An Empirical Study of Real-world Polymorphic Code Injection Attacks

Michalis Polychronakis[*] *Kostas G. Anagnostakis*[†] *Evangelos P. Markatos*[*]

Abstract

Remote code injection attacks against network services remain one of the most effective and widely used exploitation methods for malware propagation. In this paper, we present a study of more than 1.2 million polymorphic code injection attacks targeting production systems, captured using network-level emulation. We focus on the analysis of the structure and operation of the attack code, as well as the overall attack activity in relation to the targeted services. The observed attacks employ a highly diverse set of exploits, often against less widely used vulnerable services, while our results indicate limited use of sophisticated obfuscation schemes and extensive code reuse among different malware families.

1 Introduction

Despite considerable advances in host-level security hardening and network-level defenses, remote code injection attacks against network services persist as one of the most common methods for system compromise. Along with the more recently popularized client-side attacks that exploit vulnerabilities in users' software such as browsers and media players [15], remote code execution vulnerabilities continue to plague even the latest versions of popular OSes and server applications [2] and are effectively being exploited by malware, resulting in millions of infected hosts [3].

Motivated by the illicit financial gain against their victims, cyber-criminals constantly try to improve the effectiveness and evasiveness of their attacks, with the aim to compromise as many systems as possible and keep them under control for as long as possible. Code obfuscation and polymorphism [20] are among the most widely used evasion techniques employed by attackers to circumvent virus scanners and intrusion detection systems.

When polymorphism is applied to remote code injection attacks, the initial attack code is mutated so that every attack instance acquires a unique pattern, thereby making fingerprinting of the whole breed a challenge. The injected code—often dubbed *shellcode*—is the first piece of code that is executed after the instruction pointer of the vulnerable process has been hijacked, and carries out the first stage of the attack, which usually involves the download and execution of a malware binary on the compromised host. Polymorphic shellcode engines [1, 4, 7, 10, 16, 21] create different mutations of the same initial shellcode by encrypting it with a different random key, and prepending to it a decryption routine that makes the code self-decrypting. Since the decryption code itself cannot be encrypted, advanced polymorphic encoders also mutate the exposed part of the shellcode using metamorphism [20].

Although the design and implementation of polymorphic shellcode has been covered extensively in the literature [6–8, 13, 14, 16, 18], and several research works have focused on the detection of polymorphic attacks [11, 13, 14, 23], the actual prevalence and characteristics of real-world polymorphic attacks have not been studied to the same extent [12]. In this work, we present an analysis of more than 1.2 million polymorphic code injection attacks against real Internet hosts—not honeypots—detected over the course of more than 20 months. The attacks were captured by monitoring the traffic of thousands of production systems in research and education networks using network-level emulation [13, 14]. `Nemu`, our prototype implementation, uses a CPU emulator to dynamically analyze every potential instruction sequence in the inspected traffic and identify the execution behavior of self-decrypting shellcode.

Our study focuses on the attack activity in relation to the targeted network services, the structure of the polymorphic shellcode used, and the different operations performed by its actual payload. Besides common exploits against popular OS services associated with well known

[*]FORTH-ICS, Greece, {`mikepo,markatos`}`@ics.forth.gr`
[†]I^{2}R, Singapore: `kostas@i2r.a-star.edu.sg`

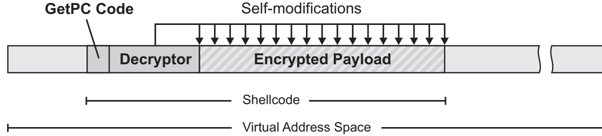

Figure 1: A typical execution of a polymorphic shellcode using network-level emulation.

vulnerabilities, we witnessed sporadic attacks against a large number of less widely used services and third-party applications. At the same time, although the bulk of the attacks use naive encryption or polymorphism, and extensive sharing of code components is prevalent among different shellcode types, we observed a few attacks employing more sophisticated obfuscation schemes.

2 Network-level Emulation

We briefly describe the design and operation of `nemu`, the detector used for capturing the attacks. The interested reader is referred to our previous work [13,14] for a more thorough description and implementation details.

The principle behind network-level emulation is that the machine code interpretation of arbitrary data results to random code, which, when it is attempted to run on an actual CPU, usually crashes soon, e.g., due to an illegal instruction. In contrast, if a network request actually contains polymorphic shellcode, then the shellcode runs normally, exhibiting a certain detectable behavior.

`Nemu` inspects the client-initiated data of each network flow, which may contain malicious requests towards vulnerable services. Each input is mapped to a random memory location in the virtual address space of an IA-32 emulator, as shown in Fig. 1. The execution of self-decrypting shellcode is identified by two key runtime behavioral characteristics: the execution of some form of *GetPC* code, and the occurrence of several *self references*, i.e., read operations from the memory addresses of the input stream itself, as illustrated in Fig 1. The GetPC code is used by the shellcode for finding the absolute address of the injected code, which is mandatory for subsequently decrypting the encrypted payload, and involves the execution of an instruction from the `call` or `fstenv` instruction groups [13].

We should note that for all captured attacks, `nemu` was able to successfully decrypt the original shellcode, while so far has resulted to zero false positives.

3 Data Set

Our analysis is based on the attacks captured by `nemu` in three deployments in European National Research Networks (referred to as NRN1-3) and one deployment in a public Educational Network in Greece (referred to as EDU). In each installation, `nemu` runs on a passive monitoring sensor that inspects all the traffic of the access link that connects the organization to the Internet.

The sensors were continuously operational for more than a year, except some occasional downtimes. The exact duration of each deployment, along with the number of detected attacks and other details, is shown in Table 1. In these four deployments, `nemu` collectively captured more than 1.2 million attacks targeting real production systems in the monitored networks.

We differentiate between *external* attacks, which originate from external IP addresses and target hosts within the monitored network, and *internal* attacks, which originate from hosts within the monitored networks. Internal attacks usually come from infected PCs that massively attempt to propagate malware in the local network and the Internet. We should note that due to NAT, DHCP, and the long duration of the data collection, a single IP may correspond to more than one physical computer.

4 Attack Analysis

4.1 Overall Attack Activity

As shown in Table 1, from the 1.240.716 attacks detected in NRN1, about one third of them were launched from 10.014 external IP addresses and targeted 769 hosts within the organization. The bulk of the attacks originated from 143 different internal hosts, targeting 331.572 different active hosts across the Internet. Interestingly, 116 of the 143 internal hosts that launched attacks are also among the 769 victim hosts, indicating that possibly some of the detected attacks were successful.

The overall attack statistics for NRN2 and NRN3 are similar to NRN1, but the number of detected attacks is orders of magnitude smaller, due to the smaller attack surface of infected or potentially vulnerable internal hosts that launched or received attacks. In contrast to the three NRNs, about two thirds of the attacks captured in the EDU deployment originated from external hosts.

An overall view of the external and internal attack activity for all deployments is presented in Fig. 2 and Fig. 3, respectively. In both figures, the upper part shows the attack activity according to the targeted port, while the bottom part shows the number of attacks per hour. For the targeted ports, the darker the color of the dot, the larger the number of attacks targeting this port in that hour. There are occasions with hundreds of attacks in one hour, mostly due to attack bursts from a single source that target all active hosts in neighboring subnets.

Network	Time Period	Total # attacks	External # attacks	External # srcIP	External # dstIP	Internal # attacks	Internal # srcIP	Internal # dstIP
NRN1	1/4/07 – 21/10/08	1240716	396899 (32.0%)	10014	769	843817 (68.0%)	143	331572
NRN2	1/4/07 – 15/2/08	12390	2617 (21.1%)	1043	82	9773 (78.9%)	66	4070
NRN3	1/4/07 – 15/2/08	1961	441 (22.5%)	113	49	1520 (77.5%)	8	1518
EDU	7/3/07 – 21/10/08	20516	13579 (66.2%)	3275	410	6937 (33.8%)	351	2253

Table 1: Number of captured attacks from four deployments of `nemu`.

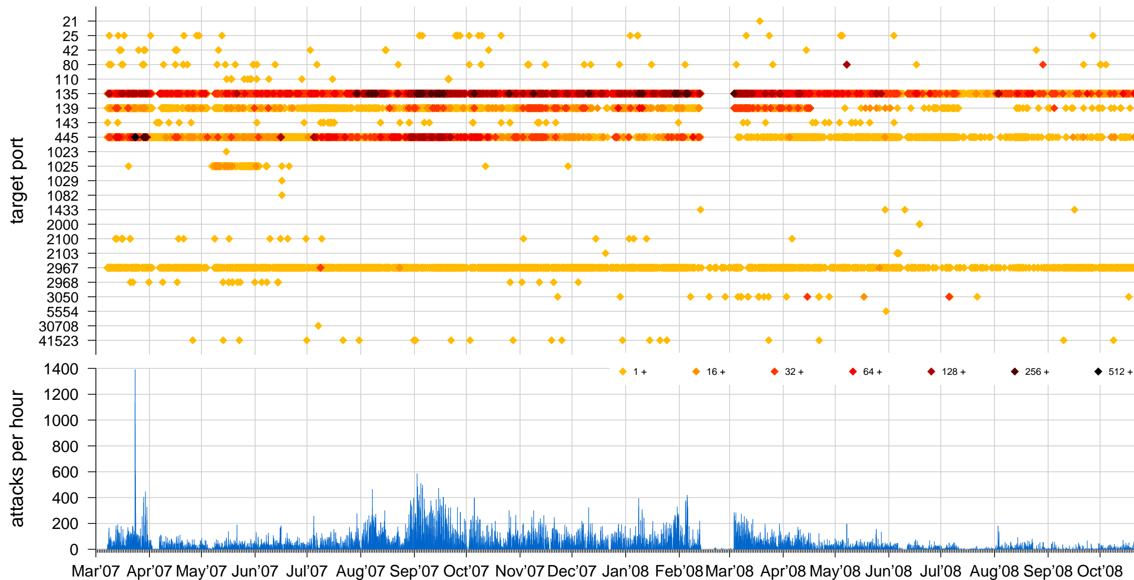

Figure 2: Overall external attack activity. Although the bulk of the attacks target well known vulnerable services, there are also sporadic attacks against less widely used services.

4.2 Targeted Services

As expected, the most highly attacked ports for both internal and external attacks include ports 135, 139, and 445, which correspond to Windows services that have been associated with multiple vulnerabilities and are still being highly exploited in the wild. Actually, the second most attacked port is port 2967, which is related to an exploit against a popular corporate virus scanner that happened to be installed in many hosts of the monitored networks. As shown in Fig. 3 several of these hosts got infected before the patch was released and were constantly propagating the attack for a long period. Other commonly attacked services include web servers (port 80) and mail servers (port 25).

It is interesting to note that there also exist sporadic attacks to many less commonly attacked ports like 3050, 5000, and 41523. With firewalls and OS-level protections now being widely deployed, attackers have started turning their attention to third-party services and applications, such as virus scanners, mail servers, backup servers, and DBMSes. Although such services are not very popular among typical home users, they are often found in corporate environments, and most importantly, they usually do not get the proper attention regarding patching, maintenance, and security hardening. Thus, these services have become attackers' next target option for remote system compromise, and as the above results show, many such exploits have been actively used in the wild. `Nemu` scans the traffic towards any port and does not rely on exploit or vulnerability specific signatures, thus it is capable to detect polymorphic attacks destined to even less widely used or "forgotten" services.

Overall, the captured attacks targeted 26 different ports. The number of attacks per port is shown in Fig. 4 (blue bars). A large number of attacks targeted port 1025, attempting to exploit the Microsoft Windows RPC malformed message buffer overflow vulnerability. Less commonly attacked services include POP3 and IMAP servers (ports 110 and 143), Oracle XDB FTP servers (port 2100), the Windows Internet Naming Service (WINS) (port 42), Microsoft SQL servers (port 1433), and the CA BrightStor Agent for Microsoft SQL Server (port 41523). The attack against port 5000

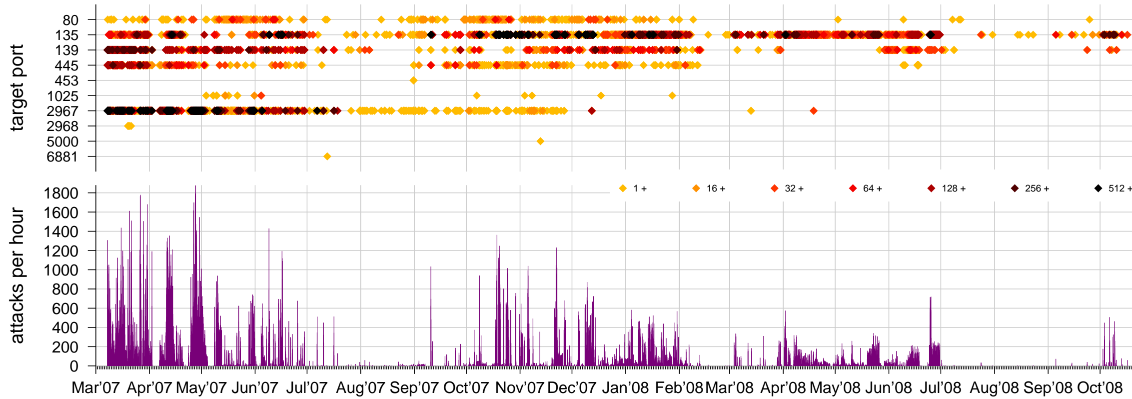

Figure 3: Overall internal attack activity.

was related to a vulnerability in the Windows XP Universal Plug and Play implementation, while the attack against port 6881 attempted to exploit a vulnerable P2P file sharing program. The single attack against port 5554 was launched by W32.dabber, a worm that propagates by exploiting a vulnerability in the FTP server started by W32.Sasser and its variants.

4.3 Shellcode Analysis

We analyzed the shellcode of the captured attacks with the aim to gain further insight on the diversity and characteristics of the attack code used by the different exploitation tools, worms, or bots in the wild. `Nemu` identifies only self-modifying shellcodes, so for each attack we can examine both the initial *shellcode*, as well as the decrypted *payload* that actually carries out the attack, and which is exposed only after successful execution of the shellcode on the emulator.

4.3.1 Shellcode Diversity

For each attack, we computed the MD5 hash of the initial shellcode as seen on the wire and plotted the number of unique shellcodes per port in Fig. 4 (purple bars). Comparing the purple and blue bars, we see that in some cases the number of unique shellcodes is quite smaller than the number of attacks. If truly polymorphic shellcode were used, we would expect the number of shellcodes to be equal to the number of attacks, since each instance of a polymorphic shellcode is different than any other instance. However, in most attacks the encryption scheme is very simple, and for the same malware family, the generated shellcodes usually have been encrypted using the same key and carry the same decryption routine. Besides code obfuscation, even such naively applied encryption is convenient for the avoidance of `NULL`, `CR`, `LF`, and depending on the exploit, other restricted bytes that should not be present in the attack vector, since this can be taken care of by the encryption engine [1].

The generated shellcodes though still look different because the encrypted body of different instances differs due to slight variations in the encrypted payload. Computing the MD5 hash of the decrypted payloads results in a number of unique payloads comparable to the number of unique shellcodes, as shown by the yellow bars in Fig. 4 when compared to the purple bars. Although the actual code of the payload used by a given malware may remain the same, variable fields like IP addresses, port numbers, URLs, and filenames result in different encrypted payloads. We discuss in detail the types of payload found and their characteristics in Section 4.3.3.

4.3.2 Decryption Routines

To gain a better understanding of whether the captured attacks are truly polymorphic or not, we analyzed further the decryption routines of the captured shellcodes. Decent polymorphic encoders generate shellcode with considerable variation in the structure of the decryption routine, and use different decryption keys (and sometimes algorithms), so that no sufficiently long common instruction sequence can be found among different shellcode instances. On the other hand, naive shellcode encoders use the same decryption routine in every instance. Even if the same decryption code and key is used, the encrypted body usually differs due to payload variations, as discussed in the previous section, so here we focus only on the variation found in the different decryption routines.

For each attack, we extracted the decryption code from the execution trace produced by `nemu`. The beginning of the decryption routine is identified by the seeding instruction of the GetPC code that stores the program counter in a memory location [13]. The end of the decryption code is identified by the branch instruction of the loop that iterates through the encrypted payload. In

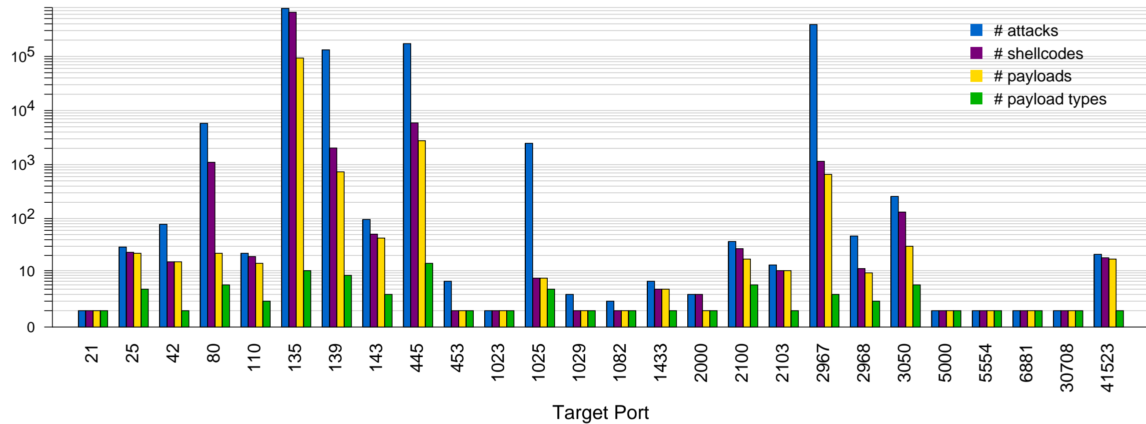

Figure 4: Number of attacks, unique shellcodes, unique decrypted payloads, and payload classes for different ports.

the execution trace of Fig. 5, this heuristic identifies the highlighted instructions as the decryption routine.

The different types of decryption routines are categorized based on the sequence of instruction opcodes in the decryption code, ignoring the actual operand values. For each inspected input, nemu maps the code into a random memory location, so memory offset operands will differ even for instances of the same decryptor. The decryption key, the length of the encrypted payload, and other parameters may also vary among different instances, resulting to different operand values. Routines with the identical instruction sequences but different register mappings are also considered the same.

After processing all captured attacks, the above process resulted in 41 unique decryption routines. This surprisingly small number of decryptors indicates that none of the malware variants that launched the attacks employs a sophisticated shellcode encoder. Despite the availability of quite advanced polymorphic shellcode encryption engines [4], none of the captured shellcodes seems to have been produced by such an engine. In contrast, most of the decryption routines are variations of simple and widely used encoders.

A larger number of shellcodes share the same decryption routine but use different decryption keys. We speculate that key variation is the result of the brute force way of operation of some encoders, which try different encryption keys until all the bad character constraints in the generated shellcode are satisfied, rather than an intentional attempt to obfuscate the shellcode.

Three of the decryptors match the code generated by the `call4_dword_xor`, `jmp_call_additive`, and `fnstenv_mov` encoders of the Metasploit Framework [1], while the decryptor shown in Fig. 5 is a variant of the decryptor used by the `countdown` encoder of the same toolkit. The GetPC code in 37 of the routines uses the `fstenv` instruction for retrieving the current value

```
 0 40000000 EB15                jmp 0x40000017
 1 40000017 E8E6FFFFFF        w call 0x40000002
 2 40000002 B98BE61341          mov ecx,0x4113e68b
 3 40000007 81F14DE61341        xor ecx,0x4113e64d
 4 4000000d 5E                r pop esi
 5 4000000e 807431FF85          xor byte [ecx+esi-0x1],0x85
 6 40000013 E2F9              S loop 0x4000000e
 7 4000000e 807431FF85          xor byte [ecx+esi-0x1],0x85
 8 40000013 E2F9              1 loop 0x4000000e
 9 4000000e 807431FF85          xor byte [ecx+esi-0x1],0x85
10 40000013 E2F9              2 loop 0x4000000e
   ...
```

Figure 5: The execution trace of a captured shellcode.

of the instruction pointer, while the rest 14 use the `call` instruction. The average length of the loop body code is 2.92 instructions (excluding the branch instruction)—the largest decryption loop uses ten instructions.

The decryption code with the largest loop body is a variant of the code used by the `alpha_mixed` encoder from Metasploit, which produces alphanumeric mixed-case shellcode, with some differences in the GetPC code (the decryption loops are identical). This type of shellcode was found in three of the attacks against port 3050, attempting to exploit an integer overflow vulnerability in the Borland Interbase 2007 database server.

The interesting aspect of these particular attacks is that the decrypted payload produced by the alphanumeric shellcode is again an instance of a self-decrypting shellcode, this time generated by yet another variant of the popular `countdown` encoder. That is, the initial payload was first encoded using a `countdown`-like encoder, and the resulting shellcode was then encoded using a `alpha_mixed`-like encoder. The overall decryption process of the shellcode is illustrated in Fig. 6. Although such layered encryption using multiple executable packers is commonly found in malware binaries, we are not aware of any previous report of in-the-wild attacks employing doubly encrypted shellcode.

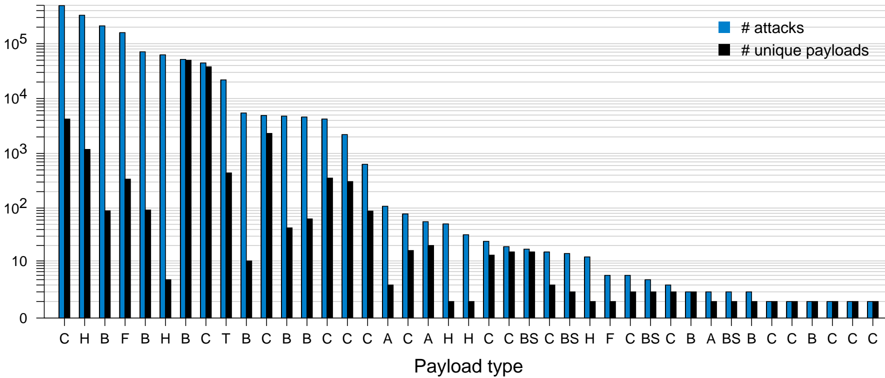

Figure 7: Number of attacks and unique payloads for the 41 payload types.

Payload Class	# Payload Types
ConnectExec	17
BindExec	9
HTTPExec	5
BindShell	4
AddUser	3
FTPExec	2
TFTPExec	1

Table 2: Payload classes.

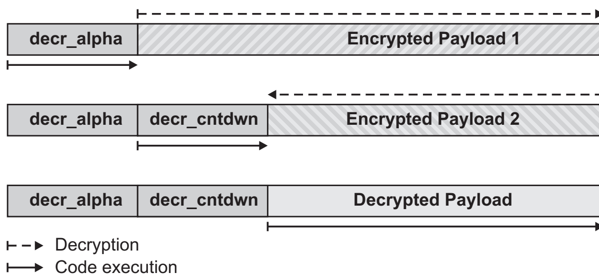

Figure 6: An illustration of the execution of the doubly encrypted shellcode found in three of the attacks.

4.3.3 Payload Categorization

The captured attacks per targeted port may come from one or more malware families, especially for ports with a high number of attacks. At the same time, the propagation mechanism of a single malware may include exploits for several different vulnerable services. Identifying the different types of payload used in the attacks can give us some insight about the diversity and functionality of the shellcode used by malware.

It is reasonable to expect that a given malware uses the same payload code in all exploitation attempts, e.g., for downloading and executing the malware binary after successful exploitation. Although a malware could choose randomly between different payloads or even use a metamorphic payload different in each attack, such second-level polymorphism is not typically seen in the wild. On the other hand, different malware may use exactly the same payload code, since in most cases the shellcode serves the same purpose, i.e., carrying out the delivery and execution of the malware binary to the victim host.

We have used a binary code clustering method to group the unique payloads with similar code from all captured attacks into corresponding payload types. As mentioned before, even exactly the same payload code may differ among different instances due to variable parameters. For example, a payload that connects back to the previous victim will contain a different IP address in each attack instance. Such differences can be manifested either as variations in instruction operands, or directly as different embedded data in the code.

To cluster the payloads, we first extract any obvious embedded strings using regular expressions, and disassemble the remaining code to derive a corresponding instruction sequence. We then group the payloads using agglomerative hierarchical clustering, with the relative edit distance over the compared instruction sequences as the distance metric. After experimenting with different thresholds and manually examining the resulting payload groups, we empirically chose a high distance criterion such that only almost identical instruction sequences are clustered together.

We also experimented with computing the edit distance over the sequence of instruction opcodes—excluding operands—instead of the complete instructions. However, due to the increased component reuse among payloads types, this approach tends to yield fewer groups that in some cases included different payload implementations. Sharing of identical code blocks is very common between different payload types due to the compartmentalized structure of modern shellcode [17].

We further analyzed each payload type to understand its behavior and intended purpose. The typical structure of Windows payloads consists of a series of steps to resolve the base address of `kernel32.dll`, potentially load other required DLLs, resolve the addresses of the API calls to be used, and finally carry out the intended task [17]. The type and sequence of library calls used by the payload provides a precise view of the payload functionality. We statically analyzed the code of each payload group, looking for patterns of known library call prologues, library function strings (used as arguments to `GetProcAddress`), library function hashes (used by

custom symbol resolution code), and shell commands, and classified each payload type according to its generic functionality.

Payload clustering and categorization resulted in 41 payload types, categorized in seven payload classes. We manually verified that only similarly implemented payloads are categorized in the same payload type. We can think of each payload type as a different implementation of the functionality corresponding to its payload class. The number of attacks and different unique payloads per payload type is shown in Fig. 7. The letter of each payload type in the *x* axis corresponds to its payload class, according to the class names listed in Table 2. We used a naming scheme similar to the one proposed by Borders et al. [5] based on the method of communication, the type of action performed, or both.

As shown in Fig. 7, for most payload types, the number of unique payloads is smaller than the number of attacks that used this type of payload. This means that exactly the same payload was used in more than one attack instance. Multiple attacks launched by the same malware running on the same host typically have identical payloads because even variable parameters, such as the IP address of the attacker, remain the same. Depending on the malware, the payload may fetch the malware binary from a predefined location, which also results in identical payloads even for attacks launched from different hosts.

On the other hand, some payload types, such as those that wait for a connection from the attacker (e.g., payloads of the *BindShell* and *BindExec* classes), may not have any variable fields at all. However, if such a payload is used by different malware families, then each malware may use it with slight modifications. For example, different malware families may bind the listening socket to a different port number, or choose a different file name for payloads that write the downloaded binary to a file before executing it. Going back to Fig. 4, the number of different payload types per attacked port is represented by the green bars. The diversity of the used payloads increases with the number of attacks for each port, indicating that highly attacked ports were attacked by several different malware families.

The most commonly used type of payload (the first pair of bars in Fig. 7), used by a little less than half of the captured attacks, is a typical "connect back, download, and execute" payload. As shown in Table 2, this payload class (*ConnectExec*) has the largest number of different implementations. Implementations may differ in many parts, including the code used to locate the base address of `kernel32.dll`, the routine and name hashing scheme for API call address resolution, (locating only `GetProcAddress` and using it for resolving the rest of the symbols is another common option), library initialization, process creation and termination, different libraries or library calls with similar functionality, as well as in the overall assembly code.

The five payloads of the *HTTPExec* class use the convenient `URLDownloadToFileA` function of `urlmon.dll` to download and execute a malicious file from a predefined URL. Other payloads first spawn a `cmd.exe` process, which is then used either for receiving commands from the attacker (*BindShell*), or for directly executing other programs as specified in the payload. For example, one of the two *FTPExec* payload types uses a command similar to the following as an argument to the `WinExec` function of `kernel32.dll`:

```
cmd /c echo open 208.111.5.228 2755 > i &
echo user 1 1 >> i &echo get 2k3.exe >> i &
echo quit >> i &ftp -n -s:i &2k3.exe&del i
```

while the *AddUser* payloads use a command like the following to create a user with administrative privileges:

```
cmd.exe /c net user Backupadmin corrie38 /ADD &&
net localgroup Administrators Backupadmin /ADD
```

`WinExec` is also used to directly execute programs without involving `cmd.exe`, such as `ftp.exe` and `tftp.exe`, in the second *FTPExec* and the *TFTPExec* payload types.

5 Related Work

Ma et al. [12] studied the variations found in shellcodes of the same exploit family, in probably the most closest work to ours. The analysis is based on attacks collected by active responders and honeypots using four well-known vulnerabilities. Our study is based on attacks against production systems that employ some form of self-modifying code, and our findings are in accordance with the results of that work. Goebel et al. [9] studied malware propagation in a university environment, focusing on the collection and analysis of the malware binaries and their post-infection activities. Yegneswaran et al. [22] studied the prevalence of scanning and intrusion attempts based on a large corpus of firewall logs.

Song et al. [19] studied the possibility of deriving a model for representing the general class of code that corresponds to all possible decryption routines, and concluded that it is infeasible.

Spector [5] is a shellcode analysis system that uses symbolic execution to extract the sequence of library calls, along with their arguments, made by the shellcode, as well as a low-level execution trace. We used a simpler technique based on static code analysis to extract the library calls used, which though was adequate since the decryption process had already been handled by `nemu`'s CPU emulator, and static analysis was performed only on the resulting decrypted payload.

6 Conclusion

In this paper, we presented a study of the polymorphic code injection attacks captured using network-level emulation in four deployments in research and education networks. We focused on the analysis of the different decryption routines and payload types used, as well as the overall attack activity and the targeted network services.

The attack activity observed in these deployments clearly shows that polymorphic attacks are extensively used in the wild, although polymorphism is mostly employed in its more naive form, using simple encryption schemes for the concealment of restricted payload bytes. Considering the wide availability of sophisticated polymorphic shellcode engines, this probably indicates that attackers are satisfied with the effectiveness of current shellcode, and they do not need to bother with more complex encryption schemes for evading existing network-level defenses. Another possible reason is the extensive code component reuse among different malware families in both decryption routines and payloads. Less skilled attackers probably rely on slight modifications of proof of concept code and existing malware, instead of implementing their own attack vectors.

However, attackers have also turned to the exploitation of less widely used services and third-party applications, while we observed attacks employing more sophisticated encryption schemes, such as doubly-encrypted shellcode. It is thus not unlikely that in the future the use of advanced polymorphic shellcode engines will be commonplace, as has already happened with executable packers, which are nowadays widely used by malware.

Acknowledgments

This work was supported in part by the project CyberScope, funded by the Greek General Secretariat for Research and Technology under contract number PENED 03ED440. Michalis Polychronakis and Evangelos Markatos are also with the University of Crete. Part of this work done while Kostas Anagnostakis was with FORTH-ICS.

References

[1] The metasploit project. `http://www.metasploit.com/`.

[2] Microsoft Security Bulletin MS08-067 – Critical, Oct. 2008. `http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx`.

[3] Calculating the Size of the Downadup Outbreak, Jan. 2009. `http://www.f-secure.com/weblog/archives/00001584.html`.

[4] P. Bania. TAPiON, 2005. `http://pb.specialised.info/all/tapion/`.

[5] K. Borders, A. Prakash, and M. Zielinski. Spector: Automatically analyzing shell code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 501–514, 2007.

[6] 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 12th ACM conference on Computer and communications security (CCS)*, pages 235–248, 2005.

[7] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(61), Aug. 2003.

[8] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *Proceedings of the* 15th *USENIX Security Symposium*, 2006.

[9] J. Goebel, T. Holz, and C. Willems. Measurement and analysis of autonomous spreading malware in a university environment. In *Proceedings of the 4th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, pages 109–128, 2007.

[10] K2. ADMmutate, 2001. `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`.

[11] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[12] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *Proceedings of the 6th Internet Measurement Conference (IMC)*, pages 53–64, 2006.

[13] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2006.

[14] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2007.

[15] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*, pages 1–16, 2008.

[16] Rix. Writing ia32 alphanumeric shellcodes. *Phrack*, 11(57), Aug. 2001.

[17] Skape. Understanding windows shellcode, 2003. `http://www.hick.org/code/skape/papers/win32-shellcode.pdf`.

[18] Skape. Implementing a custom x86 encoder. *Uninformed*, 5, Sept. 2006.

[19] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, pages 541–551, 2007.

[20] P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[21] B.-J. Wever. Alpha 2, 2004. `http://www.edup.tudelft.nl/~bjwever/src/alpha2.c`.

[22] V. Yegneswaran, P. Barford, and J. Ullrich. Internet intrusions: global characteristics and prevalence. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2003.

[23] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Lyer. Analyzing network traffic to detect self-decrypting exploit code. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 4–12, 2007.