University of Mannheim
Technical Report TR-2011-002

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

Carsten Willems[1] and Felix C. Freiling[2]

[1] University of Mannheim, Germany
[2] University of Erlangen, Germany

Abstract. Exploits that successfully attack computers are mostly based on some form of shellcode, i.e., illegitimate code that is injected by the attacker to take control of the system. Detecting and extracting such code is the first step to detailed analysis of malware containing illegitimate code. 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 of the approach is to flag critical memory pages as non-executable and use a modified page fault handler to dump corresponding memory pages. We present an implementation of the approach for the Windows platform called *CWXDetector*. Evaluations using malicious PDF documents as example show that *CWXDetector* produces no false positives and has a similarly low false negative rate.

1 Introduction

1.1 Motivation

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 caused by *illegitimate code*, i.e., code that was not intended to be executed, neither by the developer of the exploited process nor by the end-user of the system. Such code is usually injected into the target system using external data like network traffic or application files.

As a countermeasure to this increasing threat, operating systems try to *prevent* the execution of illegitimate code using techniques like *Data Execution Prevention* (DEP) [14], which ensures that no code is executed from data pages, and *address space layout randomization* (ASLR) [27]. However, prevention alone does not generally help in the analysis of illegitimate code. Therefore, it is necessary to develop mechanisms that *detect* and if possible *extract* illegitimate code from malicious data.

In this work we present CWXDetector, a new tool for the analysis of malware for the Windows operating system. *CWXDetector* follows a dynamic approach

for detecting and extracting illegitimate code by instrumenting the memory management features of the operating system itself. Roughly speaking, the idea of the approach is to mark critical 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 gets dumped for later analysis.

Note that *CWXDetector* is not meant to *protect* a system, but to monitor and analyze the execution of illegitimate code. Our system even disables some security measures like DEP for that. Nevertheless, there are some similarities in other preventive and analysis techniques that we now discuss.

1.2 Related Work

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 [29]. Other methods restrict memory write operations or control transfers. Kiriansky, Bruening and Amarasinghe [12] as well as Abadi et al.[1] use code rewriting techniques to implement the monitoring. The main difference to our work, is that those solutions terminate the monitored process in the event of a security violation, and are not able to produce any further analysis data. Furthermore, they all lack of capability to handle self-modifying or dynamically created code and they are not able to handle specific types of exploits (like SEH-related ones) under certain circumstances (e.g., if libraries are involved that have the *SafeSEH* feature disabled).

Detection of Illegitimate Code The detection of illegitimate code is an extremely difficult problem today. Early attempts relied on static signatures [10], but those 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 [15] in combination with *execution chain evaluation*. Polychronakis, Anagnostakis and Markatos [16] 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*. Overall, and in contrast to *CWXDetector*, these signature- or heuristics-based approaches are not fully generic and have to be extended when new anti-detection measures of malicious code come up.

It has been observed before that memory management can be used to detect illegitimate code execution. For example, the *PaX project* [26] proposes several

different measures to implement non-executable memory — even on architectures with no hardware support for that. Also hardware-DEP utilizes the *no-execute* (NX) flag in the Windows page table to make particular memory pages non-executable. However, and in contrast to our approach, these techniques do not allow to *extract* illegitimate code since they totally block the execution of illegitimate code.

Extraction of Illegitimate Code There exist several solutions aiming at the extraction of illegitimate code, especially automated unpacking of malware. These mechanisms usually interact deeply with the memory management of the underlying operating system. In all cases those solutions try to detect the execution of memory regions which have been written to beforehand. *OllyBone* [25] implements this by instrumenting the *translation lookaside buffer*. 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 [13] 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 unpacked memory 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 heuristic based on the concept of *trusted callers* that results in much better detection results.

Renovo [11] runs the sample in an emulated environment (TEMU [21]) and maintains a shadow memory to track written memory regions. Since this cannot be done on a native system, the system cannot be realized *without* system-emulation. Again, like with debuggers, this enables the monitored malware to easily detect the synthetic environment.

1.3 Contributions

CWXDetector is a new dynamic approach for detecting and extracting illegitimate code. The power of the 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 of operating systems, since this platform is still the major target of illegitimate code today. But since Windows is not an open source operating system, we had to perform a lot of reverse engineering regarding the internal memory

management mechanisms of the Windows kernel [29]. Based on these insights, we modified the kernel of a x86 Windows XP operating system by establishing a custom *page fault handler* and intercepting some essential memory related *system functions*.

We evaluated our approach by considering the task of detecting and extracting illegitimate code in/from a particulary relevant class of application files, namely those in Adobe's portable document format (PDF) [17]. Our approach proves to be very effective. We analyzed a set of 7278 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. The extracted shellcode data looks very promising, but remains a topic for more sophisticated analysis in future work. We also analyzed the same amount of benign PDF documents, resulting in a (false positive) detection rate of 0%. Furthemore, our detection results compare favorably to those of application specific detection tools like *Wepawet* [6], *PDF Examiner* [28] and *ADSandbox* [8].

To summarize, the contributions of this paper are twofold:

- We present a generic and fully automatic 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.

Obviously, our system is not meant to protect end consumer hosts, but its sole purpose is to support malware analysis on dedicated analysis systems. It also has to be emphasized that we are unable to detect malicious code which is embedded in arbitrary data in general, but we solely are able to detect such code when it gets executed.

To some extent our approach is similar to DEP [14], which totally disables the execution of certain memory regions. This is accomplished by employing the NX flags of the related page table entries in a similar way like we do it. Nevertheless, there are big differences between DEP and our system: first we do not completely prohibit the execution of illegitimate code, but we intentionally allow it in order to get detailed analysis results. On the attempt to execute non-executable memory, we dump the memory page that contains the code, and then continue execution in order to obtain more information. Secondly, we do not only take the type of memory into account when deciding which should be monitored respectively executed, but we also check the initiator of memory related modifications and allocations. As an effect we are able to correctly handle cases in which malicious executable modules are mapped into memory, or when DEP-conquering shellcode allocates regular executable memory.

Unfortunately, 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) [18] or *JIT-spraying* [5,19]. However, advanced attacks usually consist of multiple stages of which 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.

To some degree our system also constitutes a *reference monitor*, that is restricted to monitor accesses to executable memory. In contrast to the *inline* solutions proposed in the past [12,1], we do not modify the monitored application itself, but the underlying operating system and incorporate hardware features to perform the monitoring. This has several positive effects: it is much easier to extract the executed memory, since we are already residing within the page fault handler. Furthermore, we do not have any problems with dynamically created or self modifying code. Finally, we do not have to manually track any control transitions, i.e., some of the related work in this topic is unable to detect control flow transitions that occur due to *structured exception handling* and not due to a regular branch instruction.

1.4 Structure of this Paper

This paper is organized in the following way: Section 2 defines our attacker model and some necessary terms. In Section 3 we describe our approach in general, whereas Section 4 illustrates an implementation for the Windows XP operating system. In Section 5 we explain how our system can be applied to the analysis of malicious PDF documents and in Section 6 we present the results of an experimental evaluation. Section 7 concludes this paper and gives topics for future work.

2 Model and Definitions

In this section we specify our attacker model, define the term *illegitimate code* and further concretize our two aims: the detection and extraction of executed illegitimate code.

2.1 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 or a malicious input packet to some network application.

As mentioned above, we are aware of the threats posed by ROP [18] or JIT-spraying [5,19] 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 documented instance of such a single staged *full*-ROP/JIT-sprayed attack in the wild.

2.2 Illegitimate Code

We first define the counterpart of illegitimate code, namely legitimate code. For this, we partition the files of a system into a set of trusted files and a set of

untrusted files. We assume that such a distinction is given, e.g., by defining all files in a freshly installed system as trusted. Next we distinguish code portions contained in trusted files into trusted functions and untrusted functions. Again we assume such a distinction is given, e.g., by defining all standard memory modification 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 it is code that was dynamically created by any of the trusted functions from one of those files.

We now 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.

2.3 Problem Statement

The aim of the system described in this work is to perform two tasks automatically:

1. We wish to *detect* the execution of illegitimate code, and
2. to *extract* it, i.e., dump all relevant memory pages to disk, for a later in-depth analysis.

3 Approach: Instrumenting the Page Fault Handler

In the following we describe our approach and how it can be applied to the dynamic analysis of malware.

3.1 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. When the vulnerability is exploited, the control flow is redirected to one of the following locations:

1. ILC on the stack *(buffer overflow)*,
2. ILC in the heap *(heap-spraying)*, or
3. ILC in a static data area *(exploiting a static data buffer)*.

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.

3.2 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") the system service that is used to create files or open them with write-access.

For each trusted file we further define a set of *trusted memory modification functions*, which contains all the functions which 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*.

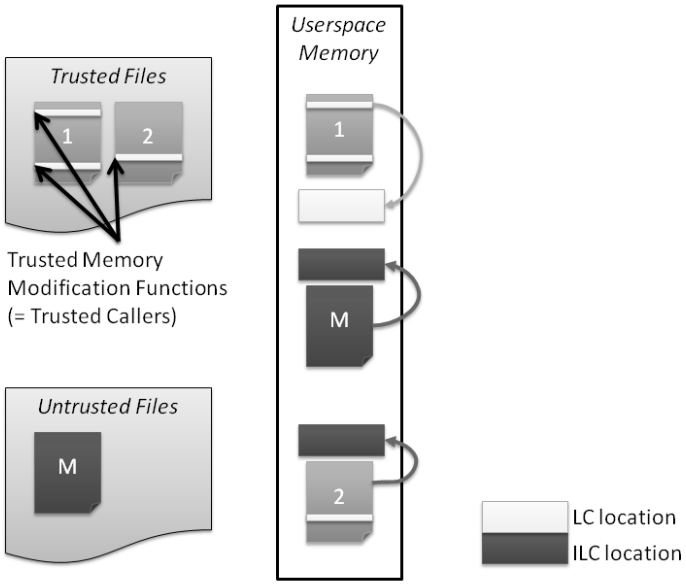


Fig. 1. Trusted Memory Modification Functions

Figure 1 illustrates our approach, showing an example with two trusted and one untrusted file. While trusted file 1 contains two trusted memory modification function, trusted file 2 only has one. The simplified version of the userspace memory 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, all mapped modules have allocated one dynamic memory area, pointed to by the corresponding arrow. That area created by file

1 was created by a trusted caller and, hence, may contain legitimate executable code. But 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.

3.3 Memory Protection Modifications

Obviously it is also necessary to intercept attempts to modify the memory protection. This is realized by hooking critical system calls. Inside these hook functions we enforce the following points 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.

All attempts that violate these rules are intercepted and the resulting memory regions becomes *non executable*.

In summary, only trusted callers can create executable memory and only trusted files can be loaded into 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 [14]. 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 *execute disable* (NX) protection flag on a page level. Nevertheless, this feature is only used for valid page table entries (PTEs) and, therefore, in most cases some additional OS memory objects may have to modified as well.

3.4 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, it has to be checked 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, we dump the related memory page to hard disk and create a log file entry. After that, the related memory region is modified 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 Windows-based Implementation

In the following section we illustrate *CWXDetector*, which is the concrete implementation of our approach for the x86 version of Windows XP. We have to use the PAE kernel version of Windows, since only this one supports the NX page table flag that we utilize 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.

In summary, to realize our approach we need to

- define trusted files and trusted callers,
- implement hook functions for memory allocations and modifications,
- implement a custom page fault handler to detect and react on 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 resulting findings are explained in detail in an additional technical report [29].

4.1 Memory Functions Hooks

In order to ensure that only legitimate code resides in executable memory, we redirect the calls of **NtAllocateVirtualMemory**, **NtProtectVirtualMemory** and **NtMapViewOfSection** to custom hook functions in order to fulfill the invariant from Section 3.1. On a lower level we realize non-executable memory by modifying the related memory structures, i.e., the *execute disable* (NX) flag of the related page table entry as well as the *VAD entry* and the *prototype PTEs* of the corresponding memory regions.

The hook of **NtAllocateVirtualMemory** first checks the caller and the wanted memory protection. If the caller is *not* trusted, that protection value is modified such that the allocated memory becomes not executable. Then the original system service is invoked to actually perform the allocation operation. All this happens transparently to the caller, i.e., no error result is returned in case of a modified protection parameter.

The proceeding within the **NtProtectVirtualMemory** hook is rather similar. Again, for all untrusted callers the protection parameter is manipulated to being non-executable. Since this system service can be used to modify the protection settings of already loaded modules, we also have to intercept when it is invoked by trusted callers. In that case it is first checked if the affected module

belongs to a trusted file or not. If it is not trusted, the protection value again is modified to non-executable. The reason behind this is to block the execution of untrusted files.

The hook function of **NtMapViewOfSection** ensures that after mapping a file into address space, all containing memory fulfills our requirement. Since we cannot simply modify a call parameter like within the previously mentioned hook functions, but have to operate directly on the VAD [29] respectively PPTE entries, the hook first calls the original system function. After that, different actions are performed, depending on the fact if a data or an image file was mapped.

For image files the effective page protection is taken from the related PPTE. Therefore, our hook enumerates all executable subsections of the related segment object and manipulates their PPTEs. In case of trusted files, only those PPTEs are modified which also indicate writable memory. In such case the PPTEs remain writable, but are no longer executable. Otherwise it would be possible for an attacker to modify the instructions which are contained in memory that is associated with legitimate code. By removing the execution property, an attacker is still able to modify it, but we detect the approach to ultimately execute the overwritten instructions. In general, a PE file should never contain executable sections which are also writable, but due to the procedure described above we are able to detect those as well.

If a data file is mapped, the situation is a bit different. For those kind of files the effective protection setting is taken from the associated VAD entry. When the related memory is actually accessed, the PTEs are created and their protection settings are directly taken from the VAD. Therefore, again we check if the section belongs to a trusted file or not. If it belongs to an untrusted file, the VAD is modified to *non-executable*. If it is trusted, the VAD protection is only modified if it is writable *and* executable.

4.2 Checking the Caller

In order to restrict the memory creation and modification attempts of executable memory, we need to check the particular initiator of such operations against the set of trusted callers. For that we have to walk the usermode call stack to a certain depth and inspect the saved return addresses. Since our hooks functions reside in the kernel, we therefore need to gather information about the current usermode context, from which the kernel call has been performed. This kind of information is stored within the so called *trap frame*, which can be accessed from kernel mode via the **KTHREAD** structure that exists for each thread.

Figure 2 shows an example to illustrate the relationship between the usermode stack layout and trapframe values, when performing a system call. The native API functions **KiFastSystemCall** and **NtCreateFile** use *frame pointer omission* (FPO), which means that they do not set up a full stack frame. Therefore, the saved frame pointer EBP cannot be used to locate their saved return addresses on the stack. Nevertheless, those saved RET addresses can be obtained from the raw usermode stack by inspecting the slots pointed to by the

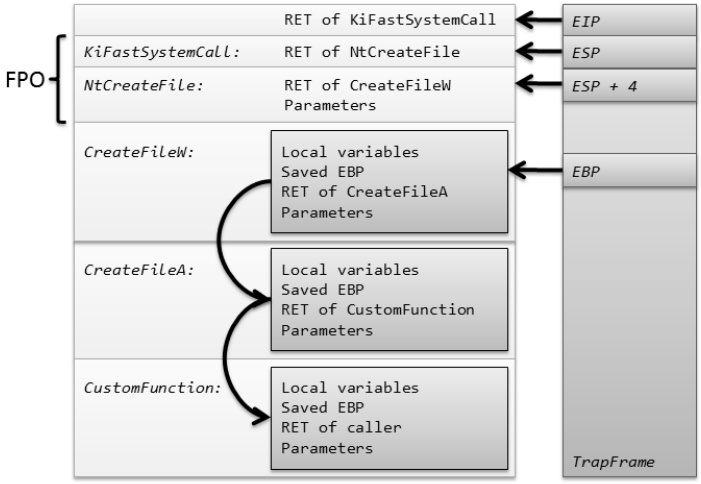


Fig. 2. Stack Frames and Trap Frame

stack pointer ESP and ESP+4. As shown, the first usermode function with a full stackframe can be reached via the saved EBP. All further calling functions can then be enumerated by using the saved EBP values, as long as they do not perform FPO. In that case we would have to involve additional information about the function's prototype and local variable area in order to further enumerate the stack frames. However, all trusted callers we are using in our experiments set up a full stack frame.

4.3 Custom Page Fault Handler

We hook the Windows system function **MmAccessFault** to implement our custom page fault handler. The code of this hook function is rather short, since all necessary prerequisites are already done by the other hook functions. First, we call the original page fault handler to actually resolve the fault, e.g., validate the related PTE. Then we check if the fault was caused by an *execute* operation and if the faulting address resides in user space. If so, we further verify if the target address is valid, which can be determined easily by inspecting the values of the related PDE and PTE. If all these conditions are met, a dump file of the related memory page is created and the protection settings of the associated PTE are modified to executable. A further modification of the PPTE and VAD entry is not necessary here, since protection settings stored with those objects are never used again, once a PTE was created from them.

4.4 Additional System Modifications

For distrusting all files which were created or modified during our analysis, we hook the system service **NtCreateFile**, which has to be called to create new and open existing files. Furthermore, we hook **NtCreateProcess** to monitor and restrict the creation of new processes.

5 Application to the Analysis of PDF Documents

Our approach is completely generic and can be used in different scenarios. In order to further illustrate it and evaluate its effectiveness we apply our tool *CWXDetector* to the field of dynamic analysis of malicious PDF documents. Malicious documents in general have become very popular in the past few years, and especially the *portable document format* (PDF) is a commonly used medium for malicious content. One reason for that is the extensive feature list of PDF documents. It offers the two programming languages *Javascript* and *Actionscript* and the possibility to embed many different object types like images, sounds and even executables. Another reason for that increasing number of exploitation attempts is the complexity, and hence error-proneness, of the PDF format itself and its viewer applications. The latest PDF reference [9] from 2008 contains 756 pages and *Adobe* already has published several extensions to it meanwhile.

5.1 Dynamic PDF Analysis

During dynamic malware analysis in general the object under investigation is not disassembled, but viewed as a *blackbox* and actually used in its intended way: an executable is run, a document is opened in its associated viewer application and so forth. This has several negative impacts: first of all the testing environment may get infected, since the malicious code actually is executed. Secondly, it may happen that though real malware is analyzed, no malicious operation may be observed at all. There are always some necessary requirements to the environment, under which particular exploits may only work, e.g., a vulnerability may be fixed in a newer version of the affected application or an exploit aims only at a particular language version of a software. Accordingly, dynamic analysis in general is *incomplete* and we try to address this disadvantage by testing each PDF sample in different viewer applications and then combine all the findings.

In order to analyze PDF documents with our system, we have 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 have used the *Adobe Acrobat Reader* versions 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 have set up one machine with *Foxit Reader* version 3.0.0 for which also some vulnerabilities are known. This particular application and version set have been 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 is analyzed in all of those machines in parallel. During the analysis we performed the following steps on each machine separately:

- We installed our customized page fault handler and our system hooks.
- We started the particular viewer application.
- 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 in our custom page fault handler.
- We opened the PDF document in the viewer application.
- If new memory was allocated or existing memory was modified during execution, we enforced the invariant from Section 3.1.
- If the execution of illegitimate code was detected, we dumped the associated memory page to a file, created a describing log entry, and modified the related PTE to being executable. We then checked the dumped memory page for typical patterns of illegal code. In case these could be found we marked this case as "*PATTERN*" in the log file.
- If a new process was created by the PDF viewer, we created an entry marked "*PROCESS*" in the log file. 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 created an entry marked "*DIALOG*" in the log file and additionally logged the contents of the window. We then simulated a user input to close the window and continue viewing the PDF document.

The analysis process was stopped after a specific timeout (two minutes) had been reached or the PDF viewer application terminates prematurely. In the latter case we marked this execution as "*CRASH*" in the log file. Finally, and if none of the aforementioned special cases above have been occurred (*PATTERN, DIALOG, PROCESS, CRASH*), the case was labeled as *NOTHING*.

After the analysis we extracted the dump and log files from the machine and then reverted it back to a clean state. What we finally got as analysis result is a logfile that contains information about:

- All attempts to allocate executable memory which are not invoked by a trusted caller,
- all attempts to modify existing memory to be executable, which are not invoked by a trusted caller,
- all attempts to execute memory that contains illegitimate code,
- all created files (needed to maintain the list of untrusted files),
- all created processes (needed for evaluation and debugging purposes), and
- all created and shown user dialog windows.

Additionally the analysis may have produced several dump files which contain the memory contents of each page from which illegitimate code was executed.

Overall, every PDF file ended up with a combination of two labels (d, c): the first label d determined whether illegal code 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:

$$ILC > CRASH > PROCESS > DIALOG > NOTHING$$

We used the highest occurring value as combined overall value.

In Section 6 we describe our findings and the results of our experiments in detail. Nevertheless, in general we are interested in the fact if a viewed 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.

5.2 Determining Trusted Callers

As explained in Section 2.2 we have to define a set of *trusted memory modification functions* for each of the trusted files. For that purpose we have to identify all the functions from each loaded image file, which are used to produce executable memory, e.g., we have to search for all calls of memory-related APIs and inspect the used parameters that specify the protection settings of the resulting memory. This may be done fully-automated, but we have used a semi-automated process, in which we have started with a white-list that only contained the loader-related function from ntdll.dll. Then we have loaded benign PDF documents into the different PDF viewer applications and if we have encountered a *false positive*, we have manually inspected the disassembly of the related function call and extended the white list appropriately. This process is fail-safe, since we may only 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 the end we came up with only three files that had to be taken into account: *ntdll.dll, AcroRd32.exe* and *authplay.dll*. All of those contain functions that allocate executable memory or modify existing one to being executable. *ntdll.dll* contains the Native API and especially the Windows loader functionality, which of course, has to be able to create executable memory. For the particular Windows version we are using, this is done from two different locations within **LdrpSnapIAT** and **LdrpSetProtection**.

Acrobat Reader from version 9 on upwards contains a JIT-compiler that also allocates executable memory. We have checked the affected library *authplay.dll* and verified that the **VirtualProtect** API is called from two different places. Since only one of those calls is used with an *executable* protection value, we only have to put this one on to the list of trusted callers. Finally, we have observed

Acrobat Reader in version 7.xx to allocate executable memory from an MFC function **CLangBarItemMgr::CreateInstance**. Hence, we also take care of these calls.

6 Evaluation

We created two sets of PDF documents of size 7278 each (a *benign* set and a *malicious* set) and used *CWXDetector* to analyze these files in order to evaluate the approach.

6.1 Benign Sampleset

In order to test the *false positive* rate of our approach, we obtained a set of known benign documents, which 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. We constructed this set using the following method:

- We retrieved URLs of the TOP 5000 Internet sites from `www.alexa.com`.
- We queried Google for the first 10 PDF documents on each site and downloaded them.
- Using the tool *pdfid* [22], we selected all documents which contained *JavaScript*, *OpenActions* or some other extended PDF features.
- We uniformly picked random samples from the other downloaded files until the total set of files consisted of 7278 samples (the same size of the malicious sample set, see below).

The final set has the following characteristic: altogether it contains 7278 samples, 600 of which contain *Javascript*, 782 contain *AcroForms*, 1573 samples have an *OpenAction* and 751 some *AdditionalAction*.

All samples have been verified by Virus Total [20] and in 3 cases one or more AV scanner delivered 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, hence we have a *false positive* rate of 0%.

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*). Since the achieved false positive rate of our experiments was so low, we can assume that it will not increase dramatically by using more different viewers.

6.2 Malicious Sampleset

We obtained a set of 7278 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 as shown in Table 1. We checked all samples with the publicly available service Virus Total [20] which confirmed that all of them were indeed malicious.

Table 1. AV Sample Sources

Source	Fraction
AV Sample Sharing	70.0 %
Found in the Wild	24.0 %
Multiscanner projects	4.8 %
Intercepted botnet traffic	1.2 %

We ran our tool on the malicious sample set and were able to detect and extract executed ILC in 93.2% of all cases. The detailed analysis results are shown in Table 2 and are explained in the following section.

Table 2. Overall Detection

	illegal code detected		**no** illegal code detected	
	Samples	Fraction	Samples	Fraction
PATTERN	6658	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	6781	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, which was *not* contained in our application set. Finally, there exist some samples that perform malicious behaviour that is not based on shellcode, but exploits built-in PDF features. For instance some samples redirect to malicious websites or exploit software bugs in third-party applications, which can be started directly from within a PDF document. Therefore, we analyzed the documents from the malicious data set in more detail.

Results without ILC Execution Detection To confirm the false positive rate, we first investigated the 497 PDF documents for which *no* ILC execution

was detected. If we would be able to prove that none of them has executed any ILC within our used environments, our detection rate would increase to 100%. However, since we are not able to manually analyze about 500 samples, we developed the following heuristics to find at least hints that lead to the assumption that these samples really failed to perform malicious computation.

We first checked those 15 files that crashed the PDF viewer. We verified if the crashing operation is contained in any of the regular modules that belong to the PDF viewer. In that case most probably some exploit went wrong or the PDF consisted of an invalid structure, which lead to an erroneous termination of the parsing application. Of course, we cannot be completely sure that no illegitimate code has been executed beforehand, but since we have seen many corrupt and broken PDF documents, it is very probable that we have not missed anything but the viewer just has crashed before any illegitimate code could be executed. After manually checking all 15 samples we found that indeed all of them performed invalid memory accesses. So overall those samples are malicious too but simply fail to execute their shellcode due to incompatibilities with the underlying PDF viewer.

We then checked those 33 samples that created a new process. In the first place one could suppose that starting a new process is a sure sign for malicious activity that was missed by us. However, we discovered that in all cases regular built-in features of the PDF viewers were used to start a new process. All checked 33 samples achieved to fork a new process without the help of illegal code. The forked processes are *Internet Explorer (iexplore.exe), Outlook Express (msimn.exe)* and the *Command Shell (cmd.exe)*. Starting those processes itself is a PDF built-in feature and does not require any exploit. Nevertheless, depending on the underlying PDF viewer the user first has to confirm a dialog message before a new process can be spawned. In most cases the parameters used when starting the Internet Explorer or Outlook Express are specially crafted to produce a parsing error and execute arbitrary operations, e.g., we have seen parameters like those shown in Figure 3. Looking at these parameters it indeed seems that these documents are malicious.

We then investigated the 295 cases where classified as showing a user dialog. Most of the dialogs we have seen contain error messages of the parsing engine, which state that the PDF document itself or some embedded JavaScript-code is invalid. In that case we assume that either the PDF document really is corrupted and will never succeed in performing any malicious operation or that we simply have not used the expected environment to trigger its functionality. Additionally, 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 dialog 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 [23,24]. The user dialog can be either an error message as a result of a broken PDF document or it can be a specially crafted dialog which is shown on behalf of the PDF document itself. Table 4 shows the three different classes of dialogs we have seen in our tests. Overall, it is unclear

```
mailto:
%/../../../../../Windows/system32/cmd".exe"" /c /q \"@echo off
&netsh firewall set opmode mode=disable&echo o 127.0.0.1>1
&echo binary>>1&echo get /ldr.exe>>1 &echo quit>>1
&ftp -s:1 -v -A>nul&del /q 1 &start ldr.exe&\" \"&\" "nul.bat
```

```
mailto:
%../../../../../windows/system32/mshta"" javascript:e=String.fromCharCode;
new ActiveXObject\('wscript.shell'\).Run\('cmd /c for /F %i IN
\('+e\(39\)+'dir /b/s %Tmp%or~1\\\\content.ie5\\\\*.pdf'+e\(39\) +'\)
DO findstr /B CZY %i>c:/a.vbs&c:/a.vbs',0\);\.close\(\)//.cmd)
```

Table 3. Malicious Process Parameters

whether these files are indeed malicious. However, it is also highly probable that none of them executed any form of ILC during execution.

PDF parsing error messages:

- A 3D data parsing error has occurred
- An unrecognized token 'aaaaaaaaa' was found
- The application is being terminated because of memory corruption

JavaScript errors messages:

- line 3 - GeneralError: Operation failed
- var ZUl8cVPKM33; var MpuldZ90lGs = new Array(); ...

Failed exploitation attempt messages :

- The application "C:\AAAAA..." is set to be launched by this PDF file...
- Could not open the file '/C/AAAAAAA AAAAAAAAAAAAAAAAAAAAAA...

Table 4. Dialog messages

Finally, 154 samples did not perform any malicious activity at all. Obviously, this does not mean necessarily that the sample is harmless. It just means that under the given environment it behaves benign. One reason for not triggering detection could be that we have not used the correct PDF viewer environment which is necessary to trigger the exploit. We manually checked a representative 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 be that they contain pattern from their virus signature databases. This may be either a pure coincidence, or it may be the result of a failed attempt to create a working malicious PDF document.

Overall, given the above findings, in all 497 cases no ILC was executed. 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 are true negatives, and so after carefull consideration one could also claim a false negative rate of 0% for our approach.

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 6658 cases that are flagged *PATTERN* we confirmed the presence of known shell code patterns in the dumped memory pages. These patterns can be for example different forms of nop sleds, commonly used methods to obtain the current instruction pointer (*call-pop, fsetenv*), or direct system calls (*int 0x2e*). Therefore, we can be sure that in fact ILC was executed.

We then checked those 20 samples that crashed the PDF viewer after the ILC detection. This is also a clear sign for (a partly failed) malicious behavior. In such cases we can be sure that the exploit from the PDF did not work well, either because it was badly programmed or because it did not discover the environment which was needed to work correctly, e.g., a wrong PDF viewer application was used. We call those samples *broken*. Even if the samples do not succeed to perform any reasonable malicious operation, we can be sure that the observed code execution really is related to ILC and, accordingly, is no false positive. All of the 20 samples crashed due to an access violation when performing reading, writing or executing memory operations on invalid regions.

Next we investigated those 83 PDF documents that spawned new processes after the ILC execution. This is clear sign of beforehand executed ILC since the test system was set up in a way that prevented new processes from being spawned "legally" as an effect of just viewing a PDF document. For all of those process types listed in Table 5, we can assume that starting them is an effect of executed malicious code: it is either tried to perform malicious operations from within a second extracted or downloaded malware, to open a benign PDF document in order to hide the maliciousness of the initial document or to gather essential information about an exploited system. Accordingly, we can be sure that all of these samples really have executed shellcode.

Finally there are 20 remaining samples with ILC execution, for which all of our previously described heuristics fail. Hence, we are not able to tell anything automatically about their maliciousness and, therefore, we have checked them manually. All of them really execute illegitimate code, from which some is simply not working correctly and other does not even consist of valid machine

Obviously malicious processes:

- `c:\a.exe`
- `c:\DOKUME~1\user\LOKALE~1\Temp\HotPAtcher.exe`

New instances of the PDF viewer:

- `c:\Programme\Adobe\Acrobat7.0\Reader\AcroRd32.exe"c:\DOKUME~1\user\LOKALE~1\Temp\ASA2010.final1.pdf"`
- `cmd.exe/cstartAcroRd32.exe"c:\DOKUME~1\user\LOKALE~1\Temp\1465.pdf"`

Common Windows system information tools:

- `c:\WINDOWS\system32\winver.exe`
- `c:\WINDOWS\system32\whoami.exe`

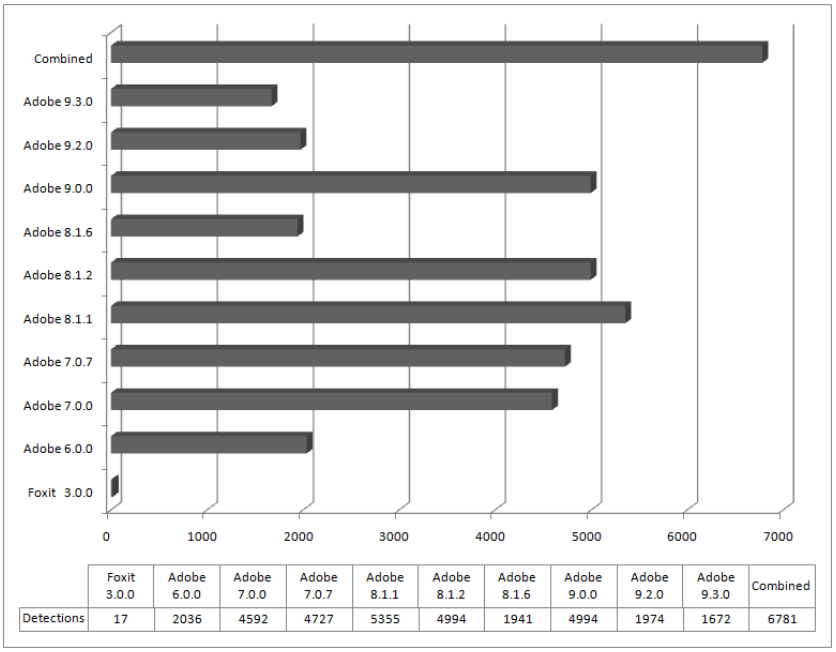
Table 5. Started Processes

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 well. Anyway, we have manually assured ourselves that in all cases ILC was executed, no matter if the resulting operations were valid or not.

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 do not contain working shellcode for any of our used environments, we get a much higher detection rate.

6.4 Detection per Viewer Application

In Figure 3 we have summarized all ILC detections and show the detection rates per PDF viewer. Note that these rates are heavily depending on the sample set, and do not necessarily reflect the quality of the PDF viewers or the number of exploits that exist for them in the wild. What definitely can be seen, is the unsurprising fact that combining the partial results of our detection scheme yields a higher detection rate. Furthermore, it is obvious that each single PDF viewer instance is vulnerable to a significant amount of malicious PDFs. One single exception to this is the *Foxit Reader*. Since this application is not nearly as widespread as the *Acrobat Reader*, not much effort has been invested by attackers into building exploits for it. Accordingly, the very low detection rate for this viewer does not necessarily mean that it has less critical software bugs.



	Foxit 3.0.0	Adobe 6.0.0	Adobe 7.0.0	Adobe 7.0.7	Adobe 8.1.1	Adobe 8.1.2	Adobe 8.1.6	Adobe 9.0.0	Adobe 9.2.0	Adobe 9.3.0	Combined
Detections	17	2036	4592	4727	5355	4994	1941	4994	1974	1672	6781

Fig. 3. Detections per Viewer

6.5 Detection Results of other Analyzers

In order to evaluate the effectiveness of our generic solution, we have compared our results against those from some popular application specific analyzers: *Wepawet* [6], *PDF Examiner* [28] and *ADSandbox* [8]. 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 then labeled either *benign, suspicious* or *malicious*. Furthermore, additional comprehensive analysis data is generated, i.e., information about the embedded objects, like PE files or URLs, or contained 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 previously learned known benign profiles. It also uses a set of signatures to detect anomalities, which are *not* based on Javascript. The author has supported our project by analyzing our samples with the new version of his tool, which currently is not available yet. A better detection rate in comparison to the current available version could be achieved with that.

PDF Examiner[28] as well extracts all embedded objects and streams from the PDF document and decrypts them if necessary. Then, signature scanning is used to detect known malicious patterns and *libemu*[3] is used to detect shell-

code. From all the findings a score value is calculated, that decides about the ultimate outcome of the analysis. Besides this value, a sophisticated GUI report is generated that highlights suspicious and malicious parts of the PDF document.

In contrast to the two aforementioned analyzers, *ADSandbox*[8] solely aims at the detection of malicious Javascript. For that purpose, all Javascript code objects are extracted from a PDF sample and then executed in an isolated environment. Finally, heuristics are used 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 such application specific analyzers to those created by *CWXDetector*, one have 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*.

Table 6 summarizes all results for the detection of malicious samples. As you can see our approach yields even better results than those of the application specific analyzers if only taking those samples into account, which were labeled as definitely being *malicious*. But still when considering also the suspicious samples, 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 also 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. The most important reason for that is that it is only capable to analyze Javascript exploits.

Table 6. Detection Results on Malicious Sampleset

Analyzer	Malicious Samples	Fraction	Suspicious Samples	Fraction
Wepawet	4737	65.1%	1739	23.9%
PDF Examiner	6089	83.7%	1108	15.2%
ADSandbox	2206	30.3%	232	3.2%
CWXDetector	6781	93.2%	0	0.0%

On the other hand, *ADSandbox* does not produce so many *false positive* as the other two PDF analyzers, as you can see in Table 7. As we already have shown in Section 6.1, our approach does not produce any *false positive* as well due to its inner course of action. Unfortunately, the *Wepawet* results for analyzing the benign sampleset have not been completed at the writing of this paper, but will be presented on the website of the authors afterwards.

Table 7. Detection Results on Benign Sampleset

Analyzer	Malicious Samples	Fraction	Suspicious Samples	Fraction
Wepawet	?	?%	?	? %
PDF Examiner	82	1.1%	246	3.4%
ADSandbox	0	0.0%	3	<0.1%
CWXDetector	0	0.0%	0	0.0%

6.6 Extraction Results

We examine those pages which were dumped due to an initial ILC execution detection. On average we have 267 dumped pages per sample, which is a rather high number. Though after removing those pages which have been identified to contain solely some form of nop sled, we are left with an average of only 1.22 pages per sample. Furthermore, we have 5 outliers with 32 (3 samples), 80 and 603 dumped pages. By manually verifying those samples we have learned that something went wrong during the exploitation attempt and all of the dumped pages contain the same (failing) shellcode. While ignoring those outliers, we finally get an average value of 1.06 pages per sample. Since most shellcodes fit within one single page, this result is not very astonishing.

7 Conclusions

In this paper we presented a generic and automatic method to detect and extract illegitimate code. We presented *CWXDetector*, an implementation of our approach for the x86 Windows XP version, and evaluated it by using malicious PDF document as examples. Our approach is very effective in supporting malware analysis, since the detection rates are very good and it directly supports the analyst by extracting a small set of memory pages for manual inspection. In our prototype, we also added a debugging feature in which the automated process can be interrupted once the execution of illegitimate code is detected. On that event automatically a debugger is attached to the affected process and a hardware breakpoint is set on the first illegitimate instruction. This enables an analyst to quickly and easily debug shellcode within its exploited process.

As an addition to the regular system we also implemented multi-version dumping, in which each detected page containing illegitimate code is not dumped only once, but several times in case that it is modified. By that we can detect *self-modifying code* (SMC), which often is used for de-obfuscation and decryption. As a result we get several versions of the same memory page, from which we can easily isolate those parts that have been modified and infer the decrypted shellcode instantly. We plan to evaluate this feature in future work.

Another topic we would like to focus on in future work is to improve our system by implementing mechanisms to detect ROP and JIT-spraying code. One way to handle ROP is a full call stack inspection on system calls and there already exists different works in this field [7]. JIT-spraying itself may be hard to

detect [4] within our system, but we may patch the Interpreter runtime system such that its allocated memory in general is non-executable and only made executable on demand and in a controlled manner.

Acknowledgments

Thanks to Tilo Müller and Thorsten Holz 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.

References

1. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.
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. Piotr Bania. Jit spraying and mitigations. *CoRR*, abs/1009.1038, 2010.
5. Dionysus Blazakis. Interpreter exploitation. In *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, pages 1–9, 2010.
6. Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.
7. Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.
8. Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: Sandboxing JavaScript to fight malicious websites. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1859–1864, New York, NY, USA, 2010. ACM.
9. Adobe Systems Incorporated. Document management, portable document format, part 1: Pdf 1.7. `http://www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf`, 2008.
10. Christopher Jordan. Writing detection signatures. *USENIX ;login:*, 30(6):55–61, 2005.
11. Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, WORM '07, pages 46–53, New York, NY, USA, 2007. ACM.
12. Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.

13. Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.
14. MSDN. A detailed description of the data execution prevention (dep) feature. `http://support.microsoft.com/kb/875352/en-us`, 2006.
15. Udo Payer, Peter Teufl, and Mario Lamberger. Hybrid engine for polymorphic shellcode detection. In *Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 19–31, 2005.
16. Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 54–73, 2006.
17. 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.
18. Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.
19. Alexey Sintsov. Writing JIT-spray shellcode for fun and profit. `http://dsecrg.com/pages/pub/show.php?id=22`, 2010.
20. Hispasec Sistemas. Virus total. `http://www.virustotal.com/`.
21. 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 *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.
22. Didier Stevens. pdfid. `http://blog.didierstevens.com/programs/pdf-tools/`.
23. Didier Stevens. `http://blog.didierstevens.com/2010/03/29/escape-from-pdf/`, 2010.
24. Didier Stevens. `http://blog.didierstevens.com/2010/03/31/escape-from-foxit-reader/`, 2010.
25. Joe Stewart. Ollybone: Semi-automatic unpacking on ia-32. *Defcon 14*, 2006.
26. PaX Team. Documentation for the PaX project - overall description. `http://pax.grsecurity.net/docs/pax.txt`, 2008.
27. The Pax team. PaX address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.
28. Malware Tracker. pdf examiner. `http://www.malwaretracker.com/pdf.php`.
29. Carsten Willems. Windows memory management internals (not only) for malware analysis. Technical report, University of Mannheim, 2011.