Mitigating Drive-by Download Attacks: Challenges and Open Problems

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

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

Abstract. Malicious web sites perform drive-by download attacks to infect their visitors with malware. Current protection approaches rely on black- or white-listing techniques that are difficult to keep up-to-date. As todays drive-by attacks already employ encryption to evade network level detection we propose a series of techniques that can be implemented in web browsers to protect the user from such threats. In addition, we discuss challenges and open problems that these mechanisms face in order to be effective and efficient.

1 Introduction

The commercialization of the Internet attracted people with malicious intents who strive to gain undeserved revenues by exploiting security flaws in Internet applications. These miscreants often attack remote systems with the intent of stealing valuable, sensitive information such as passwords, or banking credentials. Today, attacks against remote systems are often carried out as drive-by download attacks. In a typical attack, a user visits a web page that silently (i.e., without the users consent) downloads and executes malicious code on her computer. Although many solutions have been proposed to mitigate threats such as malicious e-mail attachments, network intrusions (e.g., [17, 23]), and malware (e.g., [8, 12, 27, 28]), detecting and preventing drive-by downloads has not received much attention so far.

In this paper, we discuss some challenges and open problems in detecting drive-by download attacks. Furthermore, we argue that browser vendors need to integrate mechanisms into their browsers to efficiently protect Internet users against drive-by download attacks. The paper makes the following main contributions:

- We describe different techniques that can be used to perform drive-by attacks, and elaborate on why existing protection mechanisms fail to mitigate these threats.

– We discuss new approaches that should be able to detect and mitigate drive-by attacks.
– We discuss research challenges and open problems in creating successful and efficient solutions to prevent drive-by download attacks.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of active content (i.e., scripting and plug-ins) in web browsers. Different drive-by attack scenarios are discussed in Section 3. Possible mitigation strategies for these scenarios are presented in Section 4. Section 6 discusses related work in the area, and Section 6 concludes.

2 Active Content in Web Browsers

To enrich the rather static appearance of HTML pages, browser vendors started to support active client-side content. Among the supported techniques, JavaScript is arguably the most widely known and used. Many of the so-called Web 2.0 sites heavily rely on JavaScript and the related Ajax technology to implement support for highly dynamic content. Further techniques that are widely deployed are Adobe Flash or support for executing Java applets. Additionally, Microsoft's Internet Explorer features Visual Basic Script (VBScript) support. All these techniques have in common that they download and execute code from the Internet. Since this code is under control of the respective web site's owner, it has to be regarded as being potentially malicious.

To confine the impact these programs can have on the browser and the underlying operating system, browser vendors integrated security models into their products. JavaScript, for example, follows a *same origin* policy that grants scripts only access to the data that was retrieved from the same domain as the script itself (i.e., same origin). It is, thus, not possible for a malicious script to steal sensitive information, such as a session cookie, that originated from a different site. Furthermore, files on the local system can neither be read nor written. Flash, in addition, supports general network socket communication. Similarly to the same origin policy of JavaScript, connections can only be established with the same server where the content was originally retrieved from.

Most web browsers have a concept of plug-ins. These plug-ins allow third party developers to extend the browser's functionality. Moreover, plug-ins commonly have higher privileges (e.g., reading/writing files, opening connections to arbitrary hosts) than script code embedded in a web page. Microsoft's Internet Explorer, for example, uses ActiveX technology to implement plug-in support. These plug-ins are accessible to the web browser, and unless explicitly denied, also to scripts embedded in web pages. Such plug-ins allow users, for

example, to view PDF or Microsoft Office documents in the web browser itself, instead of launching the respective application. Additionally, many third party software products contain ActiveX components to be used in the context of the web browser. Such components may support media playback, software updates, or transfer applications to hand-held devices. Since client-side scripts have access to these components, the provided functionality is available to these scripts as well.

An important similarity between all active client side content techniques is the fact that they share the address space with the web browser. For example, JavaScript objects are allocated on the browsers heap and plug-ins are loaded into the browsers address space, thus having access to the same memory contents. Because of the shared address space and the elevated privileges of plug-ins, this combination is a popular target for drive-by download attacks.

3 Drive-by Download Attacks

Drive-by download attacks are downloads that occur without the knowledge or consent of a user. After downloading, the application is invoked and is free to perform its nefarious purposes. The mere visit to a malicious web site can lead to the download and subsequent execution of malicious software on a visitor's computer. Obviously, attackers strive to infect as many victims as possible. Furthermore, an attacker can misuse a web page, even without having full control over the page's contents. Buying advertisements, for example, can sometimes allow an attacker to have her malicious code included in pages that display the advertisements. Also, by exploiting security vulnerabilities in web applications, attackers can often automatically modify these sites to host their malicious code [11]. Adding the malicious content to pages with a large number of visitors (e.g., as was the case for BusinessWeek.com [5]) raises the chance for an attacker to be able to infect many users.

Once a user visits a page that launches drive-by attacks, a common first step in the attack is to perform fingerprinting of the visitor's browser. To this end, a script collects information about the browser version and language, operating system version, or enumerates the installed plug-ins. Subsequently, the browser is instructed to load exploit code that matches the gathered information. For example, an exploit for a QuickTime vulnerability is only loaded if the fingerprinting detects the plug-in to be present.

Typically, drive-by attacks focus on exploiting vulnerabilities in popular web browsers. Moreover, applications that rely on web browser capabilities might also be vulnerable to such attacks. For example, e-mail programs that use a browser's rendering components in order to display HTML e-mails, or

media players that display additional information for currently playing songs could contain bugs that can be exploited in a drive-by download. Besides targeting the web browser directly to perform drive-by attacks, an attacker can also exploit security flaws in plug-ins that extend the browser's functionality. For example, a buffer overflow vulnerability in Flash [1] allowed an attacker to execute arbitrary code on a victim's computer. The pervasive install base of Flash products make this a viable target for attackers. Note that the advance of malicious PDF files that exploit vulnerabilities in Adobe's Acrobat products show that this problem is not confined to web browsers.

The remainder of this section introduces the two major strategies that attackers make use of to launch drive-by download attacks. First, we discuss attacks that rely on API misuse. In the second part, we focus on attacks that exploit vulnerabilities in web browsers, or their plug-ins. For each technique, a real-world example will be given to illustrate the attack.

3.1 API Misuse as Attack Vector

Insecurely designed APIs (application programming interface) allow an attacker to launch a drive-by attack. For example, the `DownloadAndInstall` API of the Sina ActiveX component [24] was intended to provide update functionality for the component itself. One of the method's parameters is a URL indicating the location of the file to download and install. As the argument was not checked to indicate a trusted source, this API could be used to download and execute arbitrary files from the Internet. In 2008, an exploit for the Microsoft Office Snapshot Viewer [14] allowed to download arbitrary files from the Internet to a local directory of choice. Downloading a malicious file to an auto-start location could infect the computer of an unsuspecting web user. Slightly more complicated, but similar in effect, was an attack based on [15]. Listing 1.1 illustrates this attack. The APIs of three different components were used to perform a drive-by download attack. The first component allowed to fetch arbitrary contents from the web (Lines 10,11). In a second stage, another component was used to save these contents to a local file on the disk (Lines 12-16). The third and final API then allowed to execute (Line 18) the downloaded file with the privileges of the browser and without the user knowing.

3.2 Exploiting Vulnerabilities in Browsers and Plug-ins

Exploiting vulnerabilities in web browsers or plug-ins also allows an attacker to perform drive-by download attacks. Such attacks typically follow the following scenario. First, the attacker loads a sequence of executable instructions,

```
1  var obj = document.createElement('object');
2  obj.setAttribute('id','obj');
3  obj.setAttribute('classid','clsid:BD96C556-65A3-11D0-983A-00C04FC29E36');
4  try {
5    var asq = obj.CreateObject('msxml2.XMLHTTP','');
6    var ass = obj.CreateObject("Shell.Application",'');
7    var asst = obj.CreateObject('adodb.stream','');
8    try {
9      asst.type = 1;
10     asq.open('GET','http://www.evil.org//load.php',false);
11     asq.send();
12     asst.open();
13     asst.Write(asq.responseBody);
14     var imya = './/..//svchosts.exe';
15     asst.SaveToFile(imya,2);
16     asst.Close();
17   } catch(e) {}
18 try { ass.shellexecute(imya); } catch(e) {}}
```

Listing 1.1. API misuse drive-by download

so-called shellcode, into the address space of the web browser. This usually happens by using client-side scripting such as JavaScript or VBScript. The second step of the attack exploits a vulnerability in the browser or a plug-in that allows the attacker to divert the control flow of the application to the shellcode. The shellcode, in turn, is responsible for downloading and executing the malicious application from the Internet. As the shellcode is provided by the attacker, it can make use of system libraries to ease its task. Again, the whole procedure is performed without the user noticing. Exploits that follow this attack vector face similar difficulties as any other control flow diverting attacks (e.g., buffer overflows). The biggest challenge for an attacker is to predict the exact location of the shellcode in memory. This information is crucial in order to successfully perform the attack. Not being able to precisely divert the control flow to the shellcode will most likely result in a crash of the browser instead of the intended download and execution of additional malicious components.

To increase the chance that the diverted control flow results in executing the shellcode, the attacker commonly prepends the shellcode with a so-called NOP sledge. This NOP sledge is a series of instructions that do not perform any action (e.g., the x86 no-operation instruction). It is thus sufficient for the attacker to divert the control flow to anywhere within the NOP sledge, as execution sleds down the NOPs, and subsequently executes the shellcode.

Heap Memory Manipulation NOP sledges increase the chance for an attacker to successfully exploit control flow diverting vulnerabilities. Within web browsers, the use of client-side scripting allows an attacker to further increase that chance. A technique called *heap spraying* relies on client-side scripting (e.g., JavaScript, VBScript) to fill large portions of the browser's heap memory with shellcode and prepended NOP sledges. For example, an attacker can embed a script in a web page that, in a loop, assigns copies of a string to different variables. If this string consists of the NOP sledge and shellcode, the attacker can easily manipulate the heap in a way that large address ranges contain these string values. If the attacker additionally leverages knowledge about how the browser's heap memory is managed [6, 25], control flow diverting vulnerabilities can be exploited reliably. Performing a series of operations that allocate and de-allocate memory allows an attacker to predict an address that will contain the NOP sledge after spraying the heap.

3.3 Observations

We now describe some observations that we made during our initial studies of drive-by download attacks in the wild. Most of the attacks we encountered so far rely on JavaScript to perform their nefarious actions. Furthermore, many attacks involve vulnerable ActiveX components. Although a fix for the MDAC vulnerability [15] is available since 2006 many malicious sites still try to launch attacks exploiting this vulnerability. Other frequently targeted components include media player plug-ins and search engine toolbars.

Obfuscation Frequently, we encountered attacks that were obfuscated. One way to obfuscate the attack code is to encrypt it and to prepend the cipher-text with a decryption routine. To hamper analysis, the decryption key can be chosen to be dependant on the source code of the decryption function. Therefore, modifying the decryption routine by adding debugging instructions modifies the key and subsequently results in distorted and invalid output.

Listing 1.2 is an excerpt from the code of a real-world drive-by attack. Before being able to analyze it, we had to reverse the encryption to get the plain text version of the attack. In Line 4, the shellcode is assigned to a variable. The loop at Lines 6 - 8 then performs heap spraying[4] and stores copies of the shellcode with a prepended NOP sledge in successive array indices. At Line 12, a SuperBuddy ActiveX component is instantiated and Line 16 exploits the vulnerable `LinkSBIcons` [3] method that transfers the control flow to the passed pointer (in this case 0x0c0c0c0c which lies in the region of the sprayed heap).

[4] This sample sprayed more than 150MB of heap memory.

```
1  function IxQUTJ9S() {
2    if (!Iw6mS7sE) {
3      var YlsElYlW = 0x0c0c0c0c;
4      var hpgfpT9z = unescape("%u00e8%u0000%u5d00%uc583% ...");
5  ...
6      for (var CCEzrp0s=0;CCEzrp0s<Wh_74Nkm;CCEzrp0s++) {
7        je9rIXgu[CCEzrp0s] = QdV7IGyr + hpgfpT9z;
8      }
9  ...
10   }
11 ...
12   var Kp1uYOjP = new ActiveXObject('Sb.SuperBuddy');
13   if (Kp1uYOjP) {
14     IxQUTJ9S();
15     oH9mUjOd(9);
16     Kp1uYOjP.LinkSBIcons(0x0c0c0c0c);
17     var Dr_RHrVa = new ActiveXObject("QuickTime.QuickTime.4");
18     if (Dr_RHrVa) {
19 ...
20       for(var vyLOQHfP=0;vyLOQHfP<3;vyLOQHfP++) {
21         Bz9o4Aco += "\x0c\x0c\x0c\x0c";
22       }
23 ...
24       param name="qtnext1" value="<rtsp://AXDOF:" + Bz9o4Aco
25 ...
```

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

If the attack was not successful (e.g., because a fixed version of the component was installed), the script creates a QuickTime ActiveX component. Lines 20 - 22 create a long argument that results in a buffer overflow when passed as the URI parameter to the component [4]. Again, the attack tries to divert control flow to the same sprayed memory location.

4 Possible Mitigation Strategies

Depending on the type of attack, we envision that different mitigation techniques can be applied to protect the user from drive-by downloads. These techniques can be applied on different levels. Google, for example, adds warning labels to search results that it found to contain malicious software. Although this measure gives some protection, it is inherently difficult for the search engine to identify and keep track of such malicious pages. As Google relies on scanners that identify malicious sites, a page that starts to launch drive-by attacks goes unnoticed at least until the next examination of the scanner. During

this interval, the user is exposed to the risk of getting infected with malware once the search result link is clicked.

Recently, AVG integrated a technology called LinkScanner [13] into their anti-virus products that actively scans the results of search engine queries and presents the user with security information regarding these sites. Although this allows for a more up-to-date assessment of possible risks, concerns have been raised for the additional load this method creates by examining all search engine results regardless of whether they are visited or not. Additionally, malicious pages might disguise their intentions by trying to distinguish between LinkScanner and a regular visitor, launching attacks only in the latter case.

To protect Internet Explorer users from attacks that exploit known vulnerabilities in ActiveX components, Microsoft regularly updates the list of components whose kill bit is set in the registry. Setting the kill bit of a component indicates that this component is not safe for scripting. Any request to instantiate such a component from a script embedded in a web site is declined by the system and results in an error. Although effective, this measure only protects from known vulnerabilities. Moreover, for third party components, Microsoft only sets the kill bit upon request by the vendor of the component.

4.1 Browser Built in Protection

For the shortcomings mentioned above, we envision protection mechanisms built into the browser itself. As the decision whether a page is malicious or not is reached during the download and interpretation of the page itself, such techniques do not suffer from outdated information. Furthermore, performing the analysis only on the currently visited page does not create additional load for other non-related sites. As many drive-by attacks rely on client-side scripting, we focus on mechanisms that allow for detection of such attacks.

The information required to cast a decision on the maliciousness of a page can be gathered from scripts in two ways. Either statically by analyzing the page and the scripts it contains, or dynamically, where the analysis is performed during the execution of the scripts. The main advantage of static analysis over dynamic analysis is that all possible execution paths can be taken into account. If a script, for example, exhibits malicious behavior only if viewed with a specific browser, static analysis could still reveal the malicious intents of the script, even when visited with a different browser. Obfuscation and encryption schemes, on the other hand, give an attacker an easy means to prevent efficient static analysis.

As our focus lies in protecting the user from drive-by attacks launched by the page currently viewed, we are mainly interested in those code paths that are actually executed. That is, an attack that is not executed does not harm the

user. In addition, obfuscation and encryption do not pose an obstacle to dynamic analysis.

Static and Machine Learning Approaches: To prevent malicious scripts from using APIs in unintended ways, we propose a technique that infers sets of possible values or domains of parameters (similar to [9]). Although the parameter types for APIs are usually checked (e.g., implicitly by the interface definition of an API) further analysis of the component might be able to derive additional meta information regarding the API function parameters. If, for example, a parameter is compared against a finite set of constants, this information helps to identify calls that contain arguments that do not appear in this set. Anomaly based detection methods, such as [22], might also prove effective in protecting users from drive-by attacks. A feasible approach might first learn the character distributions of arguments of legitimate API calls. Once the learning phase is complete, subsequent invocations of the same API are scrutinized and the character distribution of the arguments is compared against the learned information. If the difference is above a certain threshold, a possible attack is identified and the user is notified.

A further learning based approach could operate on behavioral profiles of the involved scripts. To this end, a set of known benign scripts is used to establish a body of known good profiles. The information contained in these profiles might include the size of the involved scripts, the names of invoked functions and variables, or any other meta information that can be automatically extracted. Once the benign profiles are learned, the system calculates the actual profile for every visited page. The profile is then compared against the learned benign profiles. Again, if the deviation is above a threshold, the user is notified.

Additionally, profiles can contain more abstract representations of behavior. Heap spraying, for example, might be characterized as repeatedly assigning identical contents to variables (e.g., subsequent array indices), where the accumulated memory exceeds a given threshold. If required, the assigned contents can be pattern matched for known NOP instruction sequences to lower the number of possible false positive alerts.

Emulation-Based Mitigation Technique: Drive-by attacks also make use of exploits that rely on shellcode. We envision that these threats can be mitigated by applying emulation techniques similar to approaches that are used to detect shellcodes in network streams [18]. To this end, we propose an approach where the memory contents that contain data retrieved from a web site are examined for the longest valid instruction sequence of machine instructions [7]. As the x86 instruction set is densely packed, almost all contents will contain executable in-

structions. To lower the number of false positives, we propose to introduce a threshold where only valid *instruction sequences* that exceed this value raise an alert. An overly conservative threshold value will rise the number of false positives while a too loose value will open the possibilities for attacks to go unnoticed. As the NOP sledges created by heap spraying attacks must also contain executable code, this approach would also detect attempts to spray the heap.

Performance Considerations: One challenge when dealing with integrating a protection mechanism into browsers is that such solutions need to be efficient and performant. Browser vendors invest time and money to speed up their respective JavaScript interpreters. Thus, any protection mechanisms that exhaustively reduce performance will most likely not be applied by these vendors. Performance has especially become important as more and more sites make extensive use of Web 2.0 technology (i.e., increased usage of client-side scripting). We, therefore, propose to give the user the possibility to deactivate the protection facilities for certain trusted pages. Thus, the user would benefit from the additional protection on unknown sites and experience the high performance on trusted sites.

4.2 Analysis Challenges

Developing efficient protection mechanisms against drive-by attacks requires effective analysis tools that allow to study current and future attacks. For example, to analyze a malicious script that is encrypted with its origin URL, an analysis tool needs to keep track of the URLs where the content was fetched from to allow later off-line analysis. Furthermore, many malicious sites attack a client only once in a given time-frame. To hamper analysis, these sites provide the malicious content only on the first visit of a client. All subsequent requests are redirected to harmless sites, such as search engine portals. Thus, an analysis tool needs to provide mechanisms to record and replay all requests and responses involved in a detected attack.

5 Related Work

Previous work in the area of drive-by downloads focused on measurements and identification of web sites that distribute malicious content. Provos et al. [21] investigated the mechanisms employed by web-based malware to mount their attacks. In [20], the same authors performed a measurement to determine the prevalence of malicious sites on the Internet. They came to the conclusion that the results for more than 1.3% of all Google queries contain pages that perform

drive-by download attacks. Their approach classifies web sites as malicious if visiting that site with a web browser results in the creation of additional processes.

Wang et al. [26] designed and implemented Honeymonkeys, a system that visits malicious web sites with browsers of different patch levels. The gained information allows them to classify the performed attacks and if a successful attack against a fully patched system is detected, a so-called zero-day attack is identified.

Frei et al. [10] performed measurements observing that only 60% of Google users use the latest, most secure version of their web browser. They conclude that the browser is a valuable target for attackers.

The effects of an infection with web based malware on a computer is studied by Polychronakis et al. in [19]. An empirical study of spyware that infects users via drive-by attacks was conducted by Barwinski et al. [2]. Moshchuck et al. [16] detected drive-by attacks by observing the effects the visit of a web page has on system resources such as file system, or registry. They do not, however, present a solution that would be integrated into the end-user's browser to protect her against drive-by attacks.

6 Conclusion

As the web browser advances to an integral component in every day computing, it becomes an attractive target for attackers. Outdated browsers and plug-ins allow attackers to infect computer systems via drive-by attacks that download and install malicious software upon the mere visit of a web page containing malicious content.

Mitigation approaches that are implemented in network components, such as routers or proxies, are easily evaded by obfuscated or encrypted content. White- or blacklisting techniques (as implemented by search engines) inherently suffer from non up-to-date security information.

As drive-by attacks target the browser and its components directly, we propose to have defensive mechanisms built into the browser itself to mitigate the threats that arise from drive-by download attacks. Therefore, the paper proposes and elaborates on different mitigation strategies that bare the potential to protect users from these threats. Implementing countermeasures in the browser, to some extent, also allows for the protection of otherwise vulnerable components. Therefore, users would benefit directly and immediately from the security enhancements that browser built-in protection mechanisms would provide.

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. Flash player update available to address security vulnerabilities. http://www.adobe.com/support/security/bulletins/apsb09-01.html.
2. M. Barwinski, C. Irvine, and T. Levin. Empirical study of drive-by-download spyware. http://cisr.nps.navy.mil/downloads/06paper_spyware.pdf, 2006.
3. Superbuddy activex control vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5820, 2006.
4. Buffer overflow in apple quicktime 7.1.3. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0015, 2007.
5. Dan Goodin (The Register). SQL injection taints BusinessWeek.com. http://www.theregister.co.uk/2008/09/16/businessweek_hacked/. Last accessed, December 2008.
6. M. Daniel, J. Honoroff, and C. Miller. Engineering Heap Overflow Exploits with JavaScript. In *2nd USENIX Workshop on Offensive Technologies (WOOT08)*, 2008.
7. M. Egele, E. Kirda, and C. Kruegel. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment, 6th International Conference, DIMVA 2009 (to appear)*, 2009.
8. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song. Dynamic spyware analysis. In *USENIX Annual Technical Conference*, pages 233–246, 2007.
9. M. Egele, M. Szydlowski, E. Kirda, and C. Kruegel. Using static program analysis to aid intrusion detection. In *DIMVA*, pages 17–36, 2006.
10. S. Frei, T. Dübendorfer, G. Ollmann, and M. May. Understanding the web browser threat. Technical Report 288, ETH Zurich, 06 2008. 2008.
11. 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.
12. E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *USENIX Security*, 2006.
13. Exploit Prevention Labs: LinkScanner. http://linkscanner.explabs.com/linkscanner/default.aspx.
14. Microsoft Office Snapshot Viewer ActiveX vulnerability. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2463, 2008. Last accessed, March 2009.
15. Microsoft Corporation. Microsoft Security Bulletin MS06-014 - Vulnerability in the Microsoft Data Access Components (MDAC) Function Could Allow Code Execution. http://www.microsoft.com/technet/security/Bulletin/MS06-014.mspx, 2006. Last accessed, December 2008.
16. A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*, 2006.

17. V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31, 1999.
18. 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.
19. 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.
20. N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *USENIX Security Symposium*, 2008.
21. 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.
22. W. K. Robertson, G. Vigna, C. Krügel, and R. A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*, 2006.
23. M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *13th Systems Administration Conference (LISA)*, 1999.
24. Sina dloader class activex control 'donwloadandinstall' method arbitrary file download vulnerability. `http://www.securityfocus.com/bid/30223/info`.
25. A. Sotirov. Heap Feng Shui in JavaScript. `http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html`. Last accessed, November 2008.
26. Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *NDSS*, 2006.
27. C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.
28. 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.