Journal of The Korea Society of Computer and Information
Vol. 24 No. 5, pp. 27-33, May 2019

www.ksci.re.kr
https://doi.org/10.9708/jksci.2019.24.05.027

A Discovery System of Malicious Javascript URLs hidden in Web Source Code Files

Hweerang Park*, Sang-Il Cho**, Jungkyu Park***, Youngho Cho***

Abstract

One of serious security threats is a botnet-based attack. A botnet in general consists of numerous bots, which are computing devices with networking function, such as personal computers, smartphones, or tiny IoT sensor devices compromised by malicious codes or attackers. Such botnets can launch various serious cyber-attacks like DDoS attacks, propagating mal-wares, and spreading spam e-mails over the network. To establish a botnet, attackers usually inject malicious URLs into web source codes stealthily by using data hiding methods like Javascript obfuscation techniques to avoid being discovered by traditional security systems such as Firewall, IPS(Intrusion Prevention System) or IDS(Intrusion Detection System). Meanwhile, it is non-trivial work in practice for software developers to manually find such malicious URLs which are hidden in numerous web source codes stored in web servers. In this paper, we propose a security defense system to discover such suspicious, malicious URLs hidden in web source codes, and present experiment results that show its discovery performance. In particular, based on our experiment results, our proposed system discovered 100% of URLs hidden by Javascript encoding obfuscation within sample web source files.

▸Keyword: Hidden URL Discovery, Web Defacement Attack, Javascript Obfuscation, Network Security

I. Introduction

최근 발생하는 심각한 사이버위협 중에 하나는 봇넷(botnet) 기반의 사이버공격이다. 봇넷은 수많은 봇(bot)들로 구성되는데, 봇은 악성코드에 감염되어 공격자의 통제를 받는 네트워킹 기능을 갖춘 컴퓨팅 장치(PC, 스마트폰, IoT 센서 등)를 말한다. 이러한 봇넷은 분산 서비스 거부공격(DDoS Attacks), 멀웨어(Malware)와 같은 악성프로그램 배포, 스팸메일 유포 등과 같은 다양한 사이버공격을 수행한다[1-2].

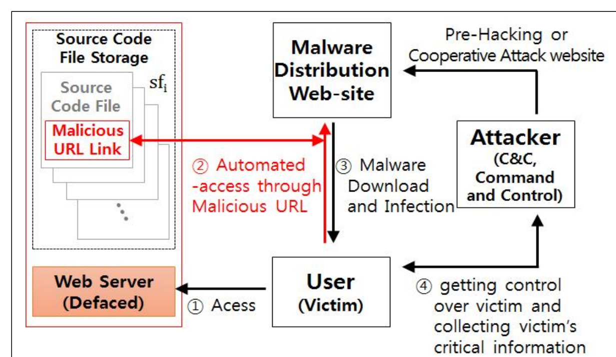


Fig. 1. Executing drive-by download attacks by using malicious URL hidden in web source codes

• First Author: Hweerang Park, Corresponding Author: Youngho Cho
*Heerang Park (sharku7@gmail.com), Air Force Operation Command, Republic of Korea
**Sang-Il Cho (csi.gecoman@gmail.com), Air Force Cyber Operations Center, Republic of Korea
***Jungkyu Park (lala16239@gmail.com), Dept. of National Defense Science, Korea National Defense University
***Youngho Cho (youngho@kndu.ac.kr), Dept. of National Defense Science, Korea National Defense University
• Received: 2019. 04. 02, Revised: 2019. 05. 08, Accepted: 2019. 05. 08.
• This paper is a revised and expanded version of a paper entitled "Detection System of Hidden Javascript URLs in Web Source Codes" presented at the 2019 winter Conference of The Korea Society of Computer and Information

이러한 봇넷을 구축하는 대표적인 방법은 홈페이지 변조 공격을 통해 이루어지는데, 예를 들어, Fig. 1에서와 같이 공격자는 홈페이지(또는 웹 소스코드)에 은밀히 악성 URL을 삽입해 놓으면 홈페이지에 접속하는 사용자(victim)의 PC는 자신도 모르게 해당 URL과 연결된 악성코드 유포사이트로부터 악성프로그램 (또는 악성코드)를 다운받아 자동으로 감염된다. 이렇게 감염된 PC들은 공격자(Command & Control)의 통제를 받는 봇넷을 형성한 후에DDoS 공격에 동원되거나, PC에 저장된 개인정보 또는 비밀자료 등의 중요한 정보를 지속적으로 공격자에게 유출한다[2-5].

한편, 공격자는 홈페이지에 악성 URL을 삽입할 때 방화벽(Firewall), 침입방지체계(IPS) 등의 기존의 정보보호체계로는 탐지가 어렵도록 Javascript 난독화 기법(obfuscation technique)을 사용하는데, 이때 가장 많이 사용되는 방법이 Javascript encoding obfuscation 기법이다[6-7].

따라서, 악성 URL을 은밀히 삽입하는 웹 변조 공격에 대해 효과적으로 대응하는 것은 정보보호 및 사이버방호를 위해 매우 중요하다. 본 논문에서는 이러한 사이버공격에 효과적으로 대응하기 위해 웹 소스코드에 은닉된 악성 Javascript URL들에 대한 일괄 점검체계를 제안한다.

이후 논문의 구성은 다음과 같다. II장에서는 웹 소스코드에 악성코드를 은닉 삽입하는 방법을 소개하고 기존의 대응방법과 한계점을 기술한다. III장에서는 은닉된 악성 URL의 점검체계를 제안하고, IV장에서는 구현된 점검체계의 점검정확성과 점검속도에 대한 실험결과를 제시한 후, V장에서 결론을 맺는다.

II. Background and Related Works

1. Various Methods of hiding URLs into Web Source Codes

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4  <script type="text/javascript">
5  var _0x2f81=[
   "\x3C\x69\x66\x72\x61\x6D\x65\x20\x66\
   x72\x61\x6D\x65\x62\x6F\x72\x64\x65\x7
   2\x3D\x22\x31\x22\x20\x77\x69\x64\x74\
   x68\x3D\x22\x32\x30\x30\x22\x20\x68\x6
   5\x69\x67\x68\x74\x3D\x22\x31\x30\x30\
   x22\x20\x73\x72\x63\x3D\x22\x68\x74\x7
   4\x70\x3A\x2F\x2F\x77\x77\x77\x2E\x68\
   x69\x64\x64\x65\x6E\x2E\x6E\x6F\x72\x6
   D\x61\x6C\x31\x2E\x6E\x65\x74\x22\x3E\
   x3C\x2F\x69\x66\x72\x61\x6D\x65\x3E",
   "\x77\x72\x69\x74\x65"];x= _0x2f81[0
   ];document[_0x2f81[1]](x)
6  </script>
7  </body>
8  </html>
9
```

Hidden Malicious URL :
http://caubr.gov.br/sh.txt

Fig. 2. An example of hidden malicious URL generated by Javascript encoding obfuscation

악성 URL을 웹 소스코드에 그대로 보이도록 삽입되면 관리자에게 쉽게 발견되어 제거되기 때문에, 관리자에게 쉽게 발견되지 않은 상태에서 오랫동안 공격을 수행할 수 있도록 은밀하게 삽입된다. 이때, 가장 많이 사용되는 방법이 Javascript 난독화(obfuscation) 기법이다. 난독화 기법은 원래 소스코드의 악의적인 분석으로부터 보호를 위해 고안되었으나, 최근에는 공격자가 자신의 공격행위를 은폐할 목적으로 악용되기도 한다. 난독화 기법은 encoding obfuscation, randomization obfuscation, data obfuscation으로 분류할 수 있으며, 이 중에서 Fig. 2와 같이 악성 URL에 해당하는 Javascript code를 16진수로 변환하여 소스코드에 삽입하는 encoding obfuscation 방법이 가장 많이 사용된다[6].

Javascript 난독화 기법을 사용하면 악성 URL을 생성해내는 Javascript code가 서버에 있을 때에는 비활성화되어 있기 때문에, 난독화 해석 기능이 없는 서버측 정보보호체계(서버용 백신, 방화벽, IPS 등)에는 탐지되지 않는다. 해당 Javascript code가 담긴 웹 페이지에 사용자가 접속했을 때, 해당 code가 사용자의 웹 브라우저에 의해 처리된 후에야 비로소 악성 URL로 활성화된다. 이후, 사용자도 모르게 악성코드를 다운로드 받아 은밀히 설치하기 때문에 악성코드에 의한 피해가 발생하여도 이것이 Javascript 악성 URL과 연관이 있다는 것을 분석해 내는 것은 일반 사용자에게는 매우 어려운 일이다[8].

2. Existing Defending Approaches

Javascript obfuscation 기법 기반의 악성코드 탐지에 관한 연구로는 소스코드의 string pattern 분석이나 머신러닝 기법을 활용한 것이 있다[7][9]. 이들은 난독화된 Javascript code들 중에서 악성코드인 것들만 선별하려고 노력하였으나, 탐지 정확도(detection accuracy)가 완전하지 않고, 사용자 PC(Web browser 등)에 별도의 점검체계가 설치되어야 한다는 점, 그리고 홈페이지 전체가 아닌 사용자가 방문한 웹 페이지에 대해서만 점검이 이루어지는 제한점이 있다.

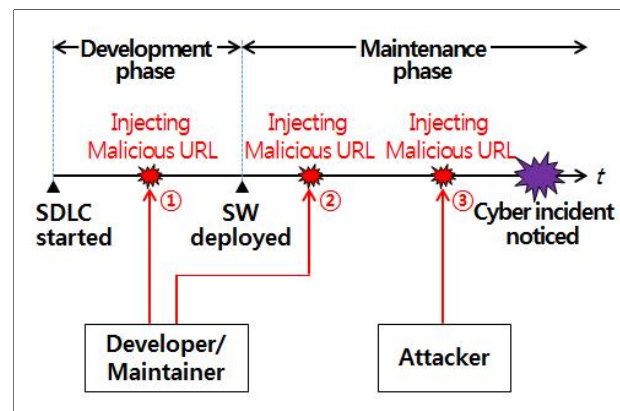


Fig. 3. Timeline showing when hidden malicious URLs can be injected into an software(source code) during its SDLC(Software Development Life Cycle)

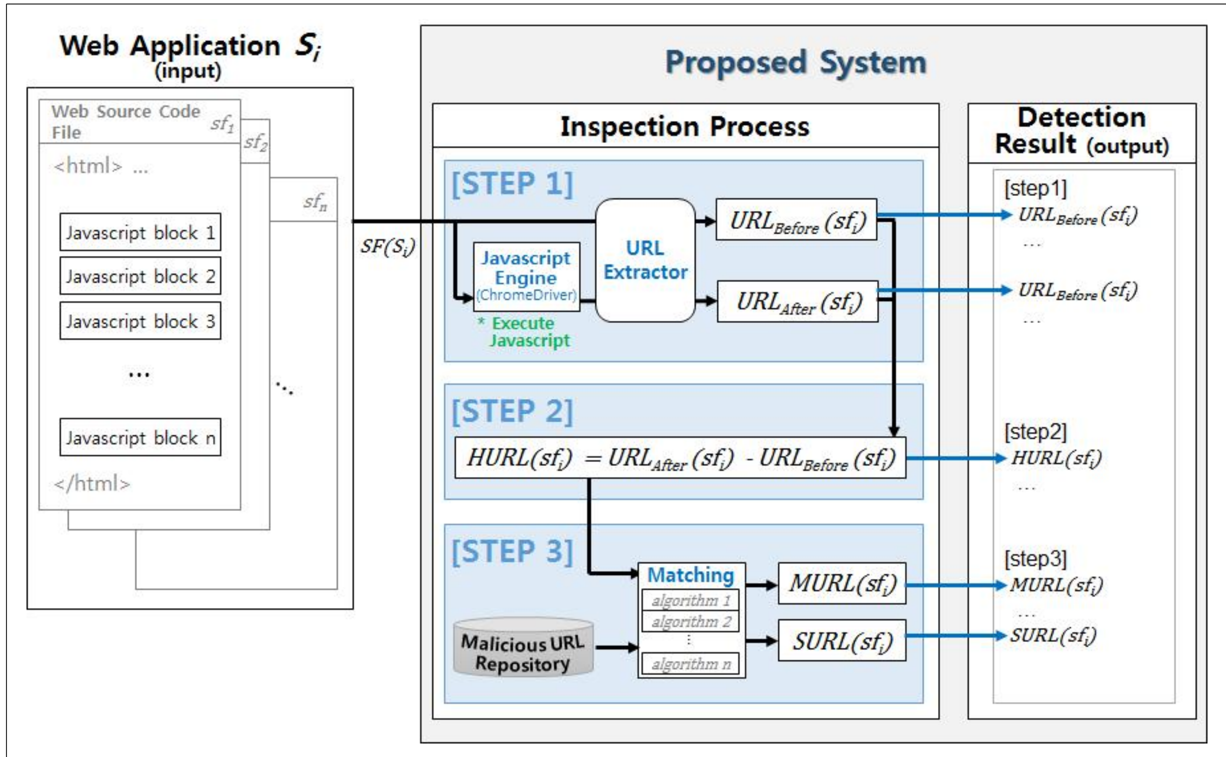


Fig. 4. The design architecture of our proposed system

한편, 악성 URL이 홈페이지 소스코드에 은밀히 삽입되는 시점은 두 가지의 경우이다 (Fig. 3 참조). 참고로, 본 연구에서는 소스코드에 명시적으로 노출된 악성 URL에 대한 점검은 어렵지 않은 것으로 보고 연구범위에서 제외한다.

우선, 악성 URL은 웹 응용체계의 개발단계에서 소스코드에 삽입될 수 있다. 예를 들어, 개발자가 공격자일 경우 의도적으로 삽입하는 경우와 개발자가 개발을 용이하게 수행하기 위해 인터넷에 공개된 악성코드에 감염된 소스코드를 의심없이 사용(copy & paste)하는 경우이다. 본 논문에서는 연구범위의 명확성을 위해 전자는 고려하지 않는다.

다음으로, 악성 URL은 웹 응용체계가 개발되어 서버에 설치된 이후 정상운영 중에 삽입될 수 있다. 예를 들어, 앞의 설명과 유사하게 체계개발자(또는 유지보수자)가 응용체계를 유지보수하는 과정에서 외부 소스코드 (악성코드 감염)를 의심 없이 사용하는 경우와 외부공격자가 서버나 웹 응용체계의 취약점을 통해 악성 URL을 은밀히 삽입하는 경우이다.

악성 URL이 은밀히 삽입되는 시점을 봤을 때, 악성 URL에 의한 피해를 최소화하기 위한 소스코드 점검 시점을 다음의 세 가지 시점으로 본다: ① 개발 후 서버에 설치되는 시점, ② 유지보수가 완료된 후 서버에 설치되는 시점, ③ 가능한 실시간으로 또는 주기적으로 서버의 소스코드를 대상으로 점검하는 것이다; 앞의 두 경우는 개발자의 실수에 대응하는 차원이고 마지막의 경우는 공격자에 의한 삽입에 대응하여 피해를 최소화하기 위해서이다. 또한, 점검대상은 개발 서버(또는 PC)와 운영 서버의 전체 웹 소스코드를 대상으로 하여야 한다.

따라서, 본 논문에서는 웹 소스코드에 은닉된 Javascript 악성 URL의 자동점검체계를 제안한다. 제안체계는 웹 응용체계의 소스코드 파일들에 은닉된 URL을 자동으로 발견한다. 또한, 악성 URL DB와 유사도 측정 알고리즘(matching algorithm)을 활용하여 악성 URL을 추가로 점검한다.

II. Design of Proposed System

본 논문에서 제안된 체계는 1) Javascript 블록 분석 및 URL 추출(STEP 1), 2) 은닉 URL 탐지(STEP 2) 악성 및 의심 URL 탐지(STEP 3)의 3단계로 동작하여 웹 소스코드에 숨겨진 악성 그리고 의심 URL들을 일괄 점검한다. 각 단계의 구체적인 동작은 다음과 같다(Fig. 4 참조).

1. (STEP 1) Extracting URLs from Javascript blocks in web source codes

제안 점검체계의 동작을 설명을 위해, 우선 점검 대상인 웹 응용체계(Web Application)를 WA_i라 하자. WA_i를 구성하는 특정 웹 소스파일을 sf_j라 하고, WA_i를 구성하는 전체 소스파일 집합을 $SF(WA_i)$라 하자. 이때, 소스파일의 총 개수가 n이면, $SF(WA_i) = \{sf_1, sf_2, \cdots, sf_n\}$이고 $|SF(WA_i)| = n$이다.

점검체계는 STEP 1에서 점검대상인 $SF(WA_i)$를 입력으로 하여 각 소스파일 sf_j에 Javascript block이 있을 경우 해당 block에 명시적으로 코딩된 URL 목록을 모두 추출한다. 이때, 추출된

URL들의 집합을 $URL_{Before}(sf_j)$라 하자. 다음으로, Javascript code 분석기(Javascript Engine)를 활용하여 sfj의 Javascript block을 parsing 후 다시 한 번 전체 URL 목록을 추출한다. 이때, 추출된 URL들의 집합을 $URL_{After}(sf_j)$라 하자. $URL_{Before}(sf_j)$과 $URL_{After}(sf_j)$는 각각 정적분석(static analysis)과 동적분석(dynamic analysis) 방법에 의해 추출된 URL 목록이다. Javascript engine은 Javascript를 지원하는 모든 웹 브라우저에 기본적으로 내장되는데, 본 연구에서는 웹 브라우저와 독립적인 별도의 점검도구로 개발하기 위해서 Javascript API를 제공하고 stand-alone으로 동작하는 ChromeDriver[10]을 활용하였다. 이에 대해서는 Ⅲ장에서 자세히 설명한다.

2. (STEP 2) Discovering Hidden URLs

STEP 1에서 추출된 $URL_{Before}(sf_j)$과 $URL_{After}(sf_j)$를 활용하면 소스코드에 숨겨진 URL들을 쉽게 찾을 수 있다. 각 소스코드 sf_j에 은닉된 Javascript URL 목록 $HURL(sf_j)$을 아래 수식 (1)과 같이 생성한다.

$$HURL(sf_j) = URL_{After}(sf_j) - URL_{Before}(sf_j) \quad (1)$$

이러한 방법으로 WA_i의 모든 소스파일에 대해 점검하면, WA_i에 은닉된 Javascript URL 목록 $HURL(WA_i)$은 (2)와 같다.

$$\begin{aligned} HURL(WA_i) &= HURL(sf_1) \cup \dots \cup HURL(sf_n) \\ &= \bigcup_{j=1}^{n} HURL(sf_j) \end{aligned} \quad (2)$$

3. (STEP 3) Discovering malicious and suspicious URLs

마지막 단계인 STEP 3에서는 앞에서 탐지한 은닉 URL 목록인 $HURL(WA_i)$로부터 알려진 악성 URL 목록을 추가로 점검한다. 이를 위해서, 제안체계에서는 악성 URL DB를 구축하고, 유사도 측정 알고리즘(matching algorithm)을 활용하여 웹 정보체계의 소스코드에 악성 URL의 포함 여부를 점검한다. 이때, 점검된 악성 URL 목록을 $MURL(WA_i)$이라 하자. 한편, $MURL(WA_i)$의 각 URL_k에 대해 matching algorithm으로 유사도를 측정한 결과값(Similarity Value) SV_k는 0과 1 사이의 값으로 나타낼 수 있으며, SV_k와 관리자가 설정하는 두 개의 경계값 T_1(하위 경계값)과 T_2(상위 경계값)에 따라 해당 URL_k에 대해서 (3)과 같이 악성(m : malicious), 의심(s : suspicious), 정상(n : normal)으로 판정할 수 있다(이때, $0 < T_1 < T_2 < 1$).

$$D(URL_k) = \begin{cases} m & \text{if } T_2 \le SV_k \le 1 \\ s & \text{if } T_1 \le SV_k < T_2 \\ n & \text{if } 0 \le SV_k < T_1 \end{cases} \quad (3)$$

즉, 유사도 값이 1보다 비교한 결과 100% 일치하지 않으나 높은 수준으로 유사한 경우에는 추가적인 조사가 필요하므로 의심(s)으로 분류하고 의심 URL 목록 $SURL(S_i)$에 포함하여 관리한다. 두 경계값 T_1과 T_2는 점검체계 운영을 통해 정확도가 향상되도록 관리자에 의해 조정될 수 있다. 제안체계는 알려진(공개된) 유사도 알고리즘이 쉽게 체계에 plug-in 될 수 있는 구조로 설계되며, 유사도 측정 알고리즘으로는 Ratcliff의 pattern matching algorithm[11]과 n-gram[12] 등이 사용될 수 있다. 참고로, 악성 URL DB는 제안체계에 직접 구축하거나 분리된 별도 체계로 구축하여 연동할 수 있다. 악성 URL 목록은 한국인터넷진흥원(KISA)과 같이 신뢰성이 높은 국내･외 정보보호 기관으로부터 제공을 받거나 Zone-H[13]과 같이 사이버 침해사고에 대한 정보를 제공하는 홈페이지를 통해 지속적으로 업데이트하여 관리한다.

아래의 Algorithm 1은 지금까지 설명한 제안체계의 점검 알고리즘의 동작에 대해 Python 스타일로 기술한 의사코드(pseudo code)이다.

[Algorithm 1] Hidden malicious URL Discovery

```
Input : SF(Si), Tlow, Thigh
Output : HURL(Si), MURL(Si), SURL(Si)

for sf in SF(Si) :
  # step 1
  # step 1.1 - static analysis
  URLbefore(sf) ← extract all urls from source file sf

  # step 1.2 - dynamic analysis
  URLafter(sf) ← extract all urls from parsed source
                 file sf by using ChromeDriver

  # step 2
  HURL(sf) = URLafter(sf) - URLbefore(sf)
  HURL(Si) ← HURL(Si) U HURL(sf)

  # step 3
  if |HURL(Si)| != 0 :
     for hurl in HURL(Si) :
        if D(hurl) ∈ [0, Tlow) :
          MURL(Si) ← MURL(Si) U hurl
        if D(hurl) ∈ [Tlow, Thigh) :
          SURL(Si) ← SURL(Si) U hurl
```

III. Implementation and Experiment

1. System Implementation and Main View

점검체계 구현을 위해 핵심 모듈인 웹 소스코드의 Javascript block을 처리하는 Javascript Engine으로는 앞에서 설명한 것과 같이 공개용 소프트웨어인 ChromeDriver[10]를 사용하였다. ChromeDriver는 자동화된 웹페이지 탐색, 사용자 입력과 Javascript 실행 등을 제공하여 웹 브라우저 기반의 테스트 수행에 활용된다. 기타 점검체계의 모듈들은 Python 언어 2.7 버전[14]으로 구현하였다.

Fig. 5. Main Page View of Our Proposed System

Fig.5는 점검체계의 메인화면을 나타내며, 3가지의 주요기능을 제공한다. Source Code Analysis는 점검체계가 설치된 장비에 저장된 점검대상 소스코드들을 점검하여 은닉, 악성, 의심 URL 목록을 제시한다. URL Analysis는 URL 형태로 제공된 소스코드들을 원격으로 점검하여 결과를 제시한다. 마지막으로, DB setting은 점검도구에서 관리하는 악성 URL DB와 점검결과의 저장에 필요한 DB의 생성/수정 등을 할 수 있는 기능이다.

2. Experiment Results

• **실험 목적 및 방법**

다음과 같은 측면에서 제안 체계의 성능을 시험한다.

① 점검 정확성(Discover Accuracy) : 시험을 위해 샘플 소스코드에 Javascript encoding obfuscation 기법으로 URL들을 숨긴 후, 제안 체계가 은닉 URL들과 악성 URL들을 정확히 찾아내는지를 시험한다. 이때, 악성 URL들은 과거 실제 공격에 사용되었던 것을 삽입하였다.

② 점검 수행시간(Discovery Execution Time) : 제안 체계의 점검 수행 시간을 다음의 두 가지 방법으로 측정하여 제시한다. 첫째, 하나의 샘플 파일 내에 은닉 URL들의 개수를 증가시키면서 그에 따라 점검이 완료되는데 소요되는 시간을 측정한다. 둘째는 점검 대상 파일의 개수를 증가시켜가면서 점검이 완료되는데 소요되는 시간을 측정한다.

실험을 위한 샘플 소스코드는 Table 1에서와 같이 6개를 준비하였다(Sample1 ~ 6). 각 샘플 소스코드에는 기본적으로 명시적으로 보이는 50개의 정상 URL들이 있으며, 샘플에 따라 은닉 URL과 악성 URL 개수를 다르게 하였다. Table 1의 Setting에서와 같이, 은닉 URL 개수는 10 ~ 50개까지 증가시켜가며 샘플들에 삽입하였으며 은닉 URL 개수 중 10%는 악성 URL로 삽입하였다. 또한, 다수의 파일에 대한 점검수행 시간 측정을 위해, URL 개수가 가장 많은 Sample 6의 파일 개수를 증가시켜 측정하였다. 실험은 Intel Core i7-3630QM PC (CPU 2.4GHz, RAM 8GBytes)에서 수행되었으며, 웹 기반의 시험환경을 제공하는 Selenium Webdriver[15]를 사용하였다.

Table 1. Test Result of Hidden, Malicious URL Discovery

Test Sample			Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6
Setting	Number of Normal URLs(N)		50	50	50	50	50	50
	Number of hidden URLs (Number of Malicious URLs)		0 (0)	10 (1)	20 (2)	30 (3)	40 (4)	50 (5)
Test result	Discovery Rate(%)	Hidden URL	-	100	100	100	100	100
		Malicious URL	-	100	100	100	100	100
	Discovery Execution Time(sec)		5.46	5.78	6.03	6.26	6.52	6.74

• 시험결과 설명 및 분석

Fig. 6. An example of discovery result (Sample 2)

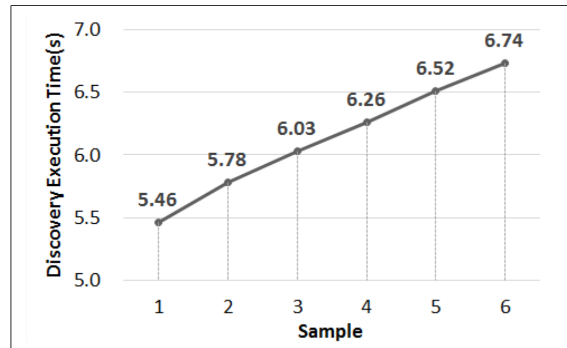


Fig. 7. Discovery Execution Time (Sample 1 ~ 6)

제안 체계로 각 샘플에 대해 점검을 수행하면 점검결과는 Fig.6과 같은 형태로 시현된다. Fig.6은 Sample 2(은닉 URL 10개, 악성 URL 1개 삽입)에 대한 세부 점검결과를 표시한 것이다. II장에서 제안체계의 각 동작 단계에 대해 설명한 것과 같이 Fig. 6 가장 상단의 STEP1의 결과에서는 샘플 소스코드에 대한 정적 및 동적분석 수행을 통해 찾아낸 URLBefore과 URLAfter가 순서대로 표시된다. STEP2에서는 은닉 URL로 최종 점검된 목록(HURL)이 표시되며, 그 아래의 STEP3에서는 추가적인 점검을 통한 악성 URL 점검목록(MURL)과 의심 URL 점검목록(SURL)이 각각 표시된다.

실험을 수행한 결과와 분석내용은 다음과 같다. 먼저, 제안체계의 점검 정확성에 대한 시험결과를 살펴보면, Table 1의 Discovery Rate에서도 볼 수 있듯이 모든 샘플의 은닉 URL들과 악성 URL들은 점검체계에 의해 모두 정확히 발견되었다. 이는 점검체계가 설계 의도에 따라 정적분석과 동적분석을 통해 숨겨진 URL들을 찾아내고 이후 점검체계가 보유한 악성 URL DB를 기반으로 matching algorithm에 의해 숨겨진 악성 URL들을 정확히 탐지한 결과이다. 점검 정확성은 삽입된 은닉/악성 URL 개수에 상관없이 모든 샘플들에 대해 100% 정확히 탐지되었다.

다음으로, 제안체계의 각 샘플들에 대한 점검수행 소요시간을 측정한 결과에 대해 설명한다. 우선, 은닉 URL의 개수가 최대 50개까지 증가될 때 제안체계의 점검수행시간을 측정한 결과를 살펴보면, 은닉 URL 1개를 점검하는데 약 0.02 ~ 0.03 sec 정도가 소요되었으며, 그림 Fig.7에서 볼 수 있듯이 은닉URL개수가 늘어남에 따라 소요시간이 그에 비례하여 선형적으로 증가함을 알 수 있다. 따라서, 하나의 소스코드 파일 내에1,000개의 은닉 URL이 존재하는 경우에도 PC급 점검 장비에서 30 sec 정도 만에 점검이 가능함을 추정할 수 있다. 또한, 점검 대상 소스파일 개수가 증가함에 따라 제안 체계의 점검 수행시간을 측정한 결과를 설명한다. Fig.8은 점검대상 소스코드 파일 개수가 10 ~ 100개로 증가할 때 소요된 점검수행시간을 나타낸 것으로, 파일 개수가 증가할수록 점검 소요시간 역시 그에 비례하여 선형적으로 증가함을 알 수 있으며, 한 개의 파일을 점검할 때 소요된 시간은 약 7초 이내로 측정되었다. 따라서, 실험에 사용된 크기 정도의 소스코드 파일을 1,000개 정도의 대규모로 일괄 검사한다고 가정할 경우에 실험에 사용된 PC급 점검 장비에서 약 2시간 이내에 점검이 가능하다고 추정할 수 있다.

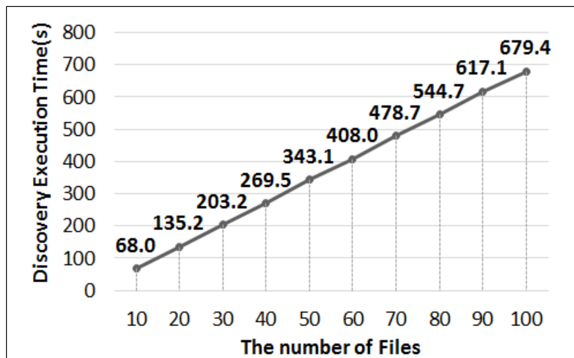


Fig. 8. Discovery Execution Time (# of files : 10 ~ 100)

IV. Conclusion and Future Works

본 논문에서는 Javascript obfuscation 기법을 악용하여 웹 응용체계의 소스코드에 은닉된 악성 Javascript URL들에 대한 일괄 점검체계를 제안한다. 제안체계는 기존 연구와는 다르게 소스코드의 Javascript 블록 분석을 통해 소스코드 상태와 실행 이후의 상태를 비교하여 은닉된 URL들을 명확히 식별한 후, 체계가 보유한 악성 URL Repository를 활용하여 의심스러운 URL을 2차적으로 식별하여 점검에 활용하도로 한다. 즉, 제안 점검체계는 웹 체계의 전체 소스코드에 대해, ① Javascript 블록 분석 및 URL 추출, ② 은닉 URL 탐지, ③ 악성 및 의심 URL 탐지의 3단계로 수행된다. 본 제안체계는 서버에 저장된 대량의 소스코드 파일들을 일괄 점검할 수 있도록 하여 은닉된 악성 URL에 의한 피해를 최소화하도록 한다. III장의 실험결과에서 볼 수 있듯이, 제안체계는 샘플 웹 소스코드에 Javascript encoding obfuscation 기법으로 은닉된 모든 URL들을 100% 탐지하였다.

향후 연구계획은 다음과 같다. 제안된 점검체계는 별도의 점검장비에 설치하여 운영하거나 소스코드가 저장되어 있는 웹 서버에 설치하여 실시간 또는 주기적으로 점검되도록 운영할 수 있다. 이때 점검 주기가 짧을수록 운영서버의 자원 소모가 커질수 있으며 서버의 정상적인 서비스 제공에 영향이 있을 수 있다. 따라서, 점검체계의 점검 주기에 따른 서버의 자원소모(CPU, 메모리 등)를 최소화할 수 있도록 알고리즘에 대한 최적화에 대한 연구를 추가적으로 수행하고, 적정 점검주기의 설정에 대한 연구 또한 함께 수행할 예정이다.

REFERENCES

[1] G Davanzo, E Medvet and A Bartoli, “Anomaly detection technique for a web defacement monitoring service,” Expert Systems with Applications(ESWA), Vol. 38, No. 10, pp.12521-12530, 2011.

[2] S. Khattak, NR. Ramay, KR Khan, AA. Syed, and SA. Khayam, “A Taxonomy of Botnet Behavior, Detection, and Defense,” IEEE Communications Survey & Tutorials, Vol. 16, No. 2, pp.898-924, Second Quarter 2014.

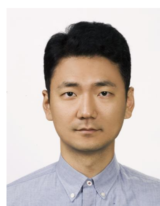
[3] Porras, Phillip, Hassen Saidi, and Vinod Yegneswaran, “A multi-perspective analysis of the storm (peacomm) worm. Technical report, Computer Science Laboratory,” SRI International, 2007.

[4] D. Dagon, “Botnet Detection and Response - The network is the infection,” Copperative Association for Internet Data Analysis DNS-OARC Workshop, July, Vol. 25, 2005.

[5] D. Dagon et al, “A taxonomy of botnet structures,” Twenty-Third Annual Computer Security Applications Conferenece ACSAC 2007, Vol. 36, pp. 325-339, 2007.

[6] W Xu, F Zhang and S Zhu, "The Power of Obfuscation Techniques in Malicious Javascript Code: A Measurement Study," Proceedings of International Conference on Malicious and Unwanted Software, pp.9-16, Oct. 2012.
[7] W Xu, F Zhang and S Zhu, "JStill : Mostly Static Detection of Obfuscated Malicious Javascript Code," Proceedings of the third ACM conference on Data and application security and privacy, pp.117-128, Feb. 2013.
[8] Mavrommatis, Niels Provos Panayiotis, and Moheeb Abu Rajab Fabian Monrose. "All your iframes point to us," Proceedings of USENIX Security Symposium. pp.1-16. 2008.
[9] C Curtsinger, B Livshits, BG Zorn and C Seifert, "Zozzle: Fast and Precise In-Browser Javascript Malware Detection," Proceedings of USENIX Security Symposium, pp.33-48, Aug. 2011.
[10] ChromeDriver, *http://chromedriver.chromium.org/ home.*
[11] JW Ratcliff and DE Metzener, “Pattern matching : The gestalt approach,” Dr. Dobb’s Journal, 13(7) 1998.
[12] N-gram, *https://pypi.org/project/ngram.*
[13] Zohn-H, *http://www.zone-h.org.*
[14] Pyhton, *https://www.python.org.*
[15] Selenium Webdriver, https://www.seleniumhq.org/projects/*webdriver.*

Authors



Hweerang Park received the B.S. degree in physics from Chonnam National University in 2010.He is currently working at the Air Force Operation Command, Republic of Korea. He is interested in network security, trust mechanism, machine learning, cyberspace operation, etc.

Sang-Il Cho received the B.S. degree in information security from Academic Credit Bank System in 2006.He is currently working at Air Force Cyber Operations Center, Republic of Korea. He is interested in network security, security system development, security vulnerability analysis, etc.

Jungkyu Park received the B.S. degree in chemistry and military science from Korea Military Academy in 2011.He is currently pursuing the M.S. degree in computer science and cyberwarfare major at the Korea National Defense University, Republic of Korea. He is interested in network security, image steganography, data hiding, etc.

Youngho Cho received the BS degree in industrial engineering from Korea Air Force Academy and the M.S. degree in computer science and industrial systems engineering from Yonsei University, Republic of Korea, in 1998 and 2006, respectively and the Ph.D. degree in electrical and computer engineering from the University of Maryland, College Park, USA, in 2013. He is currently an Assistant Professor in the Department of Computer Science and Engineering, Korea National Defense University. His research interests include network security, trust mechanism, botnet, steganography, digital forensics, game theory and network security, IoT security, etc.