Department of Computer Science
George Mason University Technical Reports

4400 University Drive MS#4A5
Fairfax, VA 22030-4444 USA
http://cs.gmu.edu/ 703-993-1530

Document Content Layout Based Exploit Protections

Charles Smutz
csmutz@gmu.edu

Angelos Stavrou
astavrou@gmu.edu

Technical Report GMU-CS-TR-2014-7

Abstract

Malware laden documents are a common exploit vector, especially in targeted attacks. Most current approaches seek to detect the malicious attributes of documents whether through signature matching, dynamic analysis, or machine learning. We take a different approach: we perform transformations on documents that render exploits inoperable while maintaining the visual interpretation of the document intact. Our exploit mitigation techniques are similar in effect to address space layout randomization, but we implement them through permutations to the document layout.

Using document content layout randomization, we randomize the data block layout of Microsoft OLE files in a manner similar to the inverse of a file system defragmention tool. This relocates malicious payloads in both the original document file and in the memory of the reader program. We demonstrate that our approach indeed subdues in the wild exploits by manual validation of both Office 2003 and Office 2007 malicious documents while the transformed documents continue to render benign content properly. The document transformation can be performed offline and requires only a single document scan while the user-perceived delay when opening the transformed document is negligible. We also show that it is possible to thwart malicious heap sprays by injecting benign content in documents. This approach, however, comes at an expense of computational and memory resources.

1 Introduction

Leveraging documents as a vehicle for exploitation remains a very popular form of malware propagation that is sometimes more effective than mere drive-by downloads [20]. Malicious documents are documents that have been modified to contain malware, but are engineered to pose as benign documents with useful content. For this reason, they are often called trojan documents or maldocs. Malware-bearing documents typically exploit a vulnerability in the document reader program, but they can also be crafted to carry exploits in the form of an embedded object such as a media file. Another class of malicious documents are used as a stepping stone and while they do not take advantage of a software flaw, they rely on the user to execute a macro or even a portable executable. Often, as part of the delivery vector, social engineering is used to enhance likelihood that victims will execute the malware contained within the document.

For years, client side exploits, including attacks against document readers, have become more prevalent [13]. Despite efforts at improving software security, new vulnerabilities in document readers are still present today. For instance, there were 17 CVEs issued for Microsoft Office in 2013, many of which were severe vulnerabilities facilitating arbitrary code execution. Document file types, such as PDF and Microsoft Office documents, are consistently among the top file types submitted to VirusTotal [3], which implies current widespread concern over the role of documents as malware carriers.

Targeted attacks rely heavily on malicious documents, especially in the case of spear phishing where they serve as the ruse [39, 15]. A recent study of targeted attacks against non-governmental organizations [8] showed that most attacks occur through malicious documents attached to emails highly customized to the recipient. Since 2010, when Adobe PDF exploits began to taper off in popularity, Microsoft Office has been the most frequently abused. The impetus to stop Trojan documents is elevated due to their pervasive use in espionage campaigns.

Numerous approaches have been proposed to detect malicious documents. Signature matching, dynamic analysis, and machine learning based approaches have been proposed and are used widely in practice. Despite these many approaches, malware authors continue to evade detection and exploit computers successfully. Operating system based mitigations such as address space layout randomization (ASLR) and data execution pre-

vention (DEP) have been implemented which seek to mitigate many classes of memory misuse based exploits. However, these protections are commonly circumvented and exploitation is still possible [34]. Despite extensive research and significant investments in protective technology, document exploits continue to remain a viable and popular vector for attack.

Our primary contribution is to design a defense that is fairly simple but effective. Indeed, we demonstrate that modifications to documents between creation and viewing can hinder misuse of document content, adding additional exploit protection, while leaving them semantically equivalent with very little end-user impact. The proposed approach is inspired by operating system based exploit protections. Instead of seeking to detect the exploits in documents, we seek to defeat exploits by making the location of malicious content unpredictable. While address space layout randomization is performed by the operating system, we induce exploit protections in the reader program through modifications in the input data. We modify documents at the file format level, resulting in a transformed document, which will render the same as the original, but in which malicious content is scrambled and inoperative. The transformed document is used in place of the original document. The document transformation should occur between the document creation and document open. Network gateways, such as web proxies or mail servers, or network clients such as web browsers or mail clients are ideal places to utilize our mechanisms. Hence, deployment of our technique requires no modification to the vulnerable document reader or client operating system.

The most successful technique is document content layout randomization (DCLR) where we rearrange the the layout of blocks in a document file, without modifying the extracted data streams. We show that it is possible to relocate malicious content in both the document file and document reader memory. In the face of ASLR circumvention techniques, such as heap sprays and egg hunts, content randomization provides an additional useful constraint on malicious payload size. In addition to DCLR, we demonstrate that we can utilize document modifications to influence memory location through memory consumption. For example, we can spray the heap with inert content. However, the memory consumption approach requires a large amount of CPU and RAM as well as modifications to user discernible document representation.

We demonstrate the strength of our approach primarily through equations showing the likelihood of determining the position of a relocated malicious payloads. We follow the pattern of other studies of probabilistic exploit defenses, such as ASLR, by using entropy (in bits) as a measure of difficulty of overcoming the location obfuscation. However, we also perform manual validation on a small number of maldocs to confirm that our methods work on current malware samples. Specifically, we will provide examples of DCLR breaking exploits by randomizing content in the document file of Office 2003 files and randomizing content in process memory while rendering Office 2007 files. The malicious documents used in this study were taken from VirusTotal [3] including documents used in apparent targeted attacks against industry [16] and non-governmental organizations [8]. Furthermore, we measured the performance of document transformation and found it to be comparable to an anti-virus scan. The performance impact of opening a transformed document is too small to be measurable. We also validated that the transformed document rendered the same as the original document.

DCLR is likely to be effective with a relatively low amount of entropy because it is difficult to retry document based attacks. Since DCLR can operate on small block sizes, it does have the potential to combine with ASLR to increase the difficulty of locating malicious data. DCLR is limited to foiling exploits that attempt to access exploit material in a manner inconsistent with the way the document reader accesses document data. Potential issues with DCLR include breaking intrusion detection system signatures and cryptographic signatures applied to the raw document file.

2 Related Work

Detection of malicious documents using signature matching or byte level analysis have long been studied and used in practice despite recognized weaknesses in detecting new or polymorphic samples [23, 31, 35, 33, 25]. The very obfuscation methods used to evade signature matching can be detected using probable plaintext attacks [43], but this approach is limited to situations where weak cryptography is used.

Dynamic analysis of documents can provide additional detection power, but it comes with computational cost, difficulty of implementation, and ambiguity in discerning malware [38]. Applying machine learning to various features extracted from documents, such as structural properties, has been shown to be effective but has also been challenged by mimicry attacks and adversarial learning [11, 21, 32, 40, 24]. We differ from most document centric defenses. Instead of seeking to detect the malicious content, we modify documents to foil exploit progression.

Our work builds upon years of research in probabilistic exploit mitigations typically implemented in the operating system. Address space layout randomization (ASLR) [37] is adopted widely. It is effective in defeating many classes of exploits, but is circumvented through limitations in implementation [30], use of heap sprays [41, 7], or data leakage [28]. As return oriented programming and similar techniques [42, 29] have become popular, mechanisms to relocate or otherwise mitigate code (gadget) reuse have been proposed [26, 44, 17].

ALSR incurs little run time overhead because it relies on virtual memory techniques where address translation and relocation are already performed.

Code level approaches such as instruction set randomization have also been proposed but are computationally prohibitive [18]. Data space randomization enciphers program data with random keys, but this method is not feasible in practice due to deployment difficulty and computational expense [6]. Other policy enforcing techniques, such as executable space protections (DEP or W⊕X), are used widely. Despite the many proposed exploit mitigations, exploits are still practical on modern systems [34].

3 Our Approach

Existing exploit protections that leverage random deviations, such as ASLR, are used widely. The intuition behind this class of protection techniques is to increase the difficulty of exploitation by making memory locations unpredictable. Moreover, often these protections are implemented in the vulnerable program or the execution environment of the vulnerable program and do not alter the functionality or compatibility of the target program or system.

Inspired by the simplicity and generality of ASLR-like techniques, we seek to obtain similar exploit mitigation outcomes through transformations to input data. The attributes of a document can often directly and predictably influence various run time attributes of the opening application. The contents the memory of a reader program often are necessarily influenced by the file which it opens. We demonstrate how documents can be modified to introduce variability in the programs which open them, making exploitation more difficult. We show that we can create these exploit breaking relocations in both the raw document file and in the memory of the document reader.

Like existing probabilistic memory protections, our aim is to reduce the predictability of the location of exploit content. We also show that it is possible to fragment malicious payloads which places additional constraints on the size of malicious payload. While inefficient, we show that through simple memory consumption we can displace or dilute malicious content also. The end result of our method is a transformed document where legitimate document content remains intact, but malicious content is relocated or scrambled. The transformed document is used in place of the original, potentially malicious document.

Our proposed transformations are envisioned to operate on documents and other file types during transfer between the malicious source and the intended victim. They could be employed in network gateways such as email relays or web proxies where modification is already supported. In practice, filtering based on blacklists and anti-virus scanning is already common at these points. The modifications to the document could also be implemented on client. For example, the web browser could employ the mechanisms presented here at download time, similar to other defenses such as blacklists and anti-virus.

One of the goals of this work is to demonstrate how files can be re-structured so that exploit mitigations are introduced into the reader program. We will focus on describing methods to implement DCLR, which can defeat exploits in many ways. We will also provide examples, using samples gathered from contemporary attacks, demonstrating how these document modifications mitigate practical exploits. However, many forms of exploit content are potentially affected by these mitigations including traditional shellcode, heap sprays, and ROP chains.

We focus on Office documents, but most of the high level principles discussed here apply to other file formats with minor or no changes. To demonstrate the mechanisms discussed, We draw examples from two document formats: the OLE based file format used in Office versions 97 to 2003 (.doc) and the OOXML based file format introduced in Office 2007 (.docx). While we refer to these file formats by their file extension (.doc/.docx) for brevity, we also included the spreadsheet (.xls/.xlsx) and presentation (.ppt/.pptx) files in this study. In practice, our exploit prevention mechanisms operated on the container level of these file formats and was therefore essentially the same across all of them.

3.1 File Access Protections

Other than restrictions levied by the document file formats themselves, there are relatively few file access based exploit mitigations, especially when compared to memory access protections. However, malicious content is often stored in the raw document file and accessed through the file system during exploitation. Typically, file level access of malicious content occurs later in the exploitation phase and this content is usually malicious code, whether it be shellcode or a portable executable. Sourcing malicious content from the file is extremely common in document based exploits.

The authors observed obfuscated portable executables embedded in the raw document file of 96% of the malicious Office 2003 documents in the contagio document corpus [27]. This set of malicious documents observed in targeted attacks includes files that were 0 day attacks when collected. Retrieving additional malicious content from the raw document file is also the norm in PDF files.

File level access most frequently is achieved through standard file access mechanisms, such reading the file handle. Because most client object exploits, including document exploits, are triggered by opening a malicious file, a handle to the exploit file is usually already available in the reader application. In many instances, the file-

name and path of the malicious document is not known in advance.

Depending upon the characteristics of the file format, there is usually a broad spectrum of options for disrupting malicious file level access, assuming the malicious data access does not occur through the file format parsing routines. The indirect access and semantically equivalent layout options provided by document file formats parsers are analogous to the virtual memory and code relocation techniques that enable ASLR. We can rearrange the raw layout of the document file without disrupting access to streams parsed out of the document. We use this disparity in access methods to unpredictably rearrange the raw access to the file content while maintaining the same reassembled content streams.

We focus on Microsoft Office documents for this study, but other document formats, such as PDF have mechanisms that permit the modification of documents in ways that break exploitation but still permit legitimate use. For example, the PDF format generally allows deviations in white space, PDFs are made of independent objects and streams that can be re-ordered, and data streams can be encoded or compressed in a few different ways. Much of our mitigation capability for Microsoft Office documents stems from characteristics that are very similar to a filesystem, allowing blocks of a stream in the document to be laid out independently without need for contiguous arrangement. An important result of the ability to fragment document content is that malicious payloads can be scrambled, forcing multiple independent pieces of the payload to be located individually or constraining the size of malicious payloads.

3.2 Memory Access Protections

Document content and structure can also influence the opening process's memory layout, paving the way for exploit protections through permutations that foil memory access misuse. These mechanisms rely on predictable processing of document data. This processing of document data can be as simple as direct loading of document data into memory but can also be much more complex. We focus on inducing changes in the program heap, as it is generally the most feasible to reliably and meaningfully modify using document data. Microsoft Office has long implemented barriers to malicious use of scripting such as document macros, including disabling execution of macros by default. Therefore, the most sophisticated malicious Office documents do not use macros, and load data into memory without using scripts. These non-scripted mechanisms for loading malicious content into memory are generally more constrained and easier to foil with content layout modifications than scripted content. Our exploit protection is applicable for content used directly in exploits, but generally does not apply to script generated content.

Permutations to document reader heap layout is obviously useful for addressing exploits that use heap sprays. We will provide an example of using DCLR to defeat heap sprays. Beyond traditional heap sprays, modern exploits often utilize heap data for object and memory corruption exploits, ASLR bypasses, malicious payloads that are egg hunted, among other techniques.

In addition to DCLR, we explore using memory consumption to influence reader process memory. It is possible to reliably influence process memory using document data, and therefore, it is possible to displace or dilute malicious payloads in memory. However, this technique incurs extremely high CPU and RAM use.

Exploit mitigations that are induced in the reader program through document content are therefore focused at preventing anomalous file and memory access. Fundamentally, the entropy introduced by the document seeks to make access to malicious content difficult by obscuring its location or scrambling it through fragmentation.

4 Exploit Content Location

Randomizing the location of exploit content is the core tenet of ASLR and similar exploit prevention mechanisms. This variability is usually introduced by the operating system and is usually measured by the amount of entropy that can be introduced.

Similarly, exploit counter-measures can be implemented by re-structuring documents so that they introduce entropy into the reader program. The intuition and primary goal of this variability is to make the location of exploit content unpredictable. The core difference is that instead of controlling the entropy through the execution environment, the entropy is introduced by unpredictable changes in the document that is the carrier of the exploit code.

Also analogous to mechanisms such as ALSR, document induced exploit prevention mechanisms are implemented such that they do not impact normal operation–benign content is used normally and is not impacted. This is possible to achieve in documents when the same fundamental assumption that enables traditional entropy based mechanisms holds: the exploits involve data access patterns that are different than normal operation. The key is the ability to probabilistically disrupt the abnormal data access patterns used in exploits while still keeping normal data access intact. This is possible to achieve when exploits access data at a different level of processing than the data used by the benign program flow.

Exploit protections that are implemented in the execution environment are limited primarily by factors such as the size of the space to randomized and the lower bound on size of the objects to be randomized. The limits in document induced entropy are bound by the number blocks inside the document that can be randomized. The exact nature of these objects and upper bound on quantity

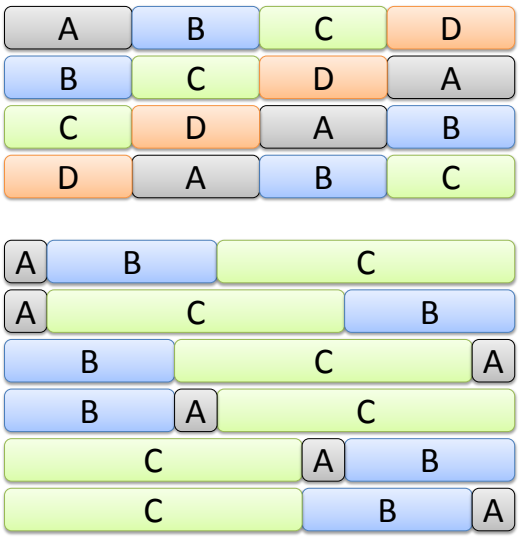

Figure 1: Object Layout Permutations

differ greatly by file format.

The lower bound of the entropy introduced by objects that are randomly arranged is based on the number of objects. This is represented as:

$$E = \log_2(n)$$

Where E is the entropy measured in bits and n is the number of objects randomized (benign or malicious).

While this represents the lower bound for entropy based on object re-ordering, this assumes that the objects are distributed among a number of fixed positions, usually because the objects to be re-ordered are done at fixed size blocks. However, in many cases the objects of are different size and are not arranged at fixed offsets. Therefore, the upper bound on the amount of entropy for a single malicious payload is derived from the sum of all the combinations of object sequences because an object can be placed at the end of any combination of n - 1 objects:

$$E = \log_2(2^{n-1}) = n - 1$$

Note that to achieve this upper bound requires that the various chains of combinations of objects have unique lengths which is a stricter requirement than all of the objects being of different size.

To further demonstrate the lower and upper bound of object order randomization, Figure 1 shows some of the permutations of two sets of objects that are randomized resulting in two bits of entropy each. The lower bound requires 4 objects while the upper bound requires only 3 objects. In both cases, any given object can reside in 4 unique locations.

These measures of entropy refer to the location of the beginning of a malicious payload and assume a single malicious payload. If the malicious payload is fragmented such that it consists of multiple blocks that must be located independently, the difficulty of guessing the location of all fragments is the product of the difficulty of each fragment. Therefore, the overall entropy expressed in bits, is the sum of entropy caused by each fragment:

$$E = m \log_2(n)$$

Where E is the entropy measured in bits, n is the number of blocks randomized, and m is the number of blocks that house the malicious payload. In practice, malicious payloads embedded in document files are done without provisions for fragmentation. Section 5 discusses malicious payload fragmentation and the inherent limitations on exploit payload size in more detail.

4.1 OLE File Content Layout Randomization

To demonstrate the effectiveness of DCLR, we provide an example based on Microsoft Office 2003 (.doc) files. In this example we focus on modifications to the document that provide location obscurity at the file level, preventing access to malware embedded in the file while still allowing normal interpretation of the document. This example is chosen because it is one of the most straightforward and widely applicable examples of document induced exploit protections.

In many cases, exploits refer back to data in the raw document for more malicious content. For example, shellcode may extract and execute more shellcode or a portable executable embedded within the document. This occurs by locating the raw file data for the document and accessing the malicious content at a predetermined fixed offset, often reading this data from the already open file handle. Other similar techniques include performing an egg hunt either through the file handle or a copy of the file in memory. Once the additional malicious content is accessed, it often requires de-obfuscation or decryption. It is common for there to be multiple levels of shellcode access and de-obfuscation, with the content increasing in size and complexity during the exploit malware bootstrap process.

The core mitigation proposed to defeat raw file reflection by malware is to perform file level document content layout randomization (DCLR). This general technique is possible in many file formats because most data files include multiple segments of data and often the layout of this data can be modified. Object Linking and Embedding (OLE) based file formats such as Microsoft Office documents (.doc) are particularly amenable to this technique.

The OLE structured storage format is a data format that allows compilation of multiple data streams and types into a single file. This general format is also called Compound File Binary Format or COM Structured Storage. OLE files implement mechanisms very similar to

a FAT file system which allows multiple files or objects to be included in the container file. OLE files include a header which is similar to a superblock, containing offsets to key data structures and definitions such as block sizes. A directory entry is included for each independent stream in the file and supports hierarchical layouts of objects embedded into the OLE file. A file allocation table (FAT) is used to track allocation of blocks to each stream in the file. In fact, there are two major FATs used to track normal sized blocks (typically 512 bytes) and small blocks (typically 64 bytes), in addition to other optional elements. Due to this flexible, if not complicated structure, OLE files can have numerous file level representations of data that are interpreted the same way. Beyond Office documents, there are many file formats based on the OLE container. For example, ActiveX objects also use an OLE container as discussed in Section 6.1.

Typically, the steams in an OLE file are sequentially stored. However, re-ordering can occur and is expressly allowed. Reordering data blocks in an OLE provides a consistently effective and quantifiable way to prevent access to malicious content in raw document files without impacting normal use.

To verify the effectiveness of this approach we built an OLE file block randomizer. It simply creates a new OLE file functionally equivalent to the original except that the layout of the data blocks is randomized. This is accomplished by randomizing the location of the data blocks, and then adjusting the FAT and directory data structures accordingly. This is essentially the opposite of running a file system defragmentation utility.

The entropy introduced by OLE file randomization is very straightforward. If reasonable simplifications and assumptions are made, the entropy is based on the number blocks in a file which is simply the quotient of the size of the file and the block size. This can be represented as:

$$E = \log_2(\frac{S_f}{S_b})$$

Where S_f is the size of the file and S_b is the block size used. If the typical 512 byte block size is used, then a 128KB file would have 8 bits of entropy and a 1MB file would have 11 bits of entropy for each file level access. If the malicious payload is larger than a single block, then the entropy increases with the count of malicious payload block. This makes the extraction of a large payload intractable in the face of file level DCLR.

4.2 OLE File DCLR Validation

In practice, the Office 2003 document content layout randomization utility performs as expected. The document content layout is fragmented in the transformed document, but the documents continue to render the same as the original document.

The ability to mitigate typical exploits was validated by manual dynamic analysis of real world malicious document samples in a virtual machine. About 20 malicious documents were selected to provide representative vulnerabilities, malware families, and exploitation techniques. Samples included vulnerabilities in Microsoft Office (CVE-2012-0158), vulnerabilities in other file formats embedded in office documents (CVE-2011-0609), and pure social engineering attacks. The files included documents, spreadsheets, and presentations.

It was observed that the document fragmentation was effective at preventing final malware execution for all documents that used an exploit. These maldocs all relied on malicious payloads embedded in the document file. Most malicious documents triggered a crash instead of malware execution, but some trojan documents resulted in a what appeared to be a an infinite loop instead of a crash. Since the pure social engineering documents do not involve any spurious file access to retrieve malware, these maldocs continued to function. However, these social engineering based maldocs required the user to change security settings or click through multiple warnings to enable the malicious content.

To test the performance impact of document content randomization, many known benign documents were converted to PDF using Microsoft Office and powershell scripting to simulate document open and rendering. 1000 documents were randomly selected from the Govdocs corpus [14]. There were 39 documents that were removed from this set because they required user input to open or printing was prohibited by Office. The most common cause of failing to print was invocation of protected view, which limits printing, apparently because they were created by old versions of Office. Other reasons including prompting for a password or prompting the user as a result of automated file repair actions. In addition, following OLE file format fragmentation, an additional 125 documents opened in protected view which prevented printing. These files apparently triggered some file validation heuristics in Office. The same mechanisms used to break exploits can also be used for malicious intent, such as evading virus scanners. All content was present, and it was discovered that the validation heuristic did not trigger reliably on independent formulations of the same original document–some transformations would trigger this protected view and some would not. Ergo, this protection built into Office must trigger on some particular block layouts but the exact criteria was not discovered by the authors.

The test data set therefore contained 836 documents totaling 197 MB. It took about 15 minutes for the documents to be converted to PDFs which equals just over 1 second per document. Performing multiple trials, there was no consistent difference in speed between the original and the fragmented documents. The differences in mean between the two sets of trials is about 1/50th of the 95% confidence interval as shown in Table 1. Therefore,

Table 1: Document Corpus Rendering Time

	Mean (s)	Std dev	95% Conf.
Original	930.3	35.4	$\pm$26.2
Transformed	929.7	42.5	$\pm$31.5

the randomized documents take no longer to open and render. This is expected as there is no additional work required to reassemble the randomized streams. Any effects resulting from less efficient read patterns seem to be masked by file caching.

Having converted both the original and fragmented documents to PDF documents, the resulting PDFs were compared for similarity. Since the PDFs had unique attributes such as creation times, none of the PDFs generated from rendering the original documents were identical to those generated from the fragmented documents. However, they were very similar in all respects. The average difference in size of the resulting PDFs was 40 bytes, with 513 of the PDF pairs having the exact same size. The average binary content similarity score of these derivative document pairs was 87 (out of 100) using the ssdeep utility [19]. Manual review of a small number of samples also confirmed the same content in the fragmented documents as in the original documents.

To evaluate the computational expense of performing the document content layout randomization, we measured the time to perform this operation on the 1000 document, 249 MB corpus. The average time to perform this task was 28.9s using a single thread on a commodity server. This equates to 68.9 Mbps of throughput. Preforming this content fragmentation on a single 248K sample (close to average document size) yielded an average 0.028s execution time. To put this execution time in perspective, we scanned the same corpus with ClamAV [1] which required an average 28.7s to complete. Hence, the operations we perform on these documents is extremely close in cost to that incurred by a common anti-virus engine.

OLE block layout randomization results in files that are semantically the same but which protect against raw file access without adversely affecting reader program performance. Our implementation requires computing resources competitive of a common anti-virus engine to perform DCLR.

5 Exploit Payload Size

Another constraint on exploits that can be levied by the document is a limit on the malicious payload size. Document induced payload size restrictions can be considered a special case of location obfuscation.

When content layout randomization is employed, malicious payloads are fragmented if they use non-standard data access mechanisms. Even if the first block of malicious payload is located, whether by chance or by mechanism such as egg hunting, the subsequent blocks are not located in order, breaking the malicious payload. The chance of arriving at each of the fragments in order is inversely proportional to the number of fragments. As stated before, this makes locating a single large fragmented payload through chance infeasible.

One of the common responses to ASLR is to use a heap spray. When a heap spray is employed, the exact location of the malicious payload is no longer critical as the target area is filled with copies of the malicious payload, any of which will advance the exploit successfully. Heap sprays can defeat ALSR in practice. Mechanisms such as heap sprays also effectively diminish the power of DCLR. Repetition increases the chance of encountering each fragment. This reduces the problem of locating a malicious payload to ensuring the proper of the fragments within the heap spray area.

The entropy induced when relying on malicious content random block ordering within an area where malicious content is repeated exclusively, such as is the case in a heap spray, is as follows:

$$E = \log_2((\frac{S_m}{S_b})^2)$$

Where S_m is the size of the malicious payload and S_b is the block size used for content fragmentation. Hence, the fundamental constraint enforced by DCLR is on the size of the malicious payload. This technique provides strong mitigation capability when the block size used for content layout re-ordering is smaller than the malicious payload. The limitations imposed on malicious payload size is most critical when the beginning of the malicious payload(s) can be located with ease and the malicious payload exclusively exists in the target area.

Content fragmentation provides an effective limit on the size of the shellcode in content based heap sprays, which will be discussed in Section 5.1. The OLE objects used in these heap sprays use a default block size of 64 bytes which is much smaller than the shellcode required for a meaningful exploit.

Table 2 lists the names and sizes of shellcode components provided in the metasploit framework [2]. Some of these schellcode components are intended to be combined with other components to implement full shellcode functionality. For example, the block_api component provides access to the windows API via hashes as identifiers. Most of the single_ and stager_ shellcode items implement a practically useful shellcode chain. The average size of all of these components is 289 bytes. In most situations, these shellcode blocks will be extended a small amount with exploit specific register setup and shellcode encoding. The size of the larger shellcode components is on par with the approximately 500 byte shellcode observed in the heap sprays used in CVE-2013-3906 .docx exploits which we use as examples in the following sections. All but the smallest shellcode components would require multiple 64 bytes content

Table 2: Metaploit Windows Shellcode Sizes

Name	Size (bytes)
stage_shell	240
stage_upexec	398
single_shell_hidden_bind_tcp	341
single_service_stuff	448
single_shell_reverse_tcp	314
single_shell_bind_tcp	341
single_loadlibrary	190 + len(libpath)
single_exec	192 + len(command)
single_create_remote_process	307
createthread	167
apc	244
executex64	75
migrate	219
block_service_change_description	448
block_service	448
block_exitfunk	31
block_api	137
block_create_remote_process	307
block_service_stopped	448
stager_sysenter_hook	202
stager_reverse_tcp_rc4	405
stager_reverse_https_proxy	274
stager_bind_tcp_nx	301
stager_reverse_ipv6_tcp_nx	298
stager_reverse_https	274
stager_reverse_tcp_nx	274
stager_bind_tcp_rc4	413
stager_reverse_tcp_dns	274
stager_reverse_tcp_nx_allports	274
stager_reverse_tcp_dns_connect_only	274
stager_reverse_http	274
stager_reverse_tcp_rc4_dns	405

blocks. The smallest components provide simple building blocks that are not useful on their own. For example, the block_exitfunk component provides functionality to terminate a process and requires the inclusion of the block_api component to function. Shellcode that provides enough malicious content to be useful in a real exploit is invariably larger than can fit within the 64 byte default size restriction imposed DCLR in these examples.

5.1 OOXML Memory Content Layout Randomization

We use Office 2007 documents (.docx) to demonstrate DCLR in reader memory. The docx format is based on the Office Open XML (OOXML) file format which utilizes a zip archive to contain the document components. The majority of OOXML documents components are XML documents, but it also supports other embedded objects. DCLR is trivial to implement at the file level through methods including reordering and modifying metadata in zip file entries. However, unlike .doc files where file level malicious payload access is the norm, the authors have not observed frequent use of file system access of raw .docx documents in current exploits. Therefore, DCLR is much more useful as a mitigation when applied memory access than file access in .docx files.

Furthermore, we use heap sprays in .docx files because these are easiest to demonstrate the importance of size based constraints levied by content fragmentation in memory. We focus on the non-script heap spray technique first utilized in CVE-2013-3906 exploits and also described in Section 6.1.

In this heap spray technique, many ActiveX objects containing primarily heap spray data are read when the document is opened and loaded into the heap. These objects are loaded into memory raw, without interpretation or parsing, and are, therefore, a highly desirable way to load arbitrary data into memory. Since ActiveX objects use the same OLE container format that Office 2003 documents use, we can use the same OLE fragmentation techniques that were described in Section 4.1. In addition to embedded Office 2003 documents and ActiveX controls, many other items that are embedded in Office 2007 documents also use the OLE format. These items, which include embedded objects such as equations, media files, office documents, and executables, can also be fragmented such that they operate as normal under typical use but are scrambled when raw access is employed. Dynamic analysis by the authors confirmed that that these embedded OLE objects are loaded directly in memory, while most other data from the document is not loaded into memory wholesale.

This example is focused predominately on shellcode. Document induced memory fragmentation could potentially disrupt other exploit mechanisms such as data used in ROP chains, object corruption exploits, etc. In each of these cases, the upper bound on the size of a single block of malicious content may prevent successful exploitation depending upon circumstances.

5.2 OOXML Content Layout Randomization Validation

To validate the efficacy of fragmenting objects to be used in heap sprays, a utility was created which randomized the layout of object embedded in Office 2007 Documents. The block randomizer created for OLE based .doc files was extended to reorder all OLE objects embedded in a .docx file. In practice, this transformation would occur between document creation and the document being opened.

Multiple samples of documents using the ActiveX object based heap spray implemented in CVE-2013-3906

were modified by fragmenting all the embedded OLE objects. The heap spray data is repeated every 4096 bytes and is about 500 bytes in length in these exploits, padded with instructions do not modify program flow. When fragmented in 64 byte blocks, the 8 blocks of shellcode are scrambled. If some simplifications are made, such as forgetting the overhead of the OLE format and assuming that all the data outside the actual payload is an effective NOP, there is a best case chance of $1/64$ of the exploit continuing to work because of the fragments being reached in order. This equates to 8 bits of entropy.

Using manual validation, it was not reasonable to perform enough permutations and observations to confirm the predicted success rate due to its very low value. There were no successful exploits observed after document content layout randomization. In the samples used for validation, much of the malicious content in the OLE object was not in a valid OLE stream and was replaced with null data by the OLE fragmenter which does not change file size. Therefore, while some of the malicious content was randomized and loaded into memory, most of the data loaded by these transformed objects was merely null data. The chance of successful exploitation after payload fragmentation was orders of magnitude lower than the potential best case for these samples. The functionality to remove data found outside valid OLE streams is a side benefit of the content layout randomizer. It should be noted that putting the heap spray data in valid OLE streams certainly would be possible with a trivial amount of effort, which would cause the content to be fragmented instead of removed.

Similar to that of the .doc content layout randomization, the computational cost of performing .docx embedded content randomization compares favorably to that of a virus scanner. We only randomize some of the objects inside of a .docx file–the binary objects that are are useful for predictably populating memory. However, since reliably determining the type of the file in the .docx zip container requires unzipping and inspecting all the contents, the majority of the cost is unavoidable even if none of the objects in the document are modified.

To measure the cost of performing our embedded object layout randomization, we compiled a corpus of benign .docx files from the Internet, using a web search with the sole criteria of seeking .docx files. The search yielded a wide diversity of sites with no known relevant bias on the part of the researchers.

This corpus consisted of 341 files weighing in at 76 MB. Executing our utility required an average 13.8s from which we derive a single threaded bandwidth of 44.2 Mbps. Scanning the same corpus with ClamAV required 28.0s, very close to double the time required for our mechanism. The time to execute on a single 225K document, which was approxiamately an average size document in this corpus, was 0.032s.

The vast majority of the documents in this benign corpus were not modified. Of the 341 documents, only 10 documents had objects on which layout randomization was performed. Since this number was so small, these samples were validated manually. Both the original and the modified document were opened and compared. In all cases there was no observable difference in the time required to open the document and the content of the document appeared pristine. As with .doc DCLR, the .docx embedded content layout randomization did not cause measurable performance differences. Randomizing the OLE objects embedded in .docx files maintained the integrity of the original document content, while breaking illegitimate memory access.

5.3 Omellete Shellcode

While not commonly seen in the wild, the documented countermeasure to limits on payload size is to perform an egg hunt per payload block, which has been styled omelette shellcode [9]. Omelette shellcode locates and combines multiple smaller eggs into a larger buffer, reconstructing a malicious payload from many small pieces. The omelette approach adds at least one more stage to exploit, in exchange for accommodating fragmentation of the malicious content.

A typical heap spray involves filling a portion of the heap with the same malicious content repeated many times, with each repetition being a valid entry point. This approach would be altered for an optimal omelette based exploit. One would spray the heap with the omelette code solely, then load a single copy of the additional shellcode eggs into memory outside the target region for the spray.

When multiple egg hunts are used to defeat malicious payload fragmentation, then the primary mitigation power is shifted to the size of a block in which the reassembly code must reside. Each egg containing the partitioned payload could have an arbitrarily small size with a few bytes overhead for a marker used to locate the egg and often an identifier to facilitate proper reordering. The size of the omelette code is invariably the bottleneck of the technique. If the omelette code can fit fully within a fragmentation block, then malicious payload fragmentation will not be effective.

Therefore, for omelette shellcode to operate, it must be loaded in a single 64 byte block or it will be fragmented and re-ordered. Most openly available examples of omelette shellcode, which are designed specifically to be as compact as possible, are about 80-90 bytes [4, 36]. Of course, it may be possible to shrink the size of the omelette functionality in a given exploit. There is an example of one omelette implementation that compiles to 53 bytes [12] but this implementation relies on the eggs being loaded in memory in sequential order, which could be defeated by document induced randomization as well.

It is also not likely that merely the bare omelette code would exist in a contiguous block. NOPs, additional

obfuscation code, register setup, etc. code would likely need to exist in the same 64 bytes block. However, even if it is possible to successfully implement an omelette based exploit in the default 64 byte block size, this block size could be dropped to a level rendering any sort of egg hunt infeasible. The size of these blocks in OLE files is tunable. It is also noteworthy that the cutoff between normal and small block stream and that the block size for the normal streams are also tunable, such that this flexibility in size applies generally to both normal and small OLE streams.

Because DCLR is not used widely, no examples of malicious documents could be found in the wild that used countermeasures such as omelette code. However, observations made during the manual validation performed for current exploits indicate that DCLR would still be successful.

6 Memory Displacement Based Mitigations

We explored alternatives for influencing process memory in Office. Another approach involves including additional non-visible content to consume memory and shift subsequent memory allocations. We sought for ways to re-arrange existing content, but could not find any practical way to influence memory load order without changing the representation of the document.

Attacking the same problem as Section 5.1, we sought to defeat .docx heap sprays. One direct counter to heap sprays is to thin out or decrease the concentration of the spray, making it less likely for malicious content to be found even if the access attempt is made in the section of the heap flooded by the spray.

To decrease the concentration of the heap spray, one injects benign content in the midst of the malicious content. Because most documents, especially malicious documents, have a relatively small amount of content compared to heap sprays which are often 10s of MB or larger, additional otherwise useless benign content must be added to the heap. When benign content is added to dilute heap sprays, the concentration of the heap spray is simply the ratio of the malicious content to the total content:

$$\phi = \frac{S_m}{S_m + S_b}$$

Where ϕ is the malicious payload density, S_m is the size of the malicious content, and S_b is size of the benign content.

This approach, while functional, has many limitations. The primary shortcoming is that it requires loading a large amount of useless content into memory which comes at great cost. Many heap sprays have very poor performance, in and of themselves. To be effective, one has to drown out the malicious content with even more inert content. This insertion of content must occur on all documents, including benign ones for this technique to be effective. However, maldocs are free to create very large heap sprays. We sought for alternative mechanisms to dilute heap spray, but we could devise no way to consume virtual memory space without incurring resource sapping data copies into memory. Using this technique incurs considerable overhead and is probably often not feasible due to performance impacts.

In addition to the cost of all the benign content needed to dilute a heap spray, there are other challenges. For this technique to be effective, the benign content must be interspersed with the malicious content. Proper alignment can be challenging depending upon how the heap spray is created and keeping the assumption that the part of the document that implements the heap spray cannot be known a priori. In the case of Office 2007 .docx files, adding additional content required changing some semantic meaning of the document, unlike OLE file fragmentation.

6.1 OOXML Heap Spray Dilution Validation

To demonstrate the effectiveness of heap spray dilution, a utility was created to modify .docx files such that when opened, the heap was sprayed with benign content. This method of spraying inert content without use of scripting is similar to that used in contemporaneous exploits [5]. This technique was pioneered in CVE-2013-3906 exploits, but has since been used in connection with other vulnerabilities and file formats [22].

ActiveX controls, with a large amount of benign superfluous content, are inserted into the document at random locations in the document. These items are set to be hidden so that they do not impact normal content. No method of adding benign content to the heap without making additions to the document could be found. However, under normal conditions, these items are hidden so the document does render as usual.

Inspection of the memory using dynamic analysis showed that the benign and malicious sprays were both loaded in memory. The individual benign and malicious objects were shuffled together such that the two sprays were interspersed according to the location of these objects in the document.

Performing manual validation via dynamic analysis on CVE-2013-3906 samples, it was confirmed that the malicious heap sprays were diluted as the ratio of exploit success matched that expected. For example, when the benign content was equal in size to the malicious heap data, the exploit success rate was consistent with the estimated rate of 50%. When the benign content was increased, the exploit success rate dropped proportionally. It was also demonstrated that in practice ensuring proper alignment and therefore optimal mixing of the benign and malicious sprays is challenging, resulting in

deviations from the anticipated exploit success rate.

Adding content to influence memory layout resulted in generally poor performance. Due to compression and the ability to reference data stored in the document multiple times, it was possible to incur only minor file size increases. However, to influence memory layout, memory had to be used, and the processing to consume the memory space was expensive. We found that optimal memory displacing benign objects added about 10 seconds of load time per 100MB of memory filled.

The performance characteristics of influencing memory layout through memory consumption, the difficulty of ensuring benign content is interspersed with malicious content, and the difficulty of overpowering potentially large heap sprays with benign content make heap spray dilution extremely computationally expensive. Document content based mitigations using displacement would be similar to operating system based memory protections that instead of using virtual memory based techniques to implement ALSR, used actual allocations that consumed memory to introduce entropy in memory addresses.

Despite high cost, interspersing malicious content with insert content is possible using document content modifications. The ability to dilute heap sprays and drop exploit success rates has been demonstrated. Since it is relatively easy to ensure that inert objects are loaded early in the document, reliable shifts to the heap are possible. While the document had to be modified, the objects used to consume memory were be marked as hidden, such that they did not change the rendering of the document.

It is possible that other exploit tactics, including memory corruption, use after free, ASLR bypass techniques, or ROP may provide situations where a limited content based memory dilution or shift could break exploits that would not be mitigated by OS level mitigations [10]. If situations are found where a small shift in they heap is useful for defeating exploits, the memory consumption based technique could be useful in practice.

7 Environmental Factors

Document content induced exploit mitigations are designed to compliment execution environment exploit protections such as ASLR. Document induced protections could provide protection against many exploit techniques if operating environment protections were not used, but this is not a plausible scenario as at least basic operating system based memory protections are used ubiquitously. Some document induced protections stand on their own but some only come into effect when attackers try to circumvent operating system exploit protections. For example, some content based protections defeat heap sprays which are only present if mechanisms such as ASLR are employed by the operating system.

The interactions between the operating environment and content driven protections are hard to generalize. Even seemingly independent mitigation strategies can operate together in ways that are difficult to quantify. For example, file based content re-ordering is generally independent from other existing protection mechanisms but file access may be preferred by attackers because other methods of retrieving malicious payloads are more difficult. Circumvention of file level content randomization could certainly be possible, but the difficulty of implementing these mechanisms is not obvious.

In the most straightforward situations, the entropy caused by operating system and document based protections could be combined, but this is not a simple combination. Imagine a situation where the operating system provides 16 bits of entropy for a malicious payload and the document layout provides 10 bits. The result is not as good as a simple 26 bit search space for some content loaded from the document into the heap. Typically, the operating system allocation entropy is limited by a 4k page size and the document induced entropy is limited by a default 64 or 512 byte block size. Given a 256K document using 512 byte blocks, the operating systemp provides variability in bits 12 to 27 and the document in bits 9 to 18. The following demonstrates this combination and overlap. Bits that represent location variability are represented by a 1 and the combination is a logical OR operation:

```
0000 1111 1111 1111 1111 0000 0000 0000 OS
0000 0000 0000 0111 1111 1110 0000 0000 Doc

0000 1111 1111 1111 1111 1110 0000 0000 Both
```

Combining these two ranges results in variability in bits 9 through 27 or 19 bits of entropy total. Note that the area of overlapping random location, bits 12 through 18, does not improve or diminish the randomness in this range. In this example, the size of the document, which dictates the entropy caused by content fragmentation is irrelevant because the same randomization is already provided by the operating system. However the granularity below the OS page size provides additional entropy. Content based protections also permit malicious payload fragmentation which is generally not possible with operation system approaches.

The manner in which malicious documents are employed also serves to further strengthen document content based protections. A fundamental difference with attacks that defeat entropy through brute force attempts is that many repeated attacks are not feasible with most document based exploits. It is often possible to issue thousands of malicious requests to a vulnerable network daemon, but sending thousands of socially engineered emails with malicious attachments is not feasible. A much lower level of entropy is required for practical and useful defenses if the cost per attempt is very high.

Malicious documents are very frequently used in situations where the attacker has to make the exploit fully self-contained and has very little ability to tailor the attack to the specific end host. Conversely, in web based exploits, the attacker can often depend on being able to load many types of content from multiple locations in a single exploit and can often make dynamic decisions about serving content based on criteria such as client software checks. The file based content fragmentation is very effective on a large number of documents due to the apparent desire to keep exploits using documents self contained. The lack of pre-exploit client information forces most document based exploit writers to generalize their exploits. Lastly, when document readers implement protections against scripting, making it more difficult for attackers to load arbitrary data in memory, then content based approaches are more likely to be applicable.

8 Discussion

Brute forcing malicious content location randomization seems unlikely through remote means due to the difficulty of repeat attempts. Usually, a user must explicitly open the document each time an exploit is attempted. This might be overcome if exploits could be formulated in a way that multiple permutations could be attempted per document open.

The most likely path to overcome DCLR seems to be circumventing it completely. If document file or document reader memory is accessed in exploits through the same means as the the reader program, then our mechanism has no affect. We consider this analogous the the return-to-libc approach to overcoming stack smashing techniques. In order to succeed, the exploit must use the data access mechanisms used as the document reader or not use relocatable document content in the exploit directly. Not all exploits are directly impacted by DCLR and many vulnerabilities may be formulated to circumvent DLCR. For example, the malicious documents foiled through OLE File Randomization in Section 4.2 could be modified to load the final malicious executable through a web download instead of extracting it from inside the document file. Similarly, the OOXML documents defeated through memory content location randomization as discussed in Section 5.1 could use a scripted heap spray instead of the scriptless heap spray relying on document content loaded into memory. However, these changes might cause the exploit to run afoul of additional mitigations such as restrictions on ability to download executables or restrictions on the execution of macros. Even if DCLR can be circumvented, it increases the difficulty of many document based exploits while incurring little performance overhead.

This work demonstrates effectiveness on some popular exploitation techniques. However, it would be beneficial to demonstrate how other techniques, such as ROP or object corruption techniques can be foiled by document content induced mechanisms. For example, it is likely that some sort of modifications made to documents could influence the stack and therefore break ROP chains. Demonstrating additional document content based protections is a promising thread for future research.

DCLR is an attractive mitigation technique because it incurs a very low performance impact. Transforming the document requires a level of computational resources that are already commonly employed to perform anti-virus on both network servers and client programs. DCLR incurs no measurable performance penalty when the transformed document is opened because this mechanism leverages the file stream reassembly routines already executed by the document reader. It seems that file system caching makes up for any potential performance impact occurred by accessing document file blocks in random order. Just as virtual memory mechanisms enable ASLR with little overhead, the parsing and reassembly that enables multiple file level representations of the same logical document allows for efficient DCLR. Any situation where data is referenced indirectly providing for multiple possible low level representation could potentially be used to implement exploit protections similar to DCLR. We focus on content layout randomization because the file formats studied here support a large degree of layout changes. However, other document and media formats might not support the same level of data fragmentation but may support arbitrary encoding or compression. Encoding or data randomization techniques generally have been unsuccessful due to computational overhead and the difficulty of deploying the technique which requires modifying system libraries as well as applications. It is likely that some document induced analogs to data randomization, using decoding or decompression, can be used to defeat exploits with low overhead also.

While DCLR does not impact the content of the document as interpreted by the document reader and viewed by the user, it does change the raw document file. This could potentially impact some signature matching systems which operate on raw files instead of interpreting as the document reader does. Also, cryptographic signatures such as those used in signed emails would not validate correctly on the transformed document. Solutions to these issues have yet to be elaborated, but potential solutions are promising. For example, signature matching systems can implement file parsing. Signatures validation systems can operate on an invariant logical representation of the parsed document, instead of a potentially arbitrary file level representation.

9 Conclusions

We designed and evaluated exploit protections using transformations performed on documents. Document content layout randomization is effective in relocating malicious content in document files and in document reader process memory. We demonstrate the ability to mitigate current exploits in Office 2003 and Office 2007 documents. The overhead of transforming documents is comparable in run time to a common anti-virus engine and the added latency of opening a content layout randomized document appears to be negligible. The transformed documents are functionally equivalent to the original documents, barring the exploit protections that are induced.

The proposed protections complement operating system based defenses by providing higher granularity in relocation and through content fragmentation which creates an additional constraint on malicious payload size. Memory consumption based approaches can be used to reliably safeguard document reader memory but it seems that such approaches require significant amounts of computational and memory resources. In general, transformation based approaches are applicable where indirect data access and arbitrary layout is used by the reader but exploits misuse direct access to data. Other file formats and exploitation techniques could possibly benefit from this approach.

References

[1] Clam AntiVirus. *http://www.clamav.net/*.

[2] Metasploit. *http://www.metasploit.com/*.

[3] VirusTotal - Free Online Virus, Malware and URL Scanner. *http://www.virustotal.com/*.

[4] Hacking/Shellcode/Egg hunt/w32 SEH omelet shellcode. *http://skypher.com/wiki/index.php?title= Hacking/Shellcode/Egg_hunt/w32_SEH_omelet_shellcode* (July 2009).

[5] 5 Attackers & Counting: Dissecting The "docx.image" Exploit Kit. *http://www.proofpoint.com /threatinsight/posts/dissecting-docx-image-exploit-kit-cve-exploitation.php* (Dec. 2013).

[6] BHATKAR, S., AND SEKAR, R. Data Space Randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, D. Zamboni, Ed., no. 5137 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2008, pp. 1–22.

[7] BLAZAKIS, D. Interpreter Exploitation. In *WOOT* (2010).

[8] BLOND, S. L., URITESC, A., GILBERT, C., CHUA, Z. L., SAXENA, P., AND KIRDA, E. A Look at Targeted Attacks Through the Lense of an NGO. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 543–558.

[9] BRADSHAW, S. The grey corner: Omlette egghunter shellcode.

[10] CHEN, X. ASLR Bypass Apocalypse in Recent Zero-Day Exploits. *FireEye Blog http://www.fireeye.com /blog/technical/cyber-exploits/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html* (Oct. 2013).

[11] CROSS, J. S., AND MUNSON, M. A. Deep PDF Parsing to Extract Features for Detecting Embedded Malware. Tech. Rep. SAND2011-7982, Sandia National Laboratories, Sept. 2011.

[12] CZUMAK, M. Windows Exploit Development – Part 5: Locating Shellcode With Egghunting. *http://www.securitysift.com/windows-exploit-development-part-5-locating-shellcode-egghunting/* (Jan. 2014).

[13] DHAMANKAR, R., PALLER, A., SACHS, M., SKOUDIS, E., ESCHELBECK, G., AND SARWATE, A. Top 20 Internet Security Risks for 2007.

[14] GARFINKEL, S., FARRELL, P., ROUSSEV, V., AND DINOLT, G. Bringing science to digital forensics with standardized forensic corpora. *Digit. Investig. 6* (Sept. 2009), S2–S11.

[15] HARDY, S., CRETE-NISHIHATA, M., KLEEMOLA, K., SENFT, A., SONNE, B., WISEMAN, G., GILL, P., AND DEIBERT, R. J. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[16] HYPPONEN, M. How We Found the File That Was Used to Hack RSA. *http://www.f-secure.com /weblog/archives/00002226.html* (Aug. 2011).

[17] KANTER, M., AND TAYLOR, S. Attack Mitigation through Diversity. In *MILCOM 2013 - 2013 IEEE Military Communications Conference* (Nov. 2013), pp. 1410–1415.

[18] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 272–280.

[19] KORNBLUM, J. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation 3, Supplement* (Sept. 2006), 91–97.

[20] LABS, S. Security Threat Report 2014: Smarter, Shadier, Stealthier Malware.

[21] LASKOV, P., AND ŠRNDIĆ, N. Static detection of malicious JavaScript-bearing PDF documents. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, ACM, pp. 373–382.

[22] LI, H., ZHU, S., AND XIE, J. RTF Attack Takes Advantage of Multiple Exploits. *http://blogs.mcafee.com/mcafee-labs/rtf-attack-takes-advantage-of-multiple-exploits* (Apr. 2014).

[23] LI, W.-J., STOLFO, S., STAVROU, A., ANDROULAKI, E., AND KEROMYTIS, A. D. A Study of Malcode-Bearing Documents. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, B. Hämmerli and R. Sommer, Eds., vol. 4579. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 231–250.

[24] MAIORCA, D., CORONA, I., AND GIACINTO, G. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious PDF files detection. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 119–130.

[25] MAIORCA, D., GIACINTO, G., AND CORONA, I. A Pattern Recognition System for Malicious PDF Files Detection. In *Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition* (Berlin, Heidelberg, 2012), MLDM'12, Springer-Verlag, pp. 510–524.

[26] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *2012 IEEE Symposium on Security and Privacy (SP)* (May 2012), pp. 601–615.

[27] PARKOUR, M. 11,355+ Malicious documents - archive for signature testing and research. *http://contagiodump.blogspot.com/2010/08/malicious-documents-archive-for.html* (Apr. 2011).

[28] SERNA, F. J. The info leak era on software exploitation. *Black Hat USA* (2012).

[29] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.

[30] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2004), CCS '04, ACM, pp. 298–307.

[31] SHAFIQ, M. Z., KHAYAM, S. A., AND FAROOQ, M. Embedded Malware Detection Using Markov n-Grams. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), DIMVA '08, Springer-Verlag, pp. 88–107.

[32] SMUTZ, C., AND STAVROU, A. Malicious PDF Detection Using Metadata and Structural Features. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 239–248.

[33] STOLFO, S., WANG, K., AND LI, W. Fileprint analysis for malware detection. *ACM CCS WORM* (2005).

[34] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy (SP)* (May 2013), pp. 48–62.

[35] TABISH, S. M., SHAFIQ, M. Z., AND FAROOQ, M. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics* (New York, NY, USA, 2009), CSI-KDD '09, ACM, pp. 23–31.

[36] TEAM, C. Exploit notes-win32 eggs-to-omelet. *https://www.corelan.be/index.php /2010/08/22/exploit-notes-win32-eggs-to-omelet/* (Aug. 2010).

[37] TEAM, P. PaX address space layout randomization. *http://pax.grsecurity.net/docs/aslr.txt* (2003).

[38] TZERMIAS, Z., SYKIOTAKIS, G., POLYCHRONAKIS, M., AND MARKATOS, E. P. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security* (New York, NY, USA, 2011), EUROSEC '11, ACM, pp. 4:1–4:6.

[39] VILLENUEVE, N., WALTON, G., GROUP., S., AND FOR INTERNATIONAL STUDIES. CITIZEN LAB., M. C. Tracking GhostNet investigating a cyber espionage network, 2009.

[40] ŠRNDIĆ, N., AND LASKOV, P. Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium* (2013), Citeseer.

[41] WEI, T., WANG, T., DUAN, L., AND LUO, J. Secure Dynamic Code Generation Against Spraying. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 738–740.

[42] WOJTCZUK, R. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine, Volume 11, Issue 58* (2001).

[43] WRESSNEGGER, C., BOLDEWIN, F., AND RIECK, K. Deobfuscating Embedded Malware Using Probable-Plaintext Attacks. In *Research in Attacks, Intrusions, and Defenses*, S. J. Stolfo, A. Stavrou, and C. V. Wright, Eds., no. 8145 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2013, pp. 164–183.

[44] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy (SP)* (May 2013), pp. 559–573.