JsSandbox: A Framework for Analyzing the Behavior of Malicious JavaScript Code using Internal Function Hooking

Hyoung Chun Kim[1], Young Han Choi[1] and Dong Hoon Lee[2]*
[1] The Attached Institute of Electronics and Telecommunications Research Institute (ETRI),
PO Box 1, Yuseong Post Office, Daejeon, Republic of Korea 305-600
[email: khche, yhch@ensec.re.kr]
[2] The Graduate School of Information Security, Korea University,
Anam-dong, Sungbuk-ku, Seoul, Republic of Korea 136-701
[email: donghlee@korea.ac.kr]
*Corresponding author: Dong Hoon Lee

Received October 5, 201X; revised November 10, 201X; accepted November 20, 201X; published December 25, 201X

Abstract

Recently, many malicious users have attacked web browsers using JavaScript code that can execute dynamic actions within the browsers. By forcing the browser to execute malicious JavaScript code, the attackers can steal personal information stored in the system, allow malware program downloads in the client's system, and so on. In order to reduce damage, malicious web pages must be located prior to general users accessing the infected pages. In this paper, a novel framework (*JsSandbox*) that can monitor and analyze the behavior of malicious JavaScript code using internal function hooking (IFH) is proposed. IFH is defined as the hooking of all functions in the modules using the debug information and extracting the parameter values. The use of IFH enables the monitoring of functions that API hooking cannot. *JsSandbox* was implemented based on a debugger engine, and some features were applied to detect and analyze malicious JavaScript code: detection of obfuscation, deobfuscation of the obfuscated string, detection of URLs related to redirection, and detection of exploit codes. Then, the proposed framework was analyzed for specific features, and the results demonstrate that *JsSandbox* can be applied to the analysis of the behavior of malicious web pages.

Keywords: Malicious JavaScript code, Sandboxing

This work was supported by the IT R&D program of MKE/KEIT [KI002113, Development of Security Technology for Car-Healthcare].

DOI: 10.3837/tiis.2012.02.019

1. Introduction

The JavaScript language is a powerful tool that can execute dynamic actions such as reading a cookie file, providing dynamic content, and generating another code in a web browser. As a result of these abilities, a remote attacker can compromise a client's system by inserting malicious JavaScript code into an HTML page, allowing web browsers to read it [1][2]. Using malicious JavaScript code, an attacker can steal personal information in the system and download other malicious software into the victim's system without the consent or knowledge of the user [3].

To defend against such threats, malicious JavaScript code must be detected before the code's activity affects the system. Research to help analyze and detect malicious JavaScript code is ongoing and widespread. Normally, static or dynamic JavaScript code analysis approaches are used, but static analyses are limited, because they cannot completely encompass the various behaviors of JavaScript. For example, static methods have difficulty analyzing and detecting obfuscated JavaScript. Another drawback is that they cannot cover JavaScript's self generating feature during run time.

Dynamic analyses can analyze the various behaviors of malicious JavaScript code. These dynamic analyses perform monitoring functions and arguments of the JavaScript engine. Many research projects use SpiderMonkey, the JavaScript engine for Mozilla, to monitor the behavior of JavaScript. Since SpiderMonkey is open source software, it is possible to implement a logger at the source code level to analyze the behavior of JavaScript.

In this paper, JsSandbox is presented, a system that dynamically analyzes malicious JavaScript code using a debugging technique. Because Microsoft Internet Explorer (IE) uses *jscript.dll* as its JavaScript engine and is not open source, JsSandbox traces all functions and arguments using the debug information which is named as a symbol file in Windows systems. The method of accessing functions using the symbol files is defined as internal function hooking (IFH).

To evaluate JsSandbox, some features for detecting malicious JavaScript code are proposed and implemented in JsSandbox.

The contributions of this paper are as follows:

- A tool, JsSandbox, is proposed that can analyze and detect malicious JavaScript code using internal function hooking. Internal function hooking supplies more sophisticated analysis results than API hooking.
- Because JsSandbox is a general framework, it can be extended to analyze other malicious codes such as VBScript and Perl.
- Most research has used the JavaScript engine of Mozilla, SpiderMonkey, because it is open source. However, JsSandbox can detect threats in IE that have been propagated from malicious JavaScript code. JsSandbox was implemented in Microsoft Internet Explorer after reverse engineering the IE modules.
- Some features for detecting malicious JavaScript code are proposed and applied to JsSandbox. The evaluation demonstrates that JsSandbox can analyze and detect several malicious JavaScript codes using the proposed features.

This paper is organized as follows. In Section 2, research related to the detection and analysis of malicious JavaScript code is introduced. Next, in Section 3, internal function

hooking is described. The method to detect malicious JavaScript code is proposed in Section 4. Then, the designed and implemented JsSandbox is discussed in Section 5, and Section 6 presents the evaluation of the JsSandbox with reference to some cases. Last, the research is concluded and the future work is explained in Section 7.

2. Related Work

JsSandbox is a framework for tracing and analyzing the behavior of malicious JavaScript code in a web browser by hooking the internal functions in the JavaScript engine and HTML parser. In this section, current research that analyzes malware using sandboxes, and detects malicious JavaScript code, is described.

Sandboxing: The representative sandbox program for the analysis of malicious code is CWSandbox [4]. This program traces and analyzes the behavior of malware using API hooking such as process creation, file creation, and network communication in Windows systems, by targeting the external functions in *kernel32.dll*, *ws2_32.dll*, and so on. After hooking these APIs, CWSandbox generates an analysis report about the malicious software behaviors. TTAnalyze [5] also uses API hooking to analyze the behavior of a malware. Norman sandbox [6] analyzes the execution of malware in simulated computer system and network environments.

These sandboxes run malware in a virtualized or emulated environment, such as VMWare [7] and QEMU [8], in order to execute the malware in an isolated system. Wang et al. [9] developed a tool, HoneyMonkey, that can detect malicious web pages using a virtual machine based on the system behavior. Because it targets drive-by-downloads, it monitors the behavior of a web browser, including the process creation, file creation, and so on; this method also uses API hooking.

However, the API hooking approach is limited because it can only access the exported functions in the DLL files. Thus, API hooking is not suitable for analyzing malicious JavaScript codes in IE because many IE modules, such as *jscript.dll*, *mshtml.dll*, and ActiveX Controls, are implemented as component object model (COM) files [10].

The export functions in the COM object avoid the analysis of sophisticated malicious JavaScript code. In order to overcome this drawback, NEPTUNE [11] uses an arbitrary hooking method that can hook internal functions in the COM object. However, the arbitrary hooking requires prior tedious and labor intensive processes to determine the target address in the memory. Arbitrary hooking methods access the functions using the hard-coded absolute address of the functions in the memory. When the target module or runtime state version is changed, the tedious and labor intensive process must be performed again.

In contrast, JsSandbox can access the internal functions in the COM object, using the relative address of the functions in the memory, by applying a debugging approach with the symbol files.

JsSandbox is a system for tracing and analyzing the behavior of malicious JavaScript code in a web browser by hooking the internal functions in the JavaScript engine and HTML parser. Chenette proposed a method that hooks the internal functions in a DLL file and performed it using DLL hooking [12]. This approach is similar to the proposed system because it focuses on internal functions for analyzing the JavaScript engine. However, because this approach must know the memory address of target functions in advance when the file is loaded in the memory, it is difficult to hook the internal functions. Alternatively, because the debug information is used here, the internal functions can be accessed in a stable manner.

Malicious JavaScript Code Detection: There has been much research on the detection and analysis of malicious JavaScript code. In reference [13], the authors studied various JavaScript redirections in spam pages and found that obfuscation techniques were prevalent among the redirections. In reference [14], the author introduced various malicious JavaScript attacks and obfuscation methods. He found that the `eval` and `document.write` functions were the most used functions in malicious web pages. This research has focused on the malicious JavaScript codes themselves.

Provos et al. [15] declared that web pages were malicious if the web pages caused the automatic installation of software without the user's knowledge or consent. They found malicious web pages by dynamically monitoring the behavior of Internet Explorer in a virtual machine. In their research, the authors observed that a number of web pages in reputable sites were obfuscated, and found that the obfuscated JavaScript is not in itself a good indicator of malice.

Ikinci et al. implemented a low interaction honeyclient system, MonkeySpider, for detecting malicious web pages [16]. Feinstein et al. analyzed JavaScript obfuscation cases and implemented an obfuscation detection tool [17]. They hooked the `eval` function and string concatenation method, based on Mozilla's SpiderMonkey. Hallaraker et al. proposed a method that monitors the JavaScript code execution, to detect malicous code behavior, and evaluated a mechanism that audits the execution of JavaScript codes [18].

These methods modified the open source JavaScript engine; nonetheless they cannot be applied to Internet Explorer (IE), which is the Internet browser with the largest market share in the world. ADSandbox [19] is a tool that analyzes JavaScript code using a sandbox implemented by the Mozilla JavaScript engine, SpiderMokey, in a Microsoft Windows and IE environment. However, the tool cannot completely analyze the threat of IE functions propagated from malicious JavaScript code, because of the imperfect emulation of IE in the SpiderMonkey-based environment. Cova et al. [20] proposed various features for detecting malicious JavaScript code in an emulated environment using the HtmlUnit and demonstrated their validation.

Another approach for detecting malicious JavaScript code is to use emulators of JavaScript engines such as DecryptJS [21], Rhino [22], NJS [23], and so on. DecryptJS is a tool that allows obfuscated JavaScript code to be readable in FireFox. Rhino is an open source implementation of JavaScript written entirely in Java, and NJS is a JavaScript interpreter. These emulators are limited because they are not real JavaScript engines, whereas JsSandbox traces the real JavaScript engine of Microsoft IE.

3. Internal Function Hooking

API hooking is mostly used for analyzing the behavior of a program in a Windows system. In a Windows system, the API is a function exported in a DLL file: API hooking changes the execution flow of the program into an arbitrary flow and extracts the values of the API parameters. Most sandboxes use API hooking approaches such as the CWSandbox [4]. However, API hooking can only access exported functions in a DLL file, except for various internal functions.

For example, when monitoring a JavaScript engine, API hooking cannot trace the flow in it, because the JavaScript engine is implemented by a COM process and does not have APIs for JavaScript. For this reason, a method that can monitor various internal functions in the modules of the web browser using debugging information is proposed.

Generally, debugging information includes the function information, global or local variable information, and so on. In a Windows system, debugging information is named as a private symbol or public symbol. Microsoft provides public symbols for the Windows OS, including Internet Explorer (IE). The public symbols include only the name and starting address of the functions and global variables. For example, the JavaScript engine (*jscript.dll*) of IE in Windows Vista includes 3,623 symbols, and the symbols represent all functions implemented in the JavaScript engine.

Using the public symbol allows the JavaScript engine of Internet Explorer to be monitored and analyzed. A symbol's function is defined as an internal function (IF), and accessing the IF using a debugging mechanism is defined as internal function hooking (IFH). That is, IFH sets the breakpoint at the starting address of the target function and catches the exception for the breakpoint using the debugger engine. The WinDBG debugger (*dbgeng.dll*) is used in this method because public symbols are being used.

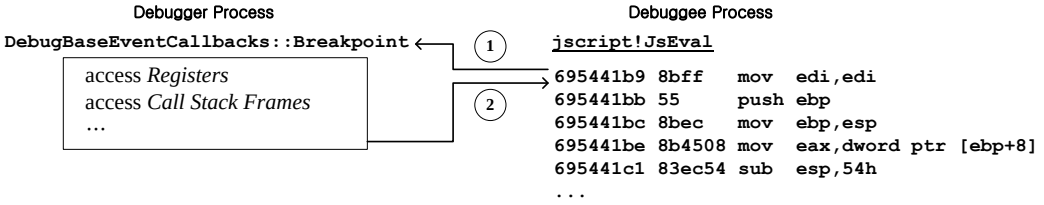

Fig. 1. Internal function hooking monitors the `eval` function of JavaScript

Fig. 1 shows an example where the IFH monitors the `eval` function of the JavaScript code. The debuggee is the target program, and the debugger is the analyzer. In the JavaScript engine (*jscript.dll*), `eval` is implemented as `JsEval`. In order to set the breakpoint at the `JsEval` function, the IFH accesses the symbol using the `Jscript!jsEval` string. This means that the `JsEval` function is in the *jscript.dll* module.

If the breakpoint is set at the function, the debugger engine substitutes the first one byte of the start address of the `JsEval` to be `0xCC`. If the flow of the program execution meets the `0xCC` byte, the program generates the exception for the breakpoint.

(1) When the execution flow of the JavaScript code reaches `JsEval`, the debugger engine catches the breakpoint event. `0x695441b9` is a starting address of a `JsEval` function. Because the IFH uses symbols, it does not need to know the memory address of the function in advance. The IFH simply accesses the location using the starting address information of the function in the symbol.

An advantage of the IFH is the use of the debugging information. As a result, the debugger engine has control of the web browser. In that case, the debugger calls the `DebugBaseEventCallbacks::Breakpoint` function, permitting a user to add their own code. The malicious JavaScript code is monitored and analyzed by extracting the values of the parameters of `JsEval`, the information for the call stack frames, and so on. For example, `[esp+0x4]` is the first parameter of the hooked function and `[esp+0x4*n]` is the *n-th* parameter of the function.

(2) After obtaining the information for the function, the IFH gives the execution control to IE. By doing this, the IFH performs monitoring for the `JsEval` function.

4. Detection Model for Malicious JavaScript Code

In this section, the execution flow of JavaScript code in a web browser is analyzed first to make a model for detection. Then, based on the analysis, some threats of malicious JavaScript code are modeled.

4.1 Execution Flow of JavaScript Code

A web browser parses and executes a HTML document after downloading it into the local computer system. In order to parse dynamic HTML (DHTML), the browser makes a document object model (DOM) tree, and then executes each node related to the HTML tags. In the web page, client script languages such as JavaScript and VBScript can access and control all nodes of the DOM tree.

To model the behavior of malicious JavaScript code, the execution flow of JavaScript code is analyzed and defined, as shown in **Fig. 2**.

Firstly, the web browser requests a web page from a web server (A0), downloads the web page (A1), and parses the HTML document file. Using the HTML document, the web browser creates a DOM tree composed of the various HTML tags (A2). In the DOM tree, each node is related to tags, such as HTML, BODY, SCRIPT, and so on. The web browser displays and executes each node.

In this paper, the focus is on the SCRIPT tag, because JavaScript codes are embedded within this tag. When a browser executes the script codes (such as JavaScript or VBScript) in the SCRIPT tag, the execution flow reaches the S0 state and the browser parses codes from the <SCRIPT> tag until the </SCRIPT> tag.

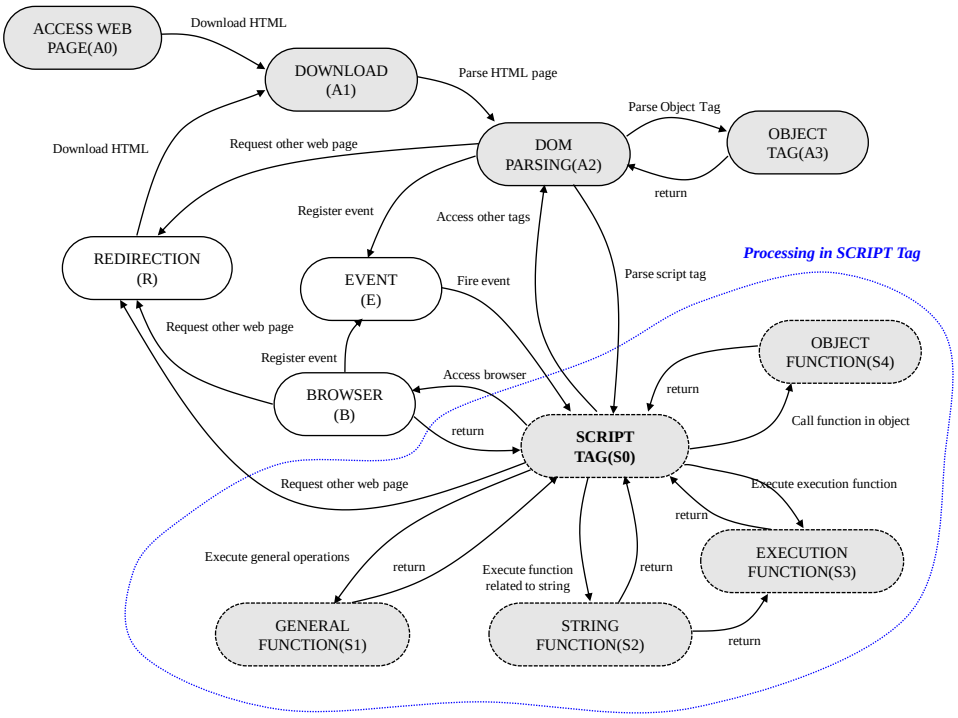

Fig. 2. Execution flow of JavaScript code

In the GENERAL FUNCTION (S1) state, the JavaScript engine only executes general JavaScript functions such as `Sqrt`, `Max`, and so on. Most malicious JavaScript code targets functions that are related to string. Therefore, in this paper, the string functions are the primary focus. In the STRING FUNCTION (S2) state, the JavaScript engine performs various string operations. In this state, malicious JavaScript code changes shape of its own accord, deobfuscates the obfuscated string, and executes the exploit code.

Recently, because many malicious JavaScript codes use obfuscation, another state was set for obfuscation, the EXECUTION FUNCTION (S3). The execution function state includes the `eval`, `document.write`, and similar functions. If a real malicious code is at another web server, the web browser is redirected to the server (REDIRECTION). Then, the browser reaches the DOWNLOAD (A1) state again.

There are many cases where the malicious JavaScript code attacks the vulnerabilities of web browsers or web applications. For example, if a malicious JavaScript code exploits the vulnerability of an ActiveX control, the browser goes to the OBJECT FUNCTION (S4) state. In this state, the internal functions implemented in the ActiveX Control are called. Because this execution flow focuses on the execution of the JavaScript code, the focus is on the string related to the exploit code, rather than the execution of exploit code in the memory.

4.2 Model for Detecting the Behavior of Malicious JavaScript Code

In this section, a model that detects malicious JavaScript is created based on the execution flow that was analyzed in the previous section. Using the proposed model, we make a model for detecting malicious JavaScript (Obfuscation, Exploit code, and Redirection) as shown in **Fig. 3**.

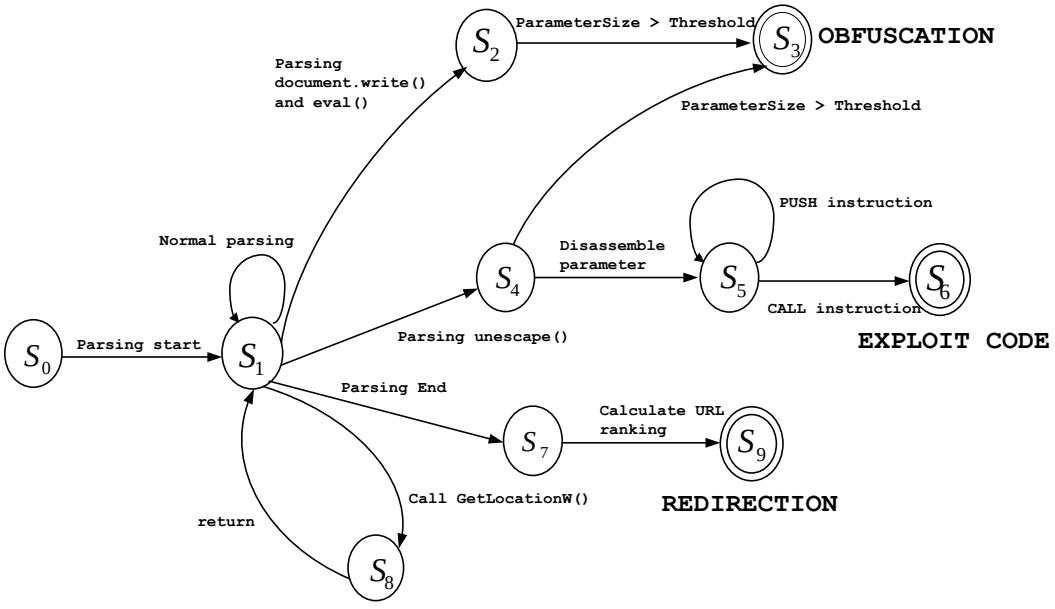

Fig. 3. Model for detecting the behavior of malicious JavaScript code

In this paper, we focus on three behaviors of malicious JavaScript code: obfuscation (S_3), exploit code (S_6), and redirection (S_9). S_0 means the initial state of JavaScript tag. S_1 represents the normal state that parses JavaScript code, and S_8 is the state that parses a URL in a HTML file. Thus, most of script codes are handled in S_1 and S_8 states.

(i) For OBFUSCATION (S_3), the model (M_o) is $S_1 \rightarrow S_2 \rightarrow S_3$ and $S_1 \rightarrow S_4 \rightarrow S_3$. M_o is related to some functions: `eval`, `document.write`, and `unescape`. After extracting the parameter value in the functions, M_o checks size of the value to determine whether it is obfuscated or not.

(ii) For EXPLOIT CODE (S_6), the model (M_e) is $S_1 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6$. M_e disassembles the parameter value of `unescape` and finds the pattern of executable code (`push`→`push`→. . .→`call`) in the value.

(iii) For REDIRECTION (S_9), the model (M_r) is $S_1 \rightarrow S_8 \rightarrow S_7 \rightarrow S_9$. M_r collects all URLs in a HTML file (S_8) and calculates ranking of them. Based on the ranking, M_r finds malicious redirection among URLs. These three components of the model are described in detail below.

Model for Detecting Obfuscated JavaScript: M_o

For M_o model, the processing flow of the JavaScript string in the target function is the primary focus. The target functions `eval`, `document.write`, and `unescape` were selected for this process. The parameters of each function are related to the obfuscated and deobfuscated strings. The detection of the obfuscated string is performed through static analyses, and extraction of the deobfuscated string is undertaken through the execution of a web page. For `eval` and `document.write`, when an obfuscated string is entered into the function code, the function attempts to process the readable string; that is, the deobfuscated string. Therefore, the IFH must access the start address of the function when the target functions are called. However, the deobfuscated string of the `unescape` function can be gained when the target function is returned. For the parameters of the `unescape` function, analyses were performed to detect exploit codes.

Based on the process flow of the JavaScript string, an algorithm that analyzes the malicious JavaScript code is proposed. **Algorithm 1** verifies whether a web page includes an obfuscated string based on the word size, and it deobfuscates the obfuscated string. For the `unescape` function, the algorithm can locate an exploit code.

Algorithm 1. Analyzing obfuscated string

INPUT : HTML document
OUTPUT : Obfuscated check, Deobfuscated string, Exploit code check

```
1  Extract code between <SCRIPT> and </SCRIPT> tags
2  Calculate WordSizes[1…n] in the code // WordSizes includes all word size in the code
3  if (Max(WordSizes) > threshold) {
4    Monitor target functions // eval, document.write, unescape
5    if (function == eval) || (function == document.write) {
6      Extract Val  // value of parameter of function)
7      Notify that Val is deobfuscated string
8    }
9    else if (function == unescape) {
10         Extract Val   // value of parameter of function)
11     Notify that Val is deobfuscated string
12     Disassemble Val as assem
13     if (assem includes push instruction) || ( Next assem is call instruction) {
14       Notify that Val is exploit code
15     }
16   }
17 }
```

The input of this algorithm is a HTML document that can include many script codes. The algorithm is applied to each SCRIPT tag. For example, if a page includes three SCRIPT tags, the proposed algorithm is called three times. First, all codes between the <SCRIPT> and

</SCRIPT> tags (LINE1) are extracted; because it is difficult to only extract the value of a parameter at the source code level, all codes in the SCRIPT tag are targetted. Statically, all word sizes of the codes (LINE 2) are calculated, and then the obfuscated string is detected based on the word size. Many obfuscated strings include unusually long word sizes. A word is defined as the character set between two spaces; for example, in the statement of "*if(a >= b)*", words are "*if(*", "*a*", ">=", "*b*" and "*)*". Maximum word sizes are used in order to decide whether a page includes an obfuscated string (LINE 3).

If the maximum size is longer than the threshold, the proposed algorithm begins to monitor the target functions in order to extract a deobfuscated string and detect an exploit code (LINE 4). The parameters of the target functions are extracted, and because the functions require a readable value as a parameter, the deobfuscated string is always transmitted into their parameter.

In the cases of the `eval` and `document.write` strings, the proposed algorithm extracts the input value of the function parameter and notifies the users of the value (LINE 5-7). For the `unescape` string, the proposed algorithm extracts the output value of the function parameter and verifies whether the string is an exploit code (LINE 9-14). The proposed pattern for detecting an exploit code is described in the following section.

Model for Detecting Exploit Code: M_e

Some malicious JavaScript code includes an exploit code in order to execute the arbitrary code. The exploit code is an assembly code that uses the vulnerability of the web browser or web applications, such as buffer overflow. By allowing a user to execute the exploit code unconsciously, the attacker can control the user's system.

A pattern related to the call instruction is used to decide whether the string is executable code or not. A call instruction is used in an exploit code, because the code must call other APIs to undertake various jobs. A pattern was found: `push` → … → `push` → `call`: that is, the push instruction was executed at least once before the call instruction. This indicates that the exploit code calls an API or a function with parameter(s). Because a normal string can only include a call instruction when it is dissembled, the push instruction is considered to be related to the function parameter. If the string includes this pattern, it was decided that the string was an exploit code.

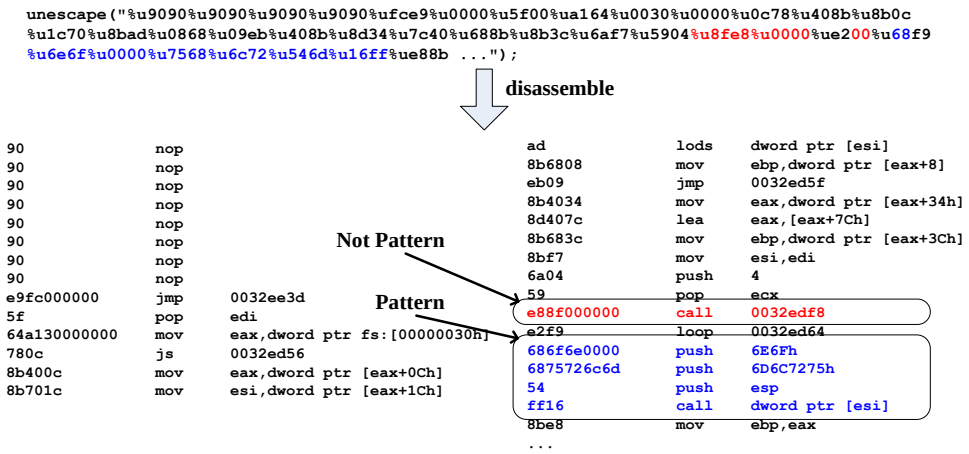

Fig. 4. Pattern for detecting exploit code in malicious JavaScript

Fig. 4 shows an example of the exploit code being disassembled. "`push 6E6Fh, push 6D6C7275h, push esp, call dword ptr [esi]`" is the pattern that the exploit code uses to call the APIs with three parameters. Although "`e88f000000`" is parsed into a call instruction, it does not match the proposed pattern because the instruction does not have parameter(s).

Model for Detecting Redirection: M_r

To force a web browser to access a malicious server, an attacker inserts malicious code for redirection into a web page. Without the user realizing it, a user's web browser accesses the malicious server, through the web pages with the embedded redirection code.

In this paper, only the execution of the web browser itself is a focus, and black list matching using a malicious URL list is not considered. Therefore, suspicious URLs are detected based on the URL's information included in the web page.

Suspicious URLs have the possibility of accessing a malicious server. A method for locating suspicious URLs is proposed based on the domain name that the web browser accesses. Suspicious URLs are verified based on two metrics:

- **Ranking** is the number of URLs which have same domain name. For example, if a web page includes a URL such as www.yahoo.com five times, the ranking of the URL is 5. Thus, the more related a URL is to the domain that a web browser accesses, the higher the ranking is.
- **Meaningful Word** is a word used several times in URLs. Generally, URLs that include meaningful words have a high ranking. For example, in *www.yahoo.com*, the meaningful word is *yahoo*. General words related to classifications such as *com*, *org*, *www* are excluded. For *mail.yahoo.com*, although the URL's ranking is one, it is not a suspicious URL because it includes a meaningful word.

Algorithm 2. Detecting redirection

```
INPUT : HTML document
OUTPUT : Suspicious URLs
 1   Extract URLs[1…n] in HTML document
 2   Calculate Ranking of URLs // Ranking is number of URLs in the document
 3   Extract meaningful words in URLs
 4   for ( i = 1 to n) {
 5     if ( URLs[i].ranking == 1) || (URLs[i] doesn't include meaningful words) {
 6       Analyze stack frame of URLs[i]
 7       if (URLs[i] is related to IFRAME) {
 8         Notify that URLs[i] is suspicious
 9       }
10     }
11   }
```

Algorithm 2 is proposed for the detection of redirection to malicious servers. Firstly, all URLs in a HTML document file (LINE 1) are extracted. Next, the rank of all URLs in the HTML document (LINE 2) are calculated. The more URLs in the web page, the higher the URL ranking. It was decided that URLs with a high rank are safe, because attackers generally do not insert multiple URL redirection codes into HTML documents. Among the URLs that have high ranking, the meaningful words (LINE 3) were extracted. The URL relationship to the redirection for all URLs (LINE 4) was verified. The URL was examined if the URL's ranking was one and did not include meaningful words (LINE 5). Next, the stack frame related to URL (LINE 6) was analyzed. In the stack frame, there is a function for the tag to which the URL is related: for IFRAME, the stack frame includes the `mshtml!CHtmIframeParseCtx::Execute` function (LINE 7). If the IFRAME tag includes the URL, it was decided that the URL is suspicious.

5. Implementation of JsSandbox

JsSanbox was implemented as a practical tool to detect and analyze malicious JavaScript code based on the detection models described in this section. JsSandbox is a debugger that monitors and extracts information related to various internal functions in the JavaScript engine and HTML parser of a web browser.

JsSandbox can access the internal functions in DLL files, as well as the exported functions, because it uses a debugging method with the public symbol files provided by Microsoft. The symbol file includes information about the functions and data structures for debugging programs in Windows systems. JsSandbox uses the WinDbg debugger engine [24], which is the representative debugger for the Windows environment. In this paper, Internet Explorer 7 (IE7) and Windows Vista are the target platforms.

Before implementing JsSandbox, the HTML parser (*mshtml.dll*), JavaScript engine (*jscript.dll*), and URL library (*shlwapi.dll*) in IE7 were analyzed through reverse engineering in order to extract the information from the functions and parameters. In IE's HTML parser, a tag is implemented as a class that includes codes related to the attributes and functions. For example, the SCRIPT tag is implemented as `CScriptElement` in *mshtml.dll*. In order to execute the script code, `CScriptElement` calls `COleScript::ParseScriptText` in *jscript.dll*. That is, the HTML parser calls the JavaScript engine because the HTML parser cannot process the JavaScript code. The JavaScript functions are defined as functions using prefix `Js`; for example, eval is `JsEval`, unescape is `JsUnescape`, and so on. If the JavaScript code wants to access the document object model (DOM), it recursively accesses the classes in *mshtml.dll*, because the DOM is related to the HTML parser. For example, if the code is `eval(document.write())`, the JavaScript engine calls `CDocument::write` in *mshtml.dll* again.

JsSandbox monitors the HTML parser (*mshtml.dll*) and JavaScript engine (*jscript.dll*). The HTML parser processes the DOM, which represents a tag as an object. In *mshtml.dll*, an object is implemented as a class. For example, the SCRIPT tag is written as a `CScriptElement` class. If the SCRIPT tag executes a JavaScript code, the HTML parser calls the `COleScript` class in *jscript.dll*. This class processes the JavaScript code after the JavaScript engine parses the code.

Based on the analysis of the IE modules, the JsSandbox system was implemented for sandboxing malicious JavaScript code in IE 7. Fig. 5 shows the architecture of the JsSandbox system.

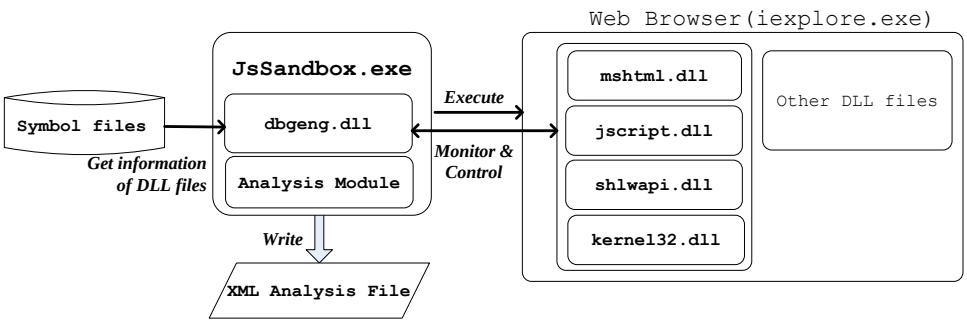

Fig. 5. Overall architecture of JsSandbox

JsSandbox traces *iexplore.exe* and sets a breakpoint for all functions related to the proposed detection model through the debugger engine (*dbgeng.dll*). A symbol file helps JsSandbox access the internal functions of IE. JsSandbox monitors some modules, such as

mshtml.dll, *jscript.dll, shlwapi.dll*, and *kernel32.dll*. After collecting data about the behavior of the JavaScript code, JsSandbox analyzes the data and prepares an XML analysis report for the analyst.

Detecting Obfuscation and Extracting Deobfuscated String

JsSandbox can detect whether the JavaScript is obfuscated or not, using the word size. Because most obfuscated strings are very long, the maximum word size of the parameter in the target function can be calculated. However, it is difficult to extract all values of the parameter of target functions, because there are various descriptions for the variables related to the parameter of target functions.

Therefore, the focus in this paper is on all statements that are in JavaScript code. JsSandbox selects the longest word in the statements and decides whether the code is obfuscated or not based on the word size. The code in the SCRIPT tag is related to `COleScript::ParseScriptText` in *jscript.dll*. Therefore, after JsSandbox accesses the second parameter of the function, it can extract all codes in the SCRIPT tag. The function is called once per SCRIPT tag. If the word size is beyond a specified threshold, JsSandbox determines that the JavaScript includes an obfuscated string.

JsSandbox monitors three functions: `eval`, `document::write`, and `unescape`, and obtains the value of the parameter in order to extract the deobfuscated string. These functions assist in executing the obfuscated string. The `eval` and `unescape` functions are implemented as `JsEval` and `JsUnescape` in *jscript.dll*, and `document.write` is implemented as `CDocument::write` in *mshtml.dll*.

In `JsEval`, the fifth parameter is directly related to the `eval` parameter. When `JsEval` is called, a deobfuscated string exists at `[[[esp+0x14]+0x8]+0x8]`. Because all variable types in JavaScript are VARIANT, its memory address is complex. The size value of the deobfuscated string exists in the memory address four bytes ahead of the deobfuscated string. `CDocument::write` receives it as a deobfuscated string through the second parameter and the type of parameter is SAFEARRAY. The address of the string is `[[[esp+0x8]+0xc]+0x8]`. Therefore, JsSandbox obtains the deobfuscated string by extracting the value of parameter of `eval` and `document::write`.

Detecting Redirection

JsSandbox extracts all URLs linked in a HTML document file using `GetLocationW` in *shlwapi.dll*. This function is always called when a URL in a tag is parsed. When the function is called, JsSandbox also examines the call stack frame by calling `IDebugControl->GetStackTrace`. Through the call stack frame, JsSandbox can know which tag is related to the URL.

For example, the IFRAME tag calls `mshtml!CHtmIframeParseCtx::Execute` and this function calls `GetLocationW`. Therefore, JsSandbox targets operations related to redirection, such as `iframe`, `document.location`, `document.URL`, `window.location`, `window.location.href`, and so on. These JavaScript codes can redirect to another site without the user's action. Therefore, JsSandbox monitors the value of parameter of `GetLocationW` and the call stack frame.

Detecting Malicious Behavior

JsSandbox finds malicious behavior based on the assembly code. The focus of JsSanbox is on an exploit code operated in a stack frame or a heap using a buffer overflow. In order to write the byte codes of a string directly into the memory, the exploit code in JavaScript uses the

`unescape` function. Therefore, JsSandbox monitors `JsUnescape` in *jscript.dll*.

Its third parameter is related to the return value of the function, and the input value is transferred into its fifth parameter. When `JsUnescape` is called, JsSandbox remembers the address of `[[esp+0xC]+0x8]+0x8`. After the function is returned, JsSandbox reads the string at the address. To disassemble the string, JsSandbox calls `IDebugControl::Disassemble` in the WinDBG engine. Using the disassembly code, JsSandbox checks for the existence of the pattern, `push` → … → `push` → `call`, as described in Section 4.2, and decides whether the JavaScript code includes an exploit code.

6. Evaluation

JsSandbox was evaluated using the proposed detecting model for malicious JavaScript: detecting obfuscation, extracting deobfuscated strings, detecting redirection, and detecting exploit codes. Undoubtedly, JsSandbox can detect malicious code using other detection models that have been presented in other research and have been developed continuously by the current authors.

In an experiment, JsSandbox and a target program were operated in virtual environment (e.g. VMWare) and controlled using a Virtualization Controller. JsSandbox executed IE and IE accessed web pages periodically based on a URL list that was collected. In this experiment, JsSandbox monitored the IE JavaScript engine for 60 seconds for each page. This time was sufficient for JsSandbox to trace and analyze the behavior of IE.

6.1 Evaluation of the Internal Function Hooking

The number of functions that the API hooking and IFH can access were compared. In this experiment, the IFH could access more functions than the API hooking because the IFH uses the public symbol file.

In particular, for the COM, IE consisted of several COMs, but the API hooking could not access the internal functions in the COM. **Table 1** shows the number of functions that the API hooking and IFH can access. Among IE's modules, *mshtml.dll*, *jscript.dll*, *vbscript.dll*, *urlmon.dll*, and *mlang.dll* were COM.

Table 1. Number of functions that API hooking and IFH can access

Name	API hooking	IFH	COM
mshtml	13	26,154	O
jscript	4	3,539	O
vbscript	4	4,066	O
wininet	247	4,049	
urlmon	219	5,457	O
mlang	14	1,056	O
shlwapi	858	3,573	

For example, in order to access the functions in *jscript.dll*, API hooking can access only four functions: `DllCanUnloadNow`, `DllGetClassObject`, `DllRegisterServer`, and `DllUnregisterServer`, because it cannot access the internal functions of COM. However, the IFH can access 3,539 internal functions, such as `JsEval` and `JsUnescape`. Furthermore, JsSandbox targeted the functions of the JavaScript engine and HTML parser such as `JsEval`, `JsUnescape`, `COleScript::ParseScriptText`, and `CDocument::write`. These functions can only be accessed by the IFH.

The IFH can monitor not only the functions that can be monitored by API hooking but also internal functions, because the public symbol includes information for exported functions and internal functions. In *wininet.dll*, the number of symbols is 4,049 and the number of exported functions is 247.

JsSandbox was applied to Internet Explorer 6, 7, and 8 on Windows XP, Vista, and 7. Without additional modification, after the analysis for each Windows and IE version, JsSandbox was used for all versions to analyze malicious JavaScript code.

For the Ultimate Deobfuscator [12], if some modules were changed, these modules in all versions needed to be analyzed in order to set the start memory address of the functions in the COM object. The memory address of `JsEval` symbol is as follows: Windows XP SP 3 & IE 7 (`0x75ba4bba`), Windows Vista & IE 7(`0x6e6f81d9`), and Windows 7 & IE 8 (`0x6bb1912a`).

6.2 Evaluation of the Malicious JavaScript Code Detection

Malicious JavaScript code was analyzed with JsSandbox using the proposed detection model. Firstly, the inclusion of an obfuscated string in a JavaScript code was analyzed. Obfuscation is the changing of the shape of a string in order to avoid a signature-based detection system. Recently, many JavaScript attacks have used obfuscation methods, and there are various methods for obfuscation [12][25].

Using Google, 452,892 web pages were collected, including 123,404 JavaScript files. In order to analyze the JavaScript code, the `js` files were collected separately. Among the web pages, it was found that the `eval` function was used 91,741 times, the `document.write` function was used 172,121 times, and the `unescape` function was used 38,657 times. Because these functions are related to obfuscation, the maximum word size of the parameter value of the functions was calculated.

Table 2 shows the distribution of the maximum word size in three functions: `eval`, `document.write`, and `unescape`. 55 bytes indicates that 90% of the parameters of the eval function were shorter than 55 bytes. The word size was selected to be 95% as the threshold for deciding obfuscation. Therefore, if the maximum word size of the strings in JavaScript was longer than the word size threshold, it was considered to be potentailly obfuscated and it was then analyzed.

Table 2. Average word size of parameters of target function

	`eval`	`document.write`	`unescape`
90%	55 bytes	82 bytes	35 bytes
95%	126 bytes	82 bytes	41 bytes

Next, all word sizes in the script codes were verified statically without considering the parameters of the target functions. **Fig. 6** shows the distribution of the word size of the string in JavaScript codes. Most sizes were concentrated within 100 bytes.

The previous two experiments demonstrated that the parameter values of the target functions and codes in script tags have similar distributions in word sizes. Most word sizes are concentrated in approximately 100 bytes: 99% of string word sizes were shorter than 120 bytes. Therefore, the threshold for deciding obfuscation was the word size longer than 120 bytes. In order to reduce false positives, the threshold was set at 500 bytes, and JsSandbox analyzed the top 500 sites in Alexa [26]. The web sites did not include JavaScript strings longer than 500 bytes.

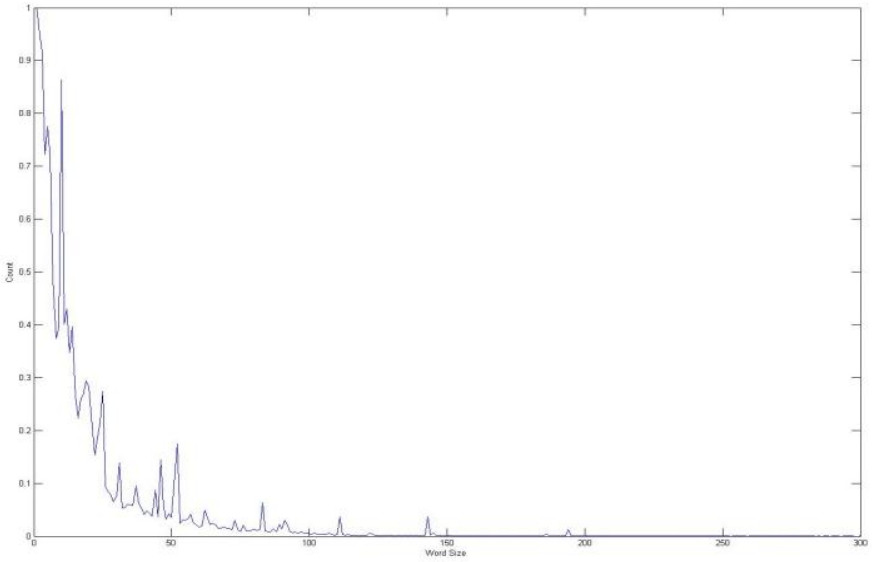

Fig. 6. Distribution of word sizes in JavaScript code strings.

Because attackers use smaller web sites more often than popular portal sites, 142,117 small web sites were collected. In these web pages, 19 obfuscated strings were found based on word size: the maximum word size varied from 536 bytes to 11,079 bytes.

For the obfuscated string, JsSandbox can extract a deobfuscated string. **Fig. 7** shows an example of a real malicious JavaScript code that JsSandbox found, and the word size of the sting, which was 758 bytes. After deobfuscating the obfuscated string, a web browser accessed 2.htm, 3.htm, and 4.htm automatically because it included the IFRAME tag. JsSanbox deobfuscated all 19 obfuscated strings that were found. JsSandbox extracted the deobfuscated strings from the parameters of the `eval`, `document.write`, and `unescape` functions.

One feature of malicious JavaScript code is redirection, which changes the connection of the web site to an arbitrary URL. By doing this, the attacker can force a web browser to download malware or force a user view an unexpected advertisement. JsSandbox extracts the URL from parameter of the `GetLocationW` function in *shlwapi.dll* and the information from call stack related to that URL. JsSandbox can know the tag name related to the redirection using the information from the call stack. For example, IFRAME tag calls functions in `CHtmIframeParseCtx` class.

BEFORE Deobfuscation

```
Eval("W144W157W143W165W155W145W156W164W56W167W162W151W164W145W154W156W50W42W74W151W146W162W141W155W145W40W
163W162W143W75W64W56W150W164W155W40W167W151W144W164W150W75W61W40W150W145W151W147W150W164W75W61W40W76W74W
134W57W151W146W162W141W155W145W76W42W51W73W15W12W144W157W143W165W155W145W156W164W56W167W162W151W164W145W
154W156W50W42W74W151W146W162W141W155W145W40W163W162W143W75W62W56W150W164W155W40W167W151W144W164W150W75W61W
40W150W145W151W147W150W164W75W61W40W76W74W134W57W151W146W162W141W155W145W76W42W51W73W15W12W144W157W143W
165W155W145W156W164W56W167W162W151W164W145W154W156W50W42W74W151W146W162W141W155W145W40W163W162W143W75W63W
56W150W164W155W40W167W151W144W164W150W75W61W40W150W145W151W147W150W164W75W61W40W76W74W134W57W151W146W162W
141W155W145W76W42W51W73")
```

AFTER Deobfuscation

```
document.writeln("<iframe src=4.htm width=1 height=1 ><W/iframe>");
document.writeln("<iframe src=2.htm width=1 height=1 ><W/iframe>");
document.writeln("<iframe src=3.htm width=1 height=1 ><W/iframe>");
```

Fig. 7. Deofuscation of obfuscated malicious JavaScript code

Among the 142,117 web pages, JsSandbox found 8 URLs related to redirection among the suspicious URLs. Two features were used to find the redirection to a malicious web page:

ranking and IFRAME tag. JsSandbox decided that a web site included redirection codes when the ranking of some URLs were one, and the redirection was related to the IFRAME tag. JsSandbox detected 8 web pages that redirected the web browser to suspicious sites. The focus was on the IFRAME tag because most web pages including redirection use this tag.

Finally, JsSandbox was evaluated for its detection of exploit code. Exploit codes were collected from real world examples and the *Milw0rm* site [27]. In *Milw0rm*, all JavaScript-related exploit codes are gathered that were published in 2009. After disassembling the parameter values of `JsUnescape` using JsSandbox, some exploit codes were detected from the disassembled code. Three malicious codes were found based on the proposed detection model.

Table 3. Patterns of exploit code in real world detected by JsSandbox

#1 exploit code			**#2 exploit code**			**#3 exploit code**		
53	push	ebx	68 00 20 00 00	push	2000h	52	push	edx
FF D0	call	eax	6A 00	push	0	FF D0	call	eax
			FF D0	call	eax			

Table 3 shows the patterns that JsSandbox detected in the real world. The `call eax` instruction indicates that the exploit code calls a function and the previous `push` instruction indicates that the code inserts a parameter of the function into the memory. Next, the exploit codes from the *Milw0rm* site were used in experiments. JsSandbox found three exploit codes from the 11 codes. Eight exploit codes did not include the proposed pattern. **Table 4** shows the proposed pattern in the three exploit codes that were found by JsSandbox.

Table 4. Patterns of exploit code from the *Milw0rm* site detected by JsSandbox

#1 exploit code	**#2 exploit code**	**#3 exploit code**
66 53 push bx	FF 37 push dword ptr [edi]	53 push ebx
66 68 33 32 push 3233h	56 push esi	FF 57 F8
68 77 73 32 5F push 5F327377h	E8 33 00 00 00 call 001fe7ec	call dword ptr [edi-8]
54 push esp		
FF D0 call eax		

7. Conclusion

A tool was proposed for analyzing malicious JavaScript code, JsSandbox, that can analyze and detect malicious JavaScript code using internal function hooking. Since JsSandbox uses the engine of the WinDBG debugger and the public symbols provided by Microsoft, it can access various internal functions in the JavaScript engine.

After attaching JsSanbox to IE, JsSandbox traced the JavaScript engine (*jscript.dll*), HTML parsing engine (*mshtml.dll*), and so on. It was found that the internal function hooking supplied more sophisticated analysis results than API hooking. Since JsSandbox is a general framework, it can be extended to analyze another malicious code such as VBScript and Perl.

Most research has used the JavaScript engine of Mozilla, SpiderMonkey, because it is open source. However, in this paper, JsSandbox was implemented in Microsoft Internet Explorer after reverse engineering of the IE modules. Therefore, JsSandbox can detect threats propagated by IE from malicious JavaScript code. Also, the functions of JsSanbox were extended in order to analyze other malicious codes such as VBScript.

Some features for detecting malicious JavaScript code were proposed and were subsequently applied to JsSandbox. Evaluation of these features demonstrated that JsSandbox

can analyze and detect several malicious JavaScript codes using the proposed features.

In order to detect malicious JavaScript more accurately, more patterns of malicious behavior in JavaScript are required. Thus, the behavior of various malicious javascript have been modeled. Using the new malicious model, JsSanbox can detect malicious JavaScript code with low false positive and false negative results.

References

[1] J. Gregoire, "JavaScript and Visual Basic Script Threats: Different scripting language for different malicious purposes," *in Proc. of 18th EICAR Annual Conference*, 2009. Article (CrossRef Link)
[2] S. Shah, "Browser Exploits: Attacks and Defense", *EUSecWest*, 2008. Article (CrossRef Link)
[3] M. Egele, E. Kirda and C. Kruegel, "Mitigating drive-by download attacks: Challenges and open problems," *in Proc. of iNetSec 2009 - Open Research Problems in Network Security Workshop*, pp.52-62, 2009. Article (CrossRef Link)
[4] C. Willems, T. Holz and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security & Privacy*, vol.5, pp.32-39, Mar.2007. Article (CrossRef Link)
[5] U. Bayer, C. Kruegel and E. Kirda, "TTanalyze: A Tool for Analyzing Malware," *in Proc. of 15th EICAR Annual Conference*, 2006. Article (CrossRef Link)
[6] Norman Sandbox, http://www.norman.com.
[7] VMWare, http://www.vmware.com.
[8] F. Bellard, "QEMU, A fast and portable dynamic translator," *in Proc. of USENIX Annual Technical Conference*, pp.41-46, 2005. Article (CrossRef Link)
[9] Y. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen and S. King, "Automated web patrol with strider HoneyMonkeys," *in Proc. of Network and Distributed System Security Symposium*, pp.35-49, 2006. Article (CrossRef Link)
[10] COM: Component Object Model Technologies, http://www.microsoft.com/com/default.mspx
[11] R. Kawach, "NEPTUNE: Detecting Web-Based Malware via Browser and OS Instrumentation," *Black Hat USA*, 2010. Article (CrossRef Link)
[12] S. Chenette, "The Ultimate Deobfuscator", *ToorCon*, 2008. Article (CrossRef Link)
[13] K. Chellapilla and A. Maykov, "A Taxonomy of JavaScript redirection Spam," *in Proc. of 3rd International Workshop on Adversarial Information Retrieval on Web*, pp.81-88, 2007. Article (CrossRef Link)
[14] J. Nazario, "Reverse engineering malicious JavaScript," *CanSecWest*, 2007. Article (CrossRef Link)
[15] N. Provos, D. McNamee, P. Mavrommatis, K. Wang and N. Modadugu, "The Ghost in the browser analysis of web based malware," *in Proc. of USENIX First Workshop on Hot Topics in Understanding Botnets*, 2007. Article (CrossRef Link)
[16] A. Ikinci, T. Holz and F. Freiling, "Monkey-Spider: Detecting malicious WebSites with low interaction Honeyclients," *in Proc. of Sicherheit, Schutz und Zuverlässigkeit,* 2008. Article (CrossRef Link)
[17] B. Feinstein and D. Peck, "Caffeine Monkey: Automated collection, detection and analysis of malicious JavaScript," *Black Hat USA*, 2007. Article (CrossRef Link)
[18] O. Hallaraker and G. Vigna, "Detecting malicious JavaScript code in Mozilla," *in Proc. of 10th IEEE Int. Conference on Engineering of Complex Computer Systems*, pp.85-94, 2005. Article (CrossRef Link)
[19] A. Dewald, T. Holz, and F.C. Freiling, "ADSandbox: Sandboxing JavaScript to fight Malicious Websites", *in Proc. of 25th Symposium on Applied Computing*, pp. 1859-1864, 2010. Article (CrossRef Link)
[20] M. Cova, C. Kruegel and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious JavaScript vode," *in Proc. of 19th International World Wide Web Conference*, pp.281-290,2010. Article (CrossRef Link)
[21] Decrypt JS, http://www.ukhoneynet.org/tools/decrypt-js/.
[22] Rhino: JavaScript for Java, http://www.mozilla.org/rhino/.
[23] NJS JavaScript Interpreter, http://sourceforget.net/projects/njs/.
[24] Debugging Tools for Windows, http://www.microsoft.com/whdc/devtools/debugging/default.mspx.
[25] Kolisar, "WhiteSpace: A Different approach to JavaScript obfuscation," *DEFCON 16*, 2008. Article (CrossRef Link)
[26] Alexa Top 500 Sites, http://www.alexa.com/.
[27] Milw0rm, http://www.milw0rm.com/.

Hyoung Chun Kim is currently a senior member of engineering staff and the team leader in the Attached Institute of Electronics and Telecommunications Research Institute. His research interests include software security, intrusion detection systems, and network security. He received his B.Sc. and M.Sc. degrees in Computer Science from Korea University, Korea, in 1999 and 2001, respectively. He got his Ph.D. degree from the Graduate School of Information Security of Korea University, Korea, in 2011.

Young Han Choi is currently a senior member of engineering staff in the Attached Institute of Electronics and Telecommunications Research Institute. His research interests include software security, intrusion detection systems, and operating system. He received his B.S. and M.S. degrees in electronic engineering from Hanyang University and Korea Advanced Institute of Science and Technology, Korea, in 2002 and 2004, respectively.

Dong Hoon Lee is a professor of Graduate School of Information Security of the Korea University in Korea. He received his B.S (1985) of Economics from the Korea University, M.Sc. (1988) from the University of Oklahoma, and Ph.D. (1992) from the University of Oklahoma. He was a faculty member at the University of Dankook of Korea from 1992 to 1993 before he joined the Korea University in 1993. He was an Editor-in-Chief at Korea Institute of Information Security and Cryptology (KIISC, 2002), Program Co-Chair of ICISC (International Conference on Information Security and Cryptology) Program Committee. He has been a chairman at Electronics Election Research (EER) and Mobile Payment Standard Association (MPSA). He is the leading researcher on Information Security, Cryptology, and Ubiquitous Security Study.