ISSN 1598-0170 (Print)
ISSN 2287-1136 (Online)
http://www.jksii.or.kr

실시간으로 악성 스크립트를 탐지하는 기술☆

The Real-Time Detection of the Malicious JavaScript

추 현 록[1] 정 종 훈[1] 김 환 국[1*]
Hyun-Lock Choo Jong-Hun Jung Hwan-Kuk Kim

요 약

자바 스크립트는 정적인 HTML 문서에 동적인 기능을 제공하기 위해 자주 사용되는 언어이며, 최근에 HTML5 표준이 발표됨으로써 더욱더 관심 받고 있다. 이렇게 자바 스크립트의 중요도가 커짐에 따라, 자바 스크립트를 사용하는 공격(DDos 공격, 개인 정보 유출 등)이 더욱 더 위협적으로 다가오고 있다. 이 악성 자바 스크립트는 흔적을 남기지 않기 때문에, 자바 스크립트 코드만으로 악성 유무를 판단해야 하며, 실제 악성 행위가 브라우저에서 자바 스크립트가 실행될 때 발생되기 때문에, 실시간으로 그 행위를 분석해야만 한다. 이러한 이유로 본 논문은 위 요구사항을 만족하는 분석 엔진을 소개하려 한다. 이 분석 엔진은 시그니쳐 기반의 정적 분석으로 스크립트 코드의 악성을 탐지하고, 행위 기반의 동적 분석으로 스크립트의 행위를 분석하여 악성을 판별하는 실시간 분석 기술이다.

☞ 주제어 : 악성 자바 스크립트, 분석 엔진, 정적 분석, 동적 분석, 스크립트 기반 사이버 공격

ABSTRACT

JavaScript is a popular technique for activating static HTML. JavaScript has drawn more attention following the introduction of HTML5 Standard. In proportion to JavaScript's growing importance, attacks (ex. DDos, Information leak using its function) become more dangerous. Since these attacks do not create a trail, whether the JavaScript code is malicious or not must be decided. The real attack action is completed while the browser runs the JavaScript code. For these reasons, there is a need for a real-time classification and determination technique for malicious JavaScript. This paper proposes the Analysis Engine for detecting malicious JavaScript by adopting the requirements above. The analysis engine performs static analysis using signature-based detection and dynamic analysis using behavior-based detection. Static analysis can detect malicious JavaScript code, whereas dynamic analysis can detect the action of the JavaScript code.

☞ Keywords: Malicious JavaScript, Analysis Engine, Static Analysis, Dynamic Analysis, Script-Based Cyber Attack

1. Introduction

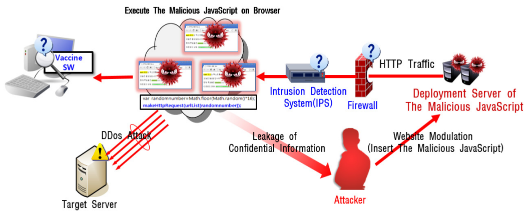
Existing web attacks are mostly of the drive-by-download type. This type of attack induces users to install the malware and execute the attack using the malware. Note, however, that a script-based cyber-attack is different because it can execute an attack only with a user accessing a particular page with a browser. The reason such attack is dangerous can be briefly described as follows:

- A script-based attack is not generated by an executable file such as malware. Since the attack is executed with the running of a script by a browser, it can easily bypass the existing intrusion detection system (IDS) and intrusion prevention system (IPS), which are used to detect the malware.
- An obfuscation method is a technology that is often used to protect the source in JavaScript; it is used by most script-based attacks to bypass the code detecting systems.
- In HTML5, JavaScript supports so many functions that it can replace ActiveX. They include not only audio, video, and other media management functions but also

[1] Korea Internet & Security Agency, Seoul, 138-950, Korea
[*] Corresponding author (rinyfeel@kisa.or.kr)
[Received 15 April 2015, Reviewed 22 April 2015, Accepted 11 June 2015]
☆ This work was supported by the ICT R&D Program of MSIP/IITP. [14-912-06-002, The Development of Script-based Cyber Attack Protection Technology]
☆ A preliminary version of this paper was presented at ICONI 2014 and was selected as an outstanding paper.

access to the local storage of the browser. An attack is very likely to use these functions.

This paper introduces the Analysis Engine for detecting malicious scripts in real time. The Analysis Engine consists of signature-based static analysis and behavior-based dynamic analysis. A signature is an object that expresses the unique pattern data defining a known malicious script. Static analysis checks if the pattern is found in a script code. Note, however, that static analysis cannot find the patterns in the variants of malicious script or obfuscated script since the code patterns are altered. So Dynamic analysis is performed to supplement it. Dynamic analysis tracks the API flow through the JavaScript engine in order to analyze the behavior of the JavaScript code. Since the behavior generated by a malicious script is completed by a native function[1] that can access the system, the syntax may be changed by code obfuscation or polymorphism, but not the API flow that determines the behavior[1][2].


Execute The Malicious JavaScript on Browser
Vaccine SW
var randomnumber=Math.floor(Math.random()*16);
makeHttpRequest(urlList[randomnumber]);
Intrusion Detection System(IPS)
Firewall
HTTP Traffic
Deployment Server of The Malicious JavaScript
DDos Attack
Target Server
Leakage of Confidential Information
Website Modulation (Insert The Malicious JavaScript)
Attacker

(Figure 1) Risk of Script-Based Cyber Attack

The rest of this paper is organized as follows: Chapter 2 discusses related research; In Chapter 3, we present the system architecture and flow and describe more details of each analysis method; Finally, the conclusion and future study direction are presented in Chapter 4.

2. Related Research

There are several ways to analyze the data flow to detect malicious scripts. Well-known FLAX[3] creates JASIL (JAvascript Simplified Instruction Language) using the JavaScript engine just like the Analysis Engine. JASIL is the simplest expression of data and execution defined by JavaScript standard[4]. FLAX finds the critical sinks accessing the data such as user privilege, code generation, and cookie and input the randomly generated data using the black box fussing method and executes the sinks to detect potential risk factors. Staging framework[5] creates the stage containing the policy to restrict data flow and DOM access. When a hole (external JavaScript loaded through the network) accesses the stage while interpreting the JavaScript code, the stage's internal restricting policy is changed according to the hole. A script is judged to be malicious if it violates the internal policy. Other methods[6] of detecting the malicious script include the method using the abstract syntax tree (AST) and control flow graph (CFG). The script code is interpreted to create an AST at first, and the main body of each function in AST is organized to CFG. CFGs are executed through symbolic execution, with the result summarized and stored in the function summary. The summaries are analyzed to detect malicious scripts. Note, however, that this technology works only with PHP.

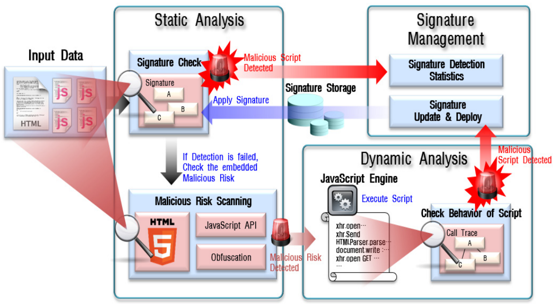
Although the detection methods described above can effectively find the latent malicious scripts in batch processing, they are not suitable for real-time processing because these require a long time and a lot of resources for the analysis in many execution paths. There are rule set-based detection methods such as IDS developed for real-time processing, but they have limitations since they cannot detect new attack types[7].

This paper proposes an Analysis Engine that supplements the aforesaid weaknesses and consolidates the strengths.

3. Proposed Technology

The Analysis Engine seeks to detect as many malicious scripts as possible through fast static analysis and find other variants of or obfuscated malicious scripts through dynamic analysis. Since the static analysis checks if a malicious pattern exists in the inputted JavaScript code, it can be quickly processed without additional code processing. Dynamic analysis tracks the flow of API created when the script is executed in the JavaScript engine to define the behavior and checks if the behavior is similar to that of a known malicious script. The malicious script data detected by the dynamic analysis is used for creating the signature used by the static analysis.

Such updating of the signature for static analysis minimizes opportunities for dynamic analysis; thus shortening the average delay time of malicious script analysis and eventually enabling real-time processing.



(Figure 2) Analysis Engine Architecture

3.1 Applied Technology

Before going into the details of the Analysis Engine, the applied technologies will be described to aid in understanding of the operation of the Analysis Engine.

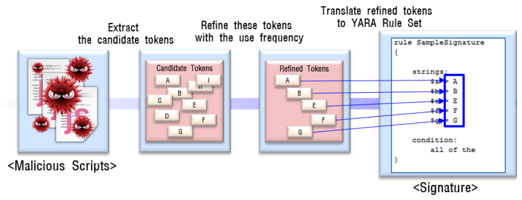
3.1.1 Conjunction Pattern

Candidate string tokens are extracted from input documents and prioritized based on how many times the tokens were used by all documents. The conjunction pattern [8] is a group of tokens refined and restructured based on the priority. In other words, it consists of the most used tokens in all input documents.

3.1.2 YARA

YARA[9] is used to detect or categorize the malware. The biggest strength of the technology is that it can include many data expressions in the detection rule. For example, usual text string, Hex character string, and regular expression can be included. It can also be applied in diverse environments since the control condition of the expressions can also be included in the rule set. And the speed was significantly increased in version 2.0 or later.

The Yara rule set containing the conjunction patterns of the malicious script is called “signature” in this paper.


Extract the candidate tokens
Refine these tokens with the use frequency
Translate refined tokens to YARA Rule Set
Candidate Tokens
Refined Tokens
rule SampleSignature
strings:
condition:
all of the
<Malicious Scripts>
<Signature>

(Figure 3) Process of Signature Creation

3.1.3 JavaScript Engine

The JavaScript engine analyzes the syntax of a script and executes the script in accordance with ECMA-262[4] as the JavaScript standard such as Chakra of IE or V8 of Chrome. When a script code is analyzed for syntax or executed in a JavaScript engine, users can check the caller/callee name and input data through API hooking. These data enable tracking of behavior of the script. Since ECMA-262 does not define the functions related to DOM (Document Object Model) [10], DOM APIs should be additionally ported so that the browser behavior is accurately understood.

The script execution data created in the JavaScript engine is called "call trace" in this paper; the call trace of the malicious script used for malicious detection is called "call trace signature."

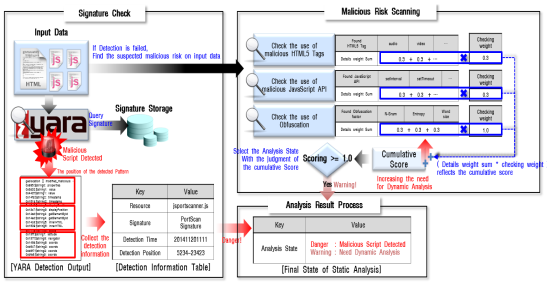
3.1.4 SimHash

SimHash is a similarity analysis algorithm using LSH (Locality Sensitive Hash). Unlike the general function used in mathematics, LSH[19] maximizes the collision probability. In other words, it generates similar result for similar items. The input data is transformed into a fingerprint arranged in byte array of a specific length regardless of size by LSH. The similarity distance[11] is measured between the created fingerprints using the Hamming distance[12].

SimHash is used to compare the similarity of call trace signature and call trace through the dynamic analysis of the script behavior.

3.2 Static Analysis

Static analysis is mainly divided into two parts: signature check, which detects the malicious scripts using the pattern data of signatures, and; malicious risk scanning, which determines the additional risk for the dynamic analysis. If the signature check fails, malicious risk scanning is performed; the input script is considered safe if the score of the malicious risk is below the threshold. Figure 4 shows the work flow of static analysis.


Signature Check
Input Data
HTML
js
If Detection is failed,
Find the suspected malicious risk on input data
yara
Query Signature
Signature Storage
Malicious Script Detected
The position of the detected Pattern
Collect the detection information
Key | Value
Resource | jsportscanner.js
Signature | PortScan Signature
Detection Time | 201411201111
Detection Position | 5234~23423
[YARA Detection Output]
[Detection Information Table]
Danger!
Malicious Risk Scanning
Check the use of malicious HTML5 Tags
Found HTML5 Tag | audio | video | ...
Details weight Sum | 0.3 + 0.3 + ...
Checking weight | 0.3
Check the use of malicious JavaScript API
Found JavaScript API | setInterval | setTimeout | ...
Details weight Sum | 0.3 + 0.3 + ...
Checking weight | 0.3
Check the use of Obfuscation
Found Obfuscation factor | N-Gram | Entropy | Word size
Details weight Sum | 0.3 + 0.3 + 0.3
Checking weight | 1.0
Select the Analysis State With the Judgment of the cumulative Score
Scoring >= 1.0
Cumulative Score
(Details weight sum * checking weight) reflects the cumulative score
Increasing the need for Dynamic Analysis
Yes Warning!
Analysis Result Process
Key | Value
Analysis State | Danger : Malicious Script Detected Warning : Need Dynamic Analysis
[Final State of Static Analysis]

(Figure 4) Static Analysis Work Flow

3.2.1 Signature Check

A stored signature in the signature storage contains the code pattern data of an already detected malicious code. If a script matches the pattern data, the script is very likely to be one of the malicious script types used for the creation of the signature. Table 1 below shows the result of the signature inspection.

This matching job is performed by the YARA module. When a specific pattern is detected, the YARA module outputs the file location of the detected character string. We will track the area where the tokens in the signature are most concentrated; the tracked area data of the malicious code will be used to remove the malicious code later.

(Table 1) Result of Signature Check

Malicious Type	XHR Dos	Hash Dos	IP Scan	Port Scan	Worker Dos	File API	Geolocation	Total
Total Number (A)	80	80	80	80	80	80	80	560
Detection Number (B)	80	80	74	74	80	71	80	539
Detection Rate (B/A)*100	100%	100%	92.5%	92.5%	100%	88.7%	100%	96.2%

3.2.2 Malicious Risk Scanning

Even if a script passed the signature check, it may still need to be checked by the dynamic analysis because it is difficult to detect new types or transformed/obfuscated malicious scripts with signatures[13]. Malicious risk scanning measures the risk of input scripts by inspecting the risk factors described below and decides whether dynamic analysis is needed by comparing the risk level with the threshold.

- HTML5 Tag Check

The existing HTML4 malware detection technology can be bypassed using the newly created tags or attributes. For example, XSS (Cross Site Scripting), which was the big issue in OWASP 2013, is more likely to use new HTML5 tags such as video and audio[14]. This function checks the use of such risky new HTML5 tags. A detailed weight factor can be assigned to each new tag to measure the risk score according to the flow of attack. The score is calculated as follows:

$$Score1 = \sum_{i=0}^{N} Tag\ W_i \begin{cases} N = \text{The Count of Found Tag} \\ i = 0, \ldots, N \\ Tag\ W = \text{The Weight of Found Tag} \end{cases}$$

- JavaScript API Check

JavaScript API contains many functions that can be abused by malicious scripts. They include eval, document.write, and setTimeout[15]. The final behavior of a malicious script is completed by the system API accessing the File I/O, DOM, and Network[16]. As such, potential risk can be detected by inspecting the use of such functions. A weight factor can be assigned to each JavaScript API function for more detailed risk analysis. The score is calculated as follows:

$$Score2 = \sum_{i=0}^{N} API\ W_i \begin{cases} N = \text{The Count of Found API} \\ i = 0, \ldots, N \\ API\ W = \text{The Weight of Found API} \end{cases}$$

- Obfuscation Check

The obfuscation technique is frequently used by malicious scripts to bypass the signature-based detection, and more than 70% of malicious scripts reportedly used the obfuscation technique[15][17]. It is difficult for signature-based detection to find such obfuscated script because there are so many different types, and new obfuscation techniques continue to be found[13]. As such, static analysis only judges the possibility of obfuscation; dynamic analysis determines whether the behavior is malicious.

Obfuscation is judged in three factors. First, entropy[18] measures the distribution rates of all characters and checks how evenly they are spread.

$$E(B) = \sum_{i=1}^{N} \left(\frac{b_i}{T}\right)\log\left(\frac{b_i}{T}\right) \begin{cases} B = b_i,\ i = 0, 1, \ldots, N \\ T = \sum_{i=1}^{N} b_i \end{cases}$$

Here, B is the set of all bytes, and b i is the individual byte in the whole document. Second, the n-gram entropy rate calculates the distributions of special characters (punctuation, symbol, etc.) and measures their ratios to the total entropy.

$$R(S) = \frac{E(S)}{E(B)} * 100,\ \text{S=\{ 0x21-0x2f, 0x3A-0x40, 0x5b-0x5f, 0x7b-0x7e \}}$$

The characters included in S are the special characters including any of "!"#$%&'{}*+,-./:;<=>?@[\]^_'{|}~". This factor represents the relative distribution of special characters that are frequently used in obfuscated statements. The last factor word size[18] is used to check character strings longer than a specific size in the script code to suspect them as obfuscated statements. Each of the three factors above is assigned a weight factor and a critical value to identify obfuscation as needed. Table 2 below shows the result of an obfuscation test.

(Table 2) Result of Obfuscation Check

Source Type	Sample Number	Detection Number
Base62/Base64/ Packed Encode	124	107(86.2%)
Not Encode	344	58(16.8%)

The obfuscation test above reveals 13.8% false negative and 16.8% false positive. Therefore, this test is used only to identify scripts suspected of being obfuscated. T_E, T_R, and T_W are the critical values of Entropy, N-gram Entropy Rate, and Word Size, respectively. If a factor exceeds its respective threshold value, the respective weight factor -- W_E, W_R, or W_W -- is reflected on the score.

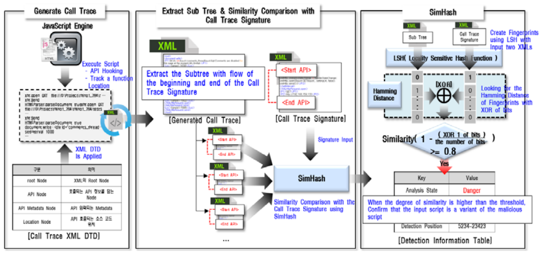
$$Score3 = (E(B) \geq T_E)?\ W_e + (R(S) \geq T_R)?\ W_R + (WordSize \geq T_W)?\ W_w$$

Malicious risk scanning has a different weight factor for each of the above test scores. For example, a higher weight factor may be assigned to the HTML5 tag check if the threat of HTML5 attacks increases; obfuscation check may be assigned a higher weight factor if there is higher occurrence of obfuscation in the malicious code. The weigh factors help users flexibly cope with risk factors. The final risk score is calculated as follows:

$$RiskScore = Score1 * W_1 + Score2 * W_2 + Score3 * W_3$$

3.3 Dynamic Analysis

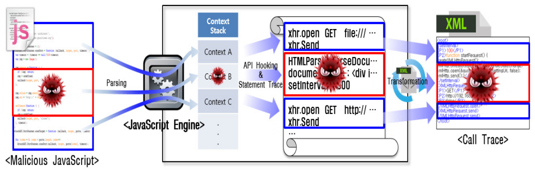
The script suspected to be malicious by the malicious risk scanning is run in the JavaScript engine to track the API flow and analyze the behavior. The API flow data are recorded in the aforementioned call trace, and malicious scripts are detected by analyzing the similarity with the call trace signature. Figure 5 shows the detailed work flow of a dynamic analysis.


Generate Call Trace
JavaScript Engine
Execute Script
- API Hooking
- Track a function Location
XML DTD Is Applied
[Call Trace XML DTD]
Extract Sub Tree & Similarity Comparison with Call Trace Signature
XML
Extract the Subtree with flow of the beginning and end of the Call Trace Signature
[Generated Call Trace]
<Start API>
<End API>
[Call Trace Signature]
Signature Input
SimHash
Similarity Comparison with the Call Trace Signature using SimHash
SimHash
Sub Tree
Call Trace Signature
Create Fingerprints using LSH with Input two XMLs
LSH(Locality Sensitive Hash Function)
Hamming Distance
[XOR]
Looking for the Hamming Distance of Fingerprints with XOR of bits
Similarity(1 - (XOR 1 of bits) / the number of bits) >= 0.8
Yes
Key
Value
Analysis State
Danger
When the degree of similarity is higher than the threshold, Confirm that the input script is a variant of the malicious script
Detection Position
5234-23423
[Detection Information Table]

(Figure 5) Dynamic Analysis Work Flow

3.3.1 Call Trace Generation

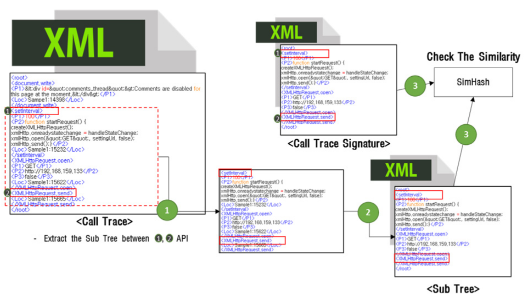
As described above, call trace is the recorded data of API functions that are sequentially called when the JavaScript engine analyzes the script syntaxes and are executed. The API record data include the names of native APIs (JavaScript API and DOM API) and the parameter data inputted to this API. The reason only native APIs are recorded is that the user-defined functions or data inputted to those functions cannot be used as the criteria for identifying the malicious behavior because they can be easily altered by users. The most important reason is that a malicious behavior is completed by the native API that can access the system; the input data determining the direction of malicious behaviors must eventually be restored to their input data form suitable for the API when the native API is executed even when the data were transformed or obfuscated[1][2][15][16]. We also tracked the location data in the original script code and recorded it in the call trace so that they can be used for removing the malicious code and generating the signature in the future. It uses the interpreter nature of the JavaScript engine analyzing the syntax in units of character string and executing it immediately. The engine always saves the location data of the original code in order to read the next character string. Figure 6 shows the details of work flow call trace creation.



(Figure 6) Process of Creating the Call Trace

3.3.1 Call Trace Match

The API flows as shown in Figure 6 because of the execution logic of the JavaScript code defined in ECMA-262 [4]. Only one JavaScript execution context[4] can be executed; other contexts are stored in the stack. If there are multiple script syntaxes, a script code will be on standby until the preceding script code execution is completed, so the sequential API flow is created. Therefore, scripts that generate similar malicious behaviors also have similar API flows. The call trace match uses it to detect the malicious script.



(Figure 7) Process of Call Trace Match

As shown in Figure 7 above, a call trace has the beginning ① and end ② function of an API flow. They generally correspond to the beginning and end of a behavior. The call trace match checks if there is a section that ends with ① and ② in a call trace containing the execution data of script codes, extracts the section into a sub tree, and compares it with the call trace signature. The reason for the process is the nature of SimHash wherein the result similarity analysis is more accurate when the sizes of the compared data are similar[11].

$$Sim_{SimHash} = 1 - \frac{Hamming\,Distance(h,h')}{b}$$

The similarity between a sub tree and a call trace signature is checked by the formula above. Here, b means the bit length of h,h' as the fingerprint created by SimHash. The number of bits -- whose XOR (Exclusive OR) is 1 -- between arrays of bits of fingerprints is then obtained. It becomes the Hamming distance [12]. If the final similarity is above the threshold value, we judge the script that created the sub tree to be a variant of a malicious script used for creating the call trace signature. If not, we judge the script to be in safe status. If a script is judged to be malicious, the we collect the location data of the malicious part from the call trace and creates the domain data of the malicious script code. The domain data will be used to generate the signature and remove the malicious code.

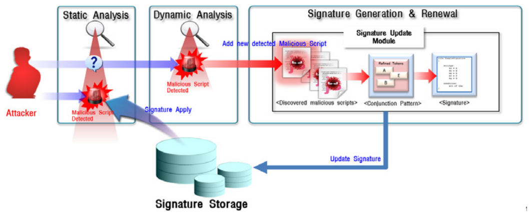
(Table 2) Result of Dynamic Analysis

Malicious Type (Obfuscation)	XHR Dos	Hash Dos	IP Scan	Port Scan	Worker Dos	File API	Geolocation	Total
Total Number (A)	240	240	240	240	240	240	240	1680
Detection Number (B)	224	223	209	209	221	178	192	1456
Detection Rate (B/A)*100	93.33%	92.92%	87.08%	87.08%	92.08%	74.17%	80.00%	86.67%

As shown in Table 2, The Accuracy of the Dynamic Analysis is enough high to detect the malicious in the obfuca. The Process cost of the Dynamic Analysis is expensive, but The Role of Dynamic Analysis is very necessary for detecting the embedded malicious and updating the signature of the static analysis from the detection result.

3.4 Signature Update

If a malicious script is detected by the dynamic analysis, a signature for static analysis can be generated using the detected data. The detected malicious script and the domain data are integrated and grouped with the existing malicious script data to extract the token corresponding to the conjunction pattern. It is then created as the signature of the YARA rule set that stores it. Figure 8 shows the detailed work flow of a signature update.

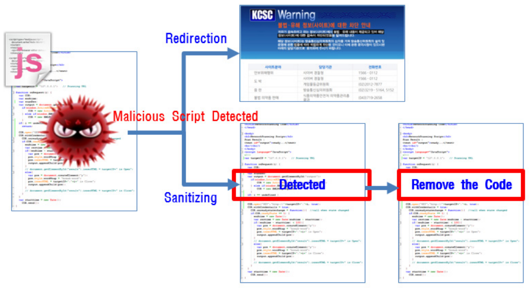


(Figure 8) Process of Signature Update

Using the updated signature, static analysis can quickly detect the same attacks. Learning through detection of transformed/obfuscated malicious script is the key function of the Analysis Engine.

3.5 Post-Process

The purpose of this job is to remove the malicious code using the script code and domain data resulting from the earlier static and dynamic analyses without interfering with the execution of safe scripts. It provides two methods. The first method is to switch the key API generating the malicious behavior with the script code in the malicious domain of the original script code with an API without any functionality. For example, the part that calls the "send" function -- which is the key API generating the traffic in the XHR Dos attack script -- may be switched with a meaningless function called "sanitizing". That way, the traffic attack can be blocked without changing the script code. The second method involves redirecting to a blocking page when the HTML page has a malicious script. Since it also blocks script functions that are not malicious, it is used for cases wherein code removal or detection is difficult.



(Figure 9) Post-Process

4. Conclusion

Fore real-time processing, it is important to minimize the time needed for JavaScript interpretation and to update the signature continuously to improve processing speed and analysis accuracy. As such, the Analysis Engine consists of static analysis, which quickly detects the malicious script that uses the signatures, and behavior-based dynamic analysis that updates the signatures. Static analysis matches the pattern of the token stored in the signature with the unprocessed script code to process the script quickly, whereas dynamic analysis analyzes the behavior using the API flow to detect the transformed or obfuscated malicious script. The result of detection by dynamic analysis is reflected on the generated signature, and the signature aids in the quick detection of the repeated attack; thus enabling real-time processing.

Table 3 shows the result of accessing 6 major domestic sites with clients and processing 1Gbps of the HTTP traffic.

(Table 3) Result of the Analysis Engine

Quantitative Performance Index	Significance
Web Access Delay Time: 2.658 sec.	Real-time processing was proven to be suitable since the delay time was kept to less than 3 seconds so as not to affect user-friendliness.
Detection Rate: 89.06%	The performance level was proven to be at the commercial level since 1995 out of 2240 test samples were detected (non-obfuscated code: 539 /obfuscated code: 1456).

The direction for future studies is to develop functions to detect totally new malicious scripts using the call trace signature, which is an important detection element of the dynamic analysis, and to advance the technology to cope with the vulnerabilities of HTML5 better. Moreover, the signature distribution and management function will be improved further for commercialization.

Reference

[1] Lu, Gen, and Saumya Debray. "Automatic simplification of obfuscated JavaScript code: A semantics-based approach." Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on. IEEE, 2012, http://www.cs.arizona.edu/~genlu/pub/js-deobf-web.pdf.

[2] Lee, Jusuk, Kyoochang Jeong, and Heejo Lee. "Detecting metamorphic malwares using code graphs." Proceedings of the 2010 ACM symposium on applied computing. ACM, 2010, http://ccs.korea.ac.kr/pds/SAC10.pdf

[3] Saxena, Prateek, et al. "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications." NDSS. 2010, http://www.andrew.cmu.edu/ user/ppoosank/papers/FLAX.pdf

[4] ECMA-262 "EMCAScript Langauge Specification", http://www.ecma-international.org/publications/standards /Ecma-262.htm

[5] Chugh, Ravi, et al. "Staged information flow for JavaScript." ACM Sigplan Notices. Vol. 44. No. 6. ACM, 2009, http://cseweb.ucsd.edu/~lerner/papers/pldi09-sif.pdf

[6] Xie, Yichen, and Alex Aiken. "Static Detection of Security Vulnerabilities in Scripting Languages." USENIX Security. Vol. 6. 2006, https://www.usenix.org/ legacy/event/sec06/tech/full_papers/xie/xie_html/

[7] Chowdhary, Mahak, Shrutika Suri, and Mansi Bhutani. "Comparative Study of Intrusion Detection System." (2014), http://www.ijcseonline.org/pub_paper/IJCSE-00229.pdf

[8] Newsome, James, Brad Karp, and Dawn Song. "Polygraph: Automatically generating signatures for polymorphic worms." Security and Privacy, 2005 IEEE Symposium on. IEEE, 2005, https://cse.sc.edu//~huangct/CSCE715F10/715presentation10.pdf

[9] YARA Documentation, http://yara.readthedocs.org/en/latest/index.html

[10] Document Object Model, http://www.w3.org/DOM/

[11] Charikar, Moses S. "Similarity estimation techniques from rounding algorithms." Proceedings of the thiry-fourth annual ACM symposium on Theory of computing. ACM, 2002, http://www.cs.princeton.edu /courses/archive/spring04/cos598B/bib/CharikarEstim.pdf

[12] Hamming, Richard W. "Error detecting and error correcting codes." Bell System technical journal 29.2 (1950): 147-160, http://www.lee.eng.uerj.br/~gil/redesII/hamming.pdf

[13] Linn, Cullen, and Saumya Debray. "Obfuscation of executable code to improve resistance to static disassembly." Proceedings of the 10th ACM conference on Computer and communications security. ACM, 2003, https://www.cs.arizona.edu/solar/papers/CCS2003.pdf

[14] Dong, Guowei, et al. "Detecting cross site scripting vulnerabilities introduced by HTML5." Computer Science and Software Engineering (JCSSE), 2014 11th International Joint Conference on. IEEE, 2014,

[15] Xu, Wei, Fangfang Zhang, and Sencun Zhu. "JStill: Mostly static detection of obfuscated malicious javascript code." Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013, http://www.cse.psu.edu/~sxz16/papers/JStill.pdf

[16] Fan, Wenqing, Xue Lei, and Jing An. "Obfuscated Malicious Code Detection with Path Condition Analysis." Journal of Networks 9.5 (2014): 1208-1214, http://ojs.academypublisher.com/index.php/jnw/article/viewFile/jnw0905120881214/9256

[17] Xu, Wei, Fangfang Zhang, and Sencun Zhu. "The power of obfuscation techniques in malicious JavaScript code: A measurement study." Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on. IEEE, 2012, http://www.cse.psu.edu/~sxz16/papers/malware.pdf

[18] Choi, YoungHan, et al. "Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis." Future Generation Information

Technology. Springer Berlin Heidelberg, http://www.sersc.org/journals/IJSIA/vol4_no2_2010/2.pdf 2009. 160-172

[19] A. Rajaraman and J. Ullman (2010). "Mining of Massive Datasets, Ch. 3.", http://www.langtoninfo.com/web_content/9781107015357_frontmatter.pdf

◐ 저 자 소 개 ◑

추 현 록 (Hyun-Lock Choo)
2007년 부경대학교 컴퓨터멀티미디어공학과(공학사)
2015년 성균관대학교 정보통신대학원 정보보호학과 석사과정
2014~현재 한국인터넷진흥원 선임 연구원
관심분야 : Network/Web Security, Cloud Service, Big Data Analysis, IoT, etc.
E-mail : hlchu@kisa.or.kr

정 종 훈 (Jong-Hun Jung)
2003년 한국해양대학교 제어컴퓨터공학과(공학사)
2010년 성균관대학교 정보통신대학원 정보보호학과(석사)
2013~현재 한국인터넷진흥원 책임 연구원
관심분야 : Cryptography, Data Security&Privacy, Mobile Security, Web Security, Cloud & IoT Security, etc.
E-mail : jjh2640@kisa.or.kr

김 환 국 (Hwan-Kuk Kim)
2000년 한국항공대학교 컴퓨터공학과(공학석사)
2011년 고려대학교 대학원 경정보공학과(공학박사)
2007~현재 한국인터넷진흥원 팀장
관심분야 : Information Security Management, VoIP security. 4G Security, Network Security, etc.
E-mail : rinyfeel@kisa.or.kr