Automatic Hooking for Forensic Analysis of Document-based Code Injection Attacks

Techniques and Empirical Analyses

Kevin Z. Snow   Fabian Monrose

University of North Carolina at Chapel Hill

{kzsnow, fabian}@cs.unc.edu

Abstract

Document-based code injection attacks, where-in malicious code (coined *shellcode*) is embedded in a document, have quickly replaced network-service based exploits as the preferred method of attack. In this paper, we present a new technique to aid in forensic and diagnostic analysis of malicious documents detected using *dynamic code analysis* techniques — namely, automated API call *hooking* and *simulation*. Our approach provides an API call trace of a shellcode in a few milliseconds. We also present the results of a large empirical analysis of malicious PDFs collected in the wild over the last few years. To our surprise, we found that 90% of shellcode embedded in documents make *no* use of machine-code level polymorphism, in stark contrast to prior shellcode studies based on samples collected from network-service level attacks. We also observed a heavy-tailed distribution of API call sequences used by contemporary shellcode.

Categories and Subject Descriptors K.6.5 [*Security and Protection*]: Invasive Software

General Terms Forensics, Security, Measurements

Keywords Shellcode, Malicious Code, Hooking

1. Introduction

Dynamic code analysis has been proposed as an effective means for detecting the presence of shellcode in an arbitrary buffer of data [2, 10, 11, 14]. The general idea is to treat a buffer of data as code and simply execute its contents in a controlled environment, starting at each offset in the buffer (called an *execution chain*). The basic premise is that if the buffer contains only benign data, then the execution would likely terminate with a fault. At that point, the execution state is reset and an execution attempt is made from the next position in the buffer, since a shellcode's location, if present, is not known a-priori. If, on the other hand, a particular execution chain within a buffer does in-fact represent shellcode, the instruction sequence executed by that shellcode will reliably generate tell-tale signs of malicious code. One of these tell-tale signs, for example, is accesses to specific fields within the Thread Environment Block (TEB) and Process Environment Block (PEB) to parse Windows API call address locations. Several profiles of these code execution sequences (called *runtime heuristics*) have been developed in the last few years [9–11, 13]. In this paper, we use a state-of-the-art dynamic code analysis framework called `ShellOS` [14] to find and analyze shellcode in document-based code injection attacks.

While dynamic code analysis has primarily been used to detect shellcode at the network packet level [11, 12], recent work has explored the application to document-based code injection attacks [14, 17] (also called malcode-*bearing* documents [7]). However, scanning documents for shellcode using dynamic code analysis creates an additional layer of complexity in that *most* document-based code injection attacks store shellcode under one or more layers of compression or encoding. These malicious documents often rely on the document reader itself to decompress the shellcode through its document parsing routines, or rely on the document reader to provide a means to dynamically decompress or decode shellcode through built-in scripting facilities (e.g. ActionScript, JavaScript, etc.).

Two general approaches have been proposed for automatically decompressing and decoding buffers in a document that may contain shellcode – *document parsing* [3, 6, 17], and *document snapshotting* [1, 14]). Document parsing attempts to manually implement lightweight document reader parsing routines and provide an environment for scripting languages to execute within. The primary difficulty with this approach is simply keeping up-to-date with all the features and nuances of a particular file format. Parsing does, however, have the advantage of precision control and greater

5th European Workshop on System Security April 10, 2012, Bern, Switzerland.
Copyright © 2012 ACM 978-1-4503-1165-6...$10.00

introspection of the decompression and/or decode routines. Document snapshotting, on the other hand, simply opens a document in its reader application and takes a snapshot of application memory after it has loaded. We use a document snapshotting approach.

Beyond merely detecting shellcode and tracing the instructions executed using dynamic code analysis, the sequence of Windows API calls executed by shellcode, along with their parameters, are particularly useful to a network operator. A network operator could, for example, blacklist URLs found in shellcode, compare those URLs with network traffic to determine if a machine was actually compromised, or provide information to their forensic team such as the location of malicious binaries on the file system. While the ShellOS framework provides state-of-the-art dynamic code analysis for shellcode detection, the forensic and diagnostic capabilities for tracing Windows API calls are limited to a small set of handcrafted functions. Indeed, once deployed in the wild, we routinely encountered new API calls used by shellcode. We present an approach for analyzing previously unseen API sequences, and show that this extension significantly improved our diagnostic capabilities.

2. API Call Diagnostics

Over the past several years, dynamic code analysis has proven to be an effective weapon in the fight against code injection attacks. However, allowing shellcode (once detected) to continue its execution and properly handle its sequence of Windows API calls is a difficult task for dynamic code analyzers, mainly due to the fact that the code runs in a simulated Windows environment. The libemu dynamic code analyzer, for example, loads a fixed set of DLLs to their default load addresses in memory and maintains a static list of Windows API call addresses that it simulates when the API address is loaded from within the instruction emulator. ShellOS takes a similar approach to libemu in that it simulates a fixed set API calls, but differs in that API calls are hooked by page-level memory *traps* (see the discussion in [14] or the notion of stealth breakpoints in [18] for a detailed overview), rather than comparing the PC to the set of simulated API call addresses at each single-stepped instruction. However, the fact remains that dynamic code analysis tools cannot easily support the simulation of the myriad of Windows API calls available, and additional third-party DLLs are often loaded by the application being analyzed. It is not uncommon, for example, that over 30,000 DLL functions are present in memory at any time.

As an alternative to dynamic code analysis, we note that that simply executing the document reader application (given the malicious document) inside Windows, all the while tracing API calls, may be the most straightforward approach. In fact, this is exactly what tools like CWSandbox [19] do. Instead of detecting shellcode, these tools are based on detecting anomalies in API, system, and network traces. Unfortunately, many shellcode have adapted to evade API hooking techniques (called *in-line code overwriting*) used by tools like CWSandbox by simply jumping a few bytes into API calls. Furthermore, the resulting traces make it exceedingly difficult to separate application-generated events from shellcode-generated events.

Fratantonio et al. [4] offer an alternative called Shellzer that focuses solely on recovering the Windows API call sequence of a given shellcode that has already been discovered and extracted by other means (e.g. using wepawet [3]). The approach they take is to compile the given shellcode into a standard Windows binary, then execute it in debug mode and single-step instructions until the PC jumps to an address in DLL-space. The API call is logged if the address is found in an external configuration file. The advantage here over typical dynamic code analysis is that Shellzer executes code in a *real* Windows OS environment allowing actual API calls and their associated kernel-level calls to complete. However, this comes at a price — obviously the analysis *must* be run in Windows, the instruction single-stepping results in sub-par performance ($\sim$15 second average analysis time), and well-known malware anti-debugging tricks can be used to evade detection and possibly compromise the OS.

To alleviate some of these problems, we developed a technique for automatically hooking all methods exported by DLLs, without the use of external DLL configuration files. We also provide a method for automatically generating code to simulate each API call, where possible.

2.1 Automated Hooking

To tackle the problem of automatically hooking the tens of thousands of exported DLL functions found in a typical Windows application, we leverage ShellOS memory *traps* along with the application snapshot that accompanies an analysis with ShellOS. As described in detail in [14], ShellOS initializes its execution environment by exactly reconstructing the virtual memory layout and content of an actual Windows application through an application snapshot. These application snapshots, called *minidumps*, are created through the Windows *DbgHelp* API and may be configured to extract the entire process state, including the code segments of DLLs[1].

The *minidump* provides ShellOS with the list of memory regions that correspond to DLLs. We use this information to set memory *traps* (a hardware supported page-level mechanism) on the entirety of each DLL region. These *traps* guarantee that any execution transfer to DLL-space is immediately caught by ShellOS, without any requirement of single-stepping each instruction to check the PC address. Once caught, we parse the export tables of each DLL loaded by the application snapshot to try and match the address that triggered the *trap* to one of the tens of thousands of DLL

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/ms679309(v=vs.85).aspx

functions. If the address does not match an exact API call address, we simply note the relation to the nearest API call entry point found $\leq$ the trapped address in the format: *function + offset*. In this way, we discover either the exact function called, or the offset into a specific function that was called. In cases where we already have a handler to simulate the API call, it may be called.

2.2 Automated Simulation

Automated hooking alone can only reveal the last function that malicious code tried to call before our diagnostic analysis failed. This certainly helps with rapidly prototyping new API calls. However, we want to automatically support the simulation of new API calls to prevent, where possible, constantly updating `ShellOS` manually.

One approach is to skip simulation altogether, for example, by simply allowing the API call code to execute as it normally would. Since `ShellOS` already maps the full process snapshot into memory, all the necessary DLL code is already present. Unfortunately, Windows API calls typically make use of kernel-level system calls. To support this without analyzing the shellcode in a real Windows environment would require simulating all of the Windows system calls – a non-trivial task.

Instead we generate a best-effort automated simulation of an API call on-the-fly. The idea is to simply return what is considered a valid result to the caller[2]. Since shellcode does not often make use of extensive error checking, this technique enables analysis of shellcode using API calls not known a-priori to run to completion. The main complication with this approach, however, is the assembly-level function calling convention used by Windows API calls (the `__stdcall` convention). The convention declares that the API call, not the caller, must clean up the function parameters pushed on the stack by the caller. Therefore, we cannot simply return to the calling instruction, which would result in a corrupted stack. Instead, we need to determine the size of the parameters pushed onto the stack for that specific function call. Unfortunately, this function parameter information is not readily available in any form within an application snapshot[3]. However, the original DLL code for the function is accessible within the application snapshot, and this code must clean up the stack before returning. We leverage this by disassembling instructions, starting at the trapped address, until we encounter a `ret` instruction. The `ret` instruction optionally takes a 2-byte source operand that specifies the number of bytes to pop off the stack. We use this information to automatically adjust the stack, allowing shellcode to continue its execution. Obviously, the automated simulation would fail to work in cases where shellcode actually requires an intelligent result (e.g. *LoadLibrary* must return a valid DLL load address). An astute attacker could therefore thwart diagnostic analysis by requiring specific results from one of these automatically simulated API calls. Our automated API hooking, however, would at least identify the offending API call.

For cases where shellcode attempts to bypass in-line code overwriting-based function hooking by jumping a few bytes into an API call, we simply adjust the stack accordingly (as also noted by [4]), and either call the manually implemented function handler (if it exists), or do on-the-fly automated simulation as described above.

[2] Windows API functions place their return value in the `eax` register, and in most cases indicate a *success* with a value ≥ 1

[3] Function parameter information could be obtained from external sources, such as library definitions or debug symbols, but these may be impossible to obtain for proprietary third-party DLLs

3. Malicious Document Analysis

In what follows, we perform an indepth forensic analysis of the document based code injection attacks. We use our dynamic code analysis capabilities to exactly pinpoint no-op sleds and shellcode for analysis, and then examine the structure of Windows API call sequences, as well as the overall behavior of the code injected into the document.

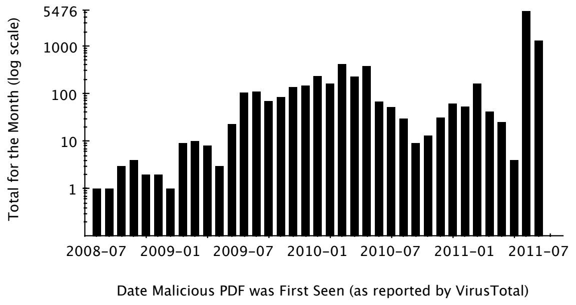
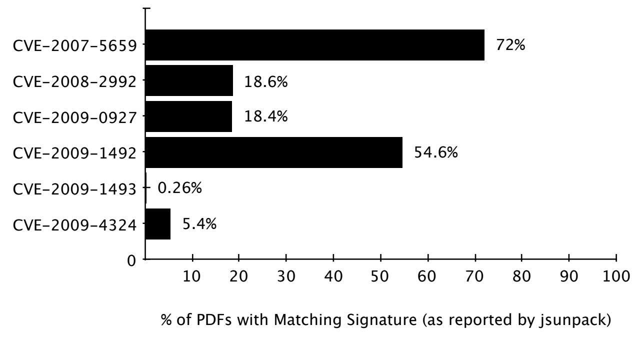


Figure 1. Timeline of PDFs used in our analysis.

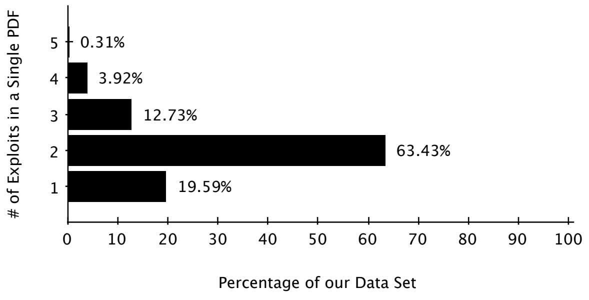
Our forensic analysis is based on nearly 10,000 distinct PDF documents collected from the wild and provided to us through several sources. Many of these were submitted directly to a submission server (running the `ShellOS` framework) available on our campus. All the documents used in this analysis had previously been labeled as *malicious*, so our subsequent analysis focuses on what we can learn about the malicious code, rather than whether the document is malicious or not; the detection capabilities of `ShellOS` is already covered in [14].

To get a sense of how varied these documents were (e.g., whether they come from different campaigns, use different exploits, use different obfuscation techniques, etc.), we performed a preliminary analysis using `jsunpack` [5] and `VirusTotal`[4]. Figure 1 shows that the set of PDFs spans from 2008, shortly after the first emergence of malicious PDF documents in 2007, up to July of 2011, with about half of the data set from more recent months in 2011. Only 16 of these documents were unknown to `VirusTotal` when our queries were submitted in January of 2012.

[4] `http://www.virustotal.com`



(a) Vulnerabilities



(b) No. of Exploits

Figure 2. Results from `jsunpack` showing (a) known vulnerabilities and (b) exploits per PDF

Figure 2(a) shows the Common Vulnerabilities and Exposure (CVE) identifiers, as reported by `jsunpack`. The CVEs reported are, of course, only for those documents that `jsunpack` could successfully unpack and match signatures to the unpacked Javascript. The percentages here do not sum to 100% because most documents contain more than one exploit. Of the successfully labelled documents, 72% of them contain the original exploit that fueled the rise of malicious PDFs in 2007 — namely, the *collab.collectEmail* exploit (CVE-2007-5659). As can be seen in Figure 2(b), most of the documents contained more than one exploit, with the second most popular exploit, *getAnnots* (CVE-2009-1492), appearing in 54.6% of the documents.

3.1 Results

Shellcode Polymorphism Polymorphism has long been used to uniquely obfuscate each instance of a shellcode to evade detection by anti-virus signatures [15, 16]. A polymorphic shellcode typically contains a few *decoder* instructions, followed by the encoded portion of the shellcode. The decoder code typically loops over the encoded portion, `xor`'ing it with a unique key, then jumps to the decoded shellcode. A typical decoder loop might like:

```
——————————— begin snippet ———————————
call 0xffffffff1      ; GetPC setup
pop ebx               ; GetPC in ebx register
xor ecx,ecx           ; Clear counter register
mov cx,0x180          ; Loop count = payload size
...
xor byte [ebx],0xef   ; Xor key with a payload byte
inc ebx               ; Move to the next byte
loop 0xfffffffc       ; counter--, repeat until 0
...
xor byte [ebx],0xef   ; Decode next position
inc ebx               ;
loop 0xfffffffc       ;
...                   ; And so on...
——————————— end snippet ———————————
```

Our approach to analyzing polymorphism is to trace the execution of the first n instructions in each shellcode (we use $n = 50$ in our evaluation). In these n instructions, we can observe either a decode loop, or the immediate execution of non-polymorphic shellcode. ShellOS detects shellcode by executing from each position in a buffer, then triggering on a heuristic (such as the `PEB` heuristic [13]). However, since shellcode is often prepended by a no-op sled, tracing the first n instructions would usually only include execution of those sled instructions. Therefore, to isolate the no-op sled and actual shellcode portions of injected code, we execute each detected shellcode several times. The first execution simply detects the presence of shellcode and indicates the buffer offset of both the execution start position (e.g., most likely the start of the no-op sled) and the offset of the instruction where the heuristic was triggered (e.g., at some location inside the shellcode itself). We then try executing the buffer multiple times, starting at the offset the heuristic was originally triggered at and moving backwards until the heuristic successfully triggers again (of course, resetting the state after each execution). This new offset indicates the first instruction required by the shellcode to properly function, and we begin our n instruction trace from here.

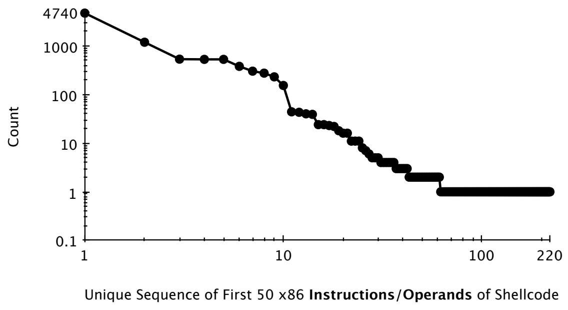


Figure 3. Unique sequences observed. 93% of shellcode use 1 of the top 10 unique starting sequences observed.

Figure 3 shows the number of shellcode found in each of the 220 unique starting sequences traced. Uniqueness, in this case is rather strict, and is determined by exactly matching instruction sequences (including opcodes). Notice the heavy-tailed distribution. Upon examining the actual instruction sequences in the tail, we found that the vast majority of these were indeed the same instruction sequence, but with varying opcode values, which is indicative of polymorphism. After re-binning the unique sequences by ignoring the opcode values, the distribution remains similar to that shown in Figure 3, but with only 108 unique starting sequences.

To our surprise, however, 90% of shellcode we analyzed were completely non-polymorphic. This is in stark contrast to prior empirical studies of shellcode [8, 12, 20]. One plausible explanation of this difference may be that prior studies examined shellcode-on-the-wire (e.g. network service-level exploits). Network-level exploits typically operate in plain-view of intrusion detection systems and therefore require obfuscation of the shellcode itself. Document-based exploits, such as those in our data set, have the benefit of using the document format itself (e.g. object compression) to obfuscate the shellcode, or the ability to pack the shellcode at the Javascript-level rather than machine code-level.

The 10% of shellcode that are actually polymorphic represent most of the heavy tail in Figure 3. Of the shellcode in this set, 11% used the `fstenv` GetPC instruction. The remaining 89% used `call` as their GetPC instruction. Of the non-polymorphic sequences, 99.6% begin by looking up the address of the `TEB` with no attempt to obfuscate their actions. Only 7 shellcode try to be evasive in their `TEB` lookup; they first push the `TEB` offset to the stack, then pop it into a register via: `push byte 0x30; pop ecx; mov eax,fs:[ecx]`.

Shellcode API Calls. To test the effectiveness of our automatic API call hooking and simulation, we allowed each shellcode in our data set to continue executing in `ShellOS`. The average analysis time, per shellcode API sequence traced, is ~ 2 milliseconds. Overall, the automatic hooking identified a number of API calls that were originally unhandled. In those cases the shellcode would crash. The new diagnostic extensions now enable the shellcode to complete its sequence of API calls, for example: *LoadLibraryA* → *GetProcAddress* → *URLDownloadToFile* → **[FreeLibrary+0]** → *WinExec* → *ExitProcess*. In this example, *FreeLibrary* is an API call that the `ShellOS` framework had no prior knowledge of how to handle. The automatic API hooking discovered the function name and that the function was directly called by shellcode, hence the $+0$ offset. Next, the automatic simulation disassembled the API code to find a `ret`, adjusted the stack appropriately, and set a valid return value. Our new API hooking techniques also identified a number of shellcode that attempt to bypass function hooks by jumping a few bytes into the API entry point. We found that the shellcode that make use of this technique usually only apply it to a small subset of their API calls. We observed hook bypassing for the following functions: *VirtualProtect*, *CreateFileA*, *LoadLibraryA*, and *WinExec*. In the following API call sequence, we automatically identified and handled hook bypassing in 2 API calls: *GetFileSize*[5] → *GetTickCount* → *ReadFile* → *GetTickCount* → *GlobalAlloc* → *GetTempPathA* → *SetCurrentDirectoryA* → **[CreateFileA+5]** → *GlobalAlloc* → *ReadFile* → *WriteFile* → *CloseHandle* → **[WinExec+5]** → *ExitProcess*. In this case, the stacks were automatically adjusted to account for the $+5$ jump into the *CreateFileA* and *WinExec* API calls. After the stack adjustment, the API calls are handled as usual.

Typically, an exploit will crash or silently terminate an exploited application. However, we observed more sophisticated shellcode that makes an effort to mask the fact that an exploit has occurred on the end-user's machine. Several API call sequences first load a secondary payload from the original document: *GetFileSize* → *VirtualAlloc* → *GetTickCount* → *ReadFile*. Then, assembly-level code decodes the payload (typically `xor`-based), and transfers control to the second payload, which goes through another round of decoding itself. The secondary payload then drops two files extracted from the original document to disk – an executable and a PDF: *GetTempPathA* → *GetTempFileNameA* → *CreateFileA* → *[LocalAlloc+0]* → *WriteFile* → *CloseHandle* → *WinExec* → *CreateFileA* → *WriteFile* → *CloseHandle* → *CloseHandle* → *[GetModuleFileNameA+0]* → *WinExec* → *ExitProcess*. The malicious executable is launched in the background, while the benign PDF is launched in the foreground via '`cmd.exe /c start`'.

Overall, we found over 50 other unique sequences of API calls in our data set. Due to space constraints, Table 1 only shows the full API call sequences for the most frequent shellcode. As with our observations of the first n assembly-level shellcode instructions, the call sequences have a heavy-tailed distribution.

4. Conclusions

As code-injection attacks have solidified their position in the new frontier of rich-media client applications, such as document readers and web browsers, diagnosing the initial code injected by an exploit becomes more critical to understanding what transpired after an exploit. The automatic hooking and simulation techniques presented in this paper greatly aid in the analysis of both known and unknown shellcode. Our empirical analysis shows that the properties of network-service level exploits and their associated shellcode, such as polymorphism, do not necessarily translate to document-based code-injection attacks. Additionally, our API call sequence analysis has revealed diversity in todays shellcode, likely due to the multitude of exploit kits available — further underscoring the usefulness of the improved shellcode diagnostics presented.

[5] A custom handler is required for *GetFileSize* and *ReadFile*. Our handler reads the original document file to provide the correct file size and contents to the shellcode.

Table 1. Shellcode API Call Sequences

Id	Count	LoadLibraryA	GetProcAddress	GetTempPathA	GetSystemDirectory	URLDownload ToFile	URLDownload ToCacheFile	CreateProcessA	ShellExecuteA	WinExec	FreeLibrary	DeleteFileA	ExitThread	TerminateProcess	TerminateThread	SetUnhandled ExceptionFilter	ExitProcess
1	5566	①		②		③				④							⑤
2	1008	②		①		③				④			⑤				
3	484	①			②	④				⑤		③	⑥				
4	397	②⑦	③⑧	①⑥		④⑨				⑤⑩							
5	317	③	②	①		④				⑤			⑥				
6	179	①				②				③							④
7	90	①					②	③							④		
8	75	①		②		③⑤				④⑥							⑦
9	46	③	②	①		④				⑤							
10	36	⑤	①-④⑥		⑦	⑧				⑨			⑩				
11	27	①		②		③⑤⑦				④⑥⑧							⑨
12	25	①②		③		④			⑤								⑥
13	20	①	②	③		④⑥⑦⑨⑩+				⑤⑧+							
14	12	①		②		③⑤⑦⑨				④⑥⑧⑩							11
15	10	①	②	③		④				⑤							

Acknowledgments

Thanks to Srinivas Krishnan, Teryl Taylor, and the anonymous reviewers for their insightful comments and suggestions. We also thank Bil Hayes for his help in setting up the networking infrastructure that supports our campus deployment of `ShellOS`. This work is supported by the National Science Foundation under award CNS-0915364 and by a Google Research Award.

References

[1] C. Andrianakis, P. Seymer, and A. Stavrou. Scalable web object inspection and malfease collection. In *USENIX Hot Topics in Security*, pages 1–16, 2010.

[2] P. Baecher and M. Koetter. Libemu - x86 shellcode emulation library. See `http://libemu.carnivore.it/`, 2007.

[3] M. Cova, C. Kruegel, and V. Giovanni. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Intl. World Wide Web conf.*, pages 281–290, 2010.

[4] Y. Fratantonio, C. Kruegel, and G. Vigna. Shellzer: a tool for the dynamic analysis of malicious shellcode. In *RAID*, Sept. .

[5] B. Hartstein. Javascript unpacker (jsunpack-n). See `http://jsunpack.jeek.org/`.

[6] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing pdf documents. In *ACSAC*, pages 373–382, 2011.

[7] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis. A study of malcode-bearing documents. In *DIMVA*, pages 231–250, 2007.

[8] U. Payer, P. Teufl, S. Kraxberger, and M. Lamberger. Massive data mining for polymorphic code detection. In *ACNS*, volume 3685 of *LNCS*, pages 448–453. Springer, 2005.

[9] M. Polychronakis and A. D. Keromytis. ROP payload detection using speculative code execution. In *MALWARE*, 2011.

[10] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level Polymorphic Shellcode Detection using Emulation. In *DIMVA*, pages 54–73, 2006.

[11] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based Detection of Non-self-contained Polymorphic Shellcode. In *RAID*, 2007.

[12] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *USENIX LEET Workshop*, 2009.

[13] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *ACSAC*, pages 287–296, 2010.

[14] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos. Shellos: enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, 2011.

[15] Y. Song, M. Locasto, A. Stavrou, A. Keromytis, and S. Stolfo. On the infeasibility of modeling polymorphic shellcode. *Machine Learning*, 81:179–205, 2010.

[16] M. Talbi, M. Mejri, and A. Bouhoula. Specification and evaluation of polymorphic shellcode properties using a new temporal logic. *Journal in Computer Virology*, 11 2008.

[17] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Fourth European Workshop on System Security*, 2011.

[18] A. Vasudevan and R. Yerraballi. Stealth breakpoints. In *ACSAC*, pages 381–392, 2005.

[19] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5:32–39, 2007.

[20] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Iyer. Analyzing Network Traffic to Detect Self-Decrypting Exploit Code. In *ASIACCS*, 2007.