PAPER Special Section on Information and Communication System Security—Against Cyberattacks—

Client Honeypot Multiplication with High Performance and Precise Detection*

Mitsuaki AKIYAMA[†a)], **Takeshi YAGI**[†], **Youki KADOBAYASHI**[††], **Takeo HARIU**[†], *and* **Suguru YAMAGUCHI**[††], *Members*

SUMMARY We investigated client honeypots for detecting and circumstantially analyzing drive-by download attacks. A client honeypot requires both improved inspection performance and in-depth analysis for inspecting and discovering malicious websites. However, OS overhead in recent client honeypot operation cannot be ignored when improving honeypot multiplication performance. We propose a client honeypot system that is a combination of multi-OS and multi-process honeypot approaches, and we implemented this system to evaluate its performance. The process sandbox mechanism, a security measure for our multi-process approach, provides a virtually isolated environment for each web browser. It prevents system alteration from a compromised browser process by I/O redirection of file/registry access. To solve the inconsistency problem of file/registry view by I/O redirection, our process sandbox mechanism enables the web browser and corresponding plug-ins to share a virtual system view. Therefore, it enables multiple processes to be run simultaneously without interference behavior of processes on a single OS. In a field trial, we confirmed that the use of our multi-process approach was three or more times faster than that of a single process, and our multi-OS approach linearly improved system performance according to the number of honeypot instances. In addition, our long-term investigation indicated that 72.3% of exploitations target browser-helper processes. If a honeypot restricts all process creation events, it cannot identify an exploitation targeting a browser-helper process. In contrast, our process sandbox mechanism permits the creation of browser-helper processes, so it can identify these types of exploitations without resulting in false negatives. Thus, our proposed system with these multiplication approaches improves performance efficiency and enables in-depth analysis on high interaction systems.

key words: *client honeypot, drive-by download, web-based malware, process sandbox, intrusion detection*

1. Introduction

Web-browser-targeted attacks called drive-by download attacks have recently become the main infection vector of malware. Security researchers must discover malicious websites hiding behind a large number of diverse websites in web space to counter these attacks. Effective methods for discovering suspicious websites have been proposed [2]–[5], and we must conduct in-depth inspection on suspicious URL candidates discovered with these methods. Moreover, exploit techniques have become complicated. In particular, an exploit obfuscation technique interferes with signature matching and makes it difficult to analyze the inside of an obfuscated code. Security researchers need an advanced tool to analyze and understand exploit techniques to counter these concealment strategies of an adversary. A network intrusion detection system has been used to monitor and detect intrusion incidents from the viewpoint of network-based host behavior. However, it is not sufficient to observe the internal situation of a victim host and comprehend detailed exploitation techniques. On the other hand, a honeypot as a decoy system functioning as a vulnerable host can be used to observe the internal situation of a victim host on an equivalent environment as an actual victim host. Thus, a honeypot is effective for grasping intrusion techniques and the behavior of compromised victim hosts in depth.

Honeypots that receive drive-by download attacks, called client honeypots, have been proposed. They must discover malicious websites in a large web space, which requires high-performance crawling. We also regard the accuracy of collected information as important for considering actual enforceable countermeasures such as filtering and take-down. There are two types of honeypots: low interaction [6]–[8], which use emulators, and high interaction [9]–[11], which use an actual system. The former emulates vulnerable hosts and simplifies detailed processing. It exhibits high performance but collects less information than that with an actual system. The latter uses an actual vulnerable host attached to a monitoring module for observing internal system behavior; therefore, the honeypot's performance is on the same level or less than that of an actual system. A low interaction system is suitable for surface analysis by high-speed crawling; however, a simplified rendering engine is an obstacle in conducting in-depth analysis of exploit techniques.

To solve the above conventional client honeypot problems of observability and performance, we propose a system that is a combination of multi-OS and multi-process honeypot multiplication approaches based on virtual isolation. We designed an advanced client honeypot as generic schema with Windows OS and implemented it based on our proposed system to confirm its feasibility. Our implemented client honeypot is based on a high interaction system in consideration of the above-mentioned advantages because such honeypots have higher detection accuracy and obtain richer information than low interaction honeypots. Our process sandbox mechanism, a security measure for our multi-process approach, provides a virtually isolated environment

Manuscript received April 28, 2014.
Manuscript revised October 2, 2014.

[†]The authors are with NTT Secure Platform Laboratories, NTT Corporation, Musashino-shi, 180–8585 Japan.
[††]The authors are with the Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.

*Previous work of this paper was presented at IEEE/IPSJ SAINT 2012 [1].

a) E-mail: akiyama.mitsuaki@lab.ntt.co.jp

DOI: 10.1587/transinf.2014ICP0002

for each web browser. In a field trial, we confirmed that the use of our multi-process approach was three or more times faster than that of a single process, and our multi-OS approach linearly improved system performance according to the number of honeypot instances.

The remainder of this paper is organized as follows. In accordance with the strategy for honeypot multiplication described in Sect. 2, we propose a client honeypot system that uses our multiplication approaches in Sect. 3. In Sect. 4, we discuss the evaluation of our client honeypot system in both an experimental environment and in actual web space. Discussion and related work are given in Sects. 5 and 6, respectively, and Sect. 7 concludes the paper.

2. Basic Strategy Toward High Performance with Precise Information Gathering

In this section, we discuss the basic strategy to improve the inspection performance of a client honeypot with precise information gathering. First, we introduce OS multiplication, which is a normal honeypot operation, to improve performance. Next, we consider process multiplication as a new operation for web-browser-based honeypots. On the basis of the discussion and the assumed attack model, we enumerate the requirements for process multiplication.

2.1 OS Multiplication

A conventional honeypot running on a high interaction system is separated by an OS boundary to prevent secondary infection from another host and limit the damage inside a compromised OS. A typical operation of a honeypot that improves its performance is using many OSs simultaneously. A virtual machine monitor (VMM) is generally used for this honeypot multiplication. A VMM provides a virtually isolated execution environment for each OS as a virtual machine (VM). The filesystem, *registry*[†], and other process on the honeypot OS are usually compromised when a specific vulnerability of a honeypot is exploited, e.g., create/delete an arbitrary entity of filesystem and registry or create/terminate an arbitrary process. Therefore, the honeypot OS should be restored to the original clean OS image after exploitation.

2.2 Process Multiplication

A web-browser-based honeypot (i.e., a client honeypot) should use many browser processes simultaneously because a client honeypot requires only web browsing functionalities. In addition, a web browser is not always busy because it asynchronously sends requests and receives replies from websites. A web browser cannot start rendering web content and remains idle when it is receiving reply web content from a website. In particular, the idle time of the web browser tends to be long when the web content is complicated and has many transactions when the round trip time of a website is long. Thus, a client honeypot can improve inspection performance efficiency by launching other browser processes when the currently running browser process is idle. Therefore, a client honeypot should simultaneously launch browser processes on the same OS to reduce OS overhead.

[†]A database that stores configuration information of Windows system.

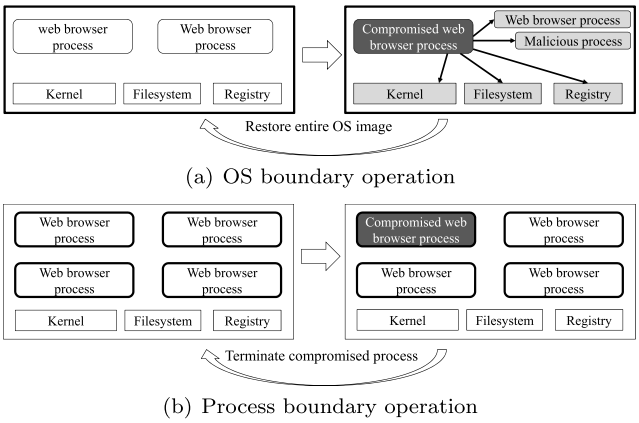


Fig. 1 OS and process boundary operation.

2.3 Process Boundary Operation for Process Multiplication

Process multiplication can resolve the above-mentioned OS overhead problem in a client honeypot. We should consider how a client honeypot can provide a process isolation mechanism for preventing interference by a compromised process because a compromised process can negatively affect other processes or can target the entire system. For example, a compromised process usually installs a rootkit to invade the kernel layer of a target system, terminates other processes to interfere with monitoring systems (e.g., anti-virus and honeypot systems), compromises other processes running on the same OS by using code injection API, or indirectly injects malicious code by replacing system dynamic link libraries (DLLs).

It is difficult to precisely determine which browser process is exploited from the simultaneously running processes when many browser processes run without a process isolation mechanism on the OS. Therefore, a critical issue is how to separate the processing boundary for honeypots to achieve process multiplication. A comparison between OS boundary operation and process boundary operation is shown in Fig. 1.

A mechanism of process isolation has been proposed that involves redirecting the input/output (I/O) on a high interaction client honeypot [11]. A file created by the web browser is redirected to a temporarily provided disk space, but a web browser can transparently access the corresponding file. When a compromised web browser attempts to create a file and execute it (i.e., a malware file), the I/O redirection mechanism can prevent the original files from being

altered, and the client honeypot can detect a newly created file as a malware file. Moreover, the client honeypot does not have to restore the OS image.

2.4 Attack Model

Drive-by downloads [12] target vulnerabilities of client applications related to web browsing and force the victim host to download/install malware executables. Conventional drive-by downloads target vulnerabilities contained only in the main components of a web browser (e.g., HTML parser, JavaScript engine). However, as the technology of the web and of developed plug-in applications evolves, these attacks also target vulnerabilities of plug-in applications. An exploit-kit is a toolkit for constructing malicious websites that conduct drive-by downloads. Known exploit-kits contain exploit codes targeting various applications [13], [14]. Since plug-in applications (e.g., Flash, Acrobat and JRE) can be installed in various web browsers, drive-by downloads target both browser-specific and plug-in vulnerabilities.

There are two types of plug-ins: those running inside and outside a browser. The former is loaded as a rendering engine by a web browser when launching the web browser or receiving specific web content. For example, rendering engines of Flash and QuickTime are loaded by a web browser into its process memory. If a loaded rendering engine has a vulnerability, a browser process is at risk of being compromised. The latter runs outside a web browser. When a web browser receives specific web content (e.g., a PDF file), it creates a browser-helper process and delegates rendering. If a rendering engine of the browser-helper process has a vulnerability, the browser-helper process is at risk of being compromised.

2.5 Requirements for Process Multiplication

The objective of this study was to analyze process behaviors by per-process sandboxing in order to reduce OS overhead. There are three requirements for improving performance of a client honeypot with precise information gathering according to the above-mentioned typical honeypot operation and an attack model.

1. Virtual isolation of process execution
   A vulnerable process, which runs a target victim application, should run independently of other processes and be accurately compromised. After being compromised, it should be prevented from negatively affecting other processes or the OS.

2. Adaptive process creation control
   (A) Restrict process behavior after exploitation
   A hijacked process becomes a stepping-stone in compromising an entire target system or intruding into the kernel layer. Thus, a honeypot should restrict process behavior to some extent; otherwise, the system will be completely hijacked. Therefore, we investigated how to prevent the hijacking of our system by malware.

   (B) Enable rendering delegation
   A web browser delegates the rendering of certain web content to *browser-helper processes* such as the plug-in process. If the process sandbox restricts process creation, web content rendering is stopped and cannot be completed to inspect an exploitation. The result of over-restriction of process creation is that an exploitation will not succeed on the honeypot system. Therefore, the process sandbox should permit the browser-helper process and inject sandbox functions into it.

3. Consistency of virtual system view between related processes
   When sandboxing each process, the filesystem and registry views are different for each process. In cross process rendering, view inconsistency occurs. For example, the browser-helper process cannot read a file for a plug-in downloaded by the browser process without file view consistency. Therefore, related processes should share common file and registry views.

According to the above requirements, we designed and implemented an original client honeypot.

3. Design and Implementation

We describe the basic design and implementation of our client honeypot system according to the following strategy; 1) OS multiplication: using a single-manager multi-agent for launching autonomous honeypot-agents and 2) process multiplication: process isolation and interprocess cooperation for running web browsers simultaneously on the same environment. First, we introduce the basic building blocks of a typical client honeypot, next we give an overview and explain the procedure of our developed client honeypot. Then, we explain our proposed system for satisfying the requirements mentioned in Sect. 2.5. Finally, we describe multi-process creation/termination and dynamic timeout control procedures.

3.1 Basic Building Blocks of Client Honeypot

The principal functionalities of a client honeypot are URL collecting, browsing, and attack detecting. We have already proposed these functionalities [5], [15]. We re-introduce their abstraction below. For URL collecting, many researchers use commercial search engines and public blacklists. In addition, for effectively discovering malicious URLs in a large web space, we proposed a method for structurally discovering potentially malicious URLs among neighboring known malicious URLs [5].

Attack detectors and a DOM inspector are also important basic building blocks of our client honeypot system. The main purposes of DOM inspection are extracting linkage URLs and tracking malware distribution networks. The

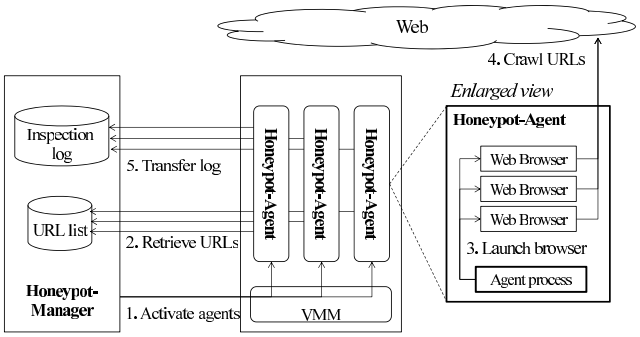


Fig. 2 Multi-OS and multi-process approaches.

DOM inspector obtains the object pointer of the DOM via the web browser control interface and extracts attributes of HTML tags from the DOM tree. In many cases, JavaScript often interacts with the DOM after rendering using Dynamic HTML for providing rich content. In the same way, malicious obfuscated JavaScript also uses Dynamic HTML for hiding the redirect-destination (i.e., URL strings). Our proposed client honeypot system parses the DOM to extract URLs set into attributions of linkage HTML tags, such as `a`, `script`, `iframe`, and `meta`, after completing web content mapping into the DOM. Therefore, we can extract redirection relationships, even if they are obfuscated. We determined that an automatically accessed URL is a result of using a kind of redirection method (e.g., `iframe` tag) when a *BeforeNavigate*, a callback event via the IWebBrowser2 interface before the browser accesses a URL, occurs.

To improve detection coverage, our client honeypot system performs stepwise detection focusing on the exploit phases; *preparation phase*, *exploitation phase*, and *post-exploitation phase*. Our system is equipped with the following detectors; scripting engine behavior anomaly (e.g., heap spraying detection), dataflow violation toward vulnerable functions (e.g., memory corruption detection), and file/registry/process event anomaly respective to the above phases.

3.2 Toward High Performance

We use both process and OS multiplication approaches; multi-OS contributes to achieving high scalability, and multi-process contributes to reducing OS overhead with high performance. Our designed honeypot system is composed of the *honeypot-manager* and *honeypot-agents*. The overview and workflow are illustrated in Fig. 2. The honeypot-manager simultaneously controls honeypot-agents. A honeypot-agent is VM as honeypot instance for URL inspection. The honeypot-manager first activates the honeypot-agent as the initialization procedure of honeypot instance. On an activated honeypot-agent, an *agent process* launches numerous web browsers. After activation, the honeypot-agent automatically refers to the honeypot-manager's database to obtain the seed URLs and dispatches them to web browsers for inspection. The web browsers then start inspecting them. The honeypot-agent reports the results to the honeypot-manager and retrieves the next seed URLs from the honeypot-manager for inspection again when it finishes inspecting the current seed URLs. The seed URL list and its status are stored in the honeypot-manager. When the honeypot-agent retrieves some URLs from the honeypot-manager, it changes the URL status to *fetched*. When the honeypot-agent finishes inspecting a fetched URL, it changes the URL status to *finished*. When a web browser finishes inspecting the input URL, the honeypot-agent sends inspection logs to the honeypot-manager. A bottleneck occurs when a single honeypot-manager controls many honeypot-agents simultaneously, so communication concentrates on the honeypot-manager side. Thus, the honeypot-manager can control honeypot-agents until it reaches that peak.

3.3 Process Sandbox for Process Multiplication

File/registry system alteration caused by a hijacked process seriously affects other processes on the same OS. To prevent direct/indirect interference between hijacked and normal processes, a honeypot should provide a virtual isolation environment for each process. We try to sandbox each process (process sandbox) by using both stealthy API hooking and filesystem/registry I/O redirection. As mentioned in Sect. 2.4, there are also exploitations targeting the browser-helper process. For detecting these types of exploitations we should enable cooperative behavior between related processes such as rendering delegation. We explain the basic mechanisms of API hooking and I/O redirection in this subsection for achieving virtual isolation of process execution. We then describe process creation control and sandbox propagation for restricting behavior after exploitation and enabling cooperative behavior between related processes.

3.3.1 Stealthy API Hooking

Event monitoring in the kernel layer can basically monitor all events. However, due to capturing all system call events of both related and unrelated processes, event monitoring in the kernel layer has a large amount of overhead. Therefore, we consider API hooking per target process. Generally, API hooking in user-land is easily detectable by malicious codes. Therefore, we must consider stealthy-hooking APIs.

To control file/registry access, we used API hooking for Win32 APIs. API hooking is used to intercept a target API procedure and alter it. When a process uses a specific API, it loads a DLL and calls an API contained in the DLL. The general API hooking strategy injects a jump code into the head instruction of the target API for altering the API procedure and jumping to a hook function. The hook function generally logs arguments of the hooked API and conducts other procedures. It then returns an instruction pointer to the original API.

Detours [16] is the most standard API hook using the `jmp` instruction. It hooks by overwriting the first six bytes

of a target function with a `jmp` instruction to a hook function. The overwriting code and hook function are loaded into the target process using any code injection method (e.g., DLL injection using `CreateRemoteThread` API). Some exploit codes attempt to determine if the target APIs are hooked by using security tools. If a hooked API is exposed by an exploit code, the exploit code stops running or attempts to prevent API hooking. We confirmed the above exploit code containing hook-prevention functionality in the wild.

```
; eax is stored address of target function
cmp byte ptr [eax], 0E9h ; hook check (jmp)
jnz short LABEL
cmp dward ptr [eax+5], 90909090h ; thunk check
jz short LABEL

; make prolog and skip first instruction
push ebp
mv ebp, esp
lea eax, [eax+5]

LABEL:
jmp eax
```

Before calling the target API, this code determines whether the first instruction of the target API is `jmp`. The code then determines that the API is hooked and skips the first instruction to prevent hooking, except that this code determines `jmp` + `nop` as *thunk*. API hook prevention causes a breakout of process sandboxing and enables the exploit code to hijack the target system. Therefore, to counter API hook prevention, we should develop a function to make it difficult to determine whether the head instruction of the target API is replaced with the jump instruction pointing toward the hook function. We developed a stealthy API hooking procedure using a combination of *adjustment* and *conditional-jump* instructions. The conditional-jump instruction jumps to an arbitrary address when specific values of the EFLAGS register satisfy the condition indicated by the type of conditional-jump instruction. The EFLAGS register stores the current state of the processer. For example, four flags; carry flag (`CF`), zero flag (`ZF`), sign flag (`SF`), and overflow flag (`OF`), are set to `0` or `1` according to the arithmetic results of the instruction. An example hook instruction is when `jz` enables the instruction pointer to jump an arbitrary address set in the operand of `jz` instruction if `ZF` is set to `0`. Then, when the typical instruction sequence example,

```
cmp eax, eax
jz hook-func-addr
```

is executed, an instruction pointer can jump an arbitrary address (i.e., `hook-func-addr`) because `ZF` is set to `0` and a condition of `jz` instruction is always satisfied. We can generate various combinations of adjustment and conditional-jump instructions because there are various kinds of flag registers and conditional-jump instructions. Therefore, we can generate numerous instruction patterns for our stealthy API hooking procedure. From the viewpoint of adversaries, however, it is difficult to recognize whether the target API is hooked before it executes the head instruction, i.e., adjustment and conditional-jump instruction sequence, because it requires processor emulation for detecting the register state, which is the conditional-jump instruction before it executes them. Due to the difficulty in creating a specific signature pattern, an exploit code cannot detect this stealthy API hook.

3.3.2 Filesystem and Registry I/O Redirection

To prevent internal alteration in a system, we developed an isolation mechanism in which a system virtually accesses resources in each process. The alteration actions affecting system behavior are filesystem, registry-system, and process-access events. We use an I/O redirection mechanism for file and registry access in each process. Each process can transparently access target file/registry entities. The process sandbox mechanism provides a virtual filesystem (VFS) for each process. A VFS conducts I/O redirection and stores the correspondence relationship between the actual target file path and redirected file path into a lookup table, called a VFS table. A VFS table has three tuples (*real file path*, *virtual file path*, and *state*). The real file path is a target file path actually input in the API argument. The virtual file path is a redirected file path, which is called a random string such as a universal unique identifier (UUID) string, and a file entity is actually in this file path. *State* denotes the accessibility of file entities. If a web browser calls the `DeleteFile` API to delete *RealFilePath_A*, the I/O redirector sets the *delete* flag to the corresponding VFS entry, and the web browser cannot look this entry up afterwards. A process executing a specific API cannot recognize the I/O redirection due to the transparent execution of I/O redirection. Because that exploit code cannot detect the actual redirected VFS path and I/O redirection is transparent, the exploit code cannot recognize the I/O redirection. Even if the exploit code attempts to alter the target system, I/O redirection can suppress filesystem and registry alteration on a specific process and prevent it in other processes. Figure 3 shows the I/O redirection procedure. A benign web browser usually accesses cache directories of the browser; thus, the VFS should exclude them.

Registry is a database that stores system configuration information such as profiles for each computer user and information about system hardware, installed programs, and property settings, and continually references this information by using the OS or applications. Therefore, we should also adapt I/O redirection to the registry system because registry system alternation seriously affects the system environment. We confirmed that the following procedure alteration fatally affects the system; create a shortcut file of an arbitrary program in the *StartUp* directory, register an arbitrary process to the *RunKey*, and set an arbitrary URL (it is often a malicious website) into the *StartPage* of the browser. We also designed a virtual registry system (VRS) by using I/O redirection. When *create*, *read*, or *delete* registry key events occur, the I/O redirector looks up a VRS table and redirects registry I/O in the same way as the VFS

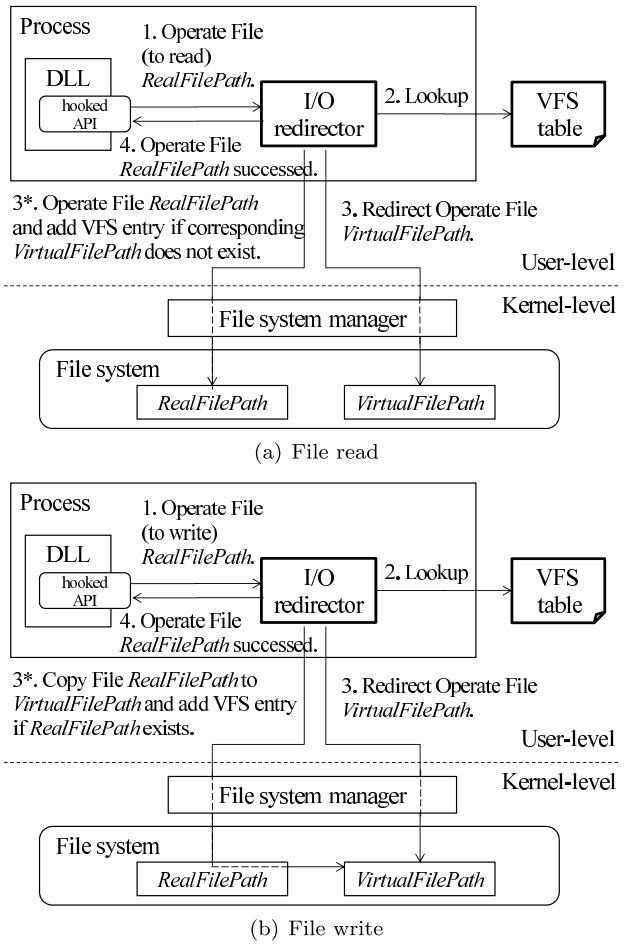


Fig. 3 I/O redirection procedure.

procedure. Thus, the VRS also should exclude registry access events in a benign browser setup procedure. The VFS and VRS should manage entity states. The I/O redirector ensures consistency of the state transition of file and registry entities. We give concrete examples of VFS entry changing in *move*/*copy*/*delete* events. When a *move* event moves *RealFilePath_A* to *RealFilePath_B*, the VFS entry (*RealFilePath_A*, *VirtualFilePath_A*) is changed to (*RealFilePath_B*, *VirtualFilePath_A*). When a *copy* event copies *RealFilePath_A* to *RealFilePath_B*, an additional VFS entry (*RealFilePath_B*, *VirtualFilePath_A*) is created. When a *delete* event deletes *RealFilePath_A*, a VFS entry (*RealFilePath_A*, *VirtualFilePath_A*) sets the "*delete*" flag and cannot be looked up, and the entity of *VirtualFilePath_A* is not actually deleted.

An entity of a created file is actually in a special working directory for the VFS, which also includes malware executables. This entity cannot be deleted and is saved to inspection logs. Once a file is created, it cannot be deleted, even if a *delete* file event occurs, because the VFS entry sets the "*delete*" flag as inaccessible to the target process.

3.3.3 Process Creation Control

When exploitation is successful, an exploit code attempts to execute malware executables on the compromised target system. To make matters worse, the system allows malware to intrude into the kernel layer when permitting arbitrary process creation. Therefore, the process sandbox should monitor an API that is able to create a process in order to restrict behavior affecting other processes (e.g., process termination and code injection). Some malware executables function as a *downloader*, which has only download functionality, and download main malware components from the Internet. Consequently, if the process sandbox restricts the creation of a malware process, the honeypot system cannot obtain the main malware components. However, we can solve the latter problem by using malware sandbox systems that have permeable internet accessibility [17]. These systems retrieve and analyze secondary executables.

3.3.4 Sandbox Propagation

General client honeypot implementation only monitors file/registry/process events; it does not restrict them. Consequently, malware can completely hijack a honeypot system. A honeypot system cleans the VM image of a honeypot and rolls it back to the primary VM image if it is compromised by malware. In other words, VM-rollback overhead cannot be prevented. Moreover, the risk of a compromised system attacking other systems until VM rollback is a serious limitation of high interaction honeypots.

A Web browser delegates plug-in applications for rendering specific web content. There are two types of rendering delegations: *in-browser* processing and *out-browser* processing. `Flash.ocx`, which is a `Flash` plug-in loaded in the browser process, renders `Flash` content inside the browser process. On the other hand, `AcroRd32.dll`, which is an `Acrobat` plug-in loaded in the browser process, launches new browser-helper processes such as `AcroRd32.exe` for rendering a PDF file. In the same way, `javaw.exe`, which renders JAR files, is launched by a browser-helper object of `JRE`. Many exploit codes target out-process rendering engines such as `Acrobat` and `JRE`. Therefore, the process sandbox should permit the launching of a specific rendering process to execute seamless processing of web content. When process creation occurs, the *process restrictor* determines whether to create or restrict the process according to a process restriction table. The process sandbox injects sandbox functionalities (i.e., I/O redirector and process restrictor) to related browser-helper processes when it is launched. Additionally, a browser-helper process always continues to run after delegated rendering of web content. Therefore, a parent process, which is a browser process, terminates a child process, which is also a browser-helper process, after rendering of web content. The above process creation control and sandbox propagation mechanism is shown in Fig. 4.

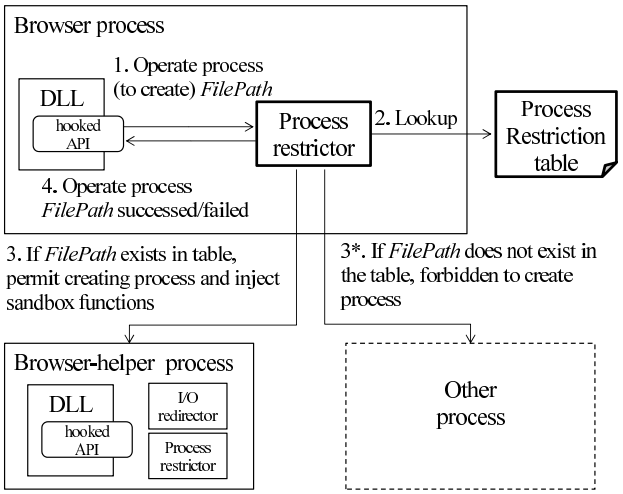


Fig. 4 Process creation control and sandbox propagation.

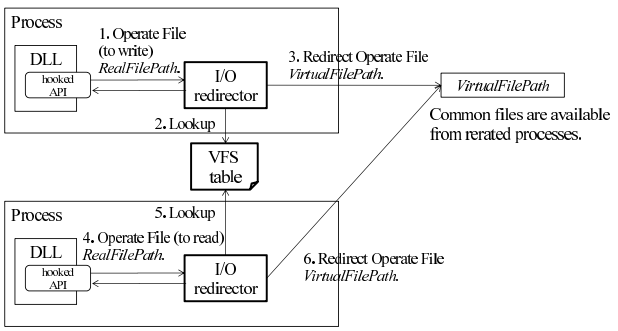


Fig. 5 Sharing virtual filesystem.

Algorithm 1 Launch browser control procedure

Number of processes limit $P^{limit} \Leftarrow LimitNumProcValue$;
Interval time $T^{interval} \Leftarrow IntervalTimeValue$;

while
 $P^{current} \Leftarrow$ number of running browser processes;
 if $P^{current} < P^{limit}$ **then**
  if system is not overloaded **then**
   launch new browser process;
  end if
 end if
sleep ($T^{interval}$);
end while

Algorithm 2 Dynamic timeout control procedure

Default *WatchDog* timeout $T^{timeout} \Leftarrow DefaltTimeOutValue$;
Max. timeout $T^{max} \Leftarrow MaxTimeOutValue$;
Additional time $T^{add} \Leftarrow AdditionalTimeValue$;
Elapsed time from launching browser T^{elapse};

DynamicTimeoutControl()
#*DocumentComplete* and *WatchDog* call this function
 while $T^{elapse} < T^{max}$
 if $T^{elapse} < T^{timeout}$ **then**
  if no established HTTP session **then** finish;
  end if
 else if established HTTP sessions exist **then**
  $T^{timeout} = T^{timeout} + T^{add}$
 else finish inspecting;
 end if
 end while timeout and finish inspecting;
return;

3.3.5 Sharing Virtual System View

Related processes can refer to the same files using the same VFS table. A child process is launched and its behaviors are controlled in the same sandbox space as the browser process (parent process). The above-mentioned sandbox propagation, therefore, enables the sharing of the same file/registry system view as the parent process and its child process by referring to common VFS/VRS tables (Fig. 5). For sharing VFS/VRS, a parent process (i.e., browser process) notifies a child process (i.e., browser-helper process) of the shared memory address of the VFS/VRS tables when it is launched. Our client honeypot system determines that the web browser process and its child process are the same crawling unit.

3.4 Multi-Process Launch/Termination Control

We describe the *launch browser control* and *dynamic timeout control* procedures. The web browser process is periodically launched to inspect URLs; in other words, web browsers sequentially access inspection candidate URLs. The web browser process terminates when the browser finishes inspection, and a honeypot-agent asynchronously launches another browser process to inspect the next inspection candidate URL. Unlimited launching of processes consumes a large amount of memory and processor resources, which destabilizes a system. Therefore, we set a limit to the number of running processes and overload conditions such as memory usage, number of TCP sessions, and Disk I/O. A honeypot-agent launches new browser process when the number of current process ($P^{current}$) is below the number of processes limit and the system is not overloaded. This process launch procedure is repeatedly conducted at constant intervals ($T^{interval}$).

In the dynamic timeout control procedure, we set a default timeout value and dynamically extend it according to the communication situation between browser and website in order to completely collect web content without interrupting communication. When the elapsed time is over the default timeout value and there are communication sessions, timeout is extended. If the elapsed time is over the maximum timeout, the browser is terminated regardless of continuing sessions. This dynamical timeout setup enables a web browser to completely download and inspect web content. To understand a browser's internal state, our implementation uses *IWebBrowser2* [18], a common web browser control interface for Internet Explorer. The *DocumentComplete* event notifies the DOM of received web-content-mapping completion, in other words, rendering of web content is finished, except for event-driven actions. *WatchDog*

Table 1 Hooked APIs for I/O redirection and process restriction.

Category	functionality	DLL	API example
File	operate file	kernel32.dll	CreateFile(A|W), MoveFileWithProgressW, CopyFileExW, DeleteFile(A|W)
	find file	kernel32.dll	FindFirstFileExW, GetFileAttributes(W|ExW)
Registry	operate registry	advapi32.dll	RegCreateKeyEx(A|W), RegOpenKeyEx(A|W), RegSetValueEx(A|W), RegDeleteKey
		ntdll.dll	ZwCreateKey, ZwOpenKey
Process	launch process	kernel32.dll	WinExec, CreateProcess(A|W)
		ntdll.dll	ZwCreateProcess(|Ex)
		shell32.dll	ShellExecuteExW
	terminate process	kernel32.dll	ExitProcess
		ntdll.dll	ZwTerminateProcess
	inject code	kernel32.dll	CreateRemoteThread

is an interruption timer that triggers certain corrective actions for the target program. These events simultaneously and asynchronously occur.

3.5 Implementation

Our implementation of our client honeypot system is based on Internet Explorer (IE) 6 and a Windows XP SP2 platform. Additionally, vulnerable versions of plug-in applications (e.g., Adobe, Flash, JRE, WinZip, and QuickTime) were installed on the system. The web browser and OS versions include various exploitable vulnerabilities, almost all of which can be attacked by many exploit packs, so they are suitable as the basis of a honeypot system. We implemented our system with a specific type of web browser. However, our system is applicable to various types of browsers because IE 7 and later versions and other browsers such as Firefox provide a browser control interface.

We indicate that the hooking APIs listed in Table 1 enable our proposed client honeypot system to effectively monitor and control the target process, e.g., web browser or browser-helper process. We confirmed that hooked APIs performed as expected. Functionalities that should be hooked are *file operation*, *file finding*, *registry operation*, *process creation*, *process termination*, and *code injection*. To implement our process sandbox, we confirmed that it was accurate and consistent in a preliminary investigation on web space and malicious websites.

We implemented honeypot-manager and honeypot-agent programs by mainly using C++, except API hook functionality, and an inline assembler. We used a DELL PowerEdge1955 Xeon 2.66 GHz with 4 core processors and 8-GB memory. Each honeypot-agent VM was assigned 1 core processor and 2 GB of memory.

4. Evaluation

Our system combines two approaches, multi-OS and multi-process, to improve our system's performance. We evaluated our system from the viewpoints of micro and macro benchmarks. In addition, we surveyed recent actual exploitation patterns detected with our system.

A client honeypot is generally used for understanding recent exploitation techniques, discovering unknown malicious websites for building blacklists, and conducting health checks for benign websites. Regarding the above utilities, we used two types of URL lists for our evaluation; public blacklist of URLs and popular site URLs (i.e., the latest lists of malwaredomainlist.com [19] and Alexa top sites [20] were on July 30th and September 29th, 2011, respectively). A public blacklist potentially includes malicious URLs; however, many malicious URLs are unstable and have already vanished, and the percentage of active URLs is only 18% (Table 2). On the other hand, almost all benign websites are stably accessible.

Table 2 URL status trend.

URL category	DNS error or connect fail (%)	HTTP error (40x or 50x) (%)	HTTP success (%)
Blacklisted websites	67.8	14.1	18.0
Benign websites	2.2	0.5	97.2

The results of the public blacklist and benign websites were obtained on September 1, 2011 and October 2, 2011, respectively.

4.1 Performance

4.1.1 API Hooking and I/O Redirection Overhead

We evaluated the overhead of API hooking and I/O redirection by using 5,000 benign websites and 2,699 exploit samples obtained from periodical inspections of a public blacklist for almost four months (2011.08.07 - 2011.11.26). The VFS entries were created in only 10.8% of the benign websites. In these cases, VFS entries corresponded to access events of plug-in applications, such as Acrobat, Flash, and JRE. For example, when rendering flash content, a browser-helper object inside the browser accesses the plug-in's working directory (i.e., plug-in's content cache and configuration files). During the rest of the inspections, file access events are only of the default cache directory of the web browser. VFS entries were also created in only 5.7% of the blacklist websites. On the contrary, 96.9% (2,618/2,699) of the detected inspection results had one or more VFS entries. VFS entries are usually created by an exploit code because such codes create files downloaded as malware. The reason no VFS entry was created during detected inspection is due to failure to exploit the target system or download malware executables. The entry numbers of the VFS table are shown in Fig. 6. As mentioned above, created VFS entries were

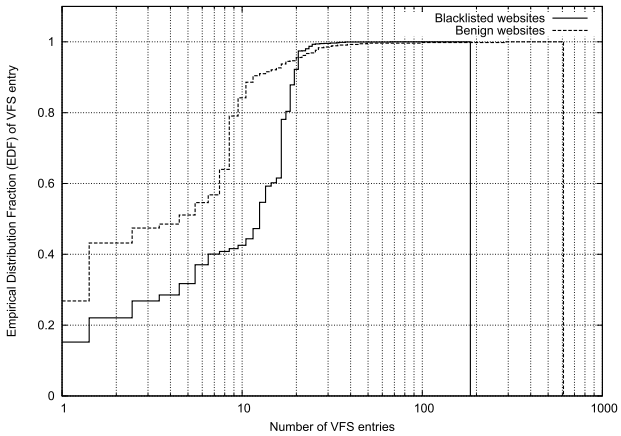
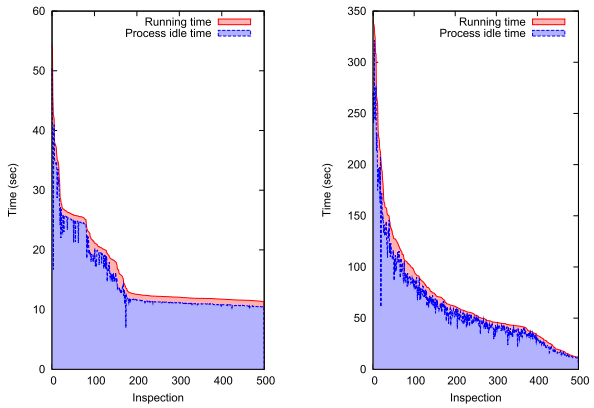


Fig. 6 Distribution of VFS entry numbers.



(a) Blacklisted websites (b) Benign websites

Fig. 7 Idle times of browser process. Inspection completion times defined as summations of running and idle time are arranged from highest to lowest. We randomly picked 500 inspection samples.

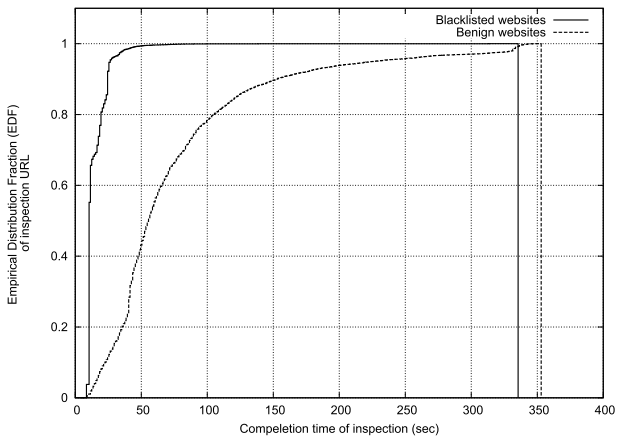


Fig. 8 Inspection completion time of each URL. T^{max} was set to 360 seconds.

at most only about 10%; moreover, VFS during the inspections of benign and blacklist websites had fewer than 20 entries in 94.6% and 92.2% of the results, respectively. Even if VFS lookup occurs, almost all VFSs contain fewer than ten entries. In addition, there is no increase in I/O redirection overhead in proportion to file size because the I/O redirector only replaces the destination file path with another file path. Thus, we believe that VFS lookup exhibits negligible low overhead.

4.1.2 Inspection Performance

In a client honeypot, the web browser must wait during the sending of a request and receiving a reply. On the same OS multiplexing applications, our system processes other applications while specific processes are idle. Therefore, we evaluated how many processes our system launches simultaneously and how long time inspection takes.

We discuss how much idle time the browser process requires. Process running time (T^{run}) is the difference between the process-launch and process-terminate timestamps. The amount of time it takes for the process to be executed in the kernel and user modes are represented as T^{kernel} and T^{user}, respectively. Idle time (T^{idle}), which mainly includes I/O waiting time, is represented as $T^{run}-(T^{kernel}+T^{user})$. We can conduct effective inspection when there are many processes running simultaneously. We obtained T^{kernel} and T^{user} by using the `GetProcessTimes` API when each browser process is terminated. The idle time of the browser process in actual inspections is shown in Fig. 7. The average percentages of the total inspection completion time of the public blacklist and benign websites taken up by idle time ($\frac{1}{N}\sum_{n=1}^{N}\frac{T^{idle}}{T^{run}}$) were 91.2% and 86.3%, respectively. Consequently, we characterize inspection completion time tendency such that idle times of both URL lists occupy most of the total inspection completion time.

Due to the fact that many blacklisted websites have already vanished, inspection finishes immediately after receiving a DNS error or server error. Benign websites contain various types of web content and cause many sessions to cross to other websites; therefore, inspection sometimes takes several minutes. We investigated the individual inspection completion times of benign and malicious websites (Fig. 8). The average individual inspection completion time of blacklisted websites was much lower than that of benign websites, and 90% of the inspections finished within 25 and 154 seconds, respectively. On the contrary, individual inspection completion times of benign websites were widely distributed due to the variety and complexity of the web content.

The number of simultaneously running processes is shown in Fig. 9. Many simultaneously running processes can conduct effective inspection. When inspecting a public blacklist, the number of running processes cannot reach the maximum process number and only about two or three processes run simultaneously due to short inspection completion time caused by DNS error of vanished websites. On the contrary, due to long inspection completion time caused by multiple sessions during the same inspection, the number of running process can easily reach the maximum process number during the inspections of benign websites. As

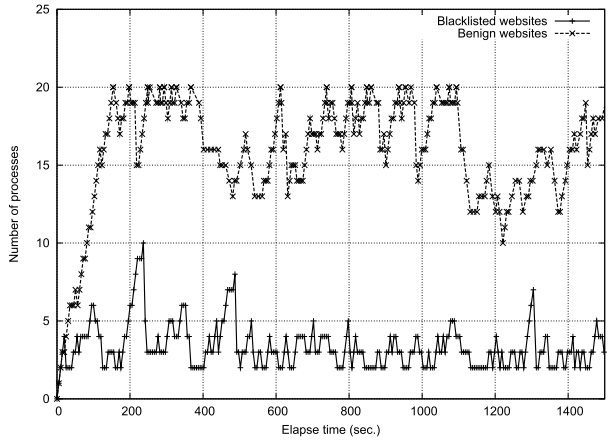


Fig. 9 Distribution of number of simultaneously running processes. P^{limit} was set to 20 processes, and $T^{interval}$ was set to 5 seconds.

mentioned in Sect. 3.4, the interval time of the launching process ($T^{interval}$) should be longer than the process setup time (T^{setup}), i.e., $T^{interval} > T^{setup}$. We found that the most efficient $T^{interval}$ was 5 seconds in our inspection environment, although this strongly depends on hardware specifications. When we set a smaller $T^{interval}$ than that of the above heuristic set, our client honeypot system exhibited an extremely high average load. Due to processor and memory resource consumption of the browser setup procedure, a system should not launch additional browser processes before the previous browser process is completely setup. Regarding the setup time of the browser process, we created an interval between the launching of the browser processes. The setup time is the process loading time and the time it takes to inject sandbox functionalities into the target process. Overlapping browser setup procedures easily causes an extremely high average load on the system. The average time of inspection completion is expressed as $\frac{1}{N}\sum_{n=1}^{N} T_n^{run}$. We estimated that the maximum logically possible number of running processes is P, satisfying the following formula: $PT^{interval} \leq \frac{1}{N}\sum_{n=1}^{N} T_n^{run}$. If the average time of individual inspection completion is less than $\frac{T^{interval}}{P}$, the system is often saturated with running processes. If it tends to be more than $\frac{T^{interval}}{P}$, P cannot reach the P^{limit}. The number of current running processes does not always reach the maximum process number when the upper limit is increased. Due to the fact that many public blacklists are DNS errors, the maximum number of processes cannot be reached. On the other hand, since sessions are successful during inspection of benign websites, multiple sessions easily occur and reach the maximum number of processes.

The total inspection completion times in a single process/multiple processes and a single OS/multiple OS are listed in Table 3. When inspecting using a single browser process, the summation of the above inspection completion time nearly equals the total inspection completion time of all the URLs on the list. When inspecting using multiple browser processes, our system can reduce total inspection completion time because it can conduct overlapped inspections. Due to short individual inspection completion time for public blacklist inspection, the number of current running processes peaked at about five. According to the number of running process, total inspection completion time peaked for five processes and was three times faster than that of a single process. On the other hand, there was a comparatively long individual inspection completion time in benign websites; there were over ten running processes. Therefore, the total inspection completion time is about five or six times faster than that of a single process depending on the number of running processes. Total inspection completion times under the multi-OS condition linearly decreased independent of the properties of both lists. By combining both multi-OS and multi-process conditions, our client honeypot performs 30 to 80 faster than that under the single-OS/single-process condition. Although these specific values depend on hardware specifications, we confirm that the performance of our client honeypot system can be improved.

Table 3 Total inspection completion times.

URL category	Honeypot-agent	Process (sec.)	
		Single process	Multi-process
Blacklisted websites	Single agent	18,565	5,123
	5 agents	3,686	1,097
	10 agents	1,982	573
Benign websites	Single agent	46,578	7,183
	5 agents	9,892	1,510
	10 agents	4,759	566

Each list includes 1,000 URLs. Blacklisted websites' URLs are the latest registered URLs picked up excluding duplication of the FQDN or IP address of URLs, and benign websites are the top 1,000 URLs from Alexa. Maximum process number is limited to 20.

4.2 Exploit Pattern

We used 2,699 exploit samples obtained during four months, similar to the experiment discussed in Sect. 4.1.1. We confirmed that the patterns, in which the browser launches a child process when they are exploited, are `Acrobat` and `JRE`, as mentioned in Sect. 3.1. When a web browser receives the MS06-001 exploit code, it launches `rundll32.exe` and loads specific vulnerable components; however, we did not observe this exploitation in our field trial. We classified the seven exploitation patterns listed in Table 4. Forty percent of exploit patterns targets a single application (i.e., pattern A, B and C), and 60% of exploit patterns (i.e., pattern D, E, F, and G) targets several application. The patterns targeting in-process (i.e., patterns A, D, E, and G) means that rendering objects inside the browser process are exploited. These rendering objects are the original browser's rendering engines and browser-helper objects such as `Flash`. In addition, 72.3% of exploit patterns (i.e., patterns B, C, D, E, F, and G) target browser-helper processes or both browser-helper processes and web browser. Moreover, 20.2% of exploit patterns only target browser-helper processes (i.e., patterns B, C, and F). Patterns excluding in-process are not successful in exploitation unless a web browser launches a browser-helper process. To increase the

Table 4 Exploit pattern distribution.

Category			Pattern	Percentage
In-browser	Out-browser			
	Acrobat	JRE		
√			A	28.0
	√		B	8.9
		√	C	3.2
√	√		D	2.1
√		√	E	20.4
	√	√	F	8.1
√	√	√	G	29.6

success rate of exploitation, many exploit codes are written to exploit multiple vulnerabilities at once. The results show that most exploitation patterns are both in-process and out-process exploitation.

5. Discussion

5.1 Sequential Exploitations and Targeted Applications

It is a limitation of high interaction systems that only an exploit code that targets a specific type of web browser expected with this honeypot implementation can be detected. On the other hand, a browser's plug-in exploitation is not affected by browser type and version because vulnerabilities of plug-ins are independent of those of a browser. As mentioned in Sect. 4.2, we confirmed that 72.0% of exploitations target both a web browser and its plug-ins. Due to these sequential exploitations, even if browser exploitation failed, a browser's plug-in exploitation will be successful and also detectable by a honeypot.

5.2 Office Document Exploitation

Vulnerability databases and many reports from security venders indicate that Microsoft `Office` applications have been targeted by recent attacks. Some of these attacks are conducted via web browsers. `SnapshotViewer` is an ActiveX object, which is an `Office` component that contains vulnerabilities (CVE-2008-2463). A web browser performs in-process rendering of this vulnerable ActiveX object. However, some exploit codes targeting office vulnerabilities require the launching of Microsoft `Word` or `Excel` due to out-process rendering. On the other hand, we confirmed that there are few URLs that have directly accessed Office document files (i.e., `.doc`, `.xls`) containing exploit codes in our blacklist inspections. This type of exploitation is out of the scope of this paper because it is usually used for mail-based target attacks.

6. Related Work

6.1 Client Honeypot

The main issue with low interaction based client honeypots [6]–[8] is how to emulate rendering engines of the browser and plug-ins. HoneyC [6] presents a basic model of a low-interaction-based client honeypot and consists of a queuer that collects inspection URLs, visitor that crawls the URLs, and analysis engine that detects an exploitation. A basic detection of low interaction is signature matching. For example, HoneyC uses a snort signature. Recent drive-by downloads have been using obfuscated malicious web content, so they can easily circumvent simple signature-based detection methods. To counter content obfuscation, PhoneyC [7] and Thug [8] attempt to emulate basic rendering engines (e.g., DOM and JavaScript) and vulnerable browser functionalities.

High-interaction-based client honeypots have also been reported [9]–[11]. HoneyMonkey [9] and Capture-HPC [10] can use many VMs to improve their inspection performance. They monitor a file/registry access and process control event to detect an intrusion in the kernel layer. Monitoring in the kernel layer does not depend on a specific browser implementation and can observe complete events and be comprehensively implemented. In addition, the current version of Capture-HPC can support multi-browser processes and discriminate which web browser is exploited based on a mapping of the state changes (e.g., file/registry accesses, process creations) to the process ID of the web browser. BLADE [11] provides file I/O redirection and safely executes browser processes without modifying the original files. The main differences between BLADE and our proposed system are described in Sects. 3.3.3, 3.3.4 and 3.3.5.

6.2 Sandbox

There have also been many studies on sandboxes for isolating running programs, while not necessarily for honeypots. Linux-VServer [21] is a *chroot*-based filesystem virtualization/isolation mechanism on Linux OS, which creates individual containers that provide many independent virtual private servers (VPS) used for web hosting services.

Tahoma [22] is a VM-based browser sandbox mechanism that uses VMs to provide sandboxes for each web browser instance. This method is based on OS boundary operation. Middlebox approaches are alternatives to the previously mentioned approaches on an end-host, for example SpyProxy [23], BrowserShield [24] and WebShield [25] are browser sandbox implementations performing as a Web-proxy. The design goal of these middlebox approaches is to block or sanitize web content transparently with low latency.

6.3 Versatility of Our Design

We designed a multi-OS and multi-process honeypot system. The implementation for OS multiplication is just controlling OSs from outside; therefore, it does not depend on OS/browser architecture. We enumerated generic requirements for process multiplication in Sect. 2.5 and created a process sandbox as a reference implementation on Windows OS. Although our implemented process sandbox controls Windows-specific APIs and other Windows/IE-specific functionalities (i.e., registry, IWebBrowser2 interface), the

basic requirements of process multiplication and the design of the process sandbox are generic and essential. Our basic design can be applicable to other web browser/OS such as FireFox on Linux at least because other architectures also have APIs and functionalities similar to Windows.

6.4 Restoration

OS boundary operation mentioned in Sect. 2.3 requires OS image restoration, e.g., restoring filesystem/registry and rebooting after that. In contrast, OS image restoration is not necessarily a procedure in our system because our process sandbox prevents the original objects (i.e., file and registry) from being altered and redirects newly created objects to VFS/VRS. In addition, deleting VFS entries is not necessarily a procedure because VFS entries are not accessible by other process and they do not affect other inspections. As we described in Sect. 4.1.1, VFS entries were created in only 5.7% – 10.8% of websites. Moreover, VFS during the inspections had fewer than 20 entries in 92.2% – 94.6% of the above results. There were few VFS entries created per inspection, although the number of VFS entries increased while the honeypot system was running.

7. Conclusion

We proposed a client honeypot system designed for achieving honeypot multiplication and implemented it in a field trial to evaluate its effectiveness. Our system uses our multi-OS and multi-process approaches. In particular, our process sandbox mechanism solved problems of process multiplication by providing a virtually isolated execution environment. In a field trial, inspection performance under the multi-OS condition linearly increases, and inspection performance under the multi-process condition is about three to six times faster than that of a single process. Consequently, our proposed client honeypot system substantially improved in performance. Our long-term observation of malicious websites indicated that 72% of exploitations target browser-helper processes. Our process sandbox enabling cooperative behavior between a browser and its helper process on our honeypot system contributes to successful exploitations and detection.

References

[1] M. Akiyama, Y. Kawakoya, and T. Hariu, “Scalable and performance-efficient client honeypot on high interaction system,” Proc. 12th IEEE/IPSJ International Symposium on Application and the Internet (SAINT2012), pp.40–50, 2012.
[2] A. Moshchuk, T. Bragin, S.D. gribble, and H.M. Levy, “A crawler-based study of spyware on the web,” 13th Annual Network and Distributed System Security Symposium (NDSS), 2006.
[3] N. Provos, P. Mavrommatis, M.A. Rajab, and F. Monrose, “All your iFRAMEs point to us,” 17th USENIX Security Symposium, pp.1–15, 2008.
[4] J.W. Stokes, R. Andersen, C. Seifert, and K. Chellapilla, “Webcop: Locating neighborhoods of malware on the web,” LEET’10: Proc. 3rd Usenix Workshop on Large-Scale Exploits and Emergent Threats, 2010.
[5] M. Akiyama, T. Yagi, and M. Itoh, “Searching structural neighborhood of malicious urls to improve blacklisting,” Proc. 11th IEEE/IPSJ International Symposium on Application and the Internet (SAINT2011), pp.1–10, 2011.
[6] C. Seifert, I. Welch, and P. Komisarczuk, “HoneyC - The low-interaction client honeypot,” NZCSRCS (Hamilton, 2007), vol.10, 2006.
[7] J. Nazario, “Phoneyc: A virtual client honeypot,” LEET’09: Proc. 3rd Usenix Workshop on Large-Scale Exploits and Emergent Threats, 2009.
[8] A. Dell’Aera, “Thug: A new low-interaction honeyclient.” http://www.honeynet.org/files/HPAW2012-Thug.pdf.
[9] 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,” 13th Annual Network and Distributed System Security Symposium (NDSS), 2006.
[10] Honeynet Project, Capture-HPC, https://projects.honeynet.org/capture-hpc
[11] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, “Blade: An attack-agnostic approach for preventing drive-by malware infection,” 17th ACM conference on Computer and communications security, 2010.
[12] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, “The ghost in the browser: Analysis of web-based malware,” Proc. 1st Workshop on Hot Topics in Understanding Botnets (HotBots2007), 2007.
[13] J. Jones, The state of web exploit kit, Blackhat USA, 2012.
[14] Contagio, “An overview of exploit packs.”
[15] M. Akiyama, K. Aoki, Y. Kawakoya, M. Iwamura, and M. Itoh, “Design and implementation of high interaction client honeypot for drive-by-download attacks,” IEICE Trans. Commun., vol.E93-B, no.5, pp.1131–1139, May 2010.
[16] G. Hunt and D. Brubacher, “Detours: Binary interception of win32 functions,” 3rd USENIX Windows NT Symposium, 1999.
[17] Anubis. http://analysis.seclab.tuwien.ac.at/
[18] MSDN, “IWebBrowser2 Interface.” http://msdn.microsoft.com/en-us/library/aa752127%28VS.85%29.aspx
[19] Malware domain List. http://malwaredomainlist.com/
[20] Alexa, “The top 100 sites on the web.” http://www.alexa.com/topsites/global
[21] H. Potzl, “Linux-vserver.” http://linux-vserver.org/
[22] R.S. Cox, S.D. Gribble, H.M. Levy, and J.G. Hansen, “A safety-oriented platform for web applications,” Proc. 2006 IEEE Symposium on Security and Privacy, SP’06, pp.350–364, 2006.
[23] A. Moshchuk, T. Bragin, D. Deville, S.D. Gribble, and H.M. Levy, “Spyproxy: Execution-based detection of malicious web content,” Proc. 16th USENIX Security Symposium on USENIX Security Symposium, SS’07, 2007.
[24] C. Reis, J. Dunagan, H.J. Wang, O. Dubrovsky, and S. Esmeir, “Browsershield: Vulnerability-driven filtering of dynamic html,” 2007.
[25] Z. Li, Y. Tang, Y. Cao, V. Rastogi, Y. Chen, B. Liu, and C. Sbisa, “Webshield: Enabling various web defense techniques without client side modifications,” Proc. Network and Distributed System Security Symposium, 2011.

Mitsuaki Akiyama received his M.E. degree and Ph.D. degree in Information Science from Nara Institute of Science and Technology, Japan in 2007 and 2013. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2007, he has been engaged in research and development of network security, especially honeypot and malware analysis. He is now with the Network Security Project of NTT Secure Platform Laboratories.

Takeshi Yagi received his B.E degree in electrical and electronic engineering and his M.E. degree in science and technology from Chiba University, Japan in 2000 and 2002. He also received his Ph.D. degree in information science and technology from Osaka University, Osaka, Japan in 2013. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2002, he has been engaged in research and design of network architecture and traffic engineering and his current research interests include network security, web security and cloud computing. He is now with the Network Security Project of NTT Secure Platform Laboratories.

Youki Kadobayashi received his Ph.D. degree in computer science from Osaka University, Osaka, Japan in 1997. From 1997 to 2000, he was with the Computation Center of Osaka University as an Assistant Professor. Since 2000, he has been an Associate Professor with the Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan. Since 2006, he has also been working as the Project Lead at the traceable network research group of the Information Security Research Center, National Institute of Information and Communications Technology (NICT). His research interests include cybersecurity, Internet architecture, and distributed systems.

Takeo Hariu received his M.S. degree in electro-communications from The University of Electro-Communications, Japan in 1991. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 1991, he has been engaged in research and development of network security. He is now a senior research engineer and supervisor in the Network Security Project of NTT Secure Platform Laboratories.

Suguru Yamaguchi received his M.E. and D.E. degrees in computer science from Osaka University in 1988 and 1991. From 1990 to 1992, he was an assistant professor in the Education Center for Information Processing, Osaka University. From 1992 to 1993, he was with the Information Technology Center, NAIST, Nara, Japan, as an associate professor. From 1993 to 2000, he was with the Graduate School of Information Science, NAIST, Nara, Japan as an associate professor. Currently, he is a professor at the Graduate School of Information Science, NAIST.