects our findings.

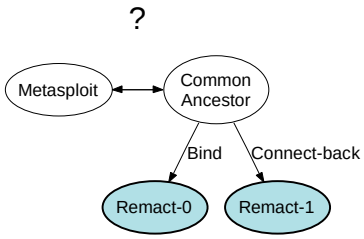


Figure 13: Relationship of the RemoteActivation families.

One peculiar feature of the RemoteActivation payloads was that they shared the same prefix—a large NOP sled sprinkled with the hex strings "\xeb\x04\xff\xff" and "\xeb\x04". A search for those strings led us to a Perl module that was part of the Metasploit Framework [14]. Metasploit is a toolkit for generating exploits, and includes options for generating encoded shellcodes and random filler characters. Although we do not directly attribute our exploits to a Metasploit generation, the similarities helped provide insight into the construction used by their authors.

The hex strings were return addresses for redirecting control flow for exploiting Windows NT, 2000 and XP. In addition, the Metasploit Perl module invoked a function Pex::Text::EnglishText(360) for filling the last portion of the exploit with random characters from ASCII 33–126, a superset of the lower-case letters we observed in the RemoteActivation exploits. More importantly, however, the fixed characters placed within the random text in the observed exploits exactly matched those from the Perl module—they were jump commands and return addresses for Windows XP and 2003. Furthermore, both were terminated with the four characters '\', NULL, 'A' and NULL. However, we could not directly attribute the exploits we observed to Metasploit because the NOP-sled in the Perl module was smaller, the distribution of random characters was different, and we could not find an instance of the decoded shellcode within the Metasploit framework.

The byte-wise encoding scheme only covered the main bodies of the exploits, but different exploits used different keys. Indeed, within each family the average edit distance between non-decoded shellcodes was 28.5% to 29.8%, whereas the average edit distance between decoded shellcodes was 19.6% to 19.9%, as shown in Table 2. And with manual inspection we confirmed that variable encoding of the exploit's main body contributed to the jump in average intra-family distance. Changing keys along with random filler characters are commonly described techniques for polymorphism, and the RemoteActivation exploits had both of these features.

Family	# Hosts	Non-decoded Edit	Decoded Edit
0	3	28.5%	19.6%
1	29	29.8%	19.9%
0 vs 1		32.5%	23.2%

Table 2: Effect of random filler: intra-family and inter-family results for the RemoteActivation vulnerability include number of hosts, average edit distance over non-decoded payloads, and average edit distance over decoded payloads.

The decoded main bodies for RemoteActivation fell into two groups—90% belonged to family Remact-1 and were 206 bytes long, while the other 10% belonged to Remact-0 and were 226 bytes long. The Remact-0 and Remact-1 exploits were similar because they shared the same function-finding block, which was placed after the same decoding routine. However, the code after the function block performed a connect-back attempt in Remact-1, and performed a bind in Remact-0 (hence the longer shellcode). Nonetheless, both exploit forms ended with a call to recv(), a load of a 4-byte authentication token, and a jump to the second exploit payload downloaded by the recv() call (we thank the Nepenthes [17] honeypot for contributing this analysis of the Remact-1 shellcode).

Summary: The exedit distance metric is able to reconstruct the phylogeny for RemoteActivation exploits even in the presence of non-trivial polymorphism such as variable encoding keys, and random padding characters embedded within the payload. Moreover, functional differences in the shellcodes primarily contributed to the distinction between Remact-0 (bind version) and Remact-1 (connect-back version). Otherwise, the shellcodes for RemoteActivation exploits did not share the same degree of diversity as the LSASS and ISystemActivator shellcodes.

4.5 Diversity Across Vulnerabilities

As our final experiment, we analyze exploit diversity in a trace from a very different network vantage point. Using this second trace, we examine the relative prevalence of vulnerability exploits

Family	# Hosts	# Exploits Sent	Vulnerabilities	Avg Intra-Family Edit
LBL-3	463	728	ISystemActivator (54.4%) LSASS (44.6%) PNP (1%)	ISystemActivator (0.4%) LSASS (0.1%) PNP (0.1%)
LBL-2	273	296	ISystemActivator	0.4%
LBL-6	152	164	ISystemActivator	2.6%
LBL-5	100	384	PNP	0.1%
LBL-4	11	11	ISystemActivator	0.5%
LBL-1	3	4	ISystemActivator	0.0%
LBL-0	1	9	RemoteActivation	19.3%

Table 3: Per-family results for the LBL trace including the number of hosts, number of exploit payloads sent, the constituent vulnerabilities (with breakdowns expressed as a percentage of the number of exploits sent), and the average intra-family edit distance over decoded shellcodes. All families were disjoint.

at a different point in the Internet, and apply our exploit clustering methodology across multiple vulnerabilities at a time to identify multi-vector exploits. The trace is a full-payload 4.5-day trace from a Windows honeyfarm running at the Lawrence Berkeley National Laboratory starting on April 19, 2006. Hosts in this honeyfarm served as active responders to incoming requests to a larger range of IP addresses, five /24 subnets. The trace was a tcpdump of network traffic, and we identified and captured exploits by running the trace through Shield.

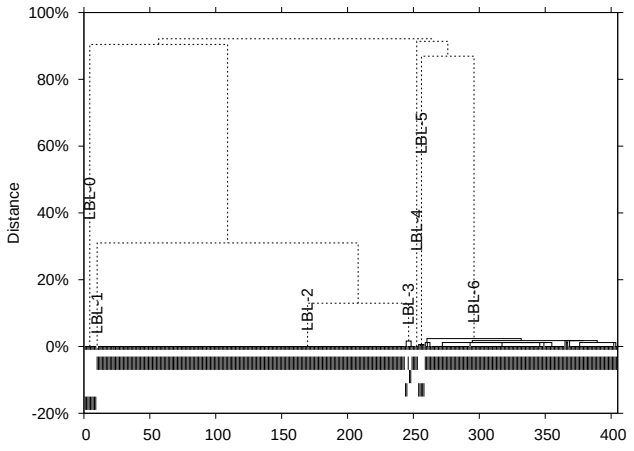


Figure 14: Dendrogram for the LBL trace exploits using exedit distance. The 1st set of hash marks just below 0% represent ISystemActivator, the 2nd represent LSASS, the 3rd represent PNP, and the 4th represent RemoteActivation.

Constructing phylogenies: In previous experiments we clustered exploits among shellcodes targeted at individual vulnerabilities. In this experiment, we cluster the exploit shellcodes to all vulnerabilities found in the LBL trace that are known to Shield at once. Figure 14 shows the results of clustering over multiple vulnerabilities in the LBL trace as a dendrogram. As with previous experiments, since most cluster merges occur below an exedit distance threshold of 10%, we use this value as the threshold for clustering exploits into families. Seven families form as a result.

Prevalence of exploit families: Manually inspecting the shellcodes confirmed that the seven families were distinct. For each shellcode family, Table 3 shows the host prevalence, the number of shellcodes sent, the vulnerabilities exploited, and the intra-family distance. The entries are sorted in decreasing order of host prevalence.

The LBL-2 and LBL-3 families were, respectively, the connect-back and bind versions of the same shellcode (as suggested by their relatively low inter-family distance of 15%). LBL-3 was the most prevalent family with 463 hosts sending its shellcodes. The most striking aspect of LBL-3 was that it was a multi-vector family, exploiting three different vulnerabilities. However, it exhibited very little intra-family variation, with an edit distance of at most 0.4% for shellcodes within each of the vulnerabilities exploited.

In contrast, LBL-0 was the least prevalent family. It targeted the RemoteActivation vulnerability with the same shellcodes found in the residential DSL trace in Section 4.4. As with the shellcodes from the residential trace, LBL-0 had the most variation within the family (a relative edit distance of 19.3%) due to the randomly generated characters at the end of the shellcodes.

Summary: Our clustering methodology with the exedit distance metric was able to cluster exploits into families across shellcodes targeting multiple vulnerabilities. As a result, it was effective in identifying multi-vector exploits. Finally, comparing the LBL and residential DSL traces, we found that the exploit shellcodes generally targeted the same set of vulnerabilities (with even the same RemoteActivation shellcodes used in both traces), and the most prevalent shellcode in the LBL trace was a multi-vector exploit.

5. DISCUSSION

The research community has been engaged in a vigorous debate over the likely future role of network intrusion detection systems (NIDS), and to what extent exploit polymorphism will limit their effectiveness. The previous section documented that polymorphism within shellcodes is already in use on the Internet. To investigate whether this polymorphism made NIDS signature construction difficult, we generated a small set of signatures that exhaustively covered all exploits we observed for each vulnerability in the DSL residential trace. Each signature was a contiguous sequence of 100 bytes. We constructed the signature set using the well-studied greedy algorithm for the set cover problem [7]. For each individual vulnerability except LSASS, one signature sufficed to cover the set of exploits. LSASS required two: one covered 1645/1769 exploits, and the other covered the rest. Manual investigation of these signatures showed that they primarily focused on the portions of the shellcode that were mostly (but not entirely) NOPs. We then tested the signatures against a 5-GB trace on our internal network for false positives. None of the signatures yielded false positives in the internal trace.

Our above experiment generating signatures for the shellcodes we traced suggests that the polymorphism was not effective for evasion. Two alternative hypotheses for the motivation of the polymorphism we observed are that perhaps the malware authors were using polymorphism to evade signatures other than the ones we constructed, or perhaps they did not realize that their polymorphism was ineffective against signature construction (e.g., because their

shellcodes were still effective at compromising a sufficient number of hosts).

Yet another hypothesis is that today's polymorphism is unrelated to evading NIDS signatures. Some of the variation in shellcodes was clearly due to functional variation (e.g., the bind and connect-back varieties of ISystemActivator and RemoteActivation). The variation due to decoding routines with variable keys might be simply motivated by increasing the difficulty of reverse engineering.

With our traces, we cannot definitively ascribe intent to the polymorphism we observed in these exploits, but we suspect that it is likely some combination of the above reasons. Improving our understanding of the motivations for the actual polymorphism on the Internet, as well as how the uses and motivations for polymorphism change over time, remain interesting open problems.

6. CONCLUSION

We have presented a methodology for constructing the phylogeny of remote code injection exploits. We evaluated this methodology on network traces taken from several vantage points. We found our methodology to be robust to the polymorphism observable in these traces. Using our methodology, we found both non-trivial code sharing and subtle variation within the exploit families such as the use of bind versus connect-back to traverse NATs.

Polymorphism and exploit diversity have been a topic of significant discussion within the research literature. We have documented the use of polymorphism in exploits currently loose on the Internet. More broadly, analyzing both the emergence of polymorphism and the phylogeny of remote code injection exploits is part of a broader effort to better understand the social and technical dynamics around malware creation. We believe our methodology and results are a promising contribution to this broader understanding.

Acknowledgments

We would like to thank Vern Paxson for access to the LBL traces and for insightful discussions, Igor Volovich for help obtaining traces from our internal network, and Michael Vrable and Michelle Panik for feedback on this paper. Also, we would like to thank the anonymous reviewers for their helpful comments. This work was supported by Microsoft, NSF grant CNS-0433668, and the UCSD Center for Networked Systems.

7. REFERENCES

[1] BBC News. Sasser Creator Avoids Jail Term. http://news.bbc.co.uk/2/hi/technology/4659329.stm, July 2005.

[2] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP)*, Brighton, UK, Oct. 2005.

[3] J. R. Crandall. Minos: A Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, Fairfax, VA, Oct. 2004.

[4] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, Nov. 2005.

[5] T. Dullien and R. Rolles. Graph-Based Comparison of Executable Objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, June 2005.

[6] H. Flake. Structural Comparison of Executable Objects. In *Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2004.

[7] D. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.

[8] G. Keizer. Sasser Worm Impacted Businesses Around the World. http://www.techweb.com/wire/story/TWB20040507S0008, May 2004.

[9] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[10] C. Kreibich and J. Crowcroft. Honeycomb — Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the 2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, MA, Nov. 2003.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, Seattle, WA, Sept. 2005.

[12] R. Lemos. MSBlast Epidemic Far Larger than Believed. http://news.com.com/2100-7349_3-5184439.html, Apr. 2004.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[14] Metasploit Project. The Metasploit Framework. http://www.metasploit.com/projects/Framework/.

[15] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[16] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Network telescopes. Technical Report CS2004-0795, UCSD, July 2004.

[17] Nepenthes Development Team. ShellcodeHandler Generic LinkTrans. http://nepenthes.mwcollect.org/.

[18] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, San Diego, CA, Feb. 2005.

[19] A. Ng, M. Jordan, and Y. Weiss. On Spectral Clustering: Analysis and an Algorithm. In *Proceedings of Advances in Neural Information Processing Systems*, 2001.

[20] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet Background Radiation. In *Proceedings of the USENIX/ACM Internet Measurement Conference*, Taormina, Sicily, Italy, Oct. 2004.

[21] P. Royal, D. Dagon, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Packed Malware. http://www-static.cc.gatech.edu/ ranma1/polyunpack/.

[22] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[23] spoonm. Recent Shellcode Developments. In *REcon*,

Montreal, QC, June 2005.

[24] A. E. Stepan. Defeating Polymorphism: Beyond Emulation. In *Proceedings of the Virus Bulletin International Conference*, Dublin, Ireland, Oct. 2005.

[25] Symantec. Trojan.Netdepix. http://www.symantec.com/avcenter/venc/data/trojan.netdepix.html.

[26] Symantec. W32.Korgo.AB. http://www.symantec.com/avcenter/venc/data/w32.korgo.ab.html, Apr. 2004.

[27] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[28] Trend Micro. Virus Encyclopedia. http://www.trendmicro.com/vinfo/virusencyclo/.

[29] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference*, Portland, Oregon, Sept. 2004.