Preventing Exploits in Microsoft Office Documents Through Content Randomization

Charles Smutz(✉) and Angelos Stavrou

George Mason University, Fairfax, USA
{csmutz,astavrou}@gmu.edu

Abstract. Malware laden documents are a common exploit vector, often used as attachments to phishing emails. Current approaches seek to detect the malicious attributes of documents 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 and data randomization, but we implement them through permutations to the document file layout.

We randomize the data block order of Microsoft OLE files in a manner similar to the inverse of a filesystem defragmention tool. This relocates malicious payloads in both the original document file and in the memory of the reader program. Through dynamic analysis, we demonstrate that our approach indeed subdues in the wild exploits in both Office 2003 and Office 2007 documents while the transformed documents continue to render benign content properly. We also show that randomizing the compression used in zip based OOXML files mitigates some attacks. The strength of these mechanisms lie in the number of content representation permutations, and the method applies where raw document content is used in attacks. Content randomization methods can be performed offline and require only a single document scan while the user-perceived delay when opening the transformed document is negligible.

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 [2]. 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. 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 social engineering is used as part of the delivery vector to enhance the likelihood that victims will execute the malware contained within the document.

© Springer International Publishing Switzerland 2015
H. Bos et al. (Eds.): RAID 2015, LNCS 9404, pp. 225–246, 2015.
DOI: 10.1007/978-3-319-26362-5_11

For years, client side exploits, including attacks against document readers, have become more prevalent [6]. 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 and 10 in 2014, 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, implying current widespread concern over the role of documents as malware carriers. The impetus to stop Trojan documents is elevated due to their pervasive use in targeted espionage campaigns, such as those waged against non-governmental organizations (NGOs) [4,8,11].

Numerous approaches have been proposed to detect malicious documents. Signature matching, dynamic analysis, and machine learning based approaches are used widely in practice. Despite these many approaches, malware authors continue to evade detection and exploit computers successfully. Mitigations which seek to defeat many classes of memory misuse based exploits, such as address space layout randomization (ASLR) and data execution prevention (DEP), are implemented in modern operating systems. However, these protections are commonly circumvented and exploitation is still possible [23]. 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 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 scrambling the malicious payloads used in attacks. While address space layout randomization is performed by the operating system, we induce exploit protections in the reader program through modifications to the input data. We modify a document at the file format level, resulting in a transformed document which will render the same as the original, but in which malicious content is disarranged and made 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.

We explore various methods of document content randomization (DCR). Microsoft Office formats are particularly amenable to document content fragment randomization (DCFR) where we rearrange 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. We also show that document content encoding randomization (DCER) breaks exploits in practice by changing the raw representation of the data in the file without changing the decoded stream. The strength of DCR

rests in the number of unique content representations that can be created. For DCFR, this is driven by the number of block order permutations. For DCER, the number of possible encodings is the limiting factor.

We evaluate these content randomization techniques by testing them against hundreds of real world maldocs taken from VirusTotal. We test DCR on the three most prevalent exploits in both .doc and .docx files in our malicious document corpus. We find that most attacks are mitigated. Furthermore, we measured the performance of document transformation and found it to be comparable to an anti-virus scan. Opening a transformed document is at most 3 % slower than normal. We also validated that the transformed document rendered the same as the original document. Lastly, we examine the limits of DCR in the face of possible bypass techniques such as omelette shellcode.

The applicability of DCR 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 DCR include breaking intrusion detection system signatures and cryptographic signatures applied to the raw document file.

2 Related Work

Detection of malicious documents using byte level analysis has long been studied and deployed widely despite recognized weaknesses in detecting new or polymorphic samples [13,22,24].

Dynamic analysis of documents can provide additional detection power, but it comes with computational cost, difficulty of implementation, and ambiguity in discerning malware [27]. 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 [14,20,21]. 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) [26] is adopted widely. It is effective in defeating many classes of exploits, but is circumvented through limitations in implementation [19], use of heap sprays [28], or data leakage. As return oriented programming and similar techniques [18] have become popular, mechanisms to relocate or otherwise mitigate code (gadget) reuse have been proposed [16,29].

ASLR 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 [9]. Data space randomization enciphers program data with random keys, but this method is not feasible in practice due to deployment difficulty and computational expense [3]. Instead of implementing these protection mechanisms in the execution environment, we seek to induce barriers to malware by modification to the document.

3 Microsoft Office File Formats and Exploit Protections

We here describe the commonly used Microsoft Office file formats. The OLE Compound Document Format was used as the default by Office 97-2003 and is used in the files whose extension is typically .doc, .ppt, and .xls. Beginning in Office 2007, the default file format is Office Open XML which use the .docx, .pptx, and .xslx extensions. We also explain the most important exploit protection mechanisms provided by Office and Windows, such as ASLR and DEP.

3.1 OLE Compound Document Format

The file format used by Office 97-2003 is called by many names including Compound File Binary Format and OLE Compound Document Format. We refer to this format as the "OLE" file format throughout this paper, as many of the libraries and utilities for parsing this format use variations of this name.

The OLE Compound Document format supports the storage of many independent data streams and borrows many structures from filesystems, especially the FAT filesystem. OLE files are used as a container for many file types including Office 97-2003 (.doc/.ppt/.xls), Outlook Message (.msg), Windows Installer (.msi), Windows Thumbnails (Thumbs.db), and ActiveX or OCX controls (.ocx). All of these file formats use the OLE format as a base container, implementing their own data structures inside of streams stored by the OLE format. In this way, the OLE file format can be compared to the zip archive which serves at the base container for diverse file formats including Java Archives (.jar), Office Open XML Documents (.docx/.pptx./.xlsx), and Mozilla Extensions (.xpi).

Like a filesystem, the OLE format is comprised of many data blocks or sectors. The vast majority of OLE files use a 512 byte sector size, but other sizes are possible. Blocks in the OLE file are allocated and linked using allocation tables. The individual data streams are logically organized in a hierarchical structure similar to the file/folder organization of a filesystem. There is a directory entry for each OLE stream containing the name of the file, the location of the first sector, and other metadata such as modification times. The OLE format also supports mini-streams which divide normal size sectors into 64 byte sectors, using a second, embedded allocation table.

Except for the header, which must be located in the first sector, all of the contents in an OLE file can be located arbitrarily within the OLE file. Furthermore, there is no requirement for the various data streams to be arranged in order or in contiguous blocks, although it is the norm.

The OLE format serves as a container for arbitrary data streams. Individual file formats use this basic container but implement their own format for the data in the various streams. We do not describe the particular OLE-based file formats as they vary widely, are often proprietary and poorly documented, and our study relies on the common container capability of OLE files.

3.2 Office Open XML File Format

The Office Open XML (OOXML) File Format became the default file format for Office documents starting in Office 2007. These files use the extensions of .docx, .pptx, and .xlsx. This format has been codified in international standards ECMA-376 and ISO/IEC 29500.

The OOXML file format uses a zip file as a container, with individual objects stored as files in the zip. The majority of the content in an OOXML file is XML data. The document content is represented as XML with markup that is unique to the OOXML format, but which is generally similar to other markup such as HTML. The contents of an OOXML document can be modified by unzipping the archive, modifying it with a text editor, and zipping the archive. However, the relatively complex markup requires extensive knowledge to make major changes.

While text and formating can be represented using XML, other content, such as images, are embedded in the document as separate files in the zip archive. Some of the binary objects embedded in OOXML files utilize the OLE format. Example of these files include Office 2003 files, some multimedia files, equations, ActiveX controls, and executables.

3.3 Microsoft Office Exploit Protections

Macro based viruses have long been an issue in Office documents. All recent Office versions disable the automatic execution of macros. The OOXML file format assigns macro based files a separate extension, such as .docm instead of .docx, making inadvertent execution of macros extremely difficult. The OOXML format is designed to minimize the amount of binary content and improve the readability in the text content of documents, making the file format easier to validate and simplifying the parser.

Data Execution Prevention (DEP) was enabled in Office 2010, which prevents execution of arbitrary shellcode in the heap and has generally forced the adoption of ROP code re-use techniques. ASLR was enabled by default in Windows Vista. Office running on versions of Windows since Vista will have the benefit of ASLR for operating system libraries, but ASLR was not enabled for all Office provided libraries until Office 2013. Hence, up until Office 2013, we observe use of ROP gadgets from Office libraries without the need for an ASLR bypass.

Throughout this paper, we refer to the container format common to Office 2003 and many other files as OLE. We refer specifically to Office 2003 or Office 2007 files by their extension: .doc or .docx respectively. While we refer to these file formats by the file extension of the document files (.doc/.docx) for brevity, we also included the presentation (.ppt/.pptx) and spreadsheet (.xls/.xlsx) files in our study. Our study relies on file format characteristics which are common across the document, presentation, and spreadsheet file variations.

4 Approach

Inspired by the simplicity and generality of ASLR-like techniques, we seek to obtain similar exploit mitigation outcomes through transformations to input

data. The properties of a document can often directly and predictably influence various run time attributes of the opening application. The memory of a reader program is necessarily influenced by the file which it opens. We seek to find practical ways to make exploitation more difficult by content induced variations to the document file and reader memory. We call this general approach document content randomization (DCR).

We study two specific forms of document entropy infusion: document content fragment randomization (DCFR) and document content encoding randomization (DCER). DCFR is analogous to ASLR in that the order of data blocks in the document file is randomized. DCER can be compared to data space randomization because we randomize file level encoding. Both of these approaches apply to data stored in document files, but they can affect memory as files and subfiles are loaded into memory.

These transformations are envisioned to operate on documents during transfer between the potentially 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 the 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.

We focus on Office documents, but most of the high level principles discussed here apply to other file formats. Specifically, we seek to mitigate exploits in two common document formats: Office 2003 (.doc) and Office 2007 (.docx).

4.1 Content Randomization in .doc Files

The most promising opportunity to apply DCR to .doc files is at the raw document file level. Malicious content is often stored in the raw document file and accessed through the filesystem 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 surprisingly 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 [17]. 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 common in PDF files used in targeted email attacks, while web based PDF exploits usually load their final payload through web download.

File level access most frequently is achieved through standard file access mechanisms, such as 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. While the malicious content may be embedded raw into the document file, exploits

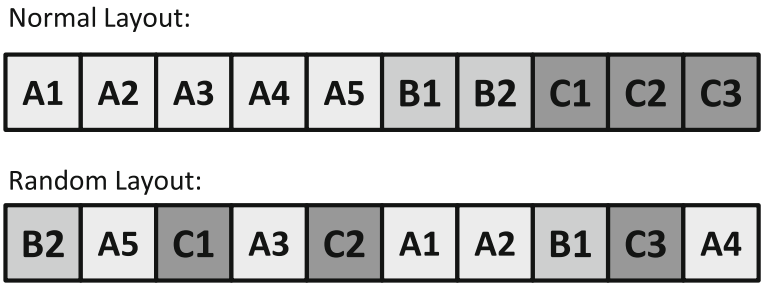

Fig. 1. OLE Fragmentation: The order of blocks in data streams is randomized, fragmenting the payloads

typically employ signature matching evasion techniques, such as trivial XOR encryption. The malicious content accessed through the raw document is sometimes accessed by offset, but typically an egg hunt is employed, where the file is searched for a specific marker. In most .doc files, we observed the shellcode and portable executables embedded within the bounds of the structure of the document, but simply appending malicious content to the end of an existing document is possible and is used sometimes.

We defeat raw file reflection by malware by performing file level content fragmentation (DCFR). OLE based file formats such as .doc files are especially accommodating of this technique. Typically, the streams in an OLE file are sequentially stored. However, re-ordering can occur and is expressly allowed. To implement 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 sector allocation tables and directory data structures accordingly. This is essentially the inverse of running a filesystem defragmentation utility. An example of fragmentation of three OLE data streams is given in Fig. 1. Note that the blocks in the data streams are typically sequentially arranged. We randomize the order of the data blocks across all the streams. In the event that any data exists in the raw document file stream but is not contained in valid OLE sectors, the data is not transfered to the new randomized document.

Re-ordering data blocks, or DCFR, in an OLE provides a consistently effective and quantifiable way to prevent access to malicious content in raw document files without impacting normal use. Since OLE files do not implement any form of encoding at the container level, DCER is not a practical option.

4.2 Content Randomization in .docx Files

We also studied the use of document content in .docx exploits. We found a small number of OOXML files where the raw zip container was accessed for a malicious payload. These attacks simply included the malicious payload, usually

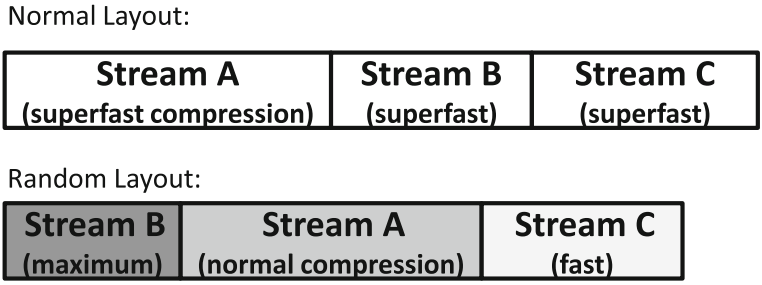

Fig. 2. ZIP Encoding Randomization: The order and compression level of data streams is randomized

an encrypted portable executable, in the zip file without compression. It is then trivially located in the file through an egg hunt, similar to that done in OLE files.

We devised two simple ways to introduce entropy in the OOXML file. First, we randomized the order of the files in the zip archive. This defeats access based on offset. We also re-compressed the zip data streams, randomly selecting one of four deflate compression levels (superfast, fast, normal, maximum). Figure 2 demonstrates transformation of a simple zip file with three subfiles. Note that the order of the files in archive and the compression used on each file is randomized. Office uses superfast compression, the lowest compression level, so our archive randomization usually results in smaller files. Compression level randomization is enough to foil simple access to file content, even if an egg hunt is used. Therefore, content encoding randomization (DCER) applied at the file level is applicable to some .docx exploits.

However, most .docx exploits gain access to the final malware payload through a web download or through an egg hunt in memory. We found a common method of performing scriptless heap sprays in contemporaneous exploits that can be mitigated by DCFR [1,12]. In this heap spray technique, first observed in CVE-2013-3906, 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. It is not clear why these objects are loaded into memory in this manner, while other embedded files do not receive the same treatment. Dynamic analysis by the authors confirmed that these embedded ActiveX objects are loaded directly in memory, while most other data from the document is not loaded into memory wholesale. Even if these ActiveX controls are not activated, they represent a simple and effective way to introduce content directly into the memory of the reader program.

Heap sprays are used to defeat ASLR. They ensure that the malicious payload can be located with high certainty through duplication of the malicious payload across a large memory address range, even if the address of a single copy can not be predicted. Only one copy of the malicious payload is needed for successful exploitation. Traditionally, heap sprays contain shellcode. However, DEP prevents execution from the heap. In the case of exploits targeting systems

with DEP, the heap is commonly sprayed with ROP gadgets and a stack pivot is used to move the stack into the sprayed region. These attacks successfully evade ASLR and DEP. We observed this technique for scriptless heap sprays used for both traditional shellcode and to implement fake stacks implementing malicious ROP chains. While these two techniques have been observed, this ability to easily and predictably influence the reader process's memory could be used for other attacks such as object corruption exploits. This general technique is also used to load single copies of arbitrary content, including portable executables, into memory which is later egg-hunted and used in exploits.

Since these ActiveX objects use the same OLE container format that Office 2003 documents use, we use the same OLE fragmentation techniques to defeat these scriptless heap sprays. We randomized the layout of all OLE files embedded in .docx files, regardless of their role. When these objects are loaded into RAM, the content is scrambled, but can still be retrieved by a document reader which implements the OLE decoding routines. These scriptless heap sprays in .docx files represent an example of how document content directly influences reader memory.

For .docx files, we perform both file level encoding randomization and fragmentation of objects to be loaded into memory.

4.3 Strength of Content Randomization Mechanisms

Like other probabilistic exploit protections, one can calculate the likelihood of exploit success in the face of brute force attacks against DCR. Methods such as ASLR obfuscate the location of malicious payloads. Document content randomization does this as well. However, content based malicious payloads are very frequently located via egg hunts or are duplicated in heap sprays, obviating randomized relocation. In practice, the primary protection power lies in randomization of the content representation, whether through fragmentation or through encoding.

In the simple case, the probability of a payload that has been randomly fragmented being in proper order is the inverse of the number of possible permutations or 1/n! where n is the number of fragments. In practice, this should be adjusted to account for other data mixed in with the malicious payload, repetition of the malicious payloads, and other limitations or constraints. For example, when OLE DCFR is employed, the number of fragments that influence the possible permutations is not just the number of fragments in the malicious payload, but includes all of the sectors that are randomized.

When we perform DCER on .docx files, we randomly select between four deflate compression levels. This is adequate for all the samples we observed where DCER has effect because they all involve data streams that are originally uncompressed. This is, however, a very small number of permutations. Part of the strength of DCER is also rooted in how difficult the encoding of the content is to reverse or circumvent. Compression makes generating a specific post compression malicious payload more difficult through transformations and restrictions in the encoded output. For example, repeated byte sequences, such

as the high order bytes in addresses used in ROP gadgets, are not found in compressed output. Unlike straightforward fragmentation, the constraining power of encoding is more difficult to quantify. Individual implementations of deflate are deterministic, but they are also allowed great latitude in how the encoding occurs. The same data stream can have many byte level representations using the same encoding method. If an entropy inducing compressor/encoder is used, the number of encoding induced permutations could be quantified.

The strength of DCR lies in the ability to fragment or encode malicious payloads in an unpredictable and constraining manner. This strength can be quantified as proportional to the number of randomized content permutations. We address possible DCR evasion approaches in Sect. 7.

5 Exploit Protection Evaluation

We evaluated the effectiveness of our content based exploit protections on hundreds of malicious Office documents sourced from VirusTotal. These documents were downloaded daily from the recent uploads to VirusTotal over the course of months. Our downloads were limited primarily by our monthly download limit on VirusTotal. We obtained 64,617 unique .doc files between May 2013 and March 2015 and 32,383 unique .docx files between November 2013 and March 2015, averaging 98 .doc and 66 .docx files per day. Of these collected documents, 40720 .doc and 2901 .docx files were labeled by at least one AV engine as malicious in a scan conducted two weeks following initial submission. Of these malicious documents, 1085 .doc and 578 .docx files were labeled by the anti-virus engines as utilizing a known exploit. The majority of the non-exploit malicious documents were identified by the anti-virus engines as utilizing macros.

As our study advances methods to break exploits using mechanisms not applicable to pure social engineering attacks, we focused our evaluation solely on maldocs leveraging a software vulnerability. Furthermore, to be able to better explain how our mechanisms applied to specific exploits, we utilized only those maldocs which were labeled by anti-virus engines to use a single exploit. We were left with 962 .doc and 363 .docx files after inconsistent exploit labels were removed. Of these documents, we found all exploits for which we were able to replicate successful exploitation and for which there were at least 20 samples. This resulted in 3 exploits in .doc files and 3 exploits in .docx files. Surprisingly, the malicious documents were distributed heavily across a small number of particularly popular exploits. For example, the three top exploits in the .docx file types comprised 306 of the 363 files, with 225 of these samples in the most popular exploit. In the event that we had many samples for a given exploit, we randomly selected a subset achieving a maximum of 100 documents to test and a maximum of 50 viable maldocs per exploit. In total, there were 343 documents tested and 217 documents demonstrating successful malware execution across these 6 sets.

To test for exploitation, we attempted dynamic execution of the Trojan documents by opening them in a virtual machine. To achieve successful exploitation,

we utilized various configurations of software including both Windows XP and Windows 7 and Office 2007 and Office 2010. The ROP based exploits required specific versions of the libraries from which they reuse code. Since one of the exploits selected for our testing is in Adobe Flash, we also installed the appropriate version of Flash player. We considered the malware execution successful when malicious code was executed or requested from the network that would have been executed. Successful exploitation occurred in 217 or 63 % of the malicious documents we tested. We attribute this relatively low malware success rate to VirusTotal being used by malware authors for testing, sometimes testing unreliable or incomplete exploits. For example, in a few of the successful exploits we observed calc.exe, the malware "hello world", as the final payload. There were a small number of apparent false positives by AV as well.

Taking these successful malicious document based exploits, we applied our document content based mitigations and re-ran the documents. We considered the exploit blocked by DCR when the final malware payload was blocked. We observed the differences in malware execution through both host based and network based instrumentation. In a very small number of cases, DCR was not possible due to the malicious document having defective structure. These failures were considered blocked as well, but are due to the rudimentary file validation provided by performing content randomization.

Generally, the malicious documents we observed employ a portable executable as the final malicious payload. Most of these executables are extracted from the raw document file, many are downloaded from an external server, and a few are extracted from document reader memory. In many of the Trojan documents, the original document file is overwritten by a benign document, which is opened and presented to the user. Most of the malware immediately beacons to a controller node, but a small minority of the malware performed other actions such as infecting other files on the local system. We observed dropped benign documents and malware that correlate to recent reports of targeted attacks against NGOs [4,8] as well as more opportunistic crimeware.

When the document based exploit is blocked by DCR, the document reader typically crashes. However, sometimes instead of crashing, the reader enters an infinite loop, presumably performing an egg hunt that is never successful. When a decoy benign document is provided by the malware, it is either never opened due to a failure in malware execution or the benign document is scrambled due to DCR and the attempt to open the document fails because the file is invalid. When DCR interrupts file-level access, shellcode that is attempting to extract a portable executable or additional shellcode from the document file is interrupted. When memory fragmentation is effective, it scrambles either shellcode or ROP chains, preventing exploitation earlier. Table 1 contains the high level results of our evaluation.

CVE-2009-3129 is triggered by a malformed spreadsheet that causes a memory corruption error. All of the successful exploits were .xls spreadsheet files. In all of these exploits, the pattern of extracting an encrypted portable executable

Table 1. DCR Exploit Protection Evaluation

CVE	File Type	Blocked	Total	Block Rate	Effective Mechanism
2009-3129	.xls	36	36	100 %	File Fragmentation
2011-0611	.doc, .xls	29	29	100 %	File Fragmentation
2012-0158	.doc, .xls	50	50	100 %	File Fragmentation
2012-0158	.pptx, .xlsx	4	10	40 %	File Encoding
2013-3906	.docx	42	42	100 %	Memory Fragmentation
2014-4114	.ppsx, .docx	2	50	4 %	File Validation
All-		163	217	75.1 %	-

and benign decoy document is employed. Due to raw access to the document file, all of these exploits were defeated by file level DCFR.

CVE-2011-0611 is actually a vulnerability in Adobe software products, including Flash player, but it is most often observed inside of Office documents. This exploit triggers a type confusion error through a malformed Flash file embedded in the Office document. We were able to observe successful exploitation in both .doc and .xls files. Like the other exploits embedded in OLE based file formats, all of the exploits are defeated by file level DCFR because the malicious executable and decoy document are extracted from the raw document file.

It is interesting to observe that this exploit in Adobe products was utilized so heavily in Office files. It is likely that part of the reason this exploit was embedded in Office documents was to leverage the social engineering of email based attacks.

CVE-2012-0158 is caused by malformed ActiveX controls that corrupt system state. While originally reported in RTF documents, our VirusTotal sourced malware contained a large number of 2012-0158 exploits in the OLE container as well. We observed successful exploitation in both .doc and .xls files, which was defeated by file level document fragmentation.

We also observed 2012-0158 in OOXML based files. These .docx based 2012-0158 were much less common than the .doc version, making this set the smallest in our evaluation. We observed both .pptx slideshows and .xlsx spreadsheet files containing viable exploits.

This vulnerability exists in the MSCOMCTL library which handles ActiveX controls. Until May 2014 (CVE-2014-1809), ASLR was not enabled on this library on all versions of Office (including Office 2013) and on all version of Windows (including Windows 7 and Windows 8). Since this library is easily locatable, it is trivial to re-use code from the same library as is used for the initial vulnerability. Due to this lack of OS level exploit mitigations and the simplicity of exploitation, DCR, including memory fragmentation, does not block this exploit. It is noteworthy that since the ActiveX controls used in this exploit are OLE files, our DCR mechanisms fragmented these objects. However, since

the access to these objects comes through legitimate means, the layout randomization provides no mitigation power.

However, some exploits are foiled because they use anomalous access to the raw document file. In the case where the raw document is accessed, the encrypted malicious payload is stored in the zip container without compression. Our re-compression of the zip streams with a randomly selected compression level defeats this file level access.

CVE-2013-3906 is a vulnerability in the TIF image format parser that permits memory corruption resulting in possible code execution. This exploit was manifest in .docx documents. Some of these exploits use ROP chains, while some use traditional shellcode. The ROP based exploits can evade DEP using a stack pivot and code re-use. Since ASLR is not enabled on the MSCOMCTL library, this library is used for gadgets in the ROP based exploits. Hence, the ROP formulation of the exploit was able to evade both ASLR and DEP as implemented at the time. However, in either the case of traditional shellcode or ROP chains, the 2013-3906 exploits are defeated through fragmentation of ActiveX objects used to implement a scriptless heap spray. The majority of the 2013-3906 samples we observed attempt to load final malware via HTTP requests. The other exploits load the final malware in memory using the same ActiveX control loading mechanism, such that these payloads are also fragmented.

The CVE-2014-4114 vulnerability is not caused by a software coding flaw, but rather policy that allows remote code to be executed. In this vulnerability, an ActiveX control allows execution of a remote .inf file which then allows execution of a portable executable. The malware is most typically downloaded via Windows file sharing (SMB/CIFS). The vast majority of these maldocs were .ppsx files which are presentations that open automatically as slide shows. There were a small number of .docx files as well. Since this vulnerability is a policy flaw, mitigations such as ASLR and DEP do not apply. Similarly, DCR does not apply even though we fragment the OLE ActiveX controls implementing the exploit. We only block a small number of these exploits because our file fragmenter identifies them as improperly formatted.

Overall, we are able to block over 75 % of the exploits in our evaluation set. If 2014-4114, which is not a traditional memory safety vulnerability, is excluded, then DCR blocks over 96 % of the exploits in our evaluation set.

6 Performance Evaluation

The core performance characteristics of DCR are the time required to perform the document transformation and the overhead incurred when opening the document. The document content randomization time was evaluated by performing DCR on a number of documents. The file open overhead was measured by timing the document reader opening and rendering the document, comparing the times from the original and randomized documents. We also validated that the view of the document presented to the user remained invariant by scripting Office to open the document and print it as a PDF. We compared the resulting PDFs

Table 2. DCR Performance

File type	Transform speed	Render overhead
.doc	68.9 Mbps	0 %
.docx	43.1 Mbps	2.9 %

created from the original and modified documents to ensure equivalence in rendering. The results of the performance evaluation of DCR are summarized in Table 2.

6.1 .doc DCR Performance

To evaluate the computational expense of performing the document content randomization, we measured the time to perform this operation on a 1000 document, 249 MB, set randomly selected from the Govdocs corpus [7]. The average time to perform the document fragmentation was 28.9 s using a single thread on a commodity server. This equates to 68.9 Mbps of throughput in a single thread. To put this execution time in perspective, we scanned the same corpus with ClamAV which required an average 28.7 s to complete. Performing this content fragmentation on a single 248 K sample (close to average document size) yielded an average 0.028 second execution time. The DCR operations are similar in cost to that incurred by a common anti-virus engine and result in a delay that should be acceptable for most situations.

To test the performance impact of DCR on document opening and rendering, this set of benign documents was converted to PDF using Microsoft Office and powershell scripting. 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 (the Govdocs corpus contains some very old documents). Other obstacles to automation included 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 automated 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 later 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. This protection built into Office triggers on some particular block layouts but the exact criteria was not discovered by the authors. If DCR is to be use widely, it would be necessary to understand and prevent triggering of this heuristic, although documents from untrusted sources (email or web) are already opened in protected view anyway.

The test data set therefore contained 836 documents totaling 197 MB. It took about 15 min 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 open times between the original and fragmented documents was 1/50th of the 95 % confidence interval. Therefore, 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 [10]. Manual review of a small number of samples also confirmed the same content in the fragmented documents as in the original documents.

6.2 .docx DCR Performance

The performance impact of .docx DCR was similarly evaluated. 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 14.3 s from which we derive a single threaded bandwidth of 43.1 Mbps. Scanning the same corpus with ClamAV required 28.0 s, nearly double the time required for our mechanism. The time to execute on a single 225 KB document, which was an average size document in this corpus, was 0.034 s.

As with .doc files, we tested the impact on rendering by converting both the original and randomized documents to PDF using Office. The outcome was a mean open time of 268.5 s for the original documents and 276.3 s for the DCR documents. This 2.9 % increase in document render time following document fragmentation is greater than the 95 % confidence interval for these trials. This slow down is very likely due to the use of higher levels of compression in the zip container. By default, Microsoft Office uses deflate compression with the fastest compression level while our randomized compression levels are spread between four compression levels. Indeed, the corpus of randomized documents was 8 % smaller than the original document set.

This performance evaluation excluded one of the 341 documents that crashed Office post randomization. This document did not appear to be malicious in any way, but simply contained a large number of ActiveX controls that triggered a bug in Office following fragmentation. We did not determine the exact cause of this crash, but did isolate it to the fragmented OLE based ActiveX objects. Since it caused a crash instead of causing a file validation/parsing error, we do

not consider it evidence of a fundamental issue with our approach, but rather a bug in Office or a special case our randomizer needs to handle.

Beyond the zip container, the vast majority of the documents in this benign corpus were not modified. Of the 341 documents, only 10 documents had OLE sub-objects on which fragmentation was performed, including the crash inducing document. Since this number was so small, the user visible representation of these samples were validated manually. Both the original and the modified document were opened and compared. Barring the aforementioned single document, randomizing the OLE objects embedded in .docx files maintained the integrity of the original document as presented to the user.

For both .doc and .docx files, the CPU time required to perform document randomization is reasonable–comparable with that of signature matching based detectors. The overhead on document open is negligible. We observed an issue with heuristic detections triggering protected view in about 12 % of .doc files. We also seemed to trigger of a bug for a single .docx file. Barring these exceptions, the transformed documents provided the same display to the user as is produced by the original.

7 Content Randomization Evasion

Document content randomization is effective against many exploits created without knowledge that it would be used. If it is to remain effective following wide-scale deployment, it must be resilient to evasion. The strength of malicious payload fragmentation lies in the number of fragments required for the payload. For fragmentation to be effective, the size of the malicious payload must be larger than the fragmentation block size.

The OLE containers used in .docx heap sprays employ a default block size of 64 bytes which is much smaller than the shellcode required for a meaningful exploit. In most of the examples we observed, the shellcode was approximately 500 bytes in length. As a comparison, we studied a collection of 32-bit windows shellcode snippets packaged with the Metasploit Framework. The functionality of these code blocks ranged from stubs that act as building blocks to complete malicious payloads. 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 comparable with the approximately 500 byte shellcode observed in the .docx scriptless heap sprays. Shellcode that provides functionality for a full malicious payload is invariably larger than can fit within the 64 byte default size restriction imposed by content fragmentation.

Current exploits are not resilient to malicious payload fragmentation because it is not currently widely deployed. However, the documented countermeasure to limits on payload size is to perform an egg hunt per payload block, which has been styled omelette shellcode [5]. 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 the 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 an identifier to facilitate proper re-ordering. 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 [25]. Of course, it may be possible to shrink the size of the omelette functionality in a given exploit and probabilistic attacks are possible.

However, if the 64 byte block size provides insufficient fragmentation, 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 streams can be changed and that the block size for the normal streams is also tunable. Ergo, this flexibility in size applies generally to both normal and small OLE streams. Due to the arbitrary tuning of OLE block sizes, it is not feasible to prevent malicious payload fragmentation by shrinking the payload size using techniques such as omelette shellcode.

In exploring malicious payload size limitations, we use shellcode because methods such as omelette shellcode are relatively well documented. The same general principles apply to other situations such as ROP based exploits. Typical ROP chains are similar in size to the shellcode, so the fragmentation of DCFR is equally effective. The ROP chains we saw in the CVE-2013-3906 heap sprays were about 1000 bytes in length. Therefore small block OLE fragmentation should be able to disrupt ROP chains as well, even if omelette style techniques are employed. The same arguments should apply to .doc file level content randomization. To the degree that exploits cannot implement malicious payload reconstruction mechanisms, then file level content randomization will remain effective.

Because document content randomization 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 DCR would still be successful.

In our study of Office documents, we saw a relatively small number of exploits that were defeated by encoding based content randomization. We observed no attempts to counter this exploit protection, and there is a dearth of studies that apply to DCER evasion. As such, counterevasion strategies are necessarily speculative.

One likely DCER evasion approach would be to anticipate the encoding and adjust the payload accordingly. Some encodings are so simplistic that they could be defeated by preparing the malicious payload so that it appears as desired post encoding. For example, if base64 were a possible encoding, it would likely be possible to prepare a malicious payload that was operable following encoding despite some restrictions in content [15]. This approach would be more difficult with encoding mechanisms such as compression which have greater complexity. Even if attackers were able to circumvent the tighter constraints caused by compression, an arbitrarily large number of compression representations are possible because of the latitude afforded in compression algorithms such as deflate. Adding a custom, entropy infusing, compressor to the existing DCR mechanisms would be operationally feasible.

Assuming there are enough possible encodings to make brute forcing infeasible, the indirect approach, analogous to omelette shellcode, would be to implement a decoder. If a very small decoder can be created, then it might be used to decode a larger payload. Trivial encodings such as hexascii or base64 may well be possible to implement in a very small decoder. Assuming an encoding method such as deflate compression is used, it is not likely that a sufficiently small decoder can be created to make this method worthwhile. We studied the compiled object size of a few common decompress only deflate implementations designed specifically for small size, including miniz and zlib's puff, and found the smallest to be 5 KB. When compared with other decoders used in exploits, this is relatively large. It seems that scenarios where using an over 5 KB decoder is useful for defeating content encoding based would be rare.

When attacked directly, DCFR's strength is driven by minimum fragment size which drives the number of fragments and the resulting number of possible permutations. It is not feasible to drop the size of a malicious payload small enough to evade the granularity provided by DCFR in OLE files. DCER's evasion resistance lies in both the constraints imposed by the encoding techniques employed and the number of possible encodings. It seems that the flexibility provided by encoding, especially compression, should allow sufficient entropy to make defeating DCER infeasible.

8 Discussion

Not all exploits are directly impacted by DCR and some vulnerabilities may be formulated to circumvent DCR. For example, the malicious documents foiled through OLE file randomization 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 could use a scripted heap spray instead of 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. Hence, DCR is enabled by environmental controls such as restrictions on web downloads, Office based protections such as disabling of scripting, and operating system controls such as DEP. If these complementary protection mechanisms are not used, DCR will not be as effective. To the degree that security controls that drive attackers to use raw file content become more prevalent, DCR should increase in applicability, including in other file formats.

Some forms of DCR are more difficult to circumvent than others because they operate much earlier in the exploitation process where the attacker has lower control over the system. For example, DCR that defeats heap sprays is more resilient than that which disrupts egg hunts that extract the final malicious payload. In our evaluation, the older exploits were interrupted later in the exploitation process while the newer exploits occur much earlier. It appears that complimentary mitigations in the operating system (ALSR and DEP) constrain exploit authors to use document content earlier in the exploits.

DCR is an attractive mitigation technique because it incurs a very low performance impact. Transforming the document requires roughly the same computational resources that are already commonly employed to perform signature matching on both network servers and client programs. DCR incurs a very small performance penalty when the transformed document is opened because this mechanism leverages the file stream reassembly routines already executed by the document reader.

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 DCR. Any situation where data is referenced indirectly, providing for multiple possible low level representations, could potentially be used to implement exploit protections similar to DCR. We focus on content fragmentation because the file formats studied here support a large degree of layout changes. Content encoding randomization is only effective in a small number of Office exploits. However, other document and media formats might not support the same level of data fragmentation but may support arbitrary encoding or compression. The PDF format is a good candidate for file level DCER to prevent raw file reflection based malware retrieval. There is an opportunity for studying the limits of DCER, especially in document formats such as PDF where there are multiple options for encoding, the encodings can be combined for the same stream, and encoding mechanisms themselves can be tweaked. For example, instead of using standard compression levels for the deflate method, one could use probabilistic Huffman coding trees and randomized use of LZ77 data deduplication. Operating system based 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. However, DCER has the potential to be computationally feasible because the content encoding already occurs.

DCR is likely to be employed in situations where many multiple repeated exploitations attempts are not easy, lowering concern of probabilistic attacks. For example, document based attacks usually require the user to take an action to view the document. Because of how client applications are used, probabilistic attacks requiring numerous attempts, similar to those employed against network daemons to defeat ASLR, are not likely to be possible.

While DCR 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. Signature validation systems could 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 between production and consumption. Document content fragment and encoding randomization are effective in scrambling exploit critical content in document files and in document reader process memory. We evaluated the ability to mitigate current exploits in Office 2003 (.doc) and Office 2007 (.docx) file formats using hundreds of malicious documents, demonstrating a memory misuse exploit block rate of over 96 %. 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 is negligible for .doc and about 3 % for .docx files. The transformed documents are functionally equivalent to the original documents, barring the exploit protections that are induced. The evasion resistance of content randomization is rooted in the number of raw content permutations possible. File content randomization should be applicable to other file formats as complementary controls force attackers to use direct access to file content to advance their attacks.

Acknowledgments. The authors would like to thank all of the reviewers for their valuable comments and suggestions. This work is supported by Lockheed Martin Corporation and the National Science Foundation Grant No. CNS 1421747 and II-NEW 1205453. Opinions, findings, conclusions and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Lockheed Martin, the NSF, or US Government.

References

1. 5 attackers & counting: Dissecting the "docx.image" exploit kit, December 2013. http://www.proofpoint.com/threatinsight/posts/dissecting-docx-image-exploit-kit-cve-exploitation.php
2. Security threat report 2014: Smarter, shadier, stealthier malware. Technical report, Sophos Labs (2014)
3. Bhatkar, S., Sekar, R.: Data space randomization. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 1–22. Springer, Heidelberg (2008)
4. Blond, S.L., Uritesc, A., Gilbert, C., Chua, Z.L., Saxena, P., Kirda, E.: A look at targeted attacks through the lense of an NGO. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 543–558, USENIX Association, San Diego (2014)
5. Bradshaw, S.: The grey corner: omlette egghunter shellcode, October 2013. http://www.thegreycorner.com/2013/10/omlette-egghunter-shellcode.html
6. Dhamankar, R., Paller, A., Sachs, M., Skoudis, E., Eschelbeck, G., Sarwate, A.: Top 20 internet security risks for 2007. http://www.sans.org/press/top20_2007.php
7. Garfinkel, S., Farrell, P., Roussev, V., Dinolt, G.: Bringing science to digital forensics with standardized forensic corpora. Digit. Investig. **6**, S2–S11 (2009)
8. Hardy, S., Crete-Nishihata, M., Kleemola, K., Senft, A., Sonne, B., Wiseman, G., Gill, P., Deibert, R.J.: Targeted threat index: characterizing and quantifying politically-motivated targeted malware. In: Proceedings of the 23rd USENIX Security Symposium (2014)
9. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, pp. 272–280. ACM, New York (2003)
10. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digit. Investig. **3**(suppl.), 91–97 (2006)
11. Li, F., Lai, A., Ddl, D.: Evidence of advanced persistent threat: a case study of malware for political espionage. In: 2011 6th International Conference on Malicious and Unwanted Software (MALWARE), pp. 102–109, October 2011
12. Li, H., Zhu, S., Xie, J.: RTF attack takes advantage of multiple exploits, April 2014. http://blogs.mcafee.com/mcafee-labs/rtf-attack-takes-advantage-of-multiple-exploits
13. Li, W.-J., Stolfo, S.J., Stavrou, A., Androulaki, E., Keromytis, A.D.: A study of malcode-bearing documents. In: Hämmerli, B.M., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 231–250. Springer, Heidelberg (2007)
14. Maiorca, D., Corona, I., 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, ASIA CCS 2013, pp. 119–130. ACM, New York (2013)
15. Mason, J., Small, S., Monrose, F., MacManus, G.: English shellcode. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009, pp. 524–533. ACM, New York (2009)
16. Pappas, V., Polychronakis, M., Keromytis, A.: Smashing the gadgets: hindering return-oriented programming using in-place code randomization. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 601–615, May 2012
17. Parkour, M.: 11,355+ malicious documents - archive for signature testing and research, April 2011. http://contagiodump.blogspot.com/2010/08/malicious-documents-archive-for.html

18. 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, CCS 2007, pp. 552–561. ACM, New York (2007)
19. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, pp. 298–307. ACM, New York (2004)
20. Smutz, C., Stavrou, A.: Malicious PDF detection using metadata and structural features. In: Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC 2012, pp. 239–248. ACM, New York (2012)
21. Srndic, N., Laskov, P.: Detection of malicious PDF files based on hierarchical document structure. In: Proceedings of the 20th Annual Network & Distributed System Security Symposium 2013 (2013)
22. Stolfo, S.J., Wang, K., Li, W.-J.: Fileprint analysis for malware detection. In: ACM CCS WORM (2005)
23. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 48–62, May 2013
24. Tabish, S.M., Shafiq, M.Z., Farooq, M.: Malware detection using statistical analysis of byte-level file content. In: Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics, CSI-KDD 2009, pp. 23–31. ACM, New York (2009)
25. Team, C.: Exploit notes-win32 eggs-to-omelet, August 2010. https://www.corelan.be/index.php/2010/08/22/exploit-notes-win32-eggs-to-omelet/
26. Team, P.: PaX address space layout randomization (2003). http://pax.grsecurity.net/docs/aslr.txt
27. Tzermias, Z., Sykiotakis, G., Polychronakis, M., Markatos, E.P.: Combining static and dynamic analysis for the detection of malicious documents. In: Proceedings of the Fourth European Workshop on System Security, EUROSEC 2011, pp. 4:1–4:6. ACM, New York (2011)
28. Wei, T., Wang, T., Duan, L., Luo, J.: Secure dynamic code generation against spraying. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 738–740. ACM, New York (2010)
29. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 559–573, May 2013