Defending Browsers against Drive-by Downloads: Mitigating Heap-spraying Code Injection Attacks

Manuel Egele[1], Peter Wurzinger[1], Christopher Kruegel[2], and Engin Kirda[3]

[1] Secure Systems Lab, Technical University Vienna, Austria
{pizzaman,pw}@seclab.tuwien.ac.at
[2] University of California, Santa Barbara
chris@cs.ucsb.edu
[3] Institute Eurecom, France
kirda@eurecom.fr

Abstract. Drive-by download attacks are among the most common methods for spreading malware today. These attacks typically exploit memory corruption vulnerabilities in web browsers and browser plug-ins to execute shellcode, and in consequence, gain control of a victim's computer. Compromised machines are then used to carry out various malicious activities, such as joining botnets, sending spam emails, or participating in distributed denial of service attacks.
To counter drive-by downloads, we propose a technique that relies on x86 instruction emulation to identify JavaScript string buffers that contain shellcode. Our detection is integrated into the browser, and performed *before* control is transfered to the shellcode, thus, effectively thwarting the attack. The solution maintains fair performance by avoiding unnecessary invocations of the emulator, while ensuring that every buffer with potential shellcode is checked. We have implemented a prototype of our system, and evaluated it over thousands of malicious and legitimate web sites. Our results demonstrate that the system performs accurate detection with no false positives.

Keywords: Drive-by download, malicious script, emulation, shellcode

1 Introduction

A drive-by download is any download of software that happens without the knowledge and consent of a user. Unfortunately, drive-by downloads present a major threat to the Internet and its users [28]. In a typical attack, the mere visit of a web site that contains the malicious content can lead to the infection of a user's computer with malware. The malicious code that is installed as part of the attack then has typically full control over the victim's machine. Often, keystrokes are recorded, passwords are stolen, and sensitive information is leaked out. Also, infected computers may join a botnet [5], a large collection of compromised hosts controlled by the attacker. The computational power of compromised hosts are valuable for attackers as these hosts can be misused for spam campaigns [17] or denial of service attacks [20].

The typical steps of a drive-by download attack are shown in Figure 1. It can be seen how the attacker first prepares a web site with malicious content. When this site is

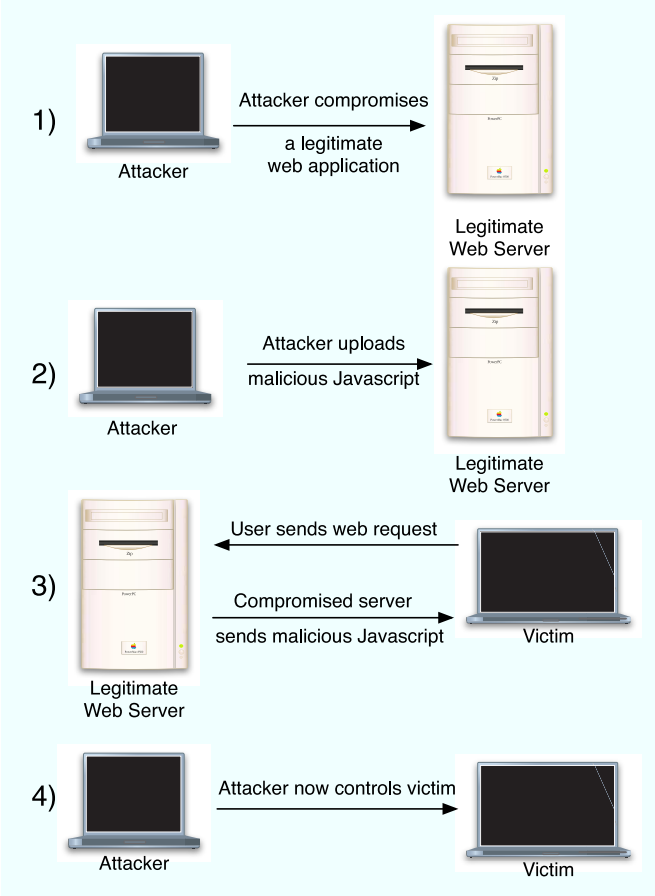

Fig. 1. The typical steps of a drive-by download attack.

later visited by a victim, hostile script code is downloaded and executed by the victim's browser. This script exploits a vulnerability in the browser or an installed browser-plugin. Once successful, the payload (shellcode) of the exploit downloads malware that provides full control to the attacker.

Most current drive-by downloads target browser plug-ins that are developed and distributed by third parties [28, 29]. The reason is that these plug-ins are less tested than the core browser, and thus, more likely to contain security vulnerabilities. Also, plug-ins are typically distributed as binary executables (at least in the case of Microsoft's Internet Explorer). As many plug-ins are written in non-safe languages such as C, they are susceptible to a wide range of vulnerabilities that are common for applications written in such languages. These vulnerabilities include buffer overflows, memory corruption issues, and pointer overwrites. Finally, plug-ins are often executed in the context of the browser, and as a result, can get full access to the browser and the underlying operating system.

As mentioned previously, as part of a drive-by download, attackers use client-side scripting code to load the shellcode (payload) into memory and execute the exploit against a vulnerable component. More precisely, JavaScript [11] is typically used to assign the binary representation of shellcode to a variable that is stored in the address space of the browser. To make their exploits more reliable, attackers resort to a technique called *heap spraying* [7, 32]. Heap spraying creates multiple instances of the shellcode, combined with a NOP sledge [34][4]. By leveraging the knowledge of how a script engine manages its heap memory, an attacker can, to a certain extent, influence where variables are stored in memory. As a result, the area of heap memory that needs to be sprayed for an attack to succeed is reduced. Once the heap memory has been "prepared," the actual exploit is launched. To this end, the hostile script typically invokes a vulnerable method (of a plug-in) with malicious arguments.

When the attacker has prepared a malicious script that can launch a drive-by download, it can be placed on a web site. Then, the attacker has to ensure that potential victims visit this site. One way is to create a new site and manipulate search engines so that they list this site high in their rankings. The higher a page is ranked, the higher the chance is that an unsuspecting Internet user will visit it. Another approach is to embed malicious content in advertisements that are placed on legitimate web pages. Here, the site embedding the advertisement becomes an unknowing accomplice for distributing the attack. Moreover, an attacker can also take advantage of vulnerabilities found in popular web applications. By exploiting these applications, they are able to place their content directly on the vulnerable web site. Automated SQL injection attacks [6, 16], for example, modify the database back-ends of web applications in order to include `iframe` tags that load the malicious pages.

Drive-by attacks belong to the most common methods for spreading malware today [29]. Thus, it is important to find solutions that mitigate the problem and protect users. In this paper, we present a proof-of-concept implementation of a system that detects shellcode-based drive-by download attacks. Our basic idea is to check the variables (strings) that are allocated by the browser (the script engine) when executing client-side scripts. When such a variable contains shellcode, we assume that the script is hostile, attempting to setup the environment for an exploit. Thus, the script is terminated, before any vulnerable function is invoked. We implemented our system in the Mozilla Firefox browser. However, our conceptual solution is general and works for arbitrary browsers. The main contributions of this paper are as follows:

- We propose a technique that uses emulation to automatically identify shell-code-based drive-by download attacks in a browser.
- We describe a proof-of-concept implementation of our approach that is integrated into the Mozilla Firefox browser.
- We present experimental results that show the feasibility of our approach. We have evaluated our prototype on more than one thousand malicious and several thousand benign sites. Our experimental results demonstrate that the system is able to accurately detect drive-by downloads with no false positives.

[4] A NOP sledge consists of a sequence of NOP instructions that increase the chance of successfully hitting the shellcode when hijacking the control flow of the vulnerable application.

2 Anatomy of a drive-by attack

In this section, we first provide a short overview of JavaScript to enable the reader to understand script-based drive-by downloads. Then, we present and discuss a real-world attack to illustrate the problem that we aim to defend against.

2.1 JavaScript basics

JavaScript is an implementation of the ECMA-262 standard that defines an object-oriented scripting language [11]. The JavaScript specification defines a set of core components, such as data types (e.g., `String`, `Integer`, `Object`), special objects (e.g., `Date`, `Math`), and operators. The most prominent use of JavaScript is for supporting dynamic content on the client-side (in web browsers). However, JavaScript is also often embedded in other software, such as Adobe's Acrobat PDF reader. Systems that use JavaScript typically provide *environments* that allow a script to interact and communicate with other components. The document object model (DOM), for example, is part of the environment provided by the web browser. It allows scripts to manipulate the web pages that are displayed and to react to user actions and inputs.

The JavaScript interpreter of the Mozilla foundation is called SpiderMonkey [11]. Microsoft's implementation of ECMA-262 is called JScript [22]. This implementation adds facilities to the environment that allow a script to instantiate and communicate with ActiveX components [21]. These components are the preferred way of providing plug-ins for the Internet Explorer. On request, the libraries implementing the components are loaded into the address space of the Internet Explorer process, and the necessary objects are instantiated. ActiveX plug-ins, thus, have the same privileges that the browser has, often including full access to the file system and the network.

Among the data types, `strings` deserves special attention. ECMA-262 defines strings as sequences of 16-bit integers, commonly interpreted as UTF-16 characters. Popular JavaScript engines, such as SpiderMonkey, implement strings as immutable. That is, once a string variable is initialized, the value does not change for the rest of its lifetime. String operations, such as substituting characters (i.e., `replace` method of the string object), do not modify the original value. Instead, a new additional string variable is instantiated with the modified content. We will see that this fact has important ramifications for the implementation of our defense technique.

2.2 An example of a real-world drive-by download

In this section, we describe a typical drive-by download attack. We actually encountered this specific attack during our experiments. On September 2, 2008, our high-interaction client honeypot visited the URL `http://www.thewebleaders.com`. This page contained an `iframe` that loaded the script presented in Listing 1.1.

The most noticeable property of the script is that it uses obfuscated variable and function names to make it difficult for a human analyst to understand the script's purpose. Manual analysis reveals that the function defined in Line 1 serves as a decryption routine. The two values that make up the key for decryption are the location currently

```
1 function XfNLVA421(IaP1EoKdg) {
2   var I833Nad64 = location.href;
3   var hOtmWAGmO = arguments.callee;
4   hOtmWAGmO = hOtmWAGmO.toString()
5   ...
6   try {
7     eval(jiiIUpFi3);
8   } catch(e)
9   ...
10  }
11  XfNLVA421('a7A7a7A7ac9bB5b261...');
```

Listing 1.1. Excerpt of an obfuscated, real-world malicious script.

visited by the browser (`location.href`, Line 2), and the source code of the decryption function itself (`arguments.callee`, Lines 3,4). Using the current location as part of the key to the decryption function allows the attacker to prevent the analysis of the script when it is loaded from a different location. That is, when the script is captured, and during a later analysis served locally, the decryption will fail. The last step of the function uses the decrypted content in an `eval`[5] statement (Line 7). Nesting the `eval` in a try-catch block suppresses the errors that would be seen by the analyst if the `eval` would fail. This failure would happen, for example, in case the key is wrong.

```
1 function IxQUTJ9S() {
2   if (!Iw6mS7sE) {
3     var YlsElYlW = 0x0c0c0c0c;
4     var hpgfpT9z = unescape("%u00e8%u0000%u5d00%uc583% ...");
      ...
5     for (var CCEzrp0s=0;CCEzrp0s<Wh_74Nkm;CCEzrp0s++) {
6           je9rIXgu[CCEzrp0s] = QdV7IGyr + hpgfpT9z;
7     }
      ...
9   }
    ...
11  var Kp1uYOjP = new ActiveXObject('Sb.SuperBuddy');
12  if (Kp1uYOjP) {
13    IxQUTJ9S();
14    oH9mUjOd(9);
15    Kp1uYOjP.LinkSBIcons(0x0c0c0c0c);
```

Listing 1.2. Excerpt of a real-world, decrypted malicious script.

[5] ECMA-262 specifies that an implementation must provide an `eval` function. This function takes an argument of type string and interprets its argument as an ECMAScript program. That is, the `eval` function executes the argument it receives as a script.

After decryption, the string passed to `eval` contains the code excerpt presented in Listing 1.2. Line 4 loads x86 shellcode into variable `hpgfpT9z`. Subsequently, the heap is sprayed by filling the memory with a large number of strings that contain a NOP sledge and a copy of the shellcode (Lines 5-7). In Line 11, the SuperBuddy ActiveX component is instantiated. If a valid object can be created, then the vulnerable method `LinkSBIcons` is invoked (Line 15). The `LinkSBIcons` vulnerability is known as CVE-2006-5820 [4]; the argument of `LinkSBIcons` is used as a function pointer, thus diverting control flow to the sprayed heap.

3 Automatically detecting drive-by attacks

As described in the previous section, drive-by downloads that target memory corruption vulnerabilities have to prepare the environment before they can successfully launch their exploits. To this end, they use client-side script code to allocate (often large numbers of) strings that are filled with x86 (shell)code. The key idea of our detection approach targets precisely this behavior. More specifically, to detect drive-by downloads that exploit memory corruption vulnerabilities, we monitor all strings that are allocated by the JavaScript interpreter. These strings are checked for the presence of shellcode. Of course, all checks occur *before* a vulnerability can be abused to redirect control flow to the shellcode. When our system detects that a script creates a string that contains shellcode, the execution of the script is stopped.

The prototype implementation of our detection technique was implemented and integrated into the Mozilla Firefox browser and SpiderMonkey, its JavaScript engine. We chose Firefox as our prototype platform as this is an open source browser. Obviously, we would have liked to have integrated our solution into the Internet Explorer. Unfortunately, we did not have access to the source code. Nevertheless, we note that our solution is conceptually generic, and is not dependent on a specific browser. We have chosen to target JavaScript because it is by far the most common language for writing scripts on the web. Of course, an attacker could make use of a different language than JavaScript to deliver an exploit (and some indeed use Visual Basic Script). However, it would be straightforward to include our technique also into other script engines.

In the following sections, we describe our technique in more detail. In particular, we discuss how we keep track of the strings that are allocated, and how we detect the shellcode that an attacker may attempt to inject. Then, we discuss two optimizations that are applied to make the proposed approach fast enough to be used in practice.

3.1 Tracking object (string) allocations

For a drive-by attack to succeed, it is important that the bytes constituting the shellcode are stored at successive addresses in memory. Otherwise, these bytes would not be interpreted as valid x86 instructions. Of course, one could consider to split a sequence of instructions over multiple segments and connect these segments with jumps, but at least the bytes of each segment need to be consecutive to be valid. In JavaScript, the only way to guarantee that bytes are stored in a consecutive manner is by using a string

variable. Here, consecutive characters of the string are allocated in adjacent memory locations.

To detect the shellcode that a malicious script might construct on the heap, we have to keep track of all string variables that the program allocates. To this end, we modified the SpiderMonkey JavaScript interpreter that is embedded in Firefox. More precisely, we added code to all points in the interpreter where string variables are created. These points were found at three locations: one for the allocation of global string variables, one for local string variables, and one for strings that are properties (members) of objects. The code that we added simply keeps track of the start address of a new string variable and its length. Here, it is important to recall that strings in JavaScript are immutable. As a result, whenever a character in a string is modified, or when two strings are concatenated, the resulting string is created in a new memory area. Thus, string manipulation is automatically handled by the code introduced for creating a new string variable.

In addition to the start address and the length of new string variables, we also keep track of the two sub-strings that are used in a string concatenation operation. That is, whenever a new string is created as a result of a concatenation operation, we keep pointers to the sub-strings. This is needed for an optimization that is discussed later.

An attacker might consider to use integer arrays to store the shellcode in successive memory addresses. However, JavaScript supports arrays of integers that follow this (packed) memory layout only for 31-bit values, where the remaining bit is always set to indicate that the value is an integer. The fact that one bit is set in each four-byte integer makes it more difficult for the attacker to craft his shellcode. Also, support for packed integer arrays can be easily disabled. For integer values that are larger than 31-bit, and for all other data types, JavaScript handles arrays differently. More precisely, such arrays only store identifiers (pointers) that reference objects that are allocated elsewhere. Since these objects contain additional management information and are allocated from a pool of memory, it is very difficult for an attacker to reliably predict where these objects will end up. As a result, our system focuses on the content of string variables. Of course, when attackers develop techniques to store shellcode in objects that are allocated in the object pool, we could easily add checks for these objects as well.

3.2 Checking strings for shellcode

Given information about the addresses and lengths of the strings in memory, the next question that needs to be answered is how shellcode can be automatically detected within these strings. More precisely, we have to discuss how shellcode can be recognized, and the points in time when this analysis is launched.

For the detection of shellcode, we are leveraging *libemu* [19]. libemu is a small library written in C that offers basic x86 emulation and shellcode detection. It is efficient in detecting shellcode and being used in projects such as Nepenthes and Honeytrap. To recognize shellcode in a string (character buffer), libemu checks, starting from each character, whether there is a sequence of valid instructions of sufficient length. When such an instruction sequence is found, libemu reports shellcode. Since most bytes can be disassembled to valid x86 instructions, libemu also uses a number of heuristics to discriminate random instructions from actual shellcode. We currently use a value of 32

bytes as the threshold for the minimal length of a shellcode sequence. We found that this value works well in our experiments, and it is also significantly shorter than all Windows shellcode encountered in the wild [26].

Note that an attacker might try to evade detection by distributing shellcode fragments over multiple strings. In this case, to be successful, each fragment must end in a jump instruction to the next fragment. Moreover, since the total length of each fragment must not exceed 32 bytes, there is almost no space for a NOP sledge. As a result, the attacker must guess the jump offset quite precisely. While modern heap manipulation techniques allow for a certain control over the heap layout, we believe that such an attack is very difficult to launch in practice. Moreover, randomizing the allocation of individual objects in the heap would be easy to do and render this hypothetical evasion vector infeasible. Note that randomizing object allocations does not help against current drive-by attacks that store the complete shellcode in one string. The reason is that the location of a particular string might not be know precisely, but the attacker can allocate thousands of such self-contained, malicious strings (sometimes worth tens or hundreds of megabyte). Then, hitting a single string is sufficient to successfully run the shellcode.

The goal of our detection approach is to ensure that the attacker *cannot* execute shellcode *before* we analyze all (string) objects that he has created. The straightforward approach to do this is to invoke the emulator whenever a new string object is created. Of course, every string object is only checked once. Nevertheless, this naive approach incurs a significant performance penalty.

3.3 Performance optimizations

To reduce the performance penalty that is incurred when checking every string that is allocated, two approaches are possible. First, one can reduce the total number of invocations of the emulation engine. Second, one can reduce the amount of data that the emulator needs to inspect. Our prototype supports techniques to leverage speedups from both of these approaches.

Since vulnerabilities exploited by drive-by download attacks are almost always found in the browser or its plug-in components, we consider the JavaScript interpreter as safe. As a result, while executing JavaScript core functionality, a script is allowed to create string objects without checks, even ones that contain shellcode. To transfer control flow to such a string buffer with shellcode, the malicious script must exploit a vulnerability in an "external" component, leaving the JavaScript core part. Thus, to detect any shellcode before it can be executed, it is not required to perform emulation immediately after creating a new string object. Instead, it is possible to only record information on all created string objects, and postpone emulation to the time at which control flow leaves the interpreter, entering an external component or the browser.

The delayed checking allows us to collect information about the involved string objects and leverage this knowledge to decrease the overall amount of data that has to be checked. First, we use information about string concatenation, a frequent operation. Consider that we observe the fact that a given string `a` consists of the concatenation of strings `x` and `y`. This allows us to skip the analysis (emulation) of `x` and `y` when `a` was already scanned and found to be clean. A second venue for optimization is the JavaScript garbage collector. By invoking garbage collection on every transition from

the interpreter to the environment, we are able to discard all objects from the emulation that are freed by the garbage collector. We have modified the garbage collector routines to remove the freed contents from the list of objects to emulate (after zeroing their content).

Note that although the detection is delayed, it is still complete in the sense that no machine instructions residing in the memory space of a JavaScript object can potentially be executed before being checked by our shellcode detector.

4 Evaluation

This section discusses how we evaluated our prototype as well as the experimental results. The evaluation was carried out in three parts. First, we evaluated our system for false positives by accessing a large number of popular benign web pages. Second, we used our system on pages that launch drive-by downloads and evaluated the detection effectiveness. Third, we examined the performance overhead of our system.

4.1 False positive evaluation

In the context of our system, a false positive is a page that is detected as malicious without actually loading shellcode to memory. To evaluate the likelihood of false positives, we extended our prototype system to visit a list of $k = 4,502$ known, benign pages. These pages were taken from the Alexa ranking of global top-sites, and simply consisted of the top k pages. We consider this to be a realistic test set that reflects a wide range of web applications and categories of content.

For the batch evaluation of URLs, we implemented a Firefox extension that visits all URLs provided in a file. After a timeout, the extension automatically visits the next URL in the list. More precisely, the extension moves to the next URL two seconds after the page finished loading, or ten seconds after page loading started. The hard limit of ten seconds was necessary to handle scripts that continuously issued page reloads.

Our prototype did not produce any false positives for this dataset. This might look suspicious at a first glance: The x86 instruction set is known to be densely packed, thus, almost any sequence of bytes makes up valid instructions. However, one has to consider the fact that JavaScript uses 16-bit Unicode characters to store text. That is, even if a given sequence of ASCII characters results in a valid x86 instruction most of the time, the JavaScript representation of the same characters most likely does not, since every other byte would contain the value 0x0. Of course, an attacker can encode the shellcode appropriately. However, benign pages typically do not contain strings that map to valid instruction sequences.

4.2 Detection effectiveness

In a next step, we evaluated the capabilities of our technique to identify drive-by attacks that rely on shellcode to perform their malicious actions. To this end, we evaluated our system on the traces of 1,187 web browsing sessions that are known to contain drive-by attacks. These traces were collected by visiting URLs that are advertised in spam

emails. We retrieved a list of such URLs from the Spamcop [33] web service, as well as from mails collected in the spam trap of a medium-size security company.

To filter those URLs that actually host drive-by attacks, we used the Capture Honeypot Client (HPC) [2]. Capture visits the URLs with a browser on a virtual machine (VM). After a site is loaded, the state of the VM is inspected, and all modifications to the file system and registry as well as new processes are logged. In addition to the logged information, the system records a trace of all network communication that was taking place. Capture simply visits a URL in a browser and performs no additional actions. Thus, by filtering URLs that caused a new process to be launched, we were able to identify those sites that perform drive-by attacks. The system running in the VM was a Windows XP Professional (Service Pack 2) installation. No additional security patches were applied, and automatic updates were turned off. Additionally, the Flash and QuickTime plug-ins were installed.

Once a URL was identified to host a drive-by attack, we used Chaosreader [14] to extract application level data from the network traces. Chaosreader is able to recognize a variety of application data from network traces. Among others, Chaosreader identifies HTML documents, binary images, or gzip compressed data, saving each response to an HTTP request in a distinct file. Files that were found to be compressed were decompressed before continuing.

Extracting a single file for every response to an HTTP request made further post-processing necessary. For example, if an HTML page references a JavaScript URI via an `src` attribute of a `script` tag, this results in another request in which the browser fetches the JavaScript. The response contains only JavaScript code without surrounding HTML tags. Visiting such a file in a web browser results in its contents being interpreted as text, and thus, no interpretation of the code takes place. We used a simple heuristics to add the necessary HTML and `script` tags to such files. More precisely, whenever a file does not already contain valid HTML, and it does contain any of the most used JavaScript reserved words (e.g., `function`, `var`), it is wrapped in appropriate tags.

Once the HTML and JavaScript files were restored, they were uploaded on a local web server. In total, 11,910 URLs (files) were associated with the 1,187 traces (since a trace can contain multiple resources that are accessed by the browser, for example, due to redirection or embedded content). Our prototype system was instructed to visit each of these URLs. The modifications required to process encrypted attack scripts are detailed in Section 5. One might ask why we did not simply use our system to visit malicious pages that are live on the Internet, but, instead, replicate malicious sites and their scripts locally. The reason is that malicious sites on the Internet are frequently taken down. Additionally, many malicious sites only perform attacks on the first visit of a client. Thus, changing the prototype and revisiting the same location could not detect attacks hosted on such pages. In our setup, we have created a corpus that allows us to replicate our experiments and better debug and understand cases in which the detection fails initially.

When running our prototype detection system on the resources associated with 1,187 traces, we detected 956 instances of shellcode. This yields an initial detection effectiveness of 81%. We then examined the remaining 231 traces to understand why

our system did not detect shellcode while the Capture honeypot client indicated an attack.

Manual analysis revealed four main causes that result in our prototype failing to detect a threat. One group (with 62 traces) contains drive-by downloads that do not make use of memory exploits. In particular, a popular attack against the Sina Downloader ActiveX component exploits insecure component behavior. More precisely, this component contains functions that allow a script to download a file and to start a program. This makes it trivial for an attacker to download malware and start it, without ever corrupting memory. However, note that this attack targets an old vulnerability (from 2006) that is very specific to a particular component. Thus, it is not a general class of vulnerabilities that our approach misses, but a specific problem in a component that basically offers all the functionality required by the attacker.

The second group of attacks (with four traces) that were missed by our system are due to exploits that use Visual Basic (VB) script code to prepare the environment and launch the exploit. As mentioned previously, our current prototype only instruments the JavaScript engine. However, similar techniques could easily be added to the VB script engine.

The third group of missed attacks (with 127 traces) are due to the way our experiments are carried out. Recall that we do not visit live pages on the Internet, but invoke individual resources (files) that we extracted from network traces. In some cases, the malicious code is distributed over several scripts that are in different files. In these cases, the browser does not see and analyze the complete, malicious script at once. This typically leads to JavaScript errors, and failure to inject shellcode into the heap. This, however, does not reflect a deficiency in our approach. If these sites were visited with a browser protected by our proposed technique, all scripts would be fetched and executed by the web browser in the same context, thus, allowing to detect the threat.

Finally, a forth group (with 38 traces) was not recognized as malicious because it contains traces that were false positives of the Capture honeypot client. More specifically, they were `.cab` archive files. Whenever a `.cab` file is downloaded, Windows automatically starts the Windows Management Instrumentation to handle this resource. While this activity results in a new process being launched, it is not because of a malicious drive-by download but due to legitimate activity. However, Capture considers the start of a new process as an indication of a successful attack.

Given the discussion of the four cases above, we argue that only the traces associated with attacks against the Sina Downloader ActiveX and similar components should be considered false negatives for our system. As a result, we can compute a detection rate of $\frac{956}{956+62} = 93.9\%$. Also, we observe that we detected all drive-by attacks that exploited a memory corruption vulnerability, which is by far the most common type of exploit found in the wild.

After evaluating the detection capabilities of our system, we also performed further analysis of the ActiveX components created by the malicious scripts. Our results show that most malicious sites perform their attacks through only a handful of vulnerable components. Figure 2 depicts a breakdown of the distribution of the involved components. It is interesting to observe that the two most prominent components (SuperBuddy and QuickTime viewer) account for almost 50% of the targets of the attacks. Note that

the figure lists the 1,688 ActiveX components that were created during our evaluation. Nonetheless, not every created component lead to a successfull exploit.

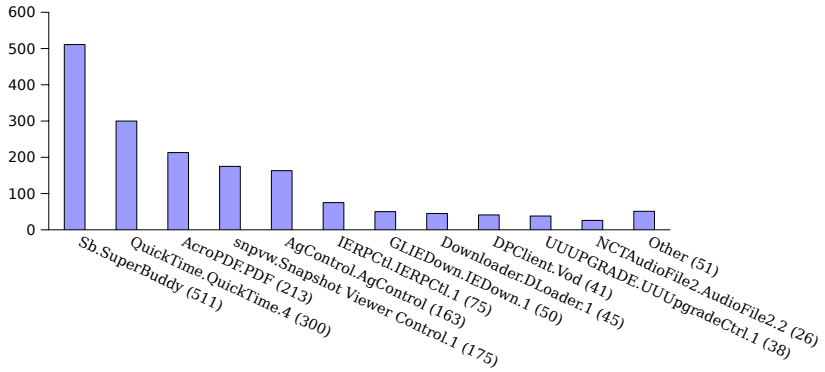

Fig. 2. ActiveX components involved in drive-by downloads.

4.3 Performance

Our approach uses x86 instruction emulation to detect shellcode within JavaScript strings. This happens online; that is, the analysis must be performed at the time the browser loads a page. Since emulation is a resource intensive task, careless invocations of the emulator may lead to a significant performance overhead. We have pursued several strategies to minimize the overhead, as explained in Section 3.3. In this section, we describe the results of our performance evaluation.

Our experiment measures the wall-clock time required to load a set of web pages. We have chosen the 150 most popular web sites (according to Alexa). The same set of pages was processed three times. First, we ran an off-the-shelf Mozilla Firefox browser without performing any additional tasks. Second, we used our modified version of the browser that provides protection against drive-by download attacks, without any performance optimizations. Third, we used the browser with protection and performance optimizations.

All measurements have been carried out on a machine with an Intel Core 2 Duo processor running at 2.66 GHz and 4 GB of main memory. Internet connectivity was established using an ADSL line with a bandwidth of 1 MBit/s.

The results of our performance evaluation are presented in Table 1. On average, an unmodified Firefox browser took 3.51 seconds to load one web page from our testing set. This time includes the download of the content over the Internet, parsing and rendering of the page, and execution of all JavaScript code. In comparison, a modified version of the browser, which provides protection against drive-by download attacks, takes additional time. The overhead can be attributed to the effort spent on tracing the

		Total time[s]	Time/page[s]	Overhead/page	Factor
Off-the-shelf browser		527	3.51		
Protected browser	w/o optimizations	1,237	8.25	4.74	2.35
	w/ optimizations	876	5.84	**2.33**	**1.66**

Table 1. Page load times (sec) with and without drive-by download protection.

allocated string objects, and more importantly, emulation of their content when executing functionality from the JavaScript environment. A basic implementation of our system, without application of performance optimization measures, took 8.25 seconds per page. This is a significant performance penalty. Our final implementation, including all optimizations took, 5.84 seconds per page. That is, the overhead of the naive version could be reduced in half.

Browsing the Web is an interactive occupation, and it is desirable for the user to experience as little latency as possible when loading a new page. Obviously, the decrease in performance introduced by our approach seems significant. However, note that the time users typically spend on consuming the downloaded content (e.g., reading an article) by far outweighs the time that is spent on waiting for new content to be loaded. Thus, we believe that the benefit of a secure browsing experience, without the risk of falling prey to a drive-by download attack, well compensates the inflicted performance penalty.

5 Implementation details

As mentioned previously, our system has been implemented by extending Mozilla Firefox and SpiderMonkey. However, all drive-by download attacks in our dataset target the Internet Explorer (IE). The astute reader might wonder how our system can actually detect such attacks, since they are not supposed to work with Firefox. In the following, we provide some (what we believe) interesting details on how we implemented our system to detect IE attacks with a modified Firefox browser. Of course, when our technique would be integrated with Internet Explorer, such extensions would not be necessary. Also, the system as introduced can readily detect drive-by downloads that target Firefox. Moreover, we discuss some additional issues that needed to be addressed because of our experimental setup.

Simulating ActiveX components. Attacks that aim to exploit a vulnerability in a specific plug-in often perform a check for the availability of this plug-in. That is, such attacks only reveal their malicious behavior when the vulnerable component is present. In the case of ActiveX plug-ins, this is done by trying to instantiate the vulnerable component. If the plug-in object is instantiated successfully, it usually means that the component is present.

Unfortunately, Mozilla Firefox does not support ActiveX plug-ins. However, as most drive-by attacks rely on ActiveX to be present, we had to modify the browser appropriately. More precisely, we extended Firefox such that it creates dummy objects

for instantiation requests to ActiveX components. Thus, whenever a malicious script attempts to instantiate an ActiveX component, the call succeeds and the corresponding dummy object is created.

These objects accept all method invocations, and also log method calls together with their respective arguments. Note that although it is not the main focus of our work, this information can be used to identify the vulnerability that is used to divert the control flow.

Browser fingerprinting. Browser fingerprinting is a technique applied by attackers to serve only exploits that match the specific browser of the sites' visitors. To this end, instead of bluntly trying a series of attacks, a script is executed to determine the browser, its version, and installed plug-ins. Based on the knowledge gathered by this script, it fetches only those exploit scripts that match this setup (e.g., if no QuickTime plug-in is detected, no QuickTime related exploits are tried). Even when no fingerprinting is performed as described above, the malicious script most likely verifies that it is executed in a browser that it intends to exploit. Therefore, the script queries the properties of the `navigator` object and only continues if the information matches its authors' intentions. Since our prototype is implemented in Mozilla Firefox, this would have prevented all scripts that perform such techniques from executing. However, the recorded traces hold proof of a successful drive-by attack. Thus, we modified our prototype to pretend to be the same browser and version[6] that was used when the traces were recorded.

To assure that the script is executing in Microsoft's Internet Explorer, attackers rely on inaccuracies of the JScript parser. More precisely, the JScript parser is more tolerant with regards to semicolons than SpiderMonkey.

```
1 try {
2   ...
3 } catch (e) {};
4 finally {
5   ...
6 }
```

Listing 1.3. Illustration of different parsing behavior.

Listing 1.3, for example, illustrates this with a try-catch-finally construct. While the JScript parser gladly accepts this syntax (notice the semicolon after the catch block in Line 3), the SpiderMonkey engine terminates the script with an error (i.e., "finally without try") at Line 4. These different parsing behaviors introduce further means for an attacker to make sure the script is interpreted by the Internet Explorer. As we could observe such attacks in the wild, we had to modify the parser of our prototype to reflect the behavior of the JScript parser.

[6] Corresponding to the user-agent string: `Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)`

Dynamic encryption keys. Most malicious scripts are encrypted in some way. The attackers' motivation to disguise malicious scripts is obviously the intention to encumber the analysis of such scripts. Encryption is a straightforward approach to do so.

An encrypted script contains a decryption routine and a cipher text. During execution, the cipher text is decrypted by the routine, and the result is executed via JavaScript's `eval` function. Two possibilities exist where the decryption routine derives the correct key from. (1) the key might be part of the script itself (e.g., stored in a variable), or (2) the key is dependent on the environment of the script. While in the first case, decryption is automatically handled by the interpreter, the second case requires that the environment presents the right information for the queried value. In our evaluation dataset, many decryption keys were partly derived from the current URL of the browser. Since the scripts were hosted at a local web server, the URLs were different, thus leading to wrong decryption keys. For wrong key values, the decryption routines produce only garbage and, as a result, no malicious behavior can be observed. Since, on the other hand, the values were correct when the network traces were recorded, we modified our prototype to report the URL that was visited during the recording of the trace as the current location. This allowed the scripts to decrypt the cipher text correctly, and we were able to analyze and detect their malicious behavior.

Batch processing time-outs. Some malicious scripts use the `setTimeout` function of JavaScript to delay their actions. During our batch processing of URLs, we use a time-out of ten seconds before moving to the next page. As a result, the usage of such timers could prevent detection. To mitigate this problem, we had to assure that these timeouts expire before the batch processing extension moves to the next URL. To this end, we modified Firefox to replace all delays of `setTimeout` calls with a delay of 50ms.

Interestingly, during our evaluation, we encountered a malicious script that implemented a custom version of a setTimeout-equivalent function. More precisely, the script looped and measured the expired time between the initial run of the loop and the current time. Once the desired delay was reached, execution continued. This sample did not use the `setTimeout` function and thus, the extension switched to the next URL before the malicious content was executed. Notice, however, that not detecting the malicious script in this sample is an artifact of the batch processing and does not indicate a weakness in our proposed approach. In fact, after removing the sleep function, the system did detect the malicious script, the shellcode it used, and the involved ActiveX components.

6 Related work

Many researchers have proposed methods to analyze, detect, and mitigate the threat posed by malicious software. For malware analysis, two different approaches exist. While dynamic analysis actually executes the malware, static analysis is performed without running the software in question. Dynamic approaches execute the malware in a controlled environment, and observe the interaction of the malicious component with the environment. Hooking API function calls results in detailed information of the behavior of a program.

CWSandbox [36] uses hooking to log the invocations of Windows API function. Similarly, Anubis [1] performs its analysis via virtual machine introspection [13] on an application that is executed in an emulated machine. A mixture of static and dynamic techniques is applied by Kirda et al. [18] to detect malicious browser plug-ins. Egele et al. performed information flow analysis on browser plug-ins [9] to identify spyware components that leak sensitive information. Information flow analysis is also the key idea of Panorama [37], where Yin et al. implemented a system to discover rootkits. While powerful, existing analysis techniques are typically too heavyweight to be used for detection on a client machine. In contrast to that, our proposed technique detects drive-by download attacks by monitoring potentially malicious scripts directly in the browser.

Previous studies have shown that drive-by download attacks pose a real threat to the Internet and its users. The mechanisms used by attackers to mount their attacks are investigated by Provos et al. in [29]. The life cycle of an infected machine is analyzed by Polychronakis in [27]. In [28], Provos et al. present a measurement study that reports that the results for 1.3% of all Google search queries contain at least one link pointing to a page that performs a drive-by attack. Also, Frei et al. [12] analyzed the vulnerability landscape of web browsers in the Internet. Apparently, only 60% of the users that navigate the Internet everyday use the latest, most secure version of their web browser. Based on a Secunia report [31], the authors argue that many browser plug-ins commonly in use have known vulnerabilities. The fact that many users only reluctantly update their web browsers and plug-ins makes it feasible for attackers to distribute attacks that target old vulnerabilities. As many of the vulnerabilities leading to control flow hijacking are present in ActiveX components, Dormann and Plakosh [8] propose fuzzy testing as a means of detecting such flaws before distributing a component.

Detecting shellcode in network traffic has a long standing history. Network intrusion detection systems, such as Snort [30] or Bro [23], rely on signatures to identify malicious network streams. While signature detection works well for known static threats, advanced polymorphic shellcode and engines that can automatically produce such shellcode can sometimes evade these detection techniques. Based on abstract payload execution, Toth and Kruegel have proposed a mechanism to detect buffer overflow attacks [34]. More precisely, their prototype implementation identifies long valid sequences of instructions in HTTP requests, thus detecting the NOP sledge that commonly accompanies shellcode. Continuing this work, Polychronakis et al. [24, 26] proposed to apply lightweight emulation on network data to identify polymorphic shellcode. This approach relies on the so-called GetPC heuristic. That is, a shellcode is only identified if a sequence of instructions is emulated that reads the current program counter value. The class of non-self-contained shellcode, however, contains code that reaches its goal without showing such behavior. In [25], the authors extend their detection techniques to also identify this class of attacks. While network-traffic-based techniques are useful, they typically cannot be used to detect drive-by downloads. The reason is that, although JavaScript contents of a web page are transmitted over the network, this code is often obfuscated. Furthermore, the shellcode contained in the JavaScript scripts are not transmitted in binary form. Instead, the ASCII representation of the individual bytes is transmitted. This sequence does not yield a valid instruction sequence in general.

Analyzing malicious JavaScript has recently gained more attention by the scientific community. Hallaraker and Vigna [15] present an approach to audit the execution of JavaScript code. These audit logs can be compared to high-level policies to detect potential attacks. Similarly, Feinstein and Peck introduced Caffeine Monkey [10], a tool that supports the collection and analysis of malicious JavaScript. To this end, they extended the Mozilla SpiderMonkey JavaScript engine by adding run-time logging facilities. Chenette et al. [3] aim at automatically reversing the obfuscation of malicious JavaScripts. Their approach relies on hooking techniques to monitor calls to relevant JavaScript functions, such as eval or document.write. These systems focus on auditing JavaScript activity, while our approach aims at detecting malicious drive-by downloads.

Vogt et al. propose a system that prevents cross-site scripting attacks performed by malicious JavaScript code [35]. To protect a user from JavaScript that tries to steal sensitive information, the propagation of such information through the JavaScript engine is tracked. Requests to a domain containing information originating from another domain raise an alert, and allow the user to stop further execution of the script.

7 Conclusion

Drive-by downloads belong to the most threatening vectors of attack that are currently used by cyber-criminals to illegitimately gain control of victims' computers. In this paper, we present a novel approach that helps protect a user against drive-by attacks that rely on shellcode.

Our system is integrated into the web browser where it monitors JavaScript code that is downloaded and executed. More precisely, our system traces all string objects that are created during run-time, and it uses x86 instruction emulation to determine whether a string buffer contains executable shellcode. The detection of the shellcode takes place before a vulnerability can be exploited (and control flow redirected). Hence, an attack can be mitigated before the security of the browser is compromised.

Our approach includes optimizations to assure a reasonable performance overhead while delivering excellent detection results for drive-by attacks that exploit binary vulnerabilities in browser plug-in software. We have built a prototype implementation with which we have verified the capability of our approach to successfully detect real-world drive-by download attacks. Our evaluation shows that our approach is feasible in practice.

Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) under grant P18764, SECoverer FIT-IT Trust in IT-Systems 2. Call, Austria, Secure Business Austria (SBA), and the WOMBAT and FORWARD projects funded by the European Commission in the 7th Framework.

References

1. U. Bayer. Anubis - analyzing unknown binaries. `http://www.anubis.iseclab.org`.

2. Capture-HPC Client Honeypot / Honeyclient. https://projects.honeynet.org/capture-hpc, 2009.
3. S. Chenette. ToorConX - the ultimate deobfuscator. http://www.toorcon.org/tcx/26_Chenette.pdf, 2008.
4. Superbuddy activex control vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5820, 2006.
5. D. Dagon, G. Gu, C. Lee, and W. Lee. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conference (ACSAC)*, 2007.
6. Dan Goodin (The Register). SQL injection taints BusinessWeek.com. http://www.theregister.co.uk/2008/09/16/businessweek_hacked/. Last accessed, December 2008.
7. M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *2nd USENIX Workshop on Offensive Technologies (WOOT08)*, 2008.
8. W. Dormann and D. Plakosh. Vulnerability detection in activex controls through automated fuzz testing. http://www.cert.org/archive/pdf/dranzer.pdf, 2008.
9. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song. Dynamic spyware analysis. In *USENIX Annual Technical Conference*, pages 233–246, 2007.
10. B. Feinstein and D. Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. http://www.dc414.org/download/confs/defcon15/Speakers/Feinstein_and%20_Peck/Whitepaper/dc-15-feinstein_and_peck-WP.pdf, 2006.
11. M. Foundation. SpiderMonkey (JavaScript-C) Engine. http://www.mozilla.org/js/spidermonkey/.
12. S. Frei, T. Dübendorfer, G. Ollmann, and M. May. Understanding the web browser threat. Technical Report 288, ETH Zurich, 06 2008. 2008.
13. T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *10th Annual Network and Distributed System Security Symposium (NDSS03)*, 2003.
14. B. Gregg. fetch application data from snoop or tcpdump logs. http://chaosreader.sourceforge.net/.
15. O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005)*, pages 85–94, 2005.
16. John Leyden. Drive-by download attack compromises 500k websites. http://www.channelregister.co.uk/2008/05/13/zlob_trojan_forum_compromise_attack/. Last accessed, February 2009.
17. C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *ACM Conference on Computer and Communications Security*, 2008.
18. E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *USENIX Security*, 2006.
19. x86 shellcode detection and emulation. http://libemu.mwcollect.org/.
20. D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Usenix Security Symposium*, 2001.
21. M. D. Network. ActiveX Controls. http://msdn.microsoft.com/en-us/library/aa751968.aspx.
22. M. D. Network. JScript (Windows Script Technologies. http://msdn.microsoft.com/en-us/library/hbxc2t98.aspx.
23. V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31, 1999.

24. M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference (DIMVA)*, pages 54–73, 2006.
25. M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Recent Advances in Intrusion Detection, 10th International Symposium (RAID)*, pages 87–106, 2007.
26. M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, 2007.
27. M. Polychronakis and N. Provos. Ghost turns zombie: Exploring the life cycle of web-based malware. In *First USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2008.
28. N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *USENIX Security Symposium*, 2008.
29. N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost In The Browser Analysis of Web-based Malware. In *First Workshop on Hot Topics in Understanding Botnets (HotBots '07)*, 2007.
30. M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *13th Systems Administration Conference (LISA)*, 1999.
31. Secunia PSI study: 28% of all detected applications are insecure. `http://secunia.com/blog/11/`, 2007.
32. A. Sotirov. Heap Feng Shui in JavaScript. `http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html`. Last accessed, November 2008.
33. Spamcop - the premier service for reporting spam. `http://www.spamcop.net/`.
34. T. Toth and C. Krugel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, pages 274–291, 2002.
35. P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007.
36. C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.
37. H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Communications Security*, pages 116–127, 2007.