FCScan: A New Lightweight and Effective Approach for Detecting Malicious Content in Electronic Documents

Master Thesis

June 11, 2013

Christiaan Leonard Schade
MSc Computer Science
Specialization Computer Security

Graduation committee:
Dr. D. Bolzoni
Prof dr. P.H. Hartel
Prof dr. F.E. Kargl

UNIVERSITY OF TWENTE.

Distributed and Embedded Security
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente, The Netherlands

Contents

List of Figures

List of Tables

Chapter 1

Introduction

According to Stone-Gross et al. [58], malware has shifted from being a "hobby" to becoming a business tool for cyber criminals. So-called Advanced Persistent Threats (APTs) often make use of custom-made malware for penetrating highly confidential computer networks and steal sensitive information, or disrupt critical processes. Unlike "regular" threats, APTs have access to skillful resources thanks to the financial support provided either by criminal organizations or state-sponsored agencies, and can be active for years before actually conducting an attack, in order to collect intelligence information about their targets. This intelligence enables APTs to successfully conduct social engineering attacks which often go together with malicious electronic documents as an attack vector.

There are two main reasons why electronic documents are a good carrier of malicious payload. First, the large installed base of software packages such as Microsoft Office or Adobe Reader, make them a safe bet for attackers. Second, victims exchange electronic documents with their colleagues on a daily basis, it is only natural for a victim to open an electronic document from a seemingly trustworthy source, referring to a relevant topic such as "*notes from a yesterday's meeting*" or "*2011 Recruitment plan*" (used in the 2011 RSA hack). Examples of APT attacks include DuQu [11, 12] where a Microsoft Office Word document was used, the 2011 RSA hack [51] using a malicious Microsoft Office Excel spreadsheet and recently a 0-day vulnerability [1] (CVE-2013-0640 and CVE-2013-0641) in Adobe Reader being used against governmental institutions around the world.

Because of the way APTs operate, current security countermeasures are seldom effective at detecting, let alone preventing, the majority of such cyber attacks. Three factors make an APT cyber attack difficult to counter, namely the exploitation of 0-day vulnerabilities, the use of electronic documents as attack vectors and the use of social engineering. In particular, by leveraging 0-day vulnerabilities, attackers are able to evade mainstream security countermeasures

[1] `http://www.fireeye.com/blog/technical/cyber-exploits/2013/02/the-number-of-the-beast.html`

like those based on signatures.

In this thesis we focus on detecting malicious electronic documents. We provide a proof of concept implementation of our new detection method in Adobe Reader to show how effective it is at detecting malicious Javascript bearing PDF documents.

1.1 Terms

Exploitation technique The method by which an attacker attempts to gain control over a targeted application. Examples include buffer overflows, format strings, use-after-free, etc.

Vulnerability Instance of an exploitation technique in an application, attackers aim to find vulnerabilities and use them to gain control of the application.

Exploitation attempt Attempt by an attacker to exploit a particular vulnerability in an application. The attempt does not need to be successful.

1.2 Problem

Current detection approaches for malicious documents are unable to perform host-based detection at a targeted victim as they suffer from computational penalties often too high for a regular user's workstation [20, 49, 56]. Secondly, novel attack vectors may remain undetected as current approaches do not use the same components as the official document reader. For example, detection tools aimed at malicious PDFs all adopt third-party PDF parsers for their detection logic. Differences in these parsers compared to mainstream PDF readers such as Adobe Reader are used by attackers to evade detection. Further, existing detection techniques focus on detecting a single or a small subset of exploitation techniques. Consequently, attackers have put a lot of effort in designing obfuscation techniques that make identification by these existing tools more difficult, and small variations of the same exploitation technique could remain undetected by these specific detection techniques.

1.3 Contribution

The aim of this research is to find the root cause of exploitation techniques used in Javascript bearing PDFs, and to design a technique to detect this root cause, independent of exploitation technique. We propose to detect malicious Javascript bearing PDFs by observing the Javascript function calls and the function parameter values. Calls to rarely used functions, or calls with exceptional function parameter values may indicate an attempt to exploit a vulnerability in the PDF reader.

Our contribution is threefold. First, we propose an effective detection technique for the detection of malicious Javascript bearing PDFs. The detection technique is general in nature and can in all likelihood be applied to similar domains, such as detection of drive-by-download attempts using Javascript in browsers. Second, our proof-of-concept implementation in the closed source Adobe Reader serves as a stepping stone for instrumenting closed source interpreters with the purpose of malware detection. Finally, since our implementation uses the same components as Adobe Reader it can be modified to extract the embedded Javascript from PDF documents. Since existing detection approaches require Javascript to be extracted from PDF documents prior to analysis, our implementation can either be used as a Javascript extractor for these tools, or to verify the Javascript extracted by third-party PDF parsers.

1.4 Research Questions

Generally, detecting exploitation attempts is a difficult problem. A great number of offensive techniques to exploit applications exist, ranging from trivial buffer overflows to sophisticated multi-stage attacks. Many detection techniques have been proposed in the literature that target a single or a small subset of exploitation techniques efficiently. There exist tools aimed at more general detection, but they suffer from computational penalties which limit a wide-spread adoption.

This research aims to answer the following research question(s), to advance the field of general detection techniques.

> *How can we detect exploitation attempts, independent of exploitation technique and at the same time efficient enough to be run on a targeted victim without affecting its computational performance.*

To support answering the main research question, we consider exploitation techniques used in malicious Javascript bearing PDFs and try to identify their root case.
We've further split the research question into the following sub-questions:

1. What exploitation techniques exist?
2. How are the exploitation techniques applied in malicious PDFs?
3. What is the current state of detecting malicious Javascript bearing PDFs?
4. What detection techniques exist aimed at detecting the above exploitation techniques?
5. What are the detection challenges specific to PDFs?
6. Do the exploitation techniques share a root cause?
7. If such a root cause exists, what detection and false positive rates can be achieved by identifying the root cause?

1.5 Thesis Outline

Chapters 2 and 3 discuss general exploitation and detection techniques, independent of application or domain to introduce the reader to the field of exploit detection. The focus does however lie on exploitation techniques commonly seen in malicious documents. Chapter 4 continues with the state of the art concerning detection of malicious Javascript bearing PDFs followed by Chapter 5 on challenges specific to the detection of malicious Javascript bearing PDFS.
Chapters 6 and 7 introduce our new detection approach and the implementation of the approach in Adobe Reader. Our approach is evaluated in Chapter 8 and finally we conclude in Chapter 9.

Chapter 2

General exploitation techniques

In this Chapter we describe current exploitation techniques that apply to any application. Given the large number of different exploitation techniques, we do limit the discussion to techniques commonly found in malicious documents.
For a successful exploitation of an application, some conditions must hold. First, the attacker needs to be able to provide some form of input to the application (for example, a document to an electronic document reader, font files, images, data over a network stream, etc.). Second, the application, or any third-party module involved in processing the user input must contain a vulnerability that gives the attacker control over the content of the memory, and eventually control over the execution flow of the application. Once the attacker has control over the memory, she can change it in such a way that the application executes instructions of the attacker's choosing.

2.1 Shellcode

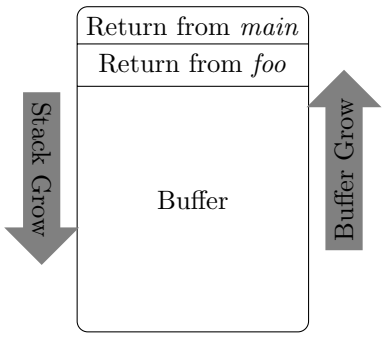
One way of executing malicious instructions is by loading *shellcode* into the application's memory, and diverting the execution flow of the application to this shellcode (Section 2.2 and 2.2.1). There may be size restrictions on the memory region that the attacker controls. Therefore, the shellcode is usually a small piece of code that starts a *reverse-shell*[1] (hence the name, *shell*code) or downloads and executes additional components from the Internet, which is referred to as *dropping*.

In some cases, the shellcode can be delivered to the application through normal user input (such as a network stream). It can even be part of the input that triggers the vulnerability to divert execution flow to the shellcode.

[1] When a reverse-shell is started on a victim's computer, it listens for incoming connections from the attacker. Once connected, the attacker can execute arbitrary commands on the victim's computer

As application hardening techniques became more commonly implemented over time, loading shellcode has shifted towards *heap spraying* which will be discussed in Section 2.4.

2.2 Buffer Overflows




```
void foo( const char* UserInput )
{
    char Buffer[ 200 ];
    // Copy until null-terminator
    strcpy( Buffer, UserInput );
}
```

Table 2.1: School board example of a stack-based buffer overflow

An easy to exploit, and prevalent type of vulnerability is a (stack-based) buffer overflow. Buffer overflows occur when the application copies input from the user into a buffer that is not large enough to hold the input. Figure 2.1 shows a simplified x86 stack. A 200 byte buffer is allocated on *foo's* stack frame and user-input is copied into this buffer without respecting the 200 byte limit. Since the buffer grows upwards in the Figure (Buffer[0] is at the bottom, Buffer[199] at the top), the attacker can overwrite the return address that was pushed onto the stack before the buffer ('*Return from foo*') [4]. On x86 architecture, whenever a CALL instruction is executed, the memory address of the instruction following the CALL instruction is pushed onto the stack. Then, when the CPU next executes a '**RET**' instruction, this address, whose value is now controlled by the attacker, is popped from the stack and execution is resumed at this address. Since the attacker controls the stack, execution is resumed at an attacker-controlled memory location, such as previously loaded shellcode.
In a correct, non-vulnerable version of the example, the '*strcpy*' call would need to be replaced with a call to '*strncpy(Buffer, UserInput, 200)*'. This would limit the number of bytes copied into *Buffer* to 200, preventing '*Return from foo*' to be overwritten when *UserInput* is larger than 200 bytes.

One technique to prevent the execution flow from diverting to an attacker controlled location is by performing sanity checks on the return addresses on the stack. This is typically done by placing a 'canary' value below a return address on the stack [9]. Each time the CPU executes a RET instruction, it checks the validity of the canary value before diverting execution flow to the return address. Clearly, the canary value must be unpredictable.

If in the example the buffer would be large enough to contain the shellcode, the attacker could divert control to the same buffer and the shellcode would be executed on the stack. With Data Execution Prevention (DEP)[18], regions in memory where executable code is not expected are marked as Non-Executable. It is uncommon for a typical application to execute instructions that are located on the stack, therefore the stack can be marked as Non-Executable in most typical applications. This does not prevent the attacker from overwriting return addresses on the stack, but she can no longer execute shellcode that is in the same buffer as the one used to overwrite the return address.

DEP may cause compatibility problems with existing applications, it is therefore possible to programmatically disable DEP which will be discussed in Section 2.4 on Heap Spraying.

2.2.1 Integer overflow

In some cases, a prerequisite to a buffer overflow is that an integer value overflows. Computers are natively only able to express numbers up to a certain value. For example, on 32-bit systems the maximum value of an unsigned integer is $2^{32} - 1$ (or '4.294.967.295'). Integer overflow [21] vulnerabilities are particularly difficult to detect [14] as integer overflows cannot (conveniently) be detected after they have happened.

Integer overflows do not allow for direct modification of the application's memory, or for control over the execution flow. A common case where integer overflows can enable buffer overflow vulnerabilities is when arithmetics cause an overflow and the result is used to determine the buffer size. Figure 2.1 shows an example of how an integer overflow can enable a buffer overflow vulnerability.

```
int* copyArray( int* input, int count )
{
    int* result = (int*)malloc( count * sizeof( int ) );
    if( NULL == result )
        return NULL;
    for( int i = 0; i < count; ++i )
        result[ i ] = input[ i ];

    return result;
}
```

Figure 2.1: Buffer overflow with integer overflow prerequisite

If an attacker controls *count*, she directly controls how much memory is allocated for *result*. By choosing a sufficiently large value for *count*, the multiplication '*count * sizeof(int)*' will overflow and become a different, smaller number. Since the overflow is undetected, the call to *malloc* will return successfully, but allocate less memory than expected, and the for loop will write past

the end of *result*, resulting in a heap overflow.

Other execution flow diversion techniques

```
int main( int argc, const char** argv )
{
    char Buffer[ 200 ];
    void (*pFoo)( const char* ) = &foo;

    // Copy until null-terminator
    strcpy( Buffer, argv[1] );

    pFoo( Buffer );
    return 0;
}
```

Figure 2.2: Overwriting function pointer to divert execution flow

```
int main( int argc, const char** argv )
{
    char Buffer[ 200 ];
    __try {
        // Copy until null-terminator
        strcpy( Buffer, argv[1] );
    } __except( GetExceptionCode() ==
       EXCEPTION_INT_DIVIDE_BY_ZERO ?
        EXCEPTION_EXECUTE_HANDLER :
            EXCEPTION_CONTINUE_SEARCH ) {
       // Handle exception
    }
    return 0;
}
```

Figure 2.3: Overwriting SEH-chain entry to divert execution flow

In the section on buffer overflows, the return addresses present on the stack was overwritten to divert the execution flow of the application. This is the most generally applicable technique, but others exist. Consider an artificial application as shown in Figure 2.2. A function pointer, pointing to the *foo* function is placed on the stack before a buffer overflow vulnerability. After input from the command line has been copied to the 200 byte buffer, *foo* is called through the function pointer *pFoo*. Exactly as with overwriting the return address in the previous example, the attacker can now overwrite the value of the *pFoo* pointer,

and thus directly control the execution flow of the application. Note that stack canaries will not be effective against this type of vulnerability as the attacker does not need to overwrite the return address. Some compilers implement array variable reordering [24] which would cause '*Buffer*' to be allocated at the top of the stack frame, in which case overflowing '*Buffer*' does not overwrite '*pFoo*'.

A similar but possibly more prevalent method is to overwrite SEH (Structured Exception Handler) pointers in Windows applications. In short, when a Windows application uses exceptions as shown in Figure 2.3, a SEH-chain is pushed onto the stack so that in case of an exception, execution flow can be diverted to a suitable exception handling routine. Again, by overwriting one of the SEH-chain entries with attacker controlled values she can divert execution flow by triggering an exception (such as divide by zero or illegal instruction) which causes the control flow to divert to the attacker controlled value.

Use-after-free

```
1  A* pA = new A;
2  delete pA; // pA is now a 'dangling pointer'
3
4  B* pB = new B; // sizeof(A) == sizeof(B)
5  // It is likely that pA == pB here
6
7  // Here, B's vtable is used for the function call.
8  pA->virtual_function_call();
```

Figure 2.4: Artificial example of use-after-free attack

Use-after-free vulnerabilities are very application-specific, and difficult to exploit [3, 61]. In general, the structure of the attack is as follows:

1. the application allocates an object 'A' on the heap;
2. the object 'A' is freed, any pointers to 'A' are now considered *dangling*, meaning that they refer to an object that no longer exists;
3. the attacker allocates one or more objects 'B' of the same size as 'A';
4. with a high probability, one of the 'B' objects will be allocated in the previous heap location of 'A';
5. the application calls a virtual function on the 'A' object, whose memory location is currently filled be the attacker-controlled 'B'.

Figure 2.4 shows in pseudo-code what happens in a use-after-free attack. Since the vtable of 'B' is used in the call on line 8, the attacker gains control of the application's control flow under the right conditions.

In object oriented languages such as C++, the vtable is used to determine the memory location of a function at run-time. This run-time, as opposed to compile time dependency allows classes to form a hierarchy, and overload functions of their base classes. We can think of the vtable as a list of 'pFoo' pointers as seen in Figure 2.2. So naturally, if the attacker control the vtable of an object, she controls the application's execution flow whenever a virtual function of this object is called.

2.3 Return Oriented Programming (ROP)

A technique to circumvent Data Execution Prevention is not to overwrite one return address, but rather to place a chain of return addresses on the stack in such a way that each 'RET' instruction executes a small portion of the malicious code. Since the attacker can not execute code on the stack, she needs to find 'gadgets'[2] in the application's code section and push the addresses of these gadgets on the stack. It has been shown that in a typical application, ROP can be used to execute arbitrary code [52]. When the attacker uses gadgets from the standard-C library, which is present in most applications, this is called a *return-to-libc* attack.

Consider that an attacker wishes to increment the EAX and EBX registers. She needs to find one or more executable locations in memory that effectively perform *inc EAX, inc EBX*, followed by a RET instruction, and place the addresses of these instructions on the stack.

Buffer overflows, ROP, and most offensive techniques in general, require the attacker to be aware of the location of certain objects in memory. Clearly when the attacker overwrites a return address to jump to her shellcode, she needs to know where in memory the shellcode resides. Address Space Layout Randomization (ASLR) [18, 53] is a technique that randomizes an application's address-space layout, either at compile time or at each new execution of the application. ASLR-enabled dynamic libraries will be loaded into memory at unpredictable offsets each time. Shacham et al.[53] show that ASLR can effectively be beaten on 32-bit systems by brute-force. The authors show that any buffer overflow can be modified to beat ASLR in on average 216 seconds of overhead. More importantly, even in modern operating systems, not all sections of the address space are randomized, due to executables that have fixed load addresses [26] or dynamic library incompatibilities with ASLR [19]. Finally, in some cases the base address of a loaded dynamic library can be brute forced [53] or calculated through a leaked pointer [64].

ROP attacks are made more difficult by ASLR because the location of the gadgets will be unpredictable. Though even with a limited number of ASLR-disabled modules, it is still possible to perform part of an attack using the ROP technique, such as disabling Data Execution Prevention.

[2]Small set of instructions ending with a 'RET' instruction. Each 'RET' pops the address of the next gadget from the stack and starts executing it

2.4 Heap Spraying

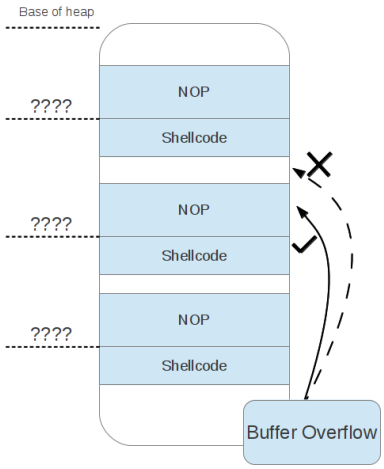


Figure 2.5: Heap littered with shellcode

With ASLR enabled, an attacker has much less certainty about where in memory her shellcode resides. This caused the rise in popularity of using heap spraying techniques. Heap spraying is a technique to litter the heap of an application with instances of the shellcode at predictable address ranges. If an application supports interpreted languages such as Javascript, then these can be used to perform the heap spray 'legitimately', i.e. without exploiting a vulnerability. Otherwise a vulnerability must exist in the application which allows the attacker to allocate large portions of heap space, and fill it with copies of the shellcode.
Once the heap spraying stage is complete, a vulnerability such as a buffer overflow can be used to divert the execution flow to a semi-random location in the application's heap which is now likely to contain the shellcode.

In practice, the shellcode is often prepended with a NOP-slide[3] to increase the likelihood of diverting the execution flow to a location that will eventually lead to execution of the shellcode. Figure 2.5 shows a heap, littered with instances of the shellcode. Somewhere in the code, a buffer overflow or other type of exploit is triggered to divert the execution flow. If a memory region marked other than '*NOP*' is hit, the application may crash.

If the heap is marked as Non-Executable by Data Execution Prevention techniques, a ROP attack is first needed to programmatically disable DEP. Techniques to disable DEP using Return Oriented Programming are described

[3] A NOP-slide is a set of instructions that have no effect on the CPU state, e.g. *inc EAX, dec EAX*

in [55]. While heap sprays in itself are not malicious, they are a crucial part of modern malware that is able to operate on systems that have been hardened using ASLR techniques. In fact, we will see in Section 6.1 that all recent examples of Adobe Reader exploits use the Javascript interpreted language to spray shellcode into the heap before triggering any vulnerabilities.

Chapter 3

Existing detection approaches

In this chapter we discuss the current state of detection for the exploitation techniques listed in the previous section. First, shellcode detection is discussed in Section 3.1 followed by techniques to detect Return Oriented Programming and heap spraying attack in Section 3.3 and 3.4 respectively.. Finally, general approaches that do not focus on detecting one particular exploitation technique are discussed in 3.5.

3.1 Shellcode

Shellcode detection has been a much discussed topic in the literature. Ideally one would like to scan an application's input such as a PDF document or a network stream and be able to detect shellcode without having to execute anything. This is known as static detection.

The early forms of static detection techniques were signature based, where input is simply scanned for sequences of bytes, called signatures, used previously in known attacks. If the signatures are carefully constructed to uniquely identify certain malware, the false positive rate is low which makes them preferred in Intrusion Prevention Systems, where false positives mean disruption to honest users of the system. Clearly, signature based systems require constant updates when new attacks are found in the wild and thus result in large signature databases.

Toth and Kruegel propose static techniques to search for common shellcode patterns such as NOP-slides [59] or structural similarities between different worm instances [32]. These methods have similar advantages as signature based systems, but due to their static nature they are easily evaded. For example, shellcode that modifies itself during execution [45] will evade detection as the static method is only able to scan the initial form of the shellcode, it is unable to scan the modified, malicious form of the shellcode. Furthermore, the shellcode

	Positive	**Negative**
Static	Speed Undetectable Safe (nothing is executed) All code available, independent of whether it is actually executed often	Prone to obfuscation Cannot scan run-time dependent input
Dynamic	Resistant to obfuscation Run-time data available	Detectable May miss rarely executed parts of code Prone to time outs

can use indirect jumps to avoid detection, as the jump location depends on data that is only available at run-time [45].

Dynamic techniques attempt to disassemble input into valid CPU instructions, and execute them on an (often emulated) CPU. These techniques are more robust against shellcode whose execution flow depends on run-time data and self-modifying shellcode, as eventually the underlying malicious shellcode will be executed and analyzed. A problem with dynamic analysis is that if the shellcode is executed on an emulated CPU, the shellcode may detect imperfections in the emulation and stop execution, or it may unintentionally stop functioning due to the imperfections. Also, in targeted attacks the attacker may verify presence of specific characteristics of its target, such as a specific user name, before showing malicious behavior. Since dynamic techniques attempt to emulate the shellcode, and only a limited time frame is available for this analysis, an attacker can evade detection by inserting long running loops to reach the detector's execution threshold before executing the malicious shellcode. Finally, emulation based detection techniques may lack a view of the system's state such as the complete address space of the target application, or CPU registers. The implications can be limited by instrumenting the emulator with commonly used system libraries which the shellcode may use, resulting in an emulator that more closely resembles a real machine.

Polychronakis et al. propose a dynamic technique for network-level detection of self-decrypting polymorphic shellcode [45]. Self-decrypting polymorphic shellcode is shellcode in encrypted form, prepended with a decryption routine to make it self-decrypting. The technique combines two heuristics to detect the decryption routine of the shellcode. First, for the decryption routine to work it must obtain the current absolute memory address of the shellcode, this is referred to as *Get Program Counter*, or *GetPC*. Three common techniques of GetPC are described in [45]. Secondly, the decryption routine will perform many read operations in the small memory region where the encrypted shellcode resides. Polychronakis et al. combine these two heuristics to effectively detect polymorphic shellcode. Since the detection technique focuses on detecting the

decryption routine present in polymorphic shellcode, it does not detect plain or metamorphic[1] shellcode.

Polychronakis et al. improve on this in [46] by proposing a general purpose dynamic shellcode detection technique based on heuristics. The heuristics attempt to detect common shellcode behavior such as kernel32.dll base address resolution, and SEH-based GetPC code.

3.2 Use-after-free

Use-after-free vulnerabilities are difficult to detect efficiently. One way is to track whether a pointer is 'dangling' (referring to a freed object). This results in a run-time cost at every memory access and is generally unacceptable. More recently research has shifted towards more secure memory allocation algorithms [3, 39, 25] with promising results.

3.3 Return Oriented Programming

Detection of Return Oriented Programming attacks have received less attention in literature, but the number of attacks using ROP is expected to grow with increasing adoption of Data Execution Prevention techniques. Currently, ROP attacks as seen in the wild are known only to have been used to disable memory protection such as DEP, and to divert control to traditional shellcode [47]. This means that the attack as a whole could still be detected if the ROP stage goes unnoticed. However, it has been shown that ROP can be used to execute arbitrary code in a typical application [52]. Such a ROP-only attack would go unnoticed by shellcode detectors, as there is no shellcode.

ROP attacks can be stopped in two ways, either by reducing the number of available gadgets in an application or by scanning memory buffers that hold user input for sequences of gadget addresses. To reduce the number of available gadgets, compiler extensions [36, 42] have been proposed that specifically limit the number of usable gadgets emitted by the compiler. While this is an effective technique, quick and widespread adoption is inhibited by the need to recompile existing applications. Secondly, run-time solutions as proposed in [16, 20] incur significant run-time overhead which limits their adoption. Finally, Vasilis Pappes et al [44] introduce a method to reduce the number of available gadgets on existing binaries using in-place code randomization. As opposed to other methods [1, 13, 30] that work on existing binaries, the in place code randomization does not require debugging symbols to be available. In fact, debugging symbols are typically not available in commercial software and thus would have similar drawbacks as the compiler extensions.

A second technique, dubbed **ROPScan** [47] by Polychronakis and Angelos takes a similar approach to dynamic shellcode detectors. ROPScan initializes

[1] Metamorphic shellcode is modified before execution, and thus does not contain a decryption routine

a CPU emulator with a snapshot of the virtual memory of the application it aims to protect. Next, the input is scanned for valid addresses within the process' virtual memory. Existence of valid addresses in an input alone is not an indication of a ROP payload, as by chance data may contain valid addresses. Therefore, when a valid address is found ROPScan attempts to start execution at that address. If the execution flow resembles ROP behavior (chain of short sequences of instructions followed by 'RET'), ROPScan marks it as malicious.

3.4 Heap Spraying

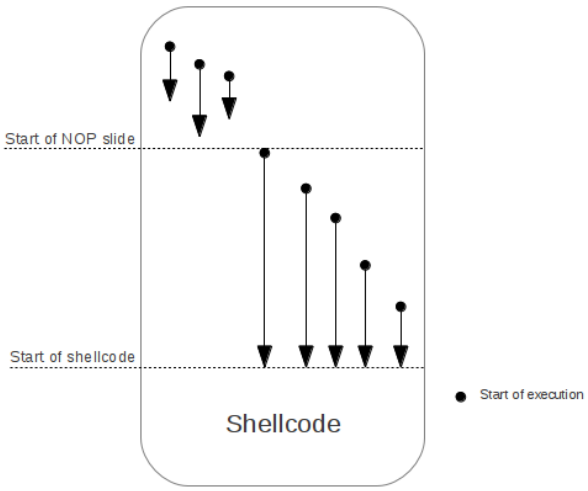


Figure 3.1: Memory block being scanned by Nozzle

Ratanaworabhan et al. introduce **Nozzle** [50]. Nozzle hooks into the family of memory allocation functions such as *malloc*. It keeps track of all currently allocated memory blocks, and periodically scans these blocks for valid x86 instructions. Since arbitrary data may contain many valid x86 instructions, these instructions are first analyzed to determine whether they look like a NOP slide. This is done by constructing execution flow graphs from arbitrary starting points. It is considered a NOP slide if many execution flows reach the same basic block, which may indicate the start of the shellcode. Figure 3.1 illustrates this.

When an application makes many heap allocations, the overhead of Nozzle is significant, making it impossible to deploy at an end user's machine.

3.5 Other general detection techniques

Besides techniques designed specifically to detect one or a small subset of exploitation techniques, more general techniques exist. A technique quite different to the previously discussed is called *automated dynamic malware analysis*. Here, the goal is to detect the observable effects of a malware infection, regardless of the exploitation technique used. Tools implementing this technique either use a fully emulated environment [10, 27] or instrument an environment [65] in such a way to monitor security-related events of the sample under analysis. Security-related events include *file system* and *network activity*, *registry modifications* and *process creation*.

Advantage of these tools are that no specific exploitation technique is being targeted, thus previously unknown malware can be detected as long as it triggers some maliciously looking set of security events.
The listed security events are perfectly legitimate and performed by many legitimate applications, so often human interaction is required to determine whether the sample is malicious or not. Consider an FTP server application. Under normal circumstances it can read, create and write files, spawn sub-processes to delegate part of the work load, etc. An attacker may want to do the same things when she exploits a vulnerability in the FTP server, the only differences is that she does it without the proper credentials. In the case of electronic document readers, this is less of a problem. It is unusual for a document reader such as Adobe Reader to download and execute a binary file, thus observing this pattern indicates that the sample is malicious with high probability.

Systems like CWSandbox [65] use API hooking to log the security events, which is easily detected by the malware, allowing it to terminate or perform only benign actions in the presence of CWSandbox. TTAnalyze [10] improves on this by completely emulating a PC in software making detection by the malware more difficult but still possible through exploiting timing differences between a real system and the emulated PC.

Techniques similar to malware analysis can be used for prevention. As noted, it is unusual for document readers to download and execute binary files and as of such, this action can be prohibited without the explicit consent of the user, to stop an infection. The process of explicitly allowing what certain applications can do, as opposed to explicitly disallowing, is called **white listing**.

Chapter 4

Related Work

Some of the early work on detecting malicious PDF documents has focused on *N-gram* [54] analysis of the PDF file. W.-J. Li et al. propose [37] a static detection method that uses N-gram to compare documents to normality-models based on benign and malicious documents. The scanned document is classified as benign or malicious depending on a similarity score between the two normality models. The downside of this technique is that is requires sufficiently large data sets of both benign and malicious documents. Furthermore, the method requires the malicious content of the scanned document to be sufficiently large for it to have a statistical meaning.

Detecting malicious Javascript, or Javascript based heap-spray attack in PDF documents is a relatively new field. However, we can learn from previous work done in the field of drive-by-download detection. Drive-by-downloads often exploit vulnerabilities in the browser's Javascript interpreter, and use Javascript to perform a heap-spraying attack, much like in malicious PDF documents.

M. Egele et al. introduce a technique [22] to detect code injection attacks in browsers through Javascript heap sprays. The authors note that most Javascript heap sprays are performed through consecutive string allocations as shown in Figure 4.1. To detect the code injection, Egele et al. instrument Mozilla's Javascript interpreter SpiderMonkey[41] to scan each newly allocated string with shellcode emulator libEmu [8]. LibEmu attempts to disassemble the string buffer and reports when it finds 32 consecutive valid CPU instructions, or when its shellcode detection heuristics detects one of the GetPC techniques [45].

As with detection of shellcode, both static and dynamic detection techniques have been proposed to detect malicious PDF documents. Laskov and Ŝrndić introduce **PJScan** [35], which is a tool for static detection of malicious PDF documents that use Javascript. The extracted Javascript is interpreted on SpiderMonkey, which is modified to only do lexical analysis of the Javascript code. The lexical tokens are then classified as benign or malicious by the One-Class Support Vector Machine(OCSV)[17] learning method. As with n-gram analysis, the OCSV model needs to be trained with a representative set of malicious documents. An advantage of the system is its high performance, compared to

```
var payload = unescape("<Ommitted for clarity>");
var nop ="";
for (iCnt=128;iCnt>=0;--iCnt)
    nop += unescape("%u9090%u9090%u9090%u9090%u9090");

heapblock  = nop + payload;
bigblock   = unescape("%u9090%u9090");
headersize = 20;
spray      = headersize+heapblock.length

while (bigblock.length<spray)
    bigblock+= bigblock;

fillblock = bigblock.substring(0, spray);
block     = bigblock.substring(0, bigblock.length-spray);

while(block.length+spray < 0x40000)
    block = block+block+fillblock;

mem = new Array();
for (i=0;i<1400;i++)
    mem[i] = block + heapblock;
```

Figure 4.1: Heap-Spraying attack in Javascript by Debasis Mohanty

dynamic methods. The authors note that the system has a high rate of false positives when analyzing benign documents that contain Javascript resulting in a tool that is only marginally better than flagging all Javascript bearing documents as malicious. A proposed solution would be to adopt a similar two-class model as in N-gram by training a separate model with benign Javascript bearing documents however at the time of writing, this has not been implemented.

The first step of a dynamic detection technique introduced by Tzermias et al. is the same as in PJScan. **MDScan** [60] feeds the extracted Javascript to SpiderMonkey, which has been extended to include the most often used extended Javascript API methods found in malicious documents. Next, the technique scans Javascript string buffers in a similar way to Egele's method against drive-by-downloads. MDScan scans the buffers using shellcode detector Nemu [46] and has later been extended to include ROPScan [47]. In Chapter 8 on evaluation, we will compare the effectiveness of MDScan with our proposed solution more thoroughly. The authors note that their technique may be ineffective if the attacker exploits vulnerabilities in the PDF Reader's Javascript interpreter that may not be present in SpiderMonkey or if extended API methods are used that the authors did not yet include in SpiderMonkey.

Any tool that uses a different Javascript interpreter than Adobe's suffers from three drawbacks. First, the different Javascript interpreter needs to be extended to support Adobe's extension to the Javascript API for PDF documents, which is large and difficult to fully implement. Secondly, malware can leverage non-standard behavior of Adobe's Javascript interpreter to detect whether it is being executed on Adobe's interpreter or a different one. For example, Adobe's interpreter does not allow the type of global variables to change after initialization, while the Javascript specification specifically allows this [66, 48]. How this can be used to detect whether the Javascript code is executed on a different Javascript interpreter is shown in Figure 4.2. On line 2, a string literal is assigned to a variable that was previously of type 'boolean'. If the Javascript is interpreted on Adobe's interpreter, the resulting value of 'testIfAdobe' will be true, as opposed to being "string variable" on a standard complying Javascript interpreter, such as SpiderMonkey.

Finally, before being able to run the Javascript found in a PDF document on a different interpreter it needs to be extracted from the PDF document. Due to the complexity of the PDF standard, this is a non-trivial task that we will further elaborate on in Chapter 5 on challenges involved in detection malicious Javascript bearing PDFs.

Automated malware analysis engines have been adopted to accept PDF files as well. This resulted in an easier combination of static and dynamic techniques. MalOffice [23] first scans the input with certain AntiVirus scanners and PE-detectors to look for known malware and embedded executable files. Next the Javascript is extracted from the PDF in a similar way to PJScan and MDScan, using the pdftoolkit and it looks for suspicious variable names, code obfuscation techniques and the use of known vulnerable Javascript functions. Code obfuscation techniques can also be used for legitimate purposes, such as protection of Intellectual Property and is thus not a definitive sign that the

```
1  testIfAdobe = false;              // boolean type
2  testIfAdobe = "string variable"; // string type
3
4  // type change 'ignored' by Adobe
5  if( testIfAdobe == true )
6  {
7      // Code is running under Adobe Reader
8  } else
9  {
10     // Code is running under custom interpreter
11 }
```

Figure 4.2: Detecting non-standard behavior of Adobe's Javascript interpreter

PDF is malicious. On top of the static analysis, MalOffice submits the sample to CWSandbox [65] and applies some heuristics to the generated report to determine if the sample is malicious. The authors have composed a black and white list of suspicious and benign actions respectively.

Discussion Figure 4.3 shows the flow of typical PDF exploits, either using Javascript, native vulnerabilities or a combination of the two. For each detection technique, and applicable general detection techniques it is shown which stage of the exploit the technique focuses on. In Chapter 8 we compare the detection performance of existing tools with each other and our proposed detection technique.

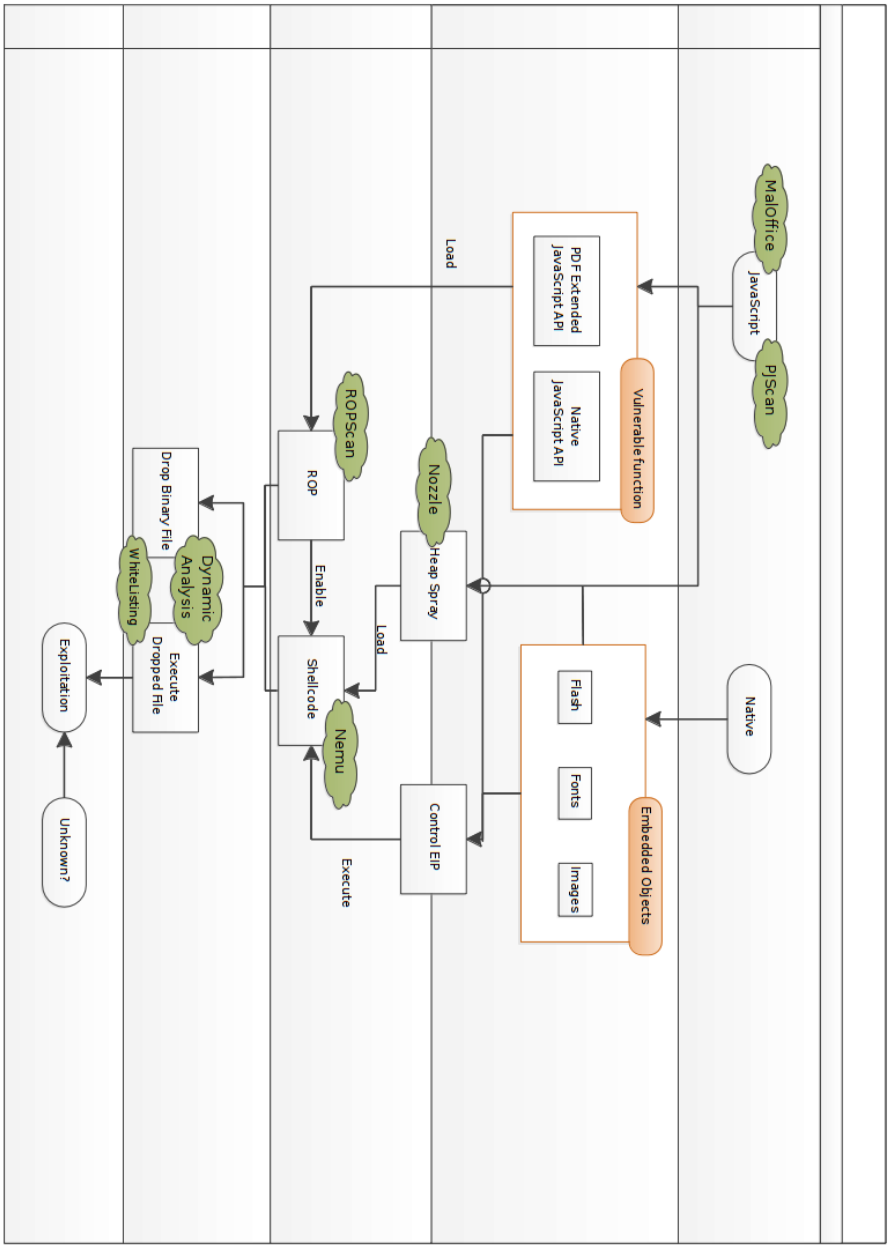


Figure 4.3: Anatomy of PDF exploits and detection techniques

Chapter 5

Challenges in malicious PDF detection

In addition to detecting exploits such as the ones listed in Chapter 2 there are additional challenges that need to be overcome when detecting malicious PDFs.

What follows is a short description of the main structure of PDF documents and how Javascript plays a role in it.
A PDF file's structure contains the following elements:

1. *header* with PDF specification version number (%PDF-x.y).
2. *body* with actual content of the PDF file (PDF Objects).
3. *cross-reference table* listing indirectly referenced objects and their location in the file.
4. *trailer* with location of the cross-reference table and some objects in the file's body.

In the *body*, the PDF specification allows for eight types of objects: *booleans*, integer and real *numbers*, *strings* (as a sequence of literal characters, or a sequence of hexadecimal numbers), *names* (used as identifiers), *arrays*, *dictionaries*, *null* and *streams* objects (dictionary objects followed by a sequence of bytes surrounded by **stream** and **endstream** keywords. Streams can be used to represent large objects, such as images or Javascript code. Streams can be encoded or uncompressed using filters listed in Table 5.1).

Some notable features [66] of Adobe Reader include, but are not limited to:

- OpenGL Rendering.
- ADBC (Adobe Database Connectivity).
- Execute embedded Flash files.
- Execute embedded Javascript code.

- Embed arbitrary files, exportable from the GUI.
- Launch arbitrary programs.
- Play embedded sound and video files.
- Digital signatures, DRM, XML parsing, bar codes
- ...

As we have seen in Chapter 4 on related work, most of the current detection tools specific to detecting malicious Javascript bearing PDFs require the Javascript to be extracted from the PDF document before it can be analyzed. Tzermias et al. [60] and Laskov et al. [35] describe some of the challenges involved with extracting Javascript from PDF documents. The complexity (over 2500 pages) and ambiguities of the PDF specification [28] make extraction of all the objects a non-trivial task. Additionally, many PDF readers including the most widespread *Adobe Reader* attempt to render non standard compliant documents to provide a better user experience. In fact, the PDF specification [28] specifically states that it does not specify "*methods for validating the conformance of PDF files or readers*". This gives attackers room to hide their exploit code in obscure places to make extraction more difficult.

Apply filters A relatively simple obfuscation method, is to apply filters to the streams that contain Javascript. The PDF format supports adding any combination of filters, listed in Table 5.1, to streams in the document. Detection tools need to be able to support all filters in order to reveal the actual contents of the stream. Also, the specification allows for filter abbreviations, such as */F1* for the */FlatDecode* filter that must be supported.

An example of a malicious document hiding with the use of filters is described in [7], the document uses a *JBIG2Decode* filter, which is defined in the PDF specification as a pure image filter. However, in the malicious document it is used on an arbitrary object stream which existing detection tools did not expect and thus did not apply the filter, leaving the underlying Javascript invisible.

Encryption Attackers can apply RC4 or AES encryption to object streams, making static analysis more difficult. The encryption key used can be made dependent on some content in the PDF, for example by calling '*getPageNthWordQuad*'. This function requires the PDF to by graphically rendered to obtain the coordinates of the word making it very difficult to implement in custom Javascript interpreters such as SpiderMonkey.

Non-standard PDF Some PDF readers, including *Adobe Reader* are relaxed towards parsing non standard compliant documents. For instance keywords such as *endobj* and *endstream* are accepted when written as *objend, streamend* respectively. Additionally, PDF documents should contain a *cross-reference table*, listing indirectly referenced object and their location in the file. Objects

not listed in this table are still parsed by many PDF readers; in fact, most PDFs are rendered correctly even when the cross-reference table is fully omitted.

In some cases, the size of a PDF object can be described in several ways such as an explicit size specification or the actual number of bytes between *stream* and *endstream* keywords. It is up to the PDF reader's implementation to chose which size specification to respect in case of a mismatch. So even though detection techniques using third party PDF parsers may claim their technique can detect exploits targeting *any PDF reader*, this is not completely true as a specifically crafted PDF may be parsed differently by different PDF reader implementations. Many more PDF parsing idiosyncrasies, including object size ambiguities, are described in [66].

The PDF specification describes extra Javascript methods in the form of an extension to the Javascript API specifically for PDF documents. The API provides functionality for document specific objects, properties and methods. This extension is important in malicious documents. Javascript code or data on which the exploit depends can be hidden in an object that is only accessible through the extension API (such as *getPageNthWord()*). The custom Javascript interpreters used in existing detection tools only implement those objects of the API which are currently used in the wild by malicious PDF documents and which are easy to implement. Functions such as *getPageNthWord()* require the complete PDF document to be rendered (correctly!) in order to return the expected result. The remaining API objects are implemented with an empty function body, so that if a PDF uses one of the unimplemented functions, no run-time error occurs. This means that the exploit will remain dormant when it is executed on a custom Javascript interpreter that does not implement the necessary PDF-specific API function.

It should be clear that extracting all Javascript code is a non-trivial task and furthermore, open source Javascript interpreters need to be extended to include the PDF specific API which has proven difficult [48, 66]. Notable work on PDF analysis has been performed by Didier Stevens [57], specifically on techniques to extract embedded Javascript. Stevens released two important tools; PDFiD and a PDF parser. The first can be used to identify characteristics of a PDF such as whether it contains any embedded Javascript code. Suspicious documents can be analyzed further using Stevens' PDF parser which can for example be used to extract Javascript and possibly embedded Shellcode.

In conclusion, no matter how well designed custom PDF parsers become, there is always a risk of interpreting a PDF file differently from "real" PDF readers such as Adobe Reader that are used by the end user. This may result in false negatives when Adobe Reader is able to extract certain Javascript where the custom PDF parser fails. While analysis of Adobe Reader's parsing capabilities has come a long way, it remains possible that hiding or obfuscation techniques are being used in the wild that are unknown to the security community. This problem is similar to detection of 0-day vulnerabilities.

ASCIIHexDecode	Decodes data encoded in an ASCII hexadecimal representation, reproducing the original binary data.
ASCII85Decode	Decodes data encoded in an ASCII base-85 representation, reproducing the original binary data.
LZWDecode	Decompresses data encoded using the LZW (Lempel-Ziv-Welch) adaptive compression method, reproducing the original text or binary data.
FlateDecode	Decompresses data encoded using the zlib/deflate compression method, reproducing the original text or binary data.
RunLengthDecode	Decompresses data encoded using a byte-oriented run-length encoding algorithm, reproducing the original text or binary data (typically monochrome image data, or any data that contains frequent long runs of a single byte value).
CCITTFaxDecode	Decompresses data encoded using the CCITT facsimile standard, reproducing the original data (typically monochrome image data at 1 bit per pixel).
JBIG2Decode	Decompresses data encoded using the JBIG2 standard, reproducing the original monochrome (1 bit per pixel) image data (or an approximation of that data).
DCTDecode	Decompresses data encoded using a DCT (discrete cosine transform) technique based on the JPEG standard, reproducing image sample data that approximates the original data.
JPXDecode	Decompresses data encoded using the wavelet-based JPEG2000 standard, reproducing the original image data.
Crypt	Decrypts data encrypted by a security handler, reproducing the data as it was before encryption.

Table 5.1: Filters supported by the PDF specification and their intended use

Chapter 6

Proposed detection technique

In this Chapter we derive our novel detection technique from a case study of Javascript vulnerabilities in Adobe Reader.
From the discussion on related work in Chapter 4 we identify 6 problems that we aim to solve in this Chapter:

1. Existing techniques are vulnerable to unknown Javascript hiding or obfuscation techniques.
2. Exploits depending on a specific Javascript interpreter, such as the one embedded in Adobe Reader, may fail to run under existing techniques.
3. Exploits depending on the PDF specific Javascript API may fail to run under existing techniques.
4. Existing dynamic techniques are detectable by the malicious document.
5. Existing dynamic techniques are susceptible to time outs.
6. No existing PDF specific technique is able to do efficient host-based, real-time detection out of the box.

At the root of most classes of vulnerabilities lies improper or missing validation of user input. This allows the attacker to control elements of the application that she is not meant to control, such as the number of bytes to be copied into a buffer and indirectly the return addresses resident on the stack, or pointers to (exception handling) routines. To exploit a vulnerability in the implementation of a function, malware has to supply a parameter that does not comply to the "function specifications", and this non-compliance is usually readily observable if the malware sample is compared to examples of legitimate use of the same function.
Automated tools exist [29, 34, 62] to find vulnerabilities in source code, such

as invalid or missing input validation. However, these tools are cumbersome to use either because of a high false positive rate, or because they require additional annotations [34] from the developer to assist in the vulnerability analysis. Additionally, static analysis tools lack run-time context that a vulnerability may depend on, while dynamic tools may miss vulnerabilities residing in rarely executed parts of the application, where vulnerabilities are likely to reside.

It is clear that it is not sufficient to simply educate developers of potential security problems, the fact is that buffer overflow vulnerabilities still exist, even though it has been over one and a half decade since Aleph One's infamous paper '*Smashing the stack for fun and profit*' [4]. Even if an application is built from the ground up with security in mind, it is only natural that rarely used routines receive less attention from developers compared to commonly used functionality. We discuss the case of the '*media.newPlayer (CVE-2009-4324)*' vulnerability in Adobe Reader. Another function, '*media.createPlayer*' exists which is similar in functionality to '*media.newPlayer*', yet only '*media.newPlayer*' is vulnerable to a use-after-free vulnerability. The Adobe PDF API specification [2] reads:

> *In most cases, it is better to use app.media.createPlayer instead of doc.media.newPlayer to create a media player.*

This suggests that internally at Adobe the '*media.createPlayer* is considered more important and received more attention from developers.

In the next section, we will analyze known vulnerabilities found in Adobe Reader's Javascript interpreter and use the knowledge gained to attempt to answer the research questions. We have implemented a proof-of-concept implementation of the proposed technique in Adobe Reader, as discussed in Chapter 7.

6.1 Case: Adobe Reader's Javascript interpreter

Adobe Reader[1] is an application to parse and display *Portable Document Format*(PDF) documents. The PDF specification is an open standard, therefore Adobe Reader is not the only application available to parse and display PDF documents. Other applications include *Foxit Reader*[2], *Evince*[3], Preview for Mac users [4], and many more. According to Avast! in July 2011, Adobe Reader was used by over 80% of its user base, compared to 4.8% of the users using the second most popular Foxit Reader [6]. We chose to analyze Adobe Reader because of its popularity, especially in corporate environments, and also because of the wide availability of exploit information.

The PDF specification (ISO 32000-1) [28] is an extremely complex document. The specification for the basic PDF functionality counts over 2500 pages [66]. This includes only the ISO specification, the Javascript API reference for

[1] `http://www.adobe.com/products/reader.html`
[2] `http://www.foxitsoftware.com/Secure_PDF_Reader/`
[3] `http://projects.gnome.org/evince/`
[4] `http://support.apple.com/kb/HT2506`

CVE	Vulnerable Function	Anomaly	Samples
CVE-2009-0927	Collab.getIcon	Unusually large string argument '*cName*'	85
CVE-2009-1492	doc.getAnnots	Large negative integer arguments	10
CVE-2009-1493	spell.customDictionaryOpen	Unusually large string argument '*cName*'	0 (linux only)
CVE-2009-4324	media.newPlayer	NULL argument	6
CVE-2008-2992	util.printf	Unusually large floating-point argument	10
CVE-2007-5659	Collab.collectEmailInfo	Unusually large string argument '*msg*'	150

Table 6.1: Existing vulnerabilities inside Adobe Reader

version 8.1 and the XML Forms architecture specification. Not included are the 3D Annotation specification, XMP specification nor any of the font specifications. This complexity makes parsing PDF documents a non-trivial task as we discussed in more detail in Chapter 5. Many vulnerabilities have been found in the parsing logic of PDF readers, but these vulnerabilities will not be the focus of this discussion. Instead, we look at vulnerabilities involving Javascript in PDF documents.

The PDF specification allows for the use of Javascript to provide dynamic content in PDF documents, such as automated form validation. Javascript is a dynamic, weakly typed scripting language commonly used to provide dynamic content on web sites. We will elaborate further on using the proposed technique to protect browsers from drive-by-downloads[5] in Chapter 9, on future work.

PDFs that contain malicious Javascript attempt to exploit vulnerabilities in the Javascript interpreter embedded in the PDF reader. In this section, we will iterate known Adobe Reader exploits that exploit vulnerabilities in the Javascript interpreter. Table 6.1 gives an overview of the most prominent vulnerabilities available today. All the listed vulnerabilities are stack-based buffer overflows, except the *media.newPlayer* vulnerability, which is a *use-after-free* vulnerability. The column *Anomaly* refers to what is unusual in a malicious call to the vulnerable function compared to a benign call. For example, an overly large buffer passed to *'Collab.getIcon'* to trigger a buffer overflow.

Figures 6.1 through 6.6 show examples of malicious function calls for each of the listed vulnerabilities.

[5]Drive-by-download exploits are often carried out through vulnerabilities in the Javascript interpreter of the browser.

CVE-2009-0927: '*Icon collab.getIcon(cName)*'

```
var buffer = unescape("%%10%%10%%10%%1f");
while(buffer.length < 0x6000)
    buffer += buffer;
app.doc.Collab.getIcon(buffer);
```

Figure 6.1: CVE-2009-0927, collab.getIcon malicious example

collab.getIcon is used to return an Icon object associated with the specified name. Example benign usage of this function is to change the icon of a button, based on some user interaction like selection from a drop-down list.

The malicious use of the function passes an extremely large buffer of 24 KB as the *cName* parameter. This triggers a buffer overflow vulnerability in the logic of the function.

CVE-2009-1492: '*Annot[] doc.getAnnots(nPage, nSortBy, bReverse, nFilterBy)*'

```
this.getAnnots
    (-134217728,-134217728,-134217728,-134217728);
```

Figure 6.2: CVE-2009-1492, doc.getAnnots malicious example

doc.getAnnots is used to return an array of Annot objects based on the optional search parameters. Example usage of this function is to retrieve all Annot objects on a given page, for example *doc.getAnnots(1)* retrieves all Annot objects from the second page in the PDF.

The malicious use of the function passes large negative numbers as all the search parameters causing memory corruption which may lead to arbitrary code execution.

CVE-2009-1493: '*bool spell.customDictionaryOpen(cDIPath, cName, bShow)*'

```
spell.customDictionaryOpen {cName:repeat(4096,
    unescape("%u0909%u0909"))} );
```

Figure 6.3: CVE-2009-1493, spell.customDictionaryOpen malicious example

customDictionaryOpen is used to add a custom dictionary to the list of available dictionaries.

The malicious use of the function involves passing an overly large buffer to parameter *cName*, similar to the *getIcon* vulnerability. This triggers a buffer overflow vulnerability in the logic of the function. This vulnerability is related to CVE-2009-1492, *doc.getAnnots* and is only applicable on Unix systems, therefore this vulnerability is not included in our tests.

CVE-2009-4324: '*MediaPlayer media.newPlayer(playerArgs)*'

```
util.printd("1.000000000.000000000.1337 : 3.13.37",
    new Date());
try {
    media.newPlayer(null);
} catch(e) {}
util.printd("1.000000000.000000000.1337 : 3.13.37",
    new Date());
```

Figure 6.4: CVE-2009-4324, media.newPlayer malicious example

media.newPlayer is used to instantiate a MediaPlayer object. The documentation on *media.newPlayer* states that in most cases it is better to call *media.createPlayer* which has the same function signature as *media.newPlayer* but is not vulnerable to the same vulnerability. This may indicate that more thought has gone into developing the *media.createPlayer* function compared to the vulnerable *media.newPlayer*.

The malicious use of the function passes a null argument, causing a use-after-free vulnerability to be triggered. A requirement is that prior to the call to *media.newPlayer* an object of the same size is placed on the stack. Samples in the wild commonly use the Date object, which due to the invalid checking of null argument will be used instead in the call to *media.newPlayer*.

CVE-2008-2992: '*string util.printf(cFormat, ...)*'

```
var num = // <long number ommited: ~12 × 10^294>
util.printf("%45000f",num);
```

Figure 6.5: CVE-2008-2992, util.printf malicious example

util.printf is used to format one or more variables as a string according to a given format. Example usage of this function is to convert a number into its hexadecimal representation.

The malicious use of the function attempts to parse an overly large number as a floating point. This triggers a buffer overflow vulnerability in the logic of the function.

CVE-2007-5659: '*collab.collectEmailInfo(msg, ...)*'

```
Collab.collectEmailInfo( {msg:repeat(4096, unescape
    ("%u0909%u0909"))} );
```

Figure 6.6: CVE-2007-5659, collab.collectEmailInfo malicious example

collab.collectEmailInfo is an undocumented method in the *Collab* API, but its purpose can be guessed from its name. The malicious use of the function passes an overly large buffer as *msg* parameter, similarly to the *Collab.getIcon* vulnerability. This triggers a buffer overflow vulnerability in the logic of the function.

6.2 Approach

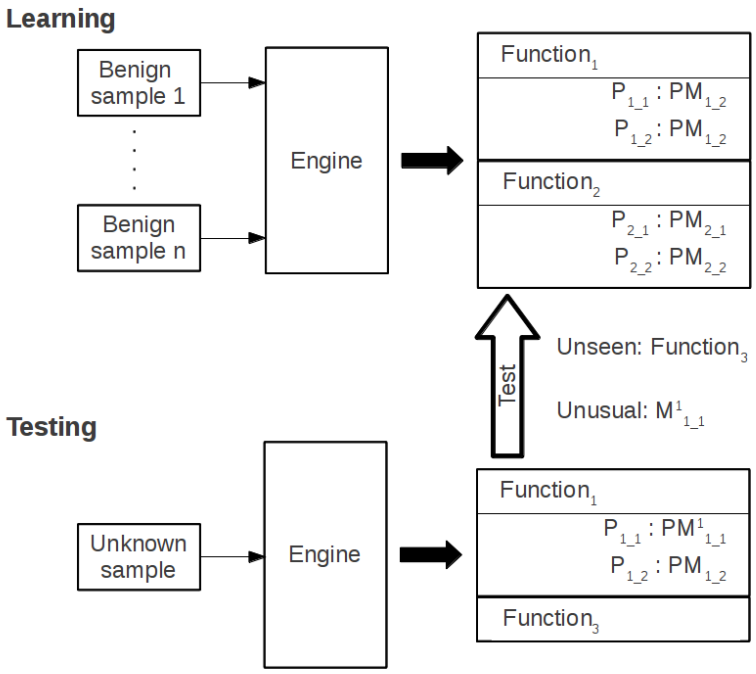


Figure 6.7: Phases of the proposed technique

In this section we propose a novel detection technique dubbed ***FCScan***. Even though the discussion throughout the thesis is focused on detecting malicious Javascript in PDFs, ***FCScan*** can in principle be applied to other domains such as detection of malicious Actionscript in Adobe Flash objects, malicious Visual Basic for Applications macros in Microsoft Office documents or detection of drive-by-download attempts using Javascript in browsers. In Chapter 9 on future work we will discuss the effort involved in porting ***FCScan*** to other domains.

On a high level, ***FCScan*** is a technique to detect anomalies in the malicious use of functions, compared to their benign use. To do this, we model the characteristics of a benign function call and compare these characteristics to future uses of the function. In the first of two phases, ***FCScan*** is fed with samples that are known or expected to be malware-free (benign samples). From these benign samples, ***FCScan*** creates a model of the function identifier and each function parameter, we call this the detection model. The approach is

deterministic, meaning that for a given input the resulting model is always the same.

In the second phase ***FCScan*** is fed with samples that are unknown to be benign or malicious. We compare function identifiers and the function parameter characteristics from the unknown sample with the previously obtained benign detection model. Intuitively, the approach is based on self-learning white listing, as originally introduced in [15, 63], then used for network intrusion detection. Advantages of this approach include:

Universal This approach can be applied to detect malware in PDF documents, for both Javascript and Flash bearing documents, as well as any other kind of electronic document, including Microsoft Office Word and Excel documents or web pages that exploit vulnerabilities in browsers through Javascript.

Platform and implementation independent The detection is completely independent of the platform the system is built on. For example, an implementation of ***FCScan*** done by extending Adobe Acrobat Reader could detect malware that exploits a vulnerability in XPdf, or in previous versions of Adobe Acrobat Reader itself unless the malicious document specifically checks for a particular version.

Lightweight The system does not require emulation, OS-based instrumentation or other computationally expensive analysis. Thanks to its minimal footprint, users can run it on their systems without observing any significant degradation in performance. The only overhead is due to the interposition with actual functions for monitoring (as low as 50 microseconds per Javascript function call (see Chapter 8)).

Amenable to privacy preserving collaborative malware detection The underlying detection model is deterministic, meaning that the outcome is easily predictable and that viewing a legitimate sample for the second time does not cause any modification in the detection model (notice that this would not be the case if we had used a machine learning technique like a neural networks). These characteristics allow the merging of legitimate learning samples, thereby decreasing the false positive rate. The merging of samples can easily be done in a privacy preserving way, as we only need to exchange function codes together with an abstraction of their parameters (length, observed ranges, etc.). This allows detection models to be shared between different organizations, as will be further discussed in Chapter 9.

Modular Unlike typical anomaly detection approaches, the learning and the detection phase do not have to be disjoint, this allows for a modular learning phase, in which the detection model is improved each time a new legitimate sample is detected. This can be done without stopping the detection phase.

A collaborative detection model, built by multiple trusted clients is envisioned and discussed further in Chapter 8

FCScan flags a sample as malicious in two cases. Either the unknown sample contains a function identifier that is not present in the detection model, or a function is called with arguments that have unusual characteristics compared to the model.

Figure 6.7 shows the learning and testing phases visually. A model is built from a certain number of benign samples containing function calls to $Function_1(P_{1\_1}, P_{1\_2})$ and $Function_2(P_{2\_1}, P_{2\_2})$. A parameter model PM_{x_y} is associated with each parameter P_{x_y}, modeling the parameter characteristics (described below). In the testing phase, the model from the unknown sample is compared to the detection model. In this case there are two anomalies: $Function_3$ is not present in the detection model, and the parameter model $PM^1_{1\_1}$ is unusual compared to the parameter detection model. Thus the unknown sample is flagged as malicious.

In a real situation with a sufficiently large number of benign training samples this would indicate that the sample flagged as malicious is "*out of the ordinary*". A previously unseen function call could mean a call to an undocumented function that is normally never called, or a call to a rarely used documented function such as *media.newPlayer*. Differences in the function parameter characteristics indicate an "*out of the ordinary*" use of a *known* function, such as an excessively large buffer passed to *collab.getIcon*.

What follows is a description of the function parameter characteristics that we record in the model.
From the discussion on general exploitation techniques in Chapter 2 and the vulnerable Javascript functions in Section 6.1 the following function parameter characteristics follow:

- Parameter length (string or buffer length, array size, ...)
- Numeric value (integer, floating point values, ...)
- String characteristics (string encoding, printable vs non-printable characters, ...)

Parameter length Often, an unusually large buffer is passed to a function to trigger the exploit, as is the case with the *collab.getIcon* vulnerability shown in Figure 6.1. A buffer of roughly 24 kilobytes is being passed containing garbage data. *collab.getIcon* is used to retrieve an icon present in the document, based on its name. It is unreasonable that under normal circumstances, the same function will be called with such a large parameter. This would mean that the document contains an icon identified by a name of over 24000 characters.

All the listed malicious Javascript samples use heap spraying to litter the memory with instances of the shellcode, but it is not uncommon to carry the shellcode in the same buffer that is used to trigger the exploit, especially in network-level exploits [5]. Since shellcode is often larger in size than typical

```
// Function declarations
function Function1( arg1, arg2, arg3 );
function Function2( arg1 );

// Learning function calls
Function1( "first argument", "second argument", 3 );
Function1( ""              , "second argument", 10 );

Function2( 1024 );
Function2( 1 );

// Resulting Model
Function1 (count: 2)
{
    arg1     (string)    [0-14]
    arg2     (string)    [15-15]
    arg3     (integer)   [3-10]
}
Function2 (count: 2)
{
    arg1     (integer)   [1-1024]
}

// Example anomalous function calls:

// Parameter length 'arg2' out of range
// 0 outside of [15-15]
Function1( "first argument", "", 3 );

// Numeric value 'arg1' out of range
// 2147483647 outside of [1-1024]
Function2( 2147483647 );
```

Figure 6.8: Example functions calls, and the resulting model

string parameters, such as names, this could be detected in the same way. The above is even more true for ROP payloads as these payloads can not be heap sprayed and many gadget addresses may be needed to create functioning shellcode making an unusually large buffer inevitable.

In the learning phase, ***FCScan*** records the minimum and maximum parameter length passed to a function. In testing phase, the parameter length is compared to the benign detection model.

Numeric value The *util.printf* vulnerability shows the importance of recording the range of numeric values of parameters. The vulnerability is triggered by attempting to parse a floating point number and passing an excessively large integer ($\sim 12 \times 10^{294}$). Similarly, in the *getAnnots* vulnerability, large negative numbers are used to trigger the exploit. Again, it is clear that both large floating point numbers, and large negative numbers are not commonly found when using these functions.

In the learning phase, ***FCScan*** records the minimum and maximum value of numeric parameters passed to a function. In testing phase, the numeric value is compared to the benign detection model.

String characteristics In some exploitations, the shellcode is passed to the function taking a buffer or string as argument. This case can be detected by recording characteristics of the string buffer in the model. Characteristics may include whether or not non-printable characters are allowed, which string encodings have been used (ASCII, UTF-8, UTF-16, ...), etc. As seen in Section 6.1, the majority of Javascript exploits use the heap spraying technique to introduce shellcode into memory, thus limiting the effectiveness of this parameter characteristic for detection of malicious Javascript bearing PDF documents. For this reason, we have only implemented the first two detection parameters.

Judging from the known malicious Javascript functions listed in 6.1 one can question the need for an anomaly-based detection technique such as ***FCScan***. It seems sufficient to flag function calls with parameter lengths or parameter values above a certain large value and check for null parameters. Unfortunately these simple static checks are insufficient. Consider the vulnerable function shown in 6.9, in this example a parameter value 'arg1' of 9 or lower would cause an integer underflow resulting in the if statement returning true. Secondly, null arguments are not necessarily incorrect, simply flagging them as such could therefore result in a large number of false positives.

```
#define HEADER_LENGTH 10
unsigned* copyBody( packet : unsigned*, packet_len :
    unsigned )
{
    unsigned bodyLen = packet_len - HEADER_LENGTH;
    unsigned* pArr = malloc( bodyLen );
    // ...
    return pArr;
}
```

Figure 6.9: Integer underflow example

Chapter 7

Proof-Of-Concept implementation

In order to support our thesis we provide a proof-of-concept implementation of ***FCScan*** to detect malicious Javascript-bearing PDF documents. In the first two sections, high level design is discussed, namely whether to implement a dynamic or a static detection technique, and whether to implement the technique as stand-alone software or natively in the PDF reader. In Section 7.2.1 the details of how to record the relevant Javascript function calls and their parameters is discussed and we conclude in Section 7.3 with an automated way of training the benign models.

Even though the proposed detection technique can in principle be applied to other domains, the Javascript interpreter embedded in Adobe Reader is particularly suitable as the use of malicious Javascript inside PDFs is a well-established exploitation technique, resulting in many different malicious samples and elaborate techniques used by attackers to hide the Javascript from existing detection techniques.

7.1 Static vs. Dynamic

In general, to detect the presence of malware one can use either a *static*, as in signature-base anti virus, or a *dynamic* approach that relies on observation of how the alleged malware actually behaves or monitors certain memory areas for changes. Static analysis is susceptible to evasion by using obfuscation techniques or techniques that rely on data that is only available at run-time. Therefore we focus on dynamic analysis.

Dynamic analysis techniques come in two flavors. First, a specially instrumented system [27, 40, 65], can be used which simulates a normal environment and analyzes the behavior of the sample under analysis. The goal is to trick the malware into thinking that it is running on a normal victim's machine, though replicating a victim's environment in a precise way, including meaningful user

behavior is a complex task. Even though it is more difficult to bypass than static approaches, it is still possible by using timers or long running loops to reach the environment's execution threshold, by using decryption routines based on information that is unique to the targeted victim (i.e. user names) or by detecting the presence of the simulated environment and only performing benign actions in that case. The second flavor of dynamic analysis techniques can be implemented on the victim's system. In this case the detection takes place *while* the malware is trying to infect the system. In the worst case, if the malware detects the presence of the analysis engine, at least the attack is mitigated. While some researchers have worked in this direction, their approaches either suffer of computational penalties often too high for the user's workstation [20, 49, 56], or require white listing several regular processes, which could be used to evade detection, in order to suppress false alerts [33].

FCScan is implemented using the second dynamic analysis flavor.

7.2 Stand-alone vs. Native

Native	**Stand-alone**
+ Guaranteed same behavior as target application	- Prone to parsing errors or differences
+ Undetectable through Javascript	- Leverage implementation differences to detect analysis engine
+ Can be used for host-based, real-time detection	- Cannot be used for real-time detection
+ Detects application-specific exploits	- Misses application-dependent exploits
- Difficult, custom implementation required	+ Open Source tools available
- Specific to single application	+ Independent of application

Table 7.1: Comparison of native vs stand-alone proof-of-concept implementation

The second choice to make is whether to implement ***FCScan*** on a third party Javascript interpreter, or to instrument Adobe Reader natively. The chosen dynamic analysis flavor already biases this choice towards a 'native' variant, but it still makes sense to look at all of the pro's and con's.

All existing detection tools for malicious Javascript bearing PDFs use custom PDF parsers and third-party Javascript interpreters. This simplifies the implementation, but as discussed in Chapter 4 this comes with certain drawbacks. Table 7.1 summarizes the pro's and con's of implementing a detection technique natively compared to using a custom, stand-alone approach.

The PDF specification is an immensely complex document and over the years attackers have become extremely skilled in leveraging these complexities to hide or obfuscate Javascript objects in PDF documents. Much research has

gone into finding these techniques and writing more robust PDF parsers. While the current state of this problem appears adequate, there is no guarantee that attackers won't find another way to hide Javascript objects from these custom parsers while remaining valid under the Javascript interpreters embedded in PDF readers. An advantage of using a third party Javascript interpreter and custom PDF parsers is that the detection technique becomes independent of the PDF reader. On the other hand, some malicious Javascript may depend on a specific implementation of the Javascript interpreter and fail to run elsewhere. Furthermore, specific non-standard behavior of the embedded Javascript interpreter can be used to detect whether the Javascript is running on a custom interpreter and change its behavior accordingly.

We choose to implement ***FCScan*** natively by instrumenting Adobe Reader's Javascript interpreter:

- Shows technical feasibility of natively instrumenting an arbitrary binary application.
- The solution can be used for host-based, real time detection.
- Additional contribution by laying bare the inner workings of the Adobe Reader Javascript interpreter.
- By understanding the embedded Javascript parser, it can be used to extract Javascript from PDFs for other detection tools, removing the need for error-prone custom parsers.
- Extracted Javascript can be used to verify third party PDF parsers used in existing detection tools.
- ***FCScan*** observes the exact same behavior as the application it's instrumented in.

7.2.1 Implementation

Figure 7.1 shows the basic steps involved in processing a Javascript function call inside a PDF document. From the Javascript parser, the function call information (function name, parameters, etc.) is extracted and passed to the MethodDispatcher. The MethodDispatcher looks up the function name in a global table with a {*function name*; *function pointer*} relation. It then calls the function pointer, passing unparsed argument information. All Javascript API function that take one or more arguments share the same preamble and call to the ArgumentParser. Functions with zero or a variable number of arguments do not call the ArgumentParser. Currently, ***FCScan*** lacks the supporting of parsing arguments to functions that take a variable number of arguments, this is left for future work. Adding this support requires two steps:

1. Identify functions with variable number of arguments.

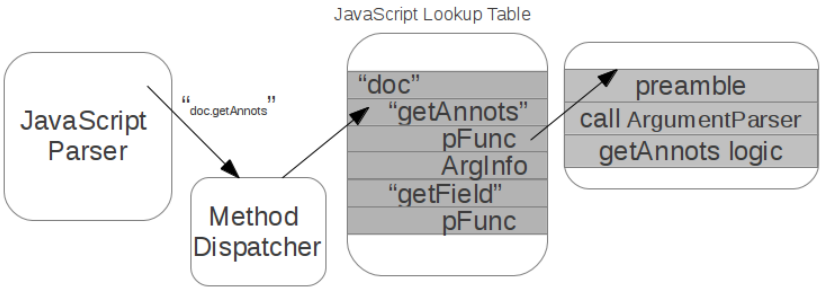


Figure 7.1: Adobe Reader execution flow when parsing Javascript function calls

2. Dissect function's logic and add hooks just like in the ArgumentParser.

To identify all functions that take a variable number of arguments, we can not solely rely on the API documentation. Only a fraction of the API is publicly documented, and we are particularly interested in the APIs that are obscure and undocumented. However, by leveraging the fact that Adobe Reader contains a Javascript lookup table (see Figure 7.1) we can write a "crawler" which scans all of the API functions for a call to the ArgumentParser. If there is no such call, the function either has zero or a variable number of arguments.
From an initial, quick implementation it seems there are roughly 10 functions with a variable number of arguments, making a custom implementation for these functions alone feasible.

All information required for the detection logic can be obtained by hooking the MethodDispatcher and ArgumentParser. From the MethodDispatcher we obtain the function name and from the ArgumentParser we obtain the type of the argument and a pointer to a buffer with the value from which we can determine the numeric value, or the buffer length.
The logic for different Javascript functionality is split over multiple dynamic libraries (.api files) that are loaded on demand by the Adobe Reader application. Each .api library contains a MethodDispatcher and an ArgumentParser. In our proof of concept we have instrumented the two most common .api libraries: *EScript* and *Multimedia*. The effort involved in instrumenting all .api files is discussed in Chapter 9.

hooking To implement our detection logic, we need to introduce code in Adobe Acrobat Reader which gets executed each time the MethodDispatcher and ArgumentParser are called. This can be achieved in several ways. First, by attaching a scriptable debugger to the Reader, breakpoints can be placed in locations where the custom logic needs to be inserted. The debugger's scripting

language can then be used to implement the logic. While this method works well for developing the solution, it is not suitable for mainstream use due to the fact that a debugger must be running alongside Adobe Reader, and the performance impact of using the debugger is unacceptable. Considering the relative ease with which the detection logic can be changed, ***FCScan*** was developed using this method.

Much work has been done on creating 'dynamic binary instrumentation' tools. These tools make it easy to instrument a binary without having to worry about architectural details. One notable example is 'Pin' [43, 38], developed by Intel. Pin would allows us to insert hooks in arbitrary locations in the application, and run custom code when they are executed. Unfortunately, Pin suffers from the same drawback as using a debugger in that it needs to run alongside the application under analysis. We did not investigate the performance impact of Pin in the context of rendering a PDF document with Adobe Acrobat Reader. Finally, the more labor intensive, but also the most rewarding technique is to implement custom hooks ourselves by patching the code of Adobe Reader while it is running. This method is chosen for ***FCScan*** as it will allow ***FCScan*** to run at the end-user without additional third-party dependencies.

To load ***FCScan*** automatically when Adobe Reader starts, we implemented ***FCScan*** as a plug-in to Adobe Reader. Plug-ins provide extra functionality to the Reader at run-time. In fact, most of the Reader's default functionality is implemented using plug-ins. Officially, third-party plug-ins need to register with Adobe to obtain a cryptographic key. This key is used to register with the Adobe Reader application. Without a valid key, a plug-in should not be loaded according to the documentation. (Un)fortunately, the implementation of this logic is flawed.

At start-up, Adobe Reader will load *any* file from the *plug_ins* directory with a *.api* extension. After loading, Adobe Reader calls two plug-in initialization routines that among other things validate the cryptographic key of the plug-in. When the second initialization function fails for whatever reason, an error dialog is shown to the user that an (unauthorized) plug-in failed to load. However, if the first initialization function fails (simply by returning *false*), *no error is generated and the plug-in is simply unloaded.*

A *.api* file is simply a dynamic link library (DLL) which is loaded by calling *LoadLibrary* and unloaded through *FreeLibrary* by Adobe Reader. On Windows, *LoadLibrary* increments a reference count for the DLL, and *FreeLibrary* decrements it. The DLL is not actually unloaded by *FreeLibrary* until the reference count reaches 0. Since our custom plug-in is not registered with Adobe, Adobe Reader will call *FreeLibrary* when the first initialization function fails. However, our plug-in can call *LoadLibrary* on itself, incrementing the reference count to 2. Now when *FreeLibrary* is called our plug-in is not actually unloaded, since its reference count is still 1. Since there is no error dialog, there is no obstruction to the end-user and our detection logic runs unobtrusively. It is even still possible to use parts of the Adobe Reader's API which should normally only be available to registered plug-ins.

7.3 Automated training of the model

Inspection of the training data set, discussed in Section 8, showed that many PDFs require user interaction to trigger execution of the Javascript. Simply opening and closing the PDF would not result in an accurate representation of the Javascript embedded in the PDF.
Most common user interaction required:

- Enter text in field
- Change field (trigger input validation)
- Tick and untick check boxes
- Press buttons (reset, print, email form etc.)

Each of the above actions can potentially open pop-up windows that need to be closed, such as the print dialog when pressing the print button or windows shown by calling Javascript's *alert()* function, to inform the user of an input error. To be able to train the model automatically, we designed a best-effort method to simulate user interaction by using a macro recording and play-back application. The application is instructed to wait for Adobe Reader's window to gain focus, after which it consecutively types '*<space>abcdef12346,./;'[]-=*', followed by TAB (to switch to the next field) and ENTER (in case the next field is a button or a check box, press it). The combination of alphanumeric and special characters results in a high chance to trigger input validation errors which are usually added to date or other numeric input fields. Since pop-ups can appear at any moment and disrupt the above 'flow' of key presses, we've instructed the macro application to close any child window of Adobe Reader as soon as it opens. The above macro application, together with a python script to iterate all PDFs in a specific folder, and wait for it to close before opening the next allows for semi-automatic (and fast) training of the model. Note that each PDF was still inspected manually for any out of the ordinary user input. Clearly, this semi-automatic training is far from being perfect, and will affect the results of the testing phase, in particular the false positive rate.

As any other anomaly-based approach, noise could be incorporated in the detection model, for example by accidentally learning from an actual malicious sample. Since the model is easily represented in human readable form, unlike approaches based on neural networks and n-gram analysis, (expert) users can review the detection model and discard any well-known malicious functions or unusual/suspicious parameter characteristics. Similarly, once the training model is built, functions with a sample count below a certain small threshold could be discarded automatically.

Figure 7.2 shows a PDF after its form has been automatically filled in. Note that check boxes are ticked and each field is filled in (some fields are configured to only accept numeric input, these fields do not show the alpha and special characters).

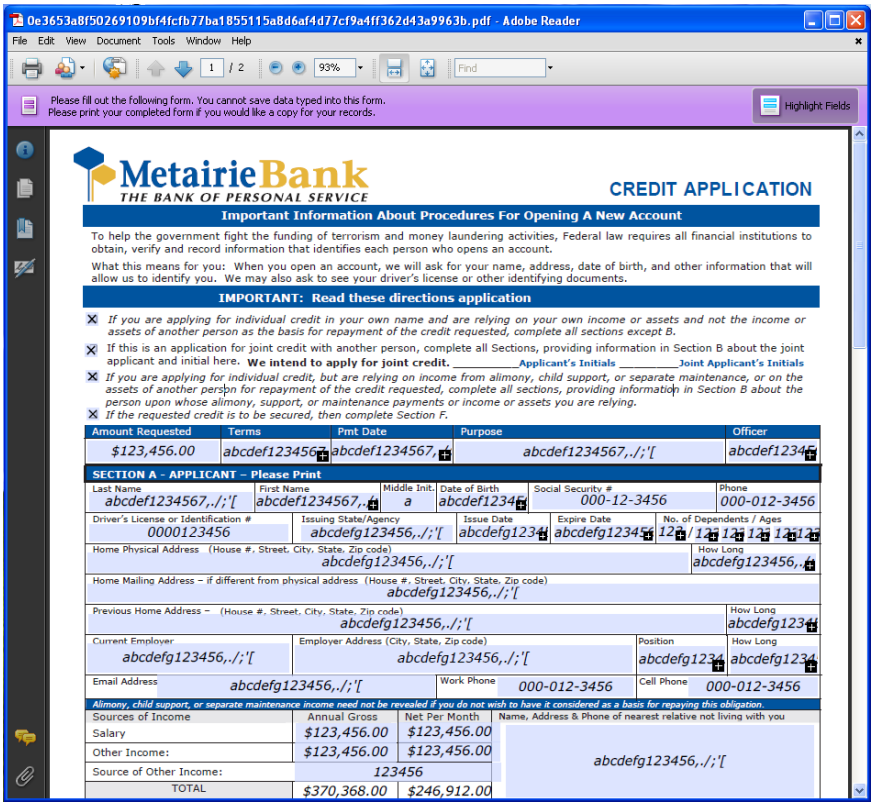


Figure 7.2: PDF with input fields automatically filled

Chapter 8

Evaluation

In this chapter, we will evaluate the proof-of-concept implementation of our detection technique, and compare it to existing techniques that were discussed in Chapter 4 aimed at detecting malicious Javascript bearing PDF.
We start with a description of the benign sample set used, followed by the false positive analysis in Section 8.0.1. The detection rate and performance are discussed in Section 8.0.2 and 8.0.3 respectively. Finally, Section 8.0.4 compares ***FCScan*** to existing PDF detection tools.

For the detection technique to work, we must obtain a set of benign PDFs containing Javascript to train the model with. We chose to obtain the data set through VirusTotal's Intelligence program[1]. The following search term gives us PDFs containing Javascript that haven't been flagged as malicious by any scanners:

tag:js-embedded type:pdf positives:0

We obtained 676 PDFs over a period of three months from VirusTotal using the previously mentioned search query. The downloaded number of samples was much higher than 676, but after inspecting the sample set it became clear that many samples were duplicates of each other, even though the SHA-1 hash was different. It appears that people submit similar PDFs over and over to Virus Total. One example that stood out was a *Chucke Cheese's* discount ticket[2]. The ticket contains an expiry date and a unique number, meaning that the SHA-1 hash of the PDF is different for each ticket. The original sample set contained dozens of these tickets. We believe that the chance that these duplicate samples contain different Javascript function calls is small, and thus leaving the duplicates in the sample set would result in a positively biased false positive rate.
Out of the 676 unique PDFs, 35 did not contain any Javascript function calls but simply a table of contents that navigate to a specific page on click by setting

[1] `http://www.virustotal.com/intelligence/`

[2] `https://www.virustotal.com/intelligence/search/?query=2e63921b5ac581600b2abd12f60583d240b2a55a980f663d4e97b67b8d6ef9bd`

'*this.pagenum = X;*'. Again, leaving these PDFs in the sample set the false positive rate would be lowered unfairly. Removing these PDFs results in an effective dataset of 640 PDFs containing 140 unique Javascript functions.

8.0.1 False positive rates

To determine the false positive rate of ***FCScan***, we analyze the 640 benign PDFs obtained from VirusTotal. To make the best use of the sample set we apply a K-fold Cross Validation (sometimes called rotation estimation) algorithm.

In K-fold cross validation, the sample set S is split into K mutually exclusive subsets (the 'folds') of the same size. The model is trained and tested **K** times, each time $t \in \{1, 2, \ldots, K\}$ it is trained on $S \setminus S_t$ and tested on S_t. In other words, the model is trained with all folds except one, which is used for testing. This process is repeated until all folds have been used for testing once.
Using a **K** equal to the size of the data set implies training the model with all samples except one, and using that single sample for testing. What this shows in our case is how many PDFs contain Javascript function calls that are not in any of the other PDFs in the sample set.
A value of *10* for **K** is a generally accepted default in literature [31].

Table 8.1 lists the result of applying the algorithm to the sample set using different values for **K**. It is worth noting that the above false positive rates refer only to samples that contain Javascript. In a normal operative situation however, only a small number of PDFs actually contain Javascript. A PDF that does not contain Javascript will never be flagged as malicious.

We note that, out of 10 function calls erroneously flagged as malicious, roughly 8 are flagged because the function is not in the detection model, and 2 are flagged as malicious because of unusual parameter characteristics. In other words, false positives are mostly due to a benign PDF using Javascript functions that haven't been seen before, rather than using known Javascript function calls in an unusual way.

K	**Number of alerts**		**False positive rate**
	Total	**Avg per fold**	
2		20	6.3%
5		6	4.6%
10		**3**	**4.3%**
50		0.8	4%
640		0.05	4%

Table 8.1: K-fold Cross Validation results for different K

The cause of false positives can be twofold. Either the training set does not sufficiently represent the real world, or the PDF in question is exceptional compared to average real world samples. The first case can be addressed by increasing the number of training samples, and include samples from multiple different sources over different time periods. Exceptional samples are a problem

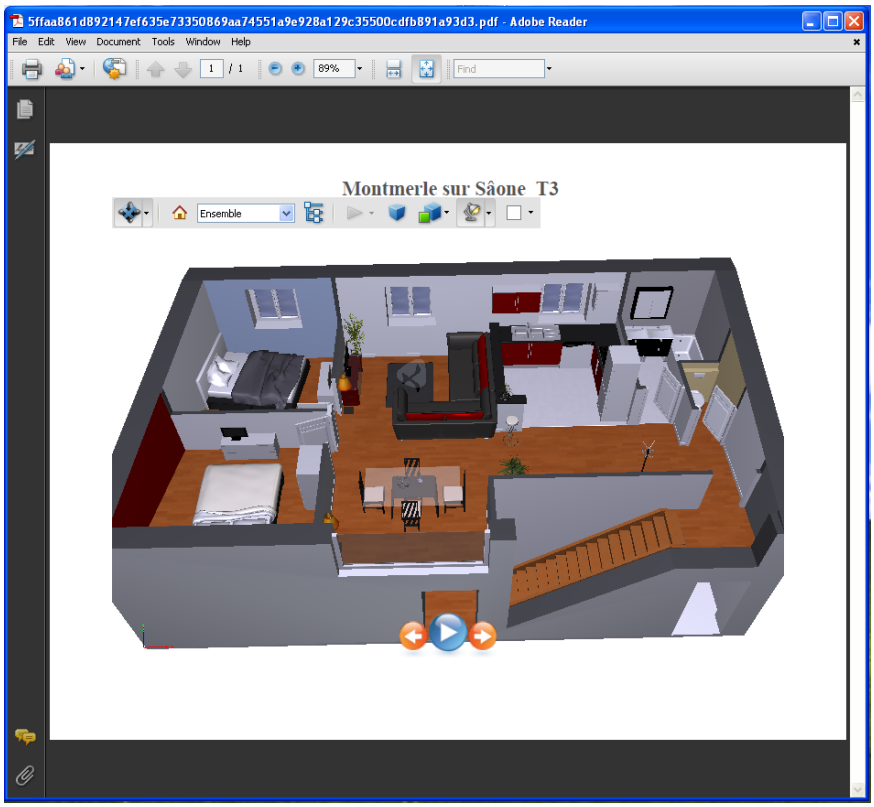


Figure 8.1: Screen shot of an 'exceptional', benign PDF

in any learning-based detection technique as by definition we aim to detect exceptional samples. An example of an exceptional PDF[3] is shown in Figure 8.1. The PDF shows an interactive 3 dimensional model of a 2 bedroom apartment for sale. Another example allows a user to dynamically add or remove fields from the input form in the PDF.
The functions that are unseen in these two exceptional PDFs are *clearInterval*, *setInterval*, *getAnnot3D* and *getAnnots3D* for the PDF of the apartment and *displayField*for the dynamically modifiable PDF.

8.0.2 Detection rates

All of the vulnerabilities listed in Table 6.1 are detected by our approach except for the *util.printf* vulnerability since this function is in the detection model, and the current version of the proof-of-concept implementation is unable to parse the parameters passed to *util.printf* since this function takes a variable

[3]VirusTotal SHA-1 hash: `https://www.virustotal.com/intelligence/search/?query=5ffaa861d892147ef635e73350869aa74551a9e928a129c35500cdfb891a93d3`

number of parameters and therefore does not use the ArgumentParser but implements its own custom argument parsing logic. The number of functions that do not use the ArgumentParser are limited, and can be identified by iterating through the "*Javascript Function Lookup Table*" as described in Section 7.2.1. We stress that this is not a limitation of the approach, the argument passed to *util.printf* is extraordinarily large and thus would be detected if the proof-of-concept implementation is extended to include functions with a variable number of arguments.

All of the other functions are detected because they are not present in the detection model, but we already discussed in Section 6.2 that even if the functions were present in the detection model, the parameters of the malicious calls would violate the parameter characteristics each time.

During our experiments we have noticed that using a threshold for setting the minimum number of anomalous function calls per PDF documents, below which a sample is considered benign, negatively affects the detection rate. This is because even a single malicious function call can be sufficient to perform a successful exploitation.

8.0.3 Performance

The performance overhead of our implementation is limited. For each Javascript call, we require one hashed table lookup for the function name and one hashed table lookup for each of its parameters. Table 8.2 shows the average time of loading a PDF with and without the detection technique loaded, when 10.000 Javascript calls are performed.

	Detection ON	Detection OFF	Impact
No Javascript	X	X	0.0s
doc.getAnnots(1, 1, 1, 1)	0.6s	0.2s	0.4s
util.printd( )	0.4s	0.2s	0.2s

Table 8.2: Average loading time of PDF document with and without detection

Over 10.000 calls to docs.getAnnots, the overhead is shown to be about 50 microseconds per function call. As described in Section 7.3, the average benign PDF contains small snippets of Javascript that trigger only on user input. The purpose of Javascript in PDFs is not to perform lengthy computations involving many function calls, therefore the overhead introduced by our implementation is not noticeable under normal usage. In real-life, a regular PDF document would hardly surpass 2000 Javascript calls. This allows for the tool to be deployed at the end host, rather than using it only to analyze suspicious PDFs. This means that the tool can be used for prevention. As soon as an unseen or unusual function call is observed, the Javascript execution can be stopped and the potential infection mitigated.

8.0.4 Comparison to existing tools

Tool	Techniques
PJScan	Static Extract embedded Javascript Look for malicious Javascript Signature-less (learning)
MDScan	Dynamic Extract embedded Javascript Interprets Javascript on SpiderMonkey Look for shellcode and ROP in memory
MalOffice	Static and Dynamic Extract embedded Javascript Look for malicious Javascript Signature based detection Dynamic analysis using CWSandbox Black & White list applied to CWSandbox report
FCScan	Dynamic Host based, real-time detection Use Adobe's internal interpreter Signature-less (learning) Detect anomalies in Javascript function calls

Table 8.3: Summary of PDF detection tools

Table 8.3 summarizes the different PDF detection tool, Table 8.4 lists techniques that would cause a false-negative when applied. Finally, Table 8.5 lists positive and negative points of each of the detection tools.

It is clear that extraction of Javascript from PDF files is a critical point. Attackers are actively looking for new methods of hiding Javascript [66, 48] which may remain unknown for a long time and would require an update of the detection tools if found. Static detection techniques suffer from run-time dependent Javascript, while dynamic techniques can by bypassed by using functions from the Adobe extended Javascript API for PDFs.

None of the techniques (except for MalOffice's dynamic analysis) detect malicious PDF's that do not depend on the use of Javascript. For example, a PDF with an embedded Flash exploit that carries the shellcode in the Flash object as well will not be detected by most of the discussed tools.

Tool	Techniques
PJScan	Javascript in unknown location Run-time dependent Javascript Non-Javascript dependent exploit
MDScan	Javascript in unknown location Reach execution threshold Javascript dependent on unimplemented extended PDF API Detect non-Adobe Javascript interpreter Non-Javascript dependent exploit
MalOffice	Javascript in unknown location Detect CWSandbox
FCScan	Vulnerability in function in un-hooked API Non-Javascript dependent exploit

Table 8.4: Scenario to bypass PDF detection tools

Tool	Positive	Negative
PJScan	Speed	Training required, High false-positive
MDScan	Shellcode & ROP, No training	Speed, Javascript interpreter
MalOffice	Combination of techniques, Black and white list	Speed, Static heuristics are simple
FCScan	Speed, Low false negative, Host based, real time detection	Training required

Table 8.5: Comparison of PDF detection tools

Chapter 9

Conclusion and future work

In this thesis we present a new and effective approach for detecting malicious content in electronic documents such as PDF and Microsoft Office documents dubbed ***FCScan***. Our approach is based on two different phases. First, ***FCScan*** observes a number of samples and builds a detection model for every function call that is invoked while processing the document. Then, when enough samples have been observed, it flags any subsequent document as malicious that does not match the built model, either because a called function is not present or because a function's parameter characteristics do not match the normal learned values.

In order to test our approach, we develop a prototype for the ubiquitous Adobe Reader. Our prototype runs at the endpoint and in real-time, while detecting all of the exploitation attempts based on malicious Javascript contained in PDF documents we collected from a renowned source, VirusTotal. On the other hand, our prototype shows a low false positive rate of around 4% when processing benign Javascript-bearing PDF documents. In real-life we believe this number could be brought down even further by sharing the learned models across different organizations, thanks to the privacy-preserving and modularity characteristics of our approach. Since our approach can run at the end-user without significant loss of computational performance, and because it is implemented natively in Adobe Reader it can be used as a prevention tool just as well as a detection tool. When ***FCScan*** observes a potentially malicious Javascript function call, it can halt execution of Adobe Reader, effectively mitigating the attack. Rather than simply terminating, the user can be asked for confirmation.

9.1 Future work

Learning phase using third-part PDF parser When building a model of the benign Javascript functions, it is important that most, if not all of the Javascript inside a benign document is analyzed. The current method adopted by ***FCScan*** relies on random input to trigger the execution of Javascript in-

side the benign document. This method does not guarantee that all of the Javascript inside the document is called, and it is time consuming to analyze a large number of samples. A better approach would be do use the same method that existing PDF detection tools use, namely using a third party PDF parser to extract the Javascript, and then run this Javascript on an instrumented version of SpiderMonkey which generates the detection model that can be loaded by ***FCScan***. This way, a large portion of the Javascript will be analyzed and many samples can be processed in a short amount of time.
FCScan still needs to be implemented natively in Adobe Reader or any other PDF reader to be most effective at detecting malicious documents, due to the limitations of third party PDF parsers as discussed in this thesis.

Support util.prinf-like functions The current proof-of-concept implementation lacks support for functions taking a variable number of arguments, such as *util.printf*. This needs to be addressed.

Support more .api files As explained in Section 7.2.1, the Javascript logic is split over multiple .api files, each containing a MethodDispatcher and an ArgumentParser. The proof of concept version of ***FCScan*** only adds hooks to the *EScript* and *Multimedia* api files, as these contain known vulnerable Javascript functions. In a future version of ***FCScan***, all of the .api files should be supported to be able to detect previously unknown vulnerabilities in the Javascript functions implemented in these .api files.
The effort involved in supporting a new .api file is minor, as the MethodDispatcher and ArgumentParser work the same way in each .api file. One way to find the location of the MethodDispatcher is to craft a PDF with a function implemented in that .api file (e.g. *media.newPlayer* in *Multimedia.api*) and open the PDF while running Adobe Reader under control of a debugger. In the debugger, a memory read breakpoint should be placed on the *"newPlayer"* string in the Javascript Lookup Table as shown in Figure 7.1. The breakpoint will hit when the MethodDispatcher reads the *"newPlayer"* string. The ArgumentParser can be found by following the *pFunc* in the Javascript Lookup Table and finding the function call matching the ArgumentParser's signature.

Support collaborative detection model In an operative situation, the model of what is benign can be stored in the cloud where it is updated by an oracle or trusted clients in case of a false positive alert. Figure 9.1 shows 'Client 1' opening a PDF that is flagged as malicious based on the current model. The alert information together with a copy of the PDF is sent to an oracle where it is analyzed further. If the oracle determines that the alert was a false positive, it updates the global model which is then pushed to all connected clients.

Mitigate attacks Since the Adobe Javascript interpreter is mono-thread, and our detection technique is implemented in-line a future enhancement would be to block the PDF reader when an alert is raised, asking the user for confirmation

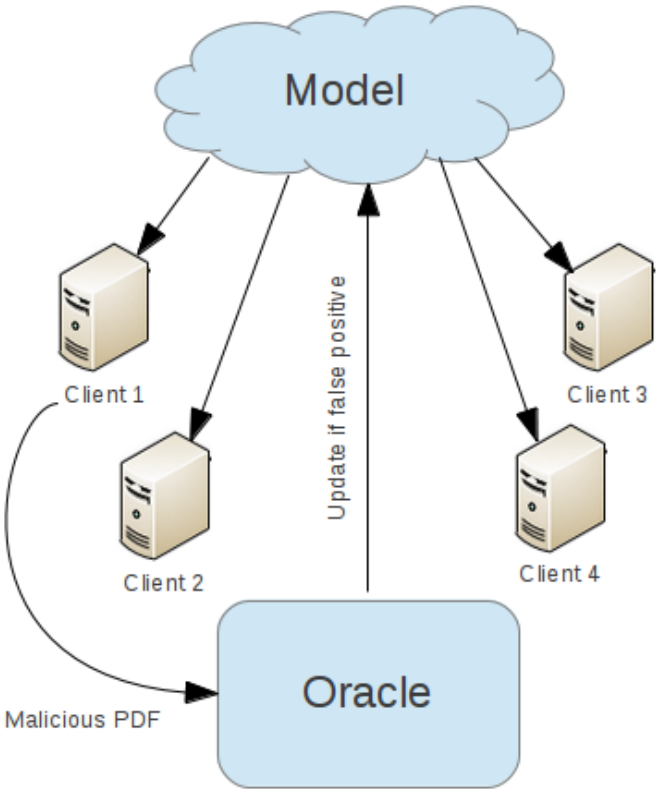


Figure 9.1: Scenario of updating model after a false positive alert.

whether to continue parsing the PDF or not. Since all arguments are parsed before the actual Javascript function is called this would mitigate the exploitation attempt when the user chooses to abort. This can be combined with the above collaborative model, giving the user the option to merge the violating functions into the detection model so that in the future the same PDF will not raise more false alerts.

Support other domains ***FCScan***'s approach is general in nature, and not limited to detection of malicious Javascript bearing PDF documents. Given the similarities between interpreted languages we will investigate the possibilities of extending the implementation of ***FCScan*** to include Actionscript in Flash objects, either stand-alone or embedded in PDF documents, Javascript in browsers to detect drive-by-download attempts and to the detection of malicious Microsoft Office documents.

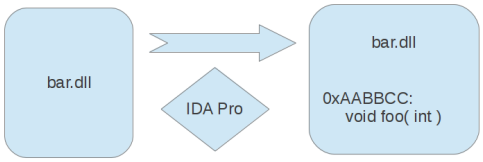


Figure 9.2: Using IDA Pro to identify functions

Support general compiled binaries Currently, ***FCScan*** is only implemented in Adobe Reader to detect malicious Javascript bearing PDFs. We believe that the technique can be applied in the same way to other interpreted languages such as Actionscript in Adobe Flash files, or Javascript in browsers. A more interesting question would be whether the technique can be applied to a compiled binary in general.

Theoretically, to implement the technique for general compiled binaries, three problems need to be solved:

- How to uniquely identify functions
- How to identify arguments, and record the parameter characteristics
- How to intercept function calls

Each function in a compiled binary resides in a module, which is loaded at a certain address in memory. Techniques such as ASLR change the addresses at which modules are loaded unpredictably. However, the relative offset of the function compared to the base address of the module remains constant and can

```
ADDRESS LoadLibrary_Hooked( String LibraryName )
    // Call original LoadLibrary to obtain module
       base address
    ADDRESS moduleBase = LoadLibrary_Original(
       LibraryName );

    if( "bar.dll" == LibraryName )
        // Hook the known 'foo' function (parameters
           read from
        // IDA Pro database generated from bar.dll)
        hookFunction( LibraryName, moduleBase, 0
           xAABBCC );

    // Return module base to caller, just like the
       original function
    return moduleBase;
```

Figure 9.3: Hooked LoadLibrary function

be used to uniquely identify the function once the current base address of the module is found.

Consider the case of an application calling a function '*void foo(int)*' that resides in a module '*bar.dll*'. To successfully instrument this application, three things need to happen: we need to find all functions in modules used by the application, in this case the address of *foo*. Secondly, we need to find the base address of the *bar* module after it has been loaded by the application and finally execution flow must be redirected to the routine containing the logic of ***FCScan***. To identify functions in a module, we use *IDA Pro*. *IDA Pro* is an interactive disassembler with advanced heuristics to provide detailed information about functions, such as types of function parameters. This greatly simplifies implementing the detection technique in compiled binaries, as *IDA Pro* provides a consistent representation of function characteristics across different architectures. To obtain the base address of the module, we divert control from a class of functions used to load modules into memory, namely the *LoadLibary* family. Whenever a module is loaded, these functions return the base address of the module and thus allow us to apply hooks to all functions previously identified using *IDA Pro*.

Figure 9.3 shows a simplified version of the hooked LoadLibrary Windows API function. When *bar.dll* is loaded we know that *foo* resides at '*moduleBase + 0xAABBCC*', allowing us to hook *foo* and identify *foo* by the module name (bar.dll) and the relative offset (0xAABBCC). The relative offset is obtained from the analysis by IDA Pro as depicted in Figure 9.2. Note that while the authors believe this scenario is possible, it is currently no more than a simplified theory.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS '05: Proc. 12th ACM conference on Computer and Communications Security*, pages 340–353. ACM Press, 2005.

[2] Adobe Systems. JavaScript for Acrobat API Reference, 2007.

[3] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[5] S. Andersson, A. Clark, and G. Mohay. Network based buffer overflow detection by exploit code analysis. In G. M. Mohay, A. J. Clark, and K. Kerr, editors, *AusCERT Asia Pacific Information Technology Security Conference: R&D Stream*, pages 39–53, Gold Coast, Australia, May 2004. University of Queensland.

[6] Avast. Six out of every ten users run vulnerable versions of Adobe Reader. `http://www.avast.com/pr-six-out-of-every-ten-users-run-vulnerable-versions-of-adobe-reader`.

[7] Avast. New PDF Exploit hiding technique tricks Antivirus engines. `https://blog.avast.com/2011/04/22/another-nasty-trick-in-malicious-pdf/`, 2011.

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

[9] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *In Proceedings of the USENIX Annual Technical Conference*, pages 251–262, 2000.

[10] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. 2006.

[11] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: A stuxnet-like malware found in the wild. Technical report, BME CrySyS Lab., October 2011. First published in cut-down form as appendix to the Duqu report of Symantec.

[12] B. Bencs'ath, G. P'ek, L. Butty'an, and M. F'elegyh'azi. Duqu: Analysis, Detection, and Lessons Learned. 2012.

[13] S. Bhatkar, R. R.S̃ekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Sec: Proc. 14th Conference on USENIX Security Symposium*, volume 14 of *SSYM '05*, pages 17–17. USENIX Association, 2005.

[14] blexim. Basic integer overflows. *Phrack*, (60), December 2002.

[15] D. Bolzoni and S. Etalle. Boosting Web Intrusion Detection Systems by Inferring Positive Signatures. In *OTM '08: On the Move to Meaningful Internet Systems Confederated International Conferences*, volume 5332 of *LNCS*, pages 938–955. Springer-Verlag, 2008.

[16] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting Return-Oriented Programming Malicious Code. In A. Prakash and I. S. Gupta, editors, *Information Systems Security*, volume 5905 of *LNCS*, pages 163–177. Springer-Verlag, 2009.

[17] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[18] C. Cowan, F. Wagle, P. Calton, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DISCEX '00: Proc. DARPA Information Survivability Conference and Exposition*, volume 2, pages 119–129, 2000.

[19] D. Dai Zovi. Practical return-oriented programming. SOURCE, 2010.

[20] L. Davi, A. Sadeghi, and M. Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS '11: Proc. 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM Press, 2011.

[21] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in c/c++. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 760–770. IEEE Press, 2012.

[22] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA '09: Proc. 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, LNCS, pages 88–106. Springer-Verlag, 2009.

[23] M. Engelberth, C. Willems, and T. Holz. Detecting malicious documents with combined static and dynamic analysis, 2009.

[24] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. 2000. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html.

[25] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 workshop on Memory system performance*, MSP '05, pages 68–77. ACM, 2005.

[26] G. R. Fresi, L. Martignoni, R. Paleari, and D. Bruschi. Surgically Returning to Randomized lib©. In *ACSAC '09: Proc. 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69. IEEE Computer Society, 2009.

[27] iseclab. Anubis. `http://anubis.iseclab.org`, 2009.

[28] Document Management - Portable Document Format - Part 1:PDF 1.7, 2008.

[29] A. Iyer and L. Liebrock. Vulnerability scanning for buffer overflow. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 2, pages 116–117 Vol.2, 2004.

[30] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC '06: Proc. 22nd Annual Computer Security Applications Conference*, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[31] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. pages 1137–1143. Morgan Kaufmann, 1995.

[32] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID '06: Proc. 8th International Symposium on Recent Advances in Intrusion Detection*, LNCS, pages 207–226. Springer-Verlag, 2006.

[33] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *CCS '10: Proc. 17th ACM conference on Computer and Communications Security*, pages 399–412. ACM Press, 2010.

[34] J. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. Rajamani, and R. Venkatapathy. Righting software. *Software, IEEE*, 21(3):92–100, 2004.

[35] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing pdf documents. In *ACSAC '11: Proc. 27th Annual Computer Security Applications ConferenceAnnual Computer Security Applications Conference*. IEEE Computer Society, 2011.

[36] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *EuroSys '10: Proc. 5th European Conference on Computer Systems*, pages 195–208. ACM Press, 2010.

[37] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis. A Study of Malcode-Bearing Documents. In *DIMVA '07: Proc. 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, LNCS, pages 231–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[38] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[39] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. *SIGPLAN Not.*, 43(3):115–124, Mar. 2008.

[40] Malheur: Automatic Analysis of Malware Behavior. `http://www.mlsec.org/malheur`.

[41] Mozilla. `https://developer.mozilla.org/en/SpiderMonkey`.

[42] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACSAC '10: Proc. 26th Annual Computer Security Applications Conference*, pages 49–58. ACM Press, 2010.

[43] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk. Controlling program execution through binary instrumentation. *SIGARCH Comput. Archit. News*, 33(5):45–50, Dec. 2005.

[44] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. *Security and Privacy, IEEE Symposium on*, 0:601–615, 2012.

[45] M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-level polymorphic shellcode detection using emulation. In *DIMVA '06: Proc. 3th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, LNCS, pages 54–73. Springer-Verlag, 2006.

[46] M. Polychronakis, K. Anagnostakis, and E. Markatos. Comprehensive shellcode detection using runtime heuristics. In *ACSAC '10: Proc. 26th Annual Computer Security Applications Conference*, pages 287–296. ACM Press, 2010.

[47] M. Polychronakis and A. Keromytis. ROP Payload Detection Using Speculative Code Execution.

[48] S. Porst. How to really obfuscate your pdf malware. Presented at ReCon, 2010.

[49] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, 2006.

[50] P. Ratanaworabhan, B. Livshits, and B. Zorn. NOZZLE: a defense against heap-spraying code injection attacks. In *USENIX Sec: Proc. 18th conference on USENIX Security Symposium*, pages 169–186. USENIX Association, 2009.

[51] RSA. Rsa, anatomy of an attack. `http://blogs.rsa.com/rivner/anatomy-of-an-attack/`, 2011.

[52] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS '07: Proc. 14th ACM conference on Computer and Communications Security*, pages 552–561. ACM Press, 2007.

[53] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proc. 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM Press, 2004.

[54] M. Shafiq, S. Khayam, and M. Farooq. Embedded malware detection using markov n-grams. In D. Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 88–107. Springer Berlin / Heidelberg, 2008.

[55] Skywing and M. Miller. Bypassing windows hardware-enforced data execution prevention. *Uninformed*, 2, September 2005.

[56] A. Slowinska, T. Stancescu, and H. J. Bos. Body armor for binaries: preventing buffer overflows without recompilation. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 11–11. USENIX Association, 2012.

[57] D. Stevens. D. stevens' blog, pdf tools. `http://blog.didierstevens.com/programs/pdf-tools/`.

[58] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *CCS '09: Proc. 16th ACM conference on Computer and Communications Security*, pages 635–647. ACM Press, 2009.

[59] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *RAID '02: Proc. 5th Internation Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.

[60] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *EUROSEC '11: Proc. 4th European Workshop on System Security*, pages 4:1–4:6. ACM Press, 2011.

[61] Use-after-free definition. `http://cwe.mitre.org/data/definitions/416.html`.

[62] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*, pages 257–267, 2000.

[63] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 21–30. ACM, 2004.

[64] P. Vreugdenhil. Pwn2own 2010 windows 7 internet explorer 8 exploit. `http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf`.

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

[66] J. Wolf. OMG WTF PDF. 27th Chaos Communication Congress, 2010.