Using Memory Management to Detect and Extract Illegitimate Code for Malware Analysis

Carsten Willems
Ruhr-University Bochum,
Horst Görtz Institute for
IT-Security (HGI), Germany

Felix C. Freiling
Friedrich-Alexander University
of Erlangen-Nuremberg,
Germany

Thorsten Holz
Ruhr-University Bochum,
Horst Görtz Institute for
IT-Security (HGI), Germany

ABSTRACT

Exploits that successfully attack computers are typically based on some form of shellcode, i.e., illegitimate code that is injected by the attacker to take control of the system. Detecting and gathering such code is the first step to its detailed analysis. The amount and sophistication of modern malware calls for automated mechanisms that perform such detection and extraction.

In this paper, we present a novel generic and fully automatic approach to detect the execution of illegitimate code and extract such code upon detection. The basic idea is to flag certain memory pages as non-executable and utilize a modified page fault handler to react on the attempt to execute code from them. Our modified page fault handler detects if legitimate code is about to be executed or if the code originates from an untrusted location. In such a case, the corresponding memory content is extracted and execution is continued to retrieve more illegitimate code for analysis.

We present an implementation of the approach for the Windows platform called *CWXDetector*, which involved reverse-engineering the proprietary memory management system of this operating system. Evaluation results using a large corpus of malicious PDF documents show that our system produces no false positives and has a very low false negative rate. To further demonstrate the universality of our approach, we also used it to detect shellcode execution in *Flash Player*, *RealVNC client*, and *VideoLan Client*.

1. INTRODUCTION

No matter what particular exploitation method or target is used, the ultimate aim of an attacker is to perform *malicious computation* on the target system, i.e., to execute machine instructions whose type and order are under the complete control of the attacker. Usually, malicious computation is conducted using *illegitimate code*, i.e., code that was not intended to be executed, neither by the developer of the exploited software nor by the end-user of the system.

ACSAC '12 Dec. 3-7, 2012, Orlando, Florida USA

Such code is usually injected into the target system using external data like network traffic or application files.

As a countermeasure to this threat, operating systems try to *prevent* the execution of illegitimate code using techniques like *Data Execution Prevention* (DEP) [17] and *Address Space Layout Randomization* (ASLR) [33]. However, prevention alone does not help in the process of analyzing malicious code since we also want to analyze what an attacker attempts to do when compromising a system. Therefore, it is necessary to develop mechanisms that *detect* and *extract* illegitimate code from malicious data. A consecutive analysis of the extracted data then can assist in developing new protection techniques and creating signatures for zero-day malware until patches are available.

In this work, we present an approach to automatically detect and extract illegitimate code by instrumenting the memory management features of operating systems. Roughly speaking, the idea is to mark certain memory pages as non-executable. This ensures that upon execution of code in these regions the page fault handler of the operating system is called. This usually suffices to *detect* illegitimate code. However, to *extract* illegitimate code, we modify the page fault handler so that the memory page that caused the page fault is written to a dump file for later analysis.

The power of our approach stems from its simplicity in using the page fault handler of the operating system itself. Therefore, the challenge is to evaluate its effectiveness in practice. This means to evaluate it for the Windows platform, since this is still the major target of illegitimate code today. To that end we have implemented *CWXDetector*, a new tool for the analysis of malware on Windows. Unfortunately, Windows is a closed-source operating system and without proper documentation it is a difficult task to integrate custom functionality into the kernel. Therefore, we had to perform a lot of substantial reverse engineering regarding the internal memory management mechanisms of the Windows kernel. Due to space constraints, most findings of this research are covered in an accompanying technical report [35]. Additionally, due to space limitations we have published an extended version of this document in a second technical report [36]. This technical report covers more implementation details of the tool *CWXDetector* and comprehensive analysis results.

We evaluated our approach by considering the task of detecting and extracting illegitimate code in/from a particularly relevant class of application files, namely those in Adobe's Portable Document Format (PDF) [22]. Our approach proves to be very effective. We analyzed a set of 7,278 malicious PDF documents using a set of vulnerable versions of Adobe Reader and achieved a detection rate of 93.2%. This can be regarded as a lower bound for our method since many of the investigated PDF files appeared to be broken although they were flagged as malicious by Antivirus products. We also analyzed the same amount of benign PDF documents, resulting in a (false positive) detection rate of 0%. Furthermore, our detection results compare favorably to those of application specific detection tools like *Wepawet* [6], *pdf examiner* [34] and *ADSandbox* [7], but outperforms them by being generic and – to some extent – also being capable of detecting zero-day exploits. To further demonstrate the universality of our approach, we also used it to detect shellcode execution in *Flash Player*, *RealVNC client*, and *VideoLan Client*. Especially the Flash file was complicated since it consists of a Flash file embedded in a Word document, which is an attack vector that is hard to analyze with concurrent approaches like ShellOS [27].

In summary, the contributions of this work are as follows:

- We present a generic and fully automatic analysis approach to detect the execution of illegitimate code and extract such code upon detection.
- We successfully evaluate our approach using malicious PDF documents as example and show that we can improve state-of-the-art tools. In addition, our approach can also be applied to other kinds of malicious documents as we demonstrate in three case studies.
- Furthermore, we have reverse engineered essential parts of the memory management internals of the Windows operating system. Due to space constraints, we only cover the relevant aspects in this paper but full details are available in an additional technical report [35].

2. RELATED WORK

There has been a lot of work on shellcode analysis in the past and we review work that is closely related to ours. Note that our approach is not meant to *protect* a system, but to monitor and analyze the execution of illegitimate code in an analysis environment. To that end, our system even disables some security measures like DEP. Nevertheless, there are some similarities in other preventive and analysis techniques that we now discuss.

2.1 Preventive Measures

A large body of related work mainly aims at the *prevention* of malicious code execution, mostly following the *reference monitor* approach. Many such methods are directly integrated into contemporary compilers and operating system [35]. However, often newly introduced protection techniques are incompatible to existing old applications and, therefore, can be disabled by the applications themselves or are deactivated per default. *Microsoft's EMET tool* [16] tries to overcome this problem by allowing a process-specific configuration of these protection methods and their enforcement. Other methods restrict memory write operations or control transfers. Kiriansky, Bruening and Amarasinghe [13] as well as Abadi et al. [1] use code rewriting techniques to implement their restrictions. Instead of modifying parts of the operating system, they inline checks into the executed applications. Unfortunately, they mostly lack the capability of handling self-modifying and dynamically created code and they are unable to detect control flow transitions that occur due to *structured exception handling* and not due to a regular branch instruction. Another difference to our work is that these solutions terminate the protected process in the event of a security violation, and are not able to produce any further analysis data. Seshadri et al. propose *SecVisor*, a tool that is capable of ensuring code and execution integrity by utilizing a tiny hypervisor [23]. In contrast to our approach, they aim at kernel code and present a solution for Linux systems.

To some extent our approach is similar to DEP [17], which also employs the *no-execute/execute-disable* (NX) flags of the page table entries to block the execution of certain memory regions. Nevertheless, there are significant differences to our system, since we are aiming at the analysis of malicious code and not at preventing its execution. Therefore, we intentionally permit the execution of illegitimate code after extracting it. Furthermore, we do not only consider the type of memory when deciding which should be monitored respectively executed, but we also check the initiator of memory related modifications and allocations. Additionally, we restrict the unintended allocation of executable memory due to programming errors, which would not be detected by DEP and enables us to interact if code is copied into and executed within those memory regions.

In summary, all of the previously described measures aim at the prevention of malware execution, but offer no assistance in their further analysis.

2.2 Detection of Illegitimate Code

The detection of illegitimate code is an extremely difficult problem today. Early attempts relied on static signatures [11], but had to be improved due to the heavy use of polymorphism, encryption and other obfuscation methods. More enhanced methods try to detect certain invariant parts of the shellcode, e.g., Akritidis et al. [2] search for the typical "sled component" in such code. Others have used heuristics in combination with *dynamic* analysis methods to detect illegitimate code. For example, machine learning methods have been used to deal with the variable parts, e.g., Payer, Teufl and Lamberger utilize a neural network [18] in combination with *execution chain evaluation*. Polychronakis, Anagnostakis and Markatos [19] use emulation to detect an ongoing decryption process which is typical for polymorphic shellcode. Also Baecher and Koetter [3] use an emulated environment to identify and isolate shellcode with the help of *GetPC heuristics*. In order to overcome the drawbacks of such emulators, Snow et al. use a KVM hypervisor to execute instruction sequences that are found in network streams or arbitrary buffer content [27]. By applying sophisticated runtime heuristics they are able to detect malicious code. Overall, and in contrast to *CWXDetector*, these signature- and heuristics-based approaches are not fully generic and have to be extended when new anti-detection measures of malicious code come up. Additional hypervisor-assisted approaches to detect and analyze the execution of malicious code have been presented by Dinaburg et al. [8] as well as Litty, Lagar-Cavilla, and Lie [14]. Different to ours these systems do not modify the operating system itself, but introspect solely from the *outside*, i.e., the hypervisor. It has been observed before that the memory system itself can be used to detect illegitimate code execution. For example, the *PaX project* [32] proposes several different measures to implement non-executable memory — even on architectures with no hardware support for that. Similar detection mechanisms can obviously be realized with hardware-DEP, like explained in the previous subsection.

2.3 Extraction of Illegitimate Code

Several solutions aiming at the extraction of illegitimate code exist, especially for automated unpacking of malware. These mechanisms usually interact deeply with the memory management of the underlying operating system and try to detect the execution of memory regions which have been written to beforehand. *OllyBone* [31] implements this by instrumenting the *translation lookaside buffers* [10]. Since *OllyBone* is a debugger-plugin, it imposes all the disadvantages of debugger-driven malware analysis, e.g., its detectability. Another disadvantage that contrasts it to our approach is that it is a semi-automated process, in which execution is stopped at the first occurrence of malicious instructions and the human analyst has to continue with further extraction steps. Finally, it is not able to deal with dynamically allocated memory regions (since it focuses on Windows PE sections).

OmniUnpack [15] uses an approach similar to the *PAGEEXEC* method proposed by PaX, i.e., the *User/Supervisor* page table flag is used to automatically break on the execution of certain monitored pages. In order to decide whether executed and previously written memory should be considered as malicious, an external detector is used to scan memory after unpacking for the existence of malicious code. That detector, again, has to use signatures or heuristics that generate a lot of false positives, especially if a JIT-compiler is involved. Finally, executed memory is only considered if a critical system call is executed afterwards. Our approach uses a more effective approach, based on the concept of *trusted callers*, that results in much better detection results.

Renovo [12] runs the sample within the emulated environment TEMU [28] and maintains shadow memory to track written memory regions. Since this cannot be done on a native system, it cannot be realized *without* system-emulation. Again, like with debuggers, this enables the monitored malware to detect the synthetic environment.

A similar approach to ours has been published by Porst in form of a *Pin-tool* [20]. It uses a simple heuristic that identifies and dumps shellcode, if its instructions are not located in the code section of any loaded module, but on the stack or the heap.

3. MODEL AND DEFINITIONS

We now specify our attacker model, define the term *illegitimate code*, and further concretize our two aims: the detection and extraction of executed illegitimate code.

3.1 Attacker Model

In this work we assume a remote attacker that provides some malicious piece of data in order to exploit a vulnerability in some handling application resulting in the execution of shellcode. This data may have arbitrary form, e.g., a specially crafted PDF document, a Flash file, or a malicious input packet to some network application.

We are aware of the threats posed by *return oriented programming* [24] (ROP) or JIT-spraying [4, 25] techniques, but nevertheless assume that an attack does not *fully* consist of such code. To the best of our knowledge, we are not aware of any incidents in the past that used single staged full-ROP/JIT-sprayed attacks.

3.2 Illegitimate Code

Our approach enforces the partitioning of executable memory into regions that contain legitimate code and those that may contain illegitimate one. Additionally, it monitors and reacts on the execution of instructions that are located in illegitimate code regions. Intuitively, all code that belongs to the operating system or to a known application is legitimate. For realizing this distinction, we first partition the files of a system into a set of trusted files and a set of untrusted ones. We assume that such a distinction is given, e.g., by defining all files in a freshly installed system as trusted. For dealing with dynamically created code, we identify those code portions contained in trusted files that should be allowed to allocate executable memory. Accordingly, we partition each trusted file into trusted memory modification functions and untrusted ones. Again we assume such a distinction is given, e.g., by defining all required code emitting memory functions of the operating system as trusted and all others as untrusted. Then, *legitimate code* (LC) is code which is either contained in a trusted (system or application) file or that was dynamically created by any of the trusted functions from one of those files.

We define *illegitimate code* (ILC) as code that is not legitimate. Intuitively, ILC is code which would not be executed if the operating system and the installed applications would function properly. In practice it is code that is either injected by or constructed on behalf of an attacker by some malicious piece of software or data. Therefore, ILC is similar to *shellcode* in its current understanding.

3.3 Problem Statement

The aim of the system described in this work is to perform two tasks in an automated way:

1. *Detect* the execution of illegitimate code, and
2. *extract* that code, i.e., dump all relevant memory pages to a file, for a later in-depth analysis.

4. APPROACH: INSTRUMENTING THE PAGE FAULT HANDLER

In the following we describe our approach and sketch our Windows implementation of it. More details are available in two accompanying technical reports [35, 36].

4.1 Enforcing an Invariant

Based on our attacker model, no matter what kind of exploit is used in an attack, the resulting effect is always the execution of illegitimate code like we have defined above. To that end, when a vulnerability is exploited, the control flow is redirected to some code hosting location on the stack, the heap, or in a static data area.

Throughout our approach, we establish and maintain the following invariant condition: *all ILC resides in* non-executable *memory.* As an effect to this invariant, all execution attempts of ILC will result in the invocation of the *page fault* handler of the operating system. By implementing our own custom page fault handler, we are able to react on such attempts appropriately.

4.2 Trusted Files and Functions

To establish the invariant, we need to identify the set of trusted files and functions. For simplicity we trust all files which have been already existing when we start our analysis, and distrust all files which were created or modified during later system operation. To achieve this, we need to keep track of file manipulation operations. Therefore it is necessary to intercept ("hook") calls of system services that are used to create or open files with write access.

For each trusted file we further define a set of *trusted memory modification functions*, which contains all the functions that are allowed to dynamically create executable memory or modify the protection settings of already existing memory to being executable. The set of all such functions from all trusted files is called *trusted callers*.

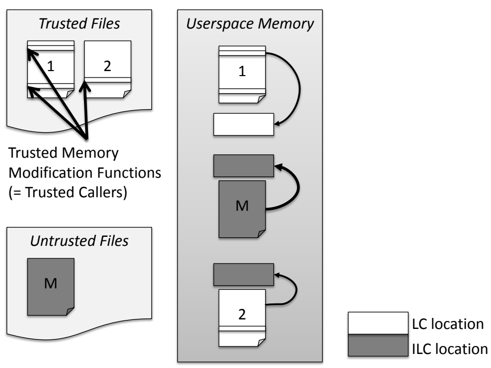

Figure 1: Trusted Memory Modification Functions

Figure 1 illustrates our understanding of trust, showing an example with two trusted and one untrusted file (left side of the figure). While trusted file 1 contains two trusted memory modification functions, trusted file 2 only has one. Obviously, the untrusted file can not contain any trusted function at all. The simplified version of the userspace memory (right side of the figure) shows that all three files have been mapped into the virtual address space. The memory related to the trusted files only contains trusted code, hence constitutes LC memory, whereas that memory of the untrusted file may contain illegitimate code. Furthermore, each mapped module has allocated one dynamic memory area, pointed to by the corresponding arrow. That area belonging to file 1 was created by a trusted caller and, hence, may contain legitimate executable code. However, the memory created by the untrusted caller from file 2 as well as the region created by the untrusted file may contain ILC and, therefore, are marked as ILC locations.

4.3 Memory Protection Modifications

Obviously it is necessary to intercept attempts to modify the memory protection, because this can result in executable memory. This is realized by hooking critical system calls. Inside these hook functions we enforce the following properties to maintain our invariant:

- only trusted callers can allocate executable memory,
- only trusted callers can modify existing memory to being executable, and
- only trusted files can be mapped into executable memory.

The parameters of all system calls that violate these rules are manipulated transparently such that the resulting memory regions become *non-executable*. In summary, only trusted files can be loaded into executable memory and only trusted callers can create executable memory. There is one exception: even if a *trusted caller* tries to modify the memory protection, intervention may be necessary under special circumstances: if the related target memory belongs to a mapped trusted file and should become writable, the executable right has to be removed. This enforces the $W \oplus X$ property [17]. In general, all legal linkers should produce files which fulfill this requirement anyway. Nevertheless, we enforce it on our own to also handle files securely which violate it by intent or accident.

The realization of making memory non-executable strongly depends on the underlying system architecture and also on the operating system. Many contemporary CPUs offer an NX protection flag on a page level. Nevertheless, this feature can be only used for valid page table entries (PTEs) and, therefore, in most cases some additional OS memory objects may have to be modified as well (more details are available in the technical report [36]).

4.4 Custom Page Fault Handler

The heart of our detection method is the *custom page fault handler*, which reacts on the attempt to execute memory regions which we have marked non-executable beforehand. As described above, all necessary prerequisites are already done when new memory is allocated or the protection of already existing one is tried to be modified. Accordingly, the custom page fault handler only has to *wait* until a protection-related page fault is triggered. In that event, we have to check if the occurred page fault is really related to our system modifications. If so, we have detected the execution of illegitimate code and, as a result, copy the content of the related memory page to a dump file. Furthermore, we log the current instruction pointer to indicate the particular instruction that should be executed within the dumped page. After that, the related memory region is modified by us to being executable, such that the current and all further execution attempts for this page will become successful. This is done because we do not want to stop our analysis process once the first illegitimate instruction is found. Finally, we resume the current process and wait for the next fault.

4.5 Multi Version Dumping

In order to avoid detection, shellcode is very often built by multiple stages that are organized like a russian stacking doll (*matrushka*). Each stage is only a small stub that deobfuscates or decrypts the next stage and then transfers control to it. Since the effective malicious instructions are mostly contained in the final stage, it is desirable to unpack it automatically. Therefore, we have developed an additional feature called *multi version dumping* (MVD), in which different versions of each executed page may be created. To that end, an internal copy of each dumped memory page content is stored and if the content is modified later on, another dump file is created. By comparing two consecutively created dumps, we can isolate those parts that have been modified and infer the decrypted shellcode instantly.

Notice, that not every shellcode is multi-staged. Therefore, sometimes only one dump file is created for a detected ILC containing memory page, and sometimes two or more.

If multiple versions are created, we mostly are interested in the final one, since we assume that it contains the fully decrypted code. However, also considering the intermediary stages may be reasonable, i.e., to gain knowledge about the used unpacking method.

4.6 Windows-based Implementation

We have developed the tool *CWXDetector* as a concrete implementation of our approach for the x86 version of Windows XP. One reason for choosing this particular version was the availability of a large sampleset of malicious documents for that target OS. Nevertheless, our system could easily been migrated to (at least the 32bit versions of) Windows Vista or 7. Due to space constraints, a detailed explanation of this tool and its internals can be found in the accompanying technical report [36]. In the following, we only provide a brief overview of the involved tasks and difficulties. As OS platform we have utilized the PAE kernel version of Windows, since this one supports the NX page table flag that we need to realize non-executable memory. Although that kernel version was originally intended to support physical memory that is larger than 4 GB, it is nowadays used on all installations of the 32 bit Windows XP version that have DEP enabled. To realize our approach we had to

- define trusted files and trusted callers,
- implement hook functions for memory allocations and protection modifications,
- implement a custom page fault handler to handle ILC execution, and
- additionally modify essential system functions to support our approach.

Since Windows is not an open source operating system, a lot of reverse engineering had to be performed previously, especially on the underlying memory objects like VADs and PPTEs. The detailed findings of this work are explained in the second accompanying technical report [35]. Though the implementation of *CWXDetector* seems to be straightforward, the unavailability of the Windows source code posed enormous difficulties when intercepting kernel system calls, customizing the page fault handler, and dynamically patching OS-controlled memory management resources.

5. ANALYSIS OF PDF DOCUMENTS

For illegitimate code detection and extraction, *CWXDetector* has to perform a dynamic analysis and actually parse and process the malicious content with the intended client application. To further illustrate and evaluate its effectiveness, we apply it to the analysis of PDF documents. Malicious documents as attacking vector have become very popular in the past years, and especially the PDF format is a commonly used medium for malicious content. One reason for that is its extensive feature list: the programming languages *Javascript* and *Actionscript* can be used within and many different object types like images, sounds and even executables can be embeeded. Accordingly, the underlying codebase is very complex and hence error-prone. For example, the latest PDF reference [9] contains 756 pages and *Adobe* already has published several extensions to it.

Since dynamic analysis in general is *incomplete*, we test each PDF sample in different viewer applications and then combine all the findings. Though not usual, a malicious functionality may only be triggered on a certain user action or input. For correctly analyzing also those files, user-simulation could be employed as an extension to the existing functionality.

We set up multiple virtual machines with 32 bit Windows XP SP2 and installed a different PDF viewer application on each of them. In particular, we used *Adobe Acrobat Reader* 6.0.0, 7.0.0, 7.0.7, 8.1.1, 8.1.2, 8.1.6, 9.0.0, 9.2.0 and 9.3.0. For comparison we also set up one machine with *Foxit Reader* 3.0.0 for which also some vulnerabilities are known. This particular application and version set was chosen to cover the most of the known vulnerabilities for PDF documents, but it should be mentioned that it may not be optimal nor have full coverage for all known existing exploits. Each PDF sample was then analyzed in all of those machines in parallel. During the analysis we performed the following steps on each machine separately:

- We installed the customized page fault handler and the system hooks.
- We disabled DEP for the viewer application, since otherwise the execution attempt of non-executable code would crash the process and we would not have any possibility to intercept it.
- We opened the document in the viewer application.
- If new memory was allocated or existing memory was modified during execution, we enforced the invariant from Section 4.1.
- If the execution of illegitimate code was detected, we dumped the associated memory page to a file and modified the related PTE to being executable. We then checked the dumped memory page for typical patterns of illegal code [36]. In case such a pattern was found, we labeled the execution with *PATTERN*.
- If a new process was created by the PDF viewer, we marked the execution with *PROCESS*. We prevented the spawning of additional processes since we are only interested in analyzing exploits in the PDF viewer application itself.
- If a dialog window was shown by the PDF viewer, we assigned the label *DIALOG* and additionally logged the contents of the window. We then simulated a user input to close the window and continued viewing the PDF document.

For scalability reasons the analysis process was stopped after a specific timeout, which was set to two minutes in our experiments. As almost all known malicious PDFs trigger their malicious operations instantly when viewing the first page of the document, it is safe to assume that we trigger most of all malicious shellcodes after this short amount of time. In many cases the viewing application was terminated prematurely, long before that timeout was reached. In that case we marked the execution as *CRASH*. Finally, and if none of the aforementioned labels have been assigned, the case was labeled as *NOTHING*.

Overall, every PDF file ended up with a combination of two labels (d, c): the first label d determined whether illegal code execution was detected or not, and the second label c was either *PATTERN*, *CRASH*, *PROCESS*, *DIALOG*, *NOTHING* as defined above. Since different PDF viewers can react differently to a single PDF file, we needed to aggregate all the different results into one overall value. We defined a lexicographic total order on the tuples as follows: $(d, c) > (d', c')$ if and only if either d had detected illegal code and d' not, or (if $d = d'$) $c > c'$ according to the following ordering:

$$\mathit{PATTERN} > \mathit{CRASH} > \mathit{PROCESS} > \mathit{DIALOG} > \mathit{NOTHING}$$

As the final result, we used the highest occurring value as combined overall value.

As stated above, we further have to determine the set of trusted files and trusted callers. Since we start with an uninfected clean system, it is safe to trust all files that already exist when the analysis starts and distrust all modified and created ones. The trusted callers are determined in a semi-automated way: we start with a clean whitelist and each time executable memory is generated, we manually inspect the call stack and verify if the caller is legitimate. This only has to be done once for a given use case, for instance with the help of a known benign PDF document. For this particular experiment we came up with two functions (`LdrpSnapIAT` and `LdrpSetProtection`) in the *Windows Native Library (ntdll.dll)* and one in the JIT-compiler of *Acrobat Reader (authplay.dll)*. It is obvious that this incremental approach is fail safe, since we only can get *false positives* if we have forgotten particular trusted callers, but will never create *false negatives* if we set up our trusted caller list correctly.

In Section 6 and 7 we describe our findings and the results of our experiments in detail. Mostly we are interested in the information if a viewed malicious PDF document triggers the execution of ILC or not. If we are able to detect such an attempt, we call our result a *true positive*. If we fail to detect it, we call it a *false negative*. If on the other hand, a benign PDF document is analyzed and in reality no ILC is executed at all, but our system erroneously reports ILC execution, we call this a *false positive*. Finally, a *true negative* stands for such a case, in which our system correctly does not report ILC execution.

6. DETECTION EVALUATION

We have evaluated the detection quality of our system by means of two different experiments. First, we have performed a comprehensive analysis of PDF documents. For that purpose we have created two sets of PDF documents of size 7,278 each (a *benign* set and a *malicious* set) and used *CWXDetector* to analyze these files. We have developed heuristics to measure the correctness and completeness of our findings. We further have compared our generic system against several other analyzers, that use application-specific knowledge to analyze PDF documents. In a second experiment we briefly illustrate the universality of our approach by applying our tool on malicious *Flash* documents and network packets.

6.1 Benign PDF Sampleset

In order to test the *false positive* rate of our approach, we obtained a set of known benign documents from the TOP 5000 Internet sites from `www.alexa.com`. We have then selected those files that contain as much different PDF features as possible. Accordingly, analyzing them enables us to monitor various different behaviors, in form of code coverage of the PDF viewing application and its plugins. The resulting set has the following characteristic: altogether it contains 7,278 samples, 600 of which contain *Javascript*, 782 contain *AcroForms*, 1,573 samples have an *OpenAction*, and 751 some *AdditionalAction*. All samples have been verified by the publicly available AV service Virus Total [26] and in 3 cases one or more supported scanners returned a positive result. We checked those samples by hand and did not find any malicious content within them. Therefore, most probably these detections are AV false positives.

We ran our system on the benign sample set. As a result, we did not detect *any* single ILC execution, resulting in a *false positive* rate of 0% for this particular set. To speed up the analysis, we only used the three "most vulnerable" PDF viewer applications for the benign sample set (namely *Adobe Acrobat Reader 7.0.7, 8.1.1* and *9.0.0*). Due to the achieved false positive rate of our experiments of zero, we can assume that it will not increase dramatically by using more different viewers.

6.2 Malicious PDF Sampleset

We obtained a set of 7,278 known malicious PDF documents from a well-known AV vendor. The set consisted of all their valid incoming PDF samples from January 2011. These samples originated from different sources: *70.0%* from sample sharing between AV vendors, *24.0%* found in the wild, *4.8%* from multiscanner projects and *1.2%* from intercepted botnet traffic. We checked all samples with Virus Total [26] which confirmed that all of them were indeed malicious. We ran our tool on the malicious sample set and were able to detect and extract ILC in 93.2% of all cases. The detailed analysis results are shown in Table 1 and are explained in the following section.

Table 1: Overall Detection on Malicious Samples

	ILC detected		**no** ILC detected	
	Samples	Fraction	Samples	Fraction
PATTERN	6,658	91.5%	—	—
CRASH	20	0.3%	15	0.2%
PROCESS	83	1.1%	33	0.4%
DIALOG	0	0.0%	295	4.1%
NOTHING	20	0.3%	154	2.1%
Total	6,781	93.2%	497	6.8%

6.3 Discussion

Given the benign and malicious data sets as stated above we end up with a false positive rate of 0% and a false negative rate of 6.8%. However, we have seen a lot of samples which were broken and, though containing malicious content, were not able to produce malicious functionality when loaded into a PDF viewer. Furthermore, some samples only triggered their exploit when using a particular PDF application that was *not* contained in our application set. Finally, there exist some samples that perform malicious behavior that is not based on shellcode, but instead uses built-in features of the PDF viewer application. For instance, some samples redirect to malicious websites or exploit software bugs in third-party applications that can be started directly from within a PDF document. To fortify our results and argue why we assume an effective false negative rate that is much lower than the measured one, we analyzed the documents from the malicious data set in more detail.

6.3.1 Results without ILC Execution Detection

Our system failed to detect ILC execution for 497 documents from the malicious sample set. Despite this fact we assume that it works correctly and those *undetected* samples either target at an untested PDF viewer application or simply corrupted and do not function at all. Since we are not able to manually analyze about 500 samples, we have developed heuristics to prove (or at least find hints for) our assumption.

We first checked those 15 files with the *CRASH* label and found that all of them performed invalid memory accesses during parsing the document. Without further investigation we can not be sure if they are really malicious or simply corrupted. Nevertheless, since they crash before executing any shellcode they seem to labeled correctly by our system.

We then examined those 33 samples that created a new process (*PROCESS*) and we discovered that in all cases regular built-in features were used for that purpose. The started applications were the *Command Shell*, *Internet Explorer*, and, *Outlook Express*. In the latter two cases the used command line parameters [36] were specially crafted to enforce a parsing error and arbitrary code execution within those started applications.

After that we investigated the 295 cases that where labeled with *DIALOG*. Most of the dialogs contained error messages [36] of the parsing engine, which state that the PDF structure itself or some embedded JavaScript-code was invalid. We assume that either the corresponding PDF documents were corrupted or that we simply have not used the expected environment to trigger the malicious functionality. In addition to this, there are some PDF exploits that solely are based on social engineering, in which the user is tricked to respond in a particular way to the shown dialog. For example the warning message for starting a new process is obfuscated in a way such that the user will not notice that a new process will actually be spawning when he clicks the *OK* button [29, 30]. Overall, it is unclear whether these files were indeed malicious or not. However, it is probable that none of them executed any form of ILC during execution.

Finally, 154 samples did not perform any suspicious activity at all (*NOTHING*). Obviously, this does not mean necessarily that the samples are harmless. It just means that under the given environment they behaved benign. We manually checked a random set of 30 samples of this class and we found that they do not contain any working exploit at all. A reason for the AV scanners to mark them as malicious may have been that they contained malicious signatures as a pure coincidence or were the results of failed attempts to create working malicious PDF documents.

In summary, it is safe to assume that in all 497 cases really no ILC was executed in any of our used environments. So while our method failed to flag these samples as malicious, it succeeded in detecting the execution of ILC: since no ILC was executed no detection was triggered. In this sense all negatives seem to be true negatives, and so after careful consideration one could also claim a false negative rate of 0% for our approach and the examined sample set.

6.3.2 Results with ILC Execution Detection

For completeness, we also discuss the cases where our method detected ILC execution. In order to show that these cases are correct, we have to ensure that all dumped memory really consists of illegitimate code and that no prior ILC execution has been missed. Again, confirming this for each individual case is impossible due to time restrictions and, therefore, again we used heuristics to get trustful hints for the correctness. In the 6,658 cases that are labeled *PATTERN* we confirmed the presence of known shell code patterns in the dumped memory pages. Therefore, we can be sure that in fact ILC was executed.

We checked those 20 samples that crashed the PDF viewer after the ILC detection (*CRASH*). This is also an evident sign for (a partly failed) malicious activity. In such cases the exploit did not work well, either because it was badly programmed or because it did not discover the environment which was needed to work correctly. Even if the samples do not succeed to perform any reasonable malicious operation, we know that the observed code execution really is related to ILC and, accordingly, is no false positive.

Next we investigated those 83 documents that spawned new processes after the ILC execution (*PROCESS*). This can also be seen as a clear sign of malicious activity, if the started process is none of those mentioned in section 6.3.1, which could be started by legitimate built-in features of the PDF viewer. We have verified that none of the spawned processes belong to those exceptions, but all fell into one of the following three categories[36]: it was either tried to start an extracted or downloaded program (with malicious content), to open a created second PDF document (to hide the maliciousness of the initial document) or to gather essential information about the exploited system. Accordingly, we can be sure that all of these samples really have executed shellcode.

Finally, there were 20 remaining samples with ILC execution, for which all of our previously described heuristics failed. Hence, we were not able to tell anything about their maliciousness in an automated way and, therefore, we checked them manually. All of these files really executed ILC, from which some was simply not working correctly and other did not even consist of valid machine instructions at all. We can only guess the reasons for that: most probably, some of these samples just were written badly or got corrupted due to some transmission error. Others may find some unexpected environment and, accordingly, do not function properly. Anyway, we had manually assured ourselves that in all cases ILC was executed, no matter if the resulting operations were valid or not.

6.3.3 Detection Summary

Though we are not able to manually verify all the samples we have analyzed with our system, the results shown in the previous subsections lead to the conclusion that our approach works well. If we aggregate all our findings with illegitimate code execution, we get a minimum detection rate of 93.2% for our particular set. If we furthermore assume that there is a serious fraction of samples that does not contain working shellcode for any of our used environments, we can assume an error-corrected detection rate that is much higher in reality.

6.3.4 Detection Results of Other Analyzers

To evaluate the effectiveness of our solution, we compared our results against those from the popular application specific analyzers *Wepawet* [6], *pdf examiner* [34], and *ADSandbox* [7]. All of these analyzers combine static and dynamic approaches, i.e., they parse the PDF document structure, extract potential malicious pieces, and then analyze them by different means. Depending on the severity of the findings, each analyzed sample is labeled as either *benign*, *suspicious*, or *malicious*. Furthermore, additional comprehensive analysis data is generated, i.e., information about the embedded objects, like PE files, URLs, or known exploits.

Wepawet [6] combines machine learning techniques with emulation. It extracts specific features while emulating Javascript code and then compares them against a set of pre-

Table 2: Detection Results on Malicious and Benign Samples

	Malicious Sampleset				Benign Sampleset			
Analyzer	Malicious		Suspicious		Malicious		Suspicious	
Wepawet	4,737	(65.1%)	1,739	(23.9%)	0	(0.0%)	0	(0.0%)
pdf examiner	6,089	(83.7%)	1,108	(15.2%)	82	(1.1%)	246	(3.4%)
ADSandbox	2,360	(32.4%)	255	(3.5%)	0	(0.0%)	3	(0.1%)
CWXDetector	6,781	(93.2%)	0	(0.0%)	0	(0.0%)	0	(0.0%)

viously learned known benign profiles. It also uses a set of signatures to detect anomalies, which are *not* based on Javascript. *pdf examiner* [34] as well extracts all embedded objects and streams from the PDF document and decrypts them if necessary. It then uses signature scanning to detect known malicious patterns and *libemu* [3] to detect shellcode. From all the findings a score value is calculated, that decides about the ultimate outcome of the analysis. Besides this value, a sophisticated report is generated that highlights suspicious and malicious parts of the PDF document. In contrast to the two aforementioned analyzers, *ADSandbox* [7] solely aims at the detection of malicious Javascript within the PDF documents. For that purpose, all Javascript snippets are extracted and then executed in an isolated environment. Subsequently, heuristics are utilized to decide from the executed operations and the involved data about the maliciousness of the particular sample. ADSandbox can be used with several different configurations settings, but we have used the defaults for simplicity.

When comparing the results of these application specific analyzers to those created by *CWXDetector*, one has to take into account that our tool only triggers on the actual execution of ILC. Accordingly, it is only capable to label a sample as *benign* or *malicious*, but not as *suspicious*. Furthermore, all of the other analyzers work with heuristics and patterns and, therefore, are presumably not able to detect all kinds of unknown exploiting methods.

Table 2 summarizes all results for the detection of malicious samples. When only taking those samples into account that have been marked as *malicious*, our approach yields better results than those of the application specific analyzers. However, also when considering the suspicious samples as well, our results are comparable, i.e., 89.0% (Wepawet) and 98.9% (pdf examiner) vs. 93.2% (*CWXDetector*). Furthermore, we know that a significant part of those malicious samples which have been not detected by *CWXDetector* are corrupted and, hence, not executable at all. A signature scanning based approach is obviously able to detect malicious parts within those broken files, but our method obviously fails on them. ADSandbox does not deliver a very high detection rate on our malicious sample set since its main focus is to analyze JavaScript code only.

When it comes to the false positive rate, the comparison is rather simple (see Table 2). As described above our approach does not produce any false positive on the used sample set. Also the other three analyzers generate good results: 0 false positives for Wepawet as well as for ADSandbox. There is a trade-off in detection accuracy of pdf examiner, since this detector has the best detection rate but also produces the most false positives of around 4.5%, which still is an acceptably low number.

6.4 Additional Experiments

In order to emphasize the universality of our approach, we briefly present detection results from different applications. We used *CWXDetector* to analyze malicious *Flash* documents as well as malicious network packets that exploit vulnerabilities in the *Real VNC viewer* and the *VideoLan Client (VLC)* tools. A more detailed description of our findings is contained in the extended version of this paper [36].

6.4.1 Flash Documents

As additional example for shellcode containing documents we have analyzed two malicious *Flash* files. The first one was created with the help of the *Metasploit Framework*[21] and the other one was *JOB_DESCRIPTION.doc*, a sample that was found in the wild and taken from the *Contagio Dump Archive*[5]. Both samples exploit the *CVE-2011-0611* vulnerability of the *Flash Player* version 10.0.45 by executing a malicious *ActionScript* that results in arbitrary code execution. Since both *Flash* samples were embedded in *Word* documents, we has to use *Microsoft Word Professional 2010* to actually view them. Both samples were detected correctly by our tool and in both cases the memory pages containing the ILC were dumped.

6.4.2 VNC Client

The traditional way to execute shellcode on remote systems has been to embed it into network packets and exploit vulnerabilities in the parsing application. The increased awareness and improved security of contemporary network applications and operating systems has driven the attackers to shift to malicious documents. However, in order to illustrate the effectiveness of our generic approach we show that it is capable to detect malicious code execution also in this traditional context. Therefore, we used the *Metasploit Framework* to setup a network server that accepts connections from *VNC* clients. After executing the *RealVNC client version 3.3.7* in combination with our *CWXDetector*, we connected to that server and received a specially crafted network packet. This packet contained an exploit for the *CVE-2001-0167* vulnerability of the *RealVNC* application. *CWXDetector* detected the execution attempt of the first contained shellcode instructions and dumped the related memory to a file.

6.4.3 VideoLan Client

An additional analysis of a network application exploited by a malicious network packet was performed with help of the *CVE-2010-3275* vulnerability, which existed in the versions 1.1.4 up to 1.1.7 of the *VideoLan Client* (VLC). By accessing a specially crafted *.amv* file, VLC can be crashed by the usage of an invalid pointer and arbitrary code can be executed. With the help of *Metasploit* we again set up a server that generated such malicious data and offered it over the network for download. Unsurprisingly, our detection mechanism triggered again and extracted the malicious instructions once the embedded shellcode was about to be executed.

7. EXTRACTION EVALUATION

In this section we try to measure the *quality* of the extracted ILC. For that purpose we determined the percentage of contained valid x86 instructions (*code ratio*) and the amount of data in terms of embedded strings (*data ratio*). Since shellcode often uses encryption and code obfuscation to avoid detection, we expected only poor *quality* when investigating the initially created dump files. To encounter this problem, we applied the MVD feature described in Section 4.5. To reduce the amount of information to be examined, we only used a subset of the malicious PDF documents and only one particular viewer. More specifically, we chose *Adobe Acrobat Reader 9.00* and those 4,869 samples for which ILC execution was detected in that viewer in the first experiment. In particular, we performed the following steps to measure the quality of the extracted ILC:

- we analyzed 4,869 PDF samples in *Acrobat Reader 9.00* with enabled MVD,
- all consecutive memory pages were concatenated to *code regions*, resulting in either one or two versions each:
  - one *initial region*, if only one dump version of each contained memory page exists,
  - and additionally one *final region*, if more than one dump version exists for at least one contained page
- the code ratio of each region was determined by using *IDA Pro*, and
- all valid strings from each region were extracted and counted.

We then selected those code regions for which an initial *and* a final version existed, i.e., those which contain self-modifying code. We found 2,534 regions of this kind. Figure 2 illustrates the code ratio for the initial and final regions, respectively. One can easily see that the percentage of valid instructions increases dramatically when applying the MVD feature. Figure 3 shows the improvement of the data ratio – in terms of valid strings – when the shellcode is deobfuscated in an automated way. When comparing with the code ratio, the improvement is only marginal. Nevertheless, in sum we extracted 7,807 strings and 1,866 URLs from the initial regions, and 8,676 strings and 2,280 valid URLs from the final ones.

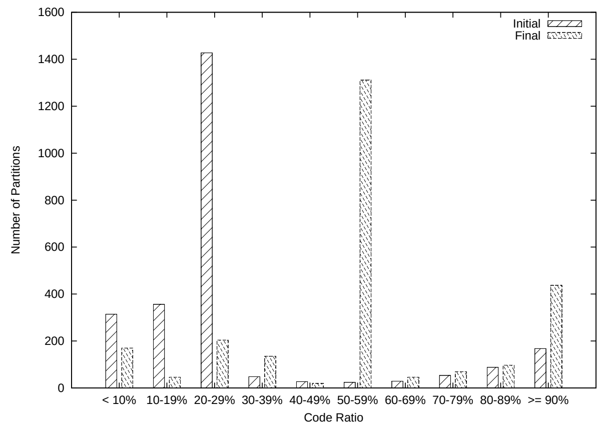

Figure 2: Code Ratio Distribution

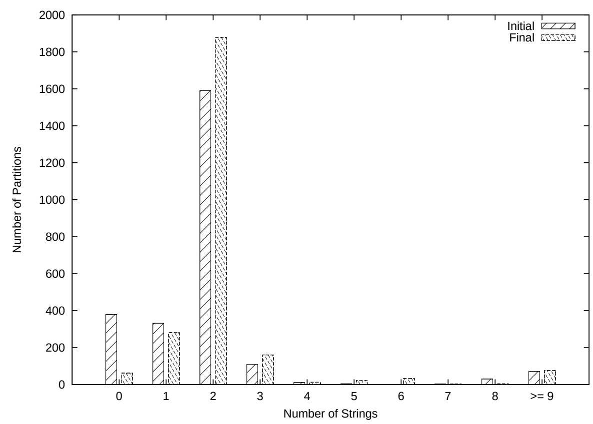

Figure 3: Data Ratio Distribution

8. LIMITATIONS

The approach described in this paper is solely based on dynamic analysis of the examined malware samples. Therefore, it suffers from all the drawbacks and limitations of dynamic analysis in general. Since during each code execution only one particular control path is taken, the gained results always may be incomplete. If a required environment condition is not met and, therefore, a certain malicious functionality is not triggered during execution, dynamic analysis is unable to reveal any information about it. Accordingly, our system is incapable to detect embedded malicious code in general, but only detects it when it gets executed. Obviously, our system is not meant to protect end consumer hosts, but its sole purpose is to support malware analysis on dedicated analysis systems.

Furthermore, the existence of malicious computation does not always imply the existence of illegitimate code. Therefore — and similar to DEP — our approach has problems with novel exploitation techniques like *return oriented programming* (ROP) [24] or *JIT-spraying* [4, 25]. However, we share those limitation with systems that are similar to ours and, furthermore, advanced attacks usually consist of multiple stages of which only the first uses ROP/JIT-spraying to set up a later stage comprising regular illegitimate code which then can be detected and extracted using our method. Nevertheless, despite its difficulty it is possible to create full ROP-shellcode that are undetectable with out approach.

9. CONCLUSIONS

In this paper, we presented a generic and automatic method to detect and extract illegitimate code during an attack. We introduced *CWXDetector*, an implementation of our approach for the 32 bit Windows XP version, and evaluated it by analyzing a large corpus of malicious PDF documents. Our system turns out to be very effective in supporting malware analysis, since the detection rates improve state-of-the-art tools and it directly supports the analyst by extracting a small set of memory pages for manual inspection. We also evaluated *CWXDetector* with different kinds of other file formats and even complex attack vectors like a Flash file embedded in a Word document can be analyzed successfully.

From an analyst point of view, especially the contained URLs and server host addresses that point to additional malware sites are valuable resources. Furthermore, the insights gained by a post processing analysis may assist in developing new protection techniques and creating signatures for zero-days until patches are available. We have also shown how the quality of the extracted ILC can be increased dramatically by applying multi-version dumping to automatically deobfuscate it.

Acknowledgments.

We would like to thank Tilo Müller for reading earlier versions of this document and making helpful suggestions for improvements. Additional thanks go to Andreas Dewald, Marco Cova, and Tyler McLellan for their great support while using their analysis tools. This work has been supported by the German Federal Ministry of Education and Research (BMBF grant 01BY1205A – JSAgents).

10. REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *20th IFIP International Information Security Conference*, 2005.

[3] P. Baecher and M. Koetter. libemu - x86 shellcode detection and emulation, 2007. `http://libemu.carnivore.it/`.

[4] Dionysus Blazakis. Interpreter exploitation. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

[5] contagio Website. Malware Sample Dump For CVE-2011-0611 Flash Player Zero day. `http://contagiodump.blogspot.com/2011/04/apr-8-cve-2011-0611-flash-player-zero.html`.

[6] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *World Wide Web Conference (WWW)*, 2010.

[7] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: Sandboxing JavaScript to fight malicious websites. In *ACM Symposium on Applied Computing (SAC)*, 2010.

[8] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[9] Adobe Systems Incorporated. Document management, portable document format, part 1: Pdf 1.7, 2008.

[10] Intel Corporation. Intel: 64 and IA-32 Architectures Software Developer's Manual. Specification, Intel, 2007. `http://www.intel.com/products/processor/manuals/index.htm`.

[11] Christopher Jordan. Writing detection signatures. *USENIX ;login:*, 30(6):55–61, 2005.

[12] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *ACM Workshop on Recurring Malcode (WORM)*, 2007.

[13] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[14] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, 2008.

[15] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[16] Microsoft. Enhanced mitigation experience toolkit (EMET). `http://support.microsoft.com/kb/2458544/de`.

[17] MSDN. A detailed description of the data execution prevention (DEP) feature. `http://support.microsoft.com/kb/875352/en-us`.

[18] Udo Payer, Peter Teufl, and Mario Lamberger. Hybrid engine for polymorphic shellcode detection. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[19] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2006.

[20] Sebastian Porst. Dumping shellcode with Pin. `http://blog.zynamics.com/2010/07/28/dumping-shellcode-with-pin/`.

[21] Rapid7. The metasploit framework. `http://metasploit.com/`.

[22] Karthik Selvaraj and Nino Fred Gutierrez. The rise of PDF malware. `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf`, 2010.

[23] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SIGOPS Symposium on OS Principles (SOSP)*, 2007.

[24] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[25] Alexey Sintsov. Writing JIT-spray shellcode for fun and profit. `http://dsecrg.com/pages/pub/show.php?id=22`.

[26] Hispasec Sistemas. Virus total. `http://www.virustotal.com/`.

[27] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. SHELLOS: enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, 2011.

[28] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, Newso James, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security (ICISS)*, 2008.

[29] Didier Stevens. `http://blog.didierstevens.com/2010/03/29/escape-from-pdf/`, 2010.

[30] Didier Stevens. `http://blog.didierstevens.com/2010/03/31/escape-from-foxit-reader/`, 2010.

[31] Joe Stewart. OllyBone: Semi-Automatic Unpacking on IA-32. *Defcon 14*, 2006.

[32] PaX Team. Documentation for the PaX project - overall description. `http://pax.grsecurity.net/docs/pax.txt`, 2008.

[33] The Pax team. PaX address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[34] Malware Tracker. pdf examiner. `http://www.malwaretracker.com/pdf.php`.

[35] Carsten Willems. Windows memory management internals (not only) for malware analysis. Technical report, University of Mannheim, 2011.

[36] Carsten Willems and Felix C. Freiling. Using memory management to detect and extract illegitimate code for malware analysis. Technical report, University of Erlangen, 2012.