Similarity Analysis of Shellcodes in Drive-by Download Attack Kits

Manoj Cherukuri
Computer Science
New Mexico Institute of Mining and Technology
Socorro, NM, USA
manoj@cs.nmt.edu

Srinivas Mukkamala
Computer Science
New Mexico Institute of Mining and Technology
Socorro, NM, USA
srinivas@cs.nmt.edu

Dongwan Shin
Computer Science
New Mexico Institute of Mining and Technology
Socorro, NM, USA
doshin@cs.nmt.edu

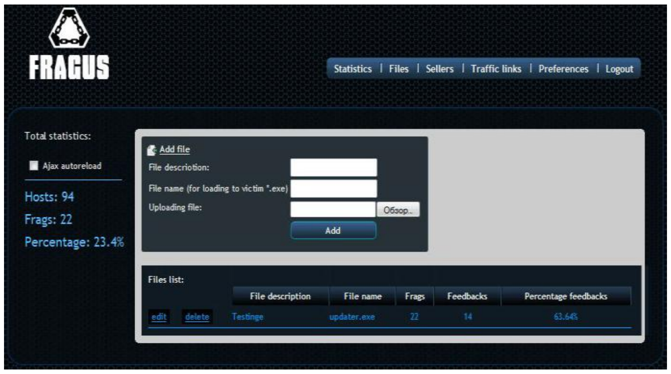
***ABSTRACT* - Drive-by downloads have become the primary attack vehicle for malware distribution in recent years. With the rise of targeted attacks, the vulnerabilities within the cloud based services and web based collaboration frameworks might end up as the principal targets for hosting drive-by download attacks. In this paper, we studied the similarity of the shellcodes among different attack kits. Shellcode is the malicious code used as the payload in drive-by download attacks. Specifically, we collected 15 different drive-by download attack kits and identified shellcodes used in each kit. As the shellcodes are transmitted to the browser as Javascript strings, we measured the similarity between regular strings and shellcodes defined in Javascript. We disassembled the shellcodes and computed the mean of Cosine Similarity, Extended Jaccard Similarity and Pearson Correlation measures based on the frequencies of the opcodes. Our analysis shows that the shellcodes, used as payloads, across different attack kits were similar with other shellcodes and dissimilar with benign Javascript strings. We observe that some of the attack kits released across different years had same shellcodes. The performance of similarity analysis was compared to an emulation based approach and observed reduction of 75% in the analysis time. Based on the results, the similarity measure of the shellcodes could be an effective static mechanism in detecting the shellcode based drive-by download attacks.**

Keywords-Cloud Services Security; Shellcodes Similarity; Web Malware; Collaboration Frameworks Security;

I. INTRODUCTION

Client-side attacks hosted by targeting web applications are ascending. As more and more people use web-based collaborative systems, vulnerabilities within the web-based collaborative systems might put the security of the entire organization at risk. Exploitation of the vulnerabilities and hosting a drive-by download campaign results in the spread of malware across the entire organization and beyond. Hosting of such attack campaigns has become easier with the assistance of the attack kits. Client-side protection ensures the security of the organizations using web-based or cloud-based collaborative systems and services.

Attack kits are a set of exploits packed together to target a set of vulnerabilities in computer systems and applications. Attack kits are also referred to as do-it-yourself (DIY) kits or crimepacks. According to a recent report from Symantec [1], about 61% of the web-based attacks observed until 2011 were from the attack kits and it was believed that a significant portion of the remaining 39% was also from the attack kits but could not be related. The attack kits make the job of launching a web-based attack easier for the attackers, who often do not have any knowledge about the internals of the kits. Attackers use the graphical user interface to select the vulnerability, operating system, and the browser to create a ready-to-use attack webpage. The attackers then compromise legitimate websites to either inject an iframe or redirect the web traffic to the link pointing to the created attack webpage. Most of the attack kits use payloads that bind a shell to the remote machine under the control of the adversaries or that download and execute a malware (drive-by download attack).


FRAGUS
Statistics | Files | Sellers | Traffic links | Preferences | Logout
Add file
File descriotion:
File name (for loading to victim *.exe)
Uploading file:
Обзор..
Add
Ajax autoreload

Total statistics:
Hosts: 94
Frags: 22
Percentage: 23.4%

Files list:

		File description	File name	Frags	Feedbacks	Percentage feedbacks
edit	delete	Testinge	updater.exe	22	14	63.64%

Figure 1. The user interface of Fragus attack kit [2]

Fig. 1 shows the user interface where the attacker can upload the malicious executable that gets installed on the victim's machine (on exploiting the vulnerability successfully).The statistics in the "Files list" shows the performance of the malware files that are used in the attack campaign. The menu bar on the top offers administrative functions like monitoring the statistics, the traffic that was generated towards various links participating in the attack campaign and options to setup the preferences of an attacker. The statistics on the left portion of the image

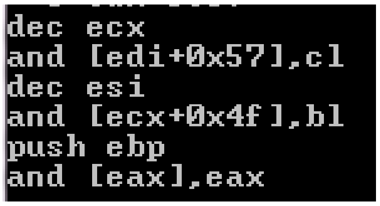
COLLABORATECOM 2012, October 14-17, Pittsburgh, United States
Copyright © 2012 ICST
DOI 10.4108/icst.collaboratecom.2012.250507

shows the hit ratio of the launched attack campaign i.e. the number of hosts that were exploited successfully to the overall number of attempts.

A. *Motivation*

There has been growing interest in research on the topic of drive-by download attacks. The lifecycle of web-based malware was studied in [3][4], and the measurement of web-based malware infection was conducted using billions of webpages crawled by Google during the ten-month period in [5]. About 3 million webpages of the analyzed webpages were found to initiate some sort of drive-by download attacks and approximately 1.3% of the Google search results had at least one result that was labeled malicious. Moreover, a recent study from Symantec shows that an average of 2,305 webpages each day serve malware or other malicious programs. Of the domains that were blocked, 31.5% were registered in the same month which proves the limitation of the blacklists based approaches [6]. Other interesting causal issues such as browser versions, plugins, and patch management were studied in relation to drive-by download attacks in [7][8].

Shellcodes used for drive-by download attacks are often defined as strings in Javascript. The x86 instruction set is so tightly packed that every byte sequence gets disassembled into a set of instructions (i.e. every string gets disassembled into a set of assembly instructions). Fig. 2 shows an example of the x86 assembly code obtained by disassembling a regular string. The tightly packed x86 instruction set poses the challenge of differentiating shellcodes from regular strings for countering such attacks.



```
dec ecx
and [edi+0x57],cl
dec esi
and [ecx+0x4f],bl
push ebp
and [eax],eax
```

Figure 2. The disassembled code generated on disassembling the string "I OWN YOU!"

More interestingly, it is very common to obfuscate shellcodes using Javascript functions, as to evade detection. Fig. 3 shows the obfuscation used by the Eleonore attack kit for hiding the malicious intent. Attackers take advantage of different functions supported by Javascript to host these attacks and evade the static detection approaches relying on signatures. The obfuscated malicious code is initially defined as the text elements in the webpage. The Javascript code accesses the stored text elements using the element ids and de-obfuscates the text using "**sOmC9bC**" and "**decryptor**" functions. The variable names, function names, and element ids are generated randomly to harden the problem of defining a signature for the detection of the attack. The last three lines of the script write the generated new script onto the window. Attackers often employ multiple levels of obfuscation using this technique.

B. *Objective*

In this paper we propose to analyze the similarity of the shellcodes found in different attack kits for the purpose of detecting shellcodes used for drive-by download attacks in an efficient and effective manner. Our approach is based on collection of drive-by download attack kits, extraction of shellcodes from those attack kits, disassembling shellcodes, and measurement of the similarity among those shellcodes. Three different measures for computing the similarity are used: Cosine Similarity, Extended Jaccard Similarity, and Pearson Correlation.

```
//Storing the malicious code in encrypted form as text across different types of elements
<i id=ABnk> f2UCUCCUffPUClUf…</i>
. . .
<u id =l3lFNI5c9>f5UfffU35UCC…</u>

<script>
//Returns the encrypted text from the elements with the corresponding ids
function h0g8Gd2(e7T0W33, HqJ0FCc) {…}
…

//Splits the given string based on the delimiter and forms a new string from the resulting
array, which contains the numerical character codes. Returns the final decrypted code
function decryptor(vlBwpdW)
{
O5Bj2zl = vlBwpdW.split('N');
 for (var i=0;i<O5Bj2zl.length-1;i++) {
  O5Bj2zl[i]++;
  kBVDK1e += INgGEcQ(O5Bj2zl[i]);
 }
 return(kBVDK1e);
}

function sOmC9bC(V99xNCj)
{
//First level decoding of the encrypted text
 Var B59ILwD,LL91CyE,C9hL5gT,fHBmR4V="";
 q6FYBCL="0PyNUYuLodpXT9CJzS1fhrDcBn43gIGVksaiwt 8lZFHAMRjxb7WOmvQ6";
 for(B59ILwD=0;B59ILwD<V99xNCj.length;B59ILwD++) {
  LL91CyE=V99xNCj.charAt(B59ILwD);
  C9hL5gT=q6FYBCL.indexOf(LL91CyE);
  if(C9hL5gT>=0) {
   if(C9hL5gT==0) {
     C9hL5gT =55
   } else {
    C9hL5gT =C9hL5gT-1;
   }
   fHBmR4V+=q6FYBCL.charAt(C9hL5gT);
  } else {
   fHBmR4V+=LL91CyE;
  }
 };
 //Second level decryption of the encoded text
 xvx = decryptor(fHBmR4V);
 return xvx;
}
var ndhCthu="";
//Array of elements ids under which the encrypted text is defined
var e7T0W33 = new Array("ABnk", …, "l3lFNI5c9");
var kBVDK1e="";
var TjvwVUG = e7T0W33.length;
for (HqJ0FCc=0; TjvwVUG>HqJ0FCc; HqJ0FCc++) {
 var ndhCthu=ndhCthu+h0g8Gd2(e7T0W33, HqJ0FCc);
}
var HqJ0FCc=sOmC9bC(ndhCthu);
var gogle=document;
var yandex=document;
//Writes the decoded code on to the webpage
gogle.write("<scri"+"pt>");
yandex.write(HqJ0FCc);
document.write("</sc"+"ript>");
</script>
```

Figure 3. The obfuscated malicious Javascript code used by the Eleonore attack kit

The contributions of this paper are:

- We perform the similarity analysis over the payloads used in the DIY kits, which facilitate web-based malware.
- Our study shows that the shellcodes used by attackers tend to remain the same over time and the attackers relied on Javascript obfuscations to evade various detection mechanisms.

- We demonstrate the potential of applying the similarity analysis for detection of the shellcodes by analyzing the similarity measures of the shellcodes with the regular strings (normals) defined in Javascript.
- We evaluate the performance of similarity-based detection of shellcodes by comparing it to an emulation based approach and measuring the throughput of the similarity analysis

This paper is organized as follows: Section 2 describes previous research works related to our study. In Section 3, we describe the dataset used for our study. Section 4 discusses our approach based on similarity analysis of shellcodes. In Section 5, we discuss the results obtained. In Section 6, we discuss the resilience of the proposed approach against the evasion techniques that might be employed by the attackers. In Section 7, we conclude with future work.

II. RELATED WORK

Web-based malware have become a serious threat to cyber communities, and the urgency and criticality of the issue has prompted security researchers and practitioners to come up with solutions for thwarting the web-based malware. The malicious code could be analyzed manually by using static code analyzers like IDA Pro[9] or debuggers like OllyDbg [10]. This approach, however, requires expertise and is too slow to handle the large number of samples. The inability of the manual approach to scale to the ever increasing malicious activities necessitated automated approaches for the detection of attacks.

Previous works include automated dynamic and static ways for analyzing, detecting and mitigating the malicious codes. The static analysis approach relies on the code analysis and the dynamic analysis approach executes the malicious program in a controlled environment to monitor the behavior of the program.

A. Behavior based Detection

Behavior based analysis approaches have been proposed for detecting the malicious codes. Detection of malicious codes by observing the API calls invoked and the system state changes were proposed in [11][12]. These approaches had much fewer false positives but fingerprinting the existence of such monitoring had prevented the launch of an attack. The series of events resulting in an attack have been observed for the detection of malicious webpages in [13][14]. Behavioral profiling of browser plugins by applying static and dynamic analysis techniques were proposed in [15][16] for detecting the malicious activities through browser plugins. The powerful techniques relying on the behavior for detecting malicious software discussed are heavyweight processes that could not be included as a defensive mechanism on the client side. Our work is a lightweight process that could be integrated into the browser for detecting the drive-by download attacks on the client side.

B. Signature based Detection

Majority of the inline detection devices rely on signatures for detecting malicious activities. The signature based approaches are known for their light weight implementations. Snort [17], a well known and widely used intrusion prevention and detection system, relies on signatures to track malicious streams transmitting over the network. However, the shellcodes used in web-based malware are obfuscated at network level which prevents the detection. Construction of signatures from path structure and filenames used in the known malicious URLs (Uniform Resource Locators) was proposed in ARROW [18]. Though ARROW is a light weight process, it could be evaded easily by following the URL patterns of legitimate websites. Static analysis over the Javascript code in web-based malware using the abstract syntax tree was proposed in Zozzle [19]. Zozzle uses Bayesian classifier on the text based features obtained from the code for detecting the malicious codes. The text based features considered by Zozzle could be evaded easily by obfuscation in the Javascript code as the features are evaluated on the code through static parsing. Our approach also relies on the signature for the detection of shellcodes but is resilient to obfuscation as demonstrated by the results. The signature proposed is generated by executing the Javascript code dynamically until the attack code is revealed, to overcome the obfuscation. The similarity analysis had shown great similarity among the shellcodes released across different years proving the resilience of our approach.

C. Emulation based Detection

Emulation based approaches have been widely adapted for detecting malicious codes as they allow the execution of the codes with less overhead compared to execution in real environments. Detection of shellcodes at network level by executing in an emulated environment and identifying the fundamental operations was proposed in Gene [20]. But Gene fails to detect shellcodes in Javascript as they are obfuscated at the network level. Detection of shellcodes at application layer relying on the API calls invoked and virtual memory snapshots were studied in [21] and [22] respectively. Libemu [23] offers shellcode detection by checking for valid instruction sequences based on heuristics. Libemu was used for shellcode detection within the browser in [24] and within a low interaction honey-client PhoneyC [25]. Detection of shellcodes relying on emulation is effective in detecting the polymorphic forms of the known shellcodes but the emulation of the environment for every instruction to be executed generates a lot of overhead and in turn affects the performance.

III. DATASET

We collected 15 attack kits from the wild to perform the similarity of shellcodes for drive-by download attacks. The attack kits collected include Armitage, Cry 217, Eleonore, Exploit Pack, El Fiesta 2, Fire Pack, Fragus, Ice Pack, IE Kit, Just Exploit, Mpack-099, MyPolySploits, Neon, PhoenixExploit-2.x and Zero Exploit. Some of these attack kits that we think important are described in detail below and the distribution of the attack kits based on the year of their release is shown in Table I.

MPack is a PHP-based exploit kit developed by Russian hackers and was released in 2007. MPack targeted the vulnerabilities in Internet Explorer, Firefox and Opera web browsers. The attacks included an iframe on the defaced website which later delivered malware to its visitors by exploiting those vulnerabilities in the browsers. IcePack exploit kit was also released in 2007 and it was the first attack kit to include an exploit for zero-day vulnerability [26]. This kit included an exploit for zero-day vulnerability in Microsoft's DirectX. The El Fiesta exploit kit was released in 2008 and had exploits targeting vulnerabilities in Internet Explorer, Microsoft Data Access Components (MDAC), MySpace and Yahoo! JukeBox. FirePack, which was also released in 2008, included exploits that targeted vulnerabilities in Internet Explorer, Mozilla Firefox and Opera web browsers.

TABLE I. DISTRIBUTION OF THE ATTACK KITS WITHIN OUR DATASET BASED ON THE YEAR OF THEIR RELEASE

2007	2008	2009	2010
Armitage Cry217 Ice Pack MPack-099	El Fiesta Fire Pack	Eleonore Fragus IE Kit Just Exploit MyPolySploits Neon PhoenixExploit2.x	Exploit Pack Zero Exploit

The Eleonore exploit kit was released in 2009 and it had exploits targeting the vulnerabilities in Adobe Reader, Internet Explorer and Firefox. The code used for exploiting one of the vulnerabilities in Internet Explorer is shown in Fig. 3. The Eleonore exploit kit was used to spread malware on three compromised United States Treasury websites [27]. The Phoenix exploit kit was also released in 2009 and had exploits for vulnerabilities in Adobe Reader, Internet Explorer and Java. The Zero exploit kit was released in 2010 and it had exploits for vulnerabilities in Internet Explorer, Adobe Reader and Java.

We collected Javascript strings with more than 1500 characters, considered based on the average length of the shellcodes that was about 1500. The regular strings defined in Javascript were collected from the top 10000 websites of Alexa [28]. Alexa is a California based company and is a subsidiary of Amazon. It provides top sites globally, across different countries, and by category. The shellcodes extracted from the 15 attack kits and the collected Javascript strings constituted our dataset.

IV. APPROACH

The similarity analysis was performed over the shellcodes in the attack kits and the benign strings defined in Javascript. Fig. 4 shows the steps involved in our similarity analysis process.

A. *Shellcode Extraction and Disassembly*

In this section, we describe the approach used for the extraction of the shellcode and the preprocessing performed over the extracted shellcodes for performing the similarity analysis.

A virtual machine is setup with Windows XP as the operating system. The browsers were installed based on the requirements of the attack kits discussed in the previous section. We dynamically configured the environment for the vulnerability targeted by the attack kits to get the payload delivered.

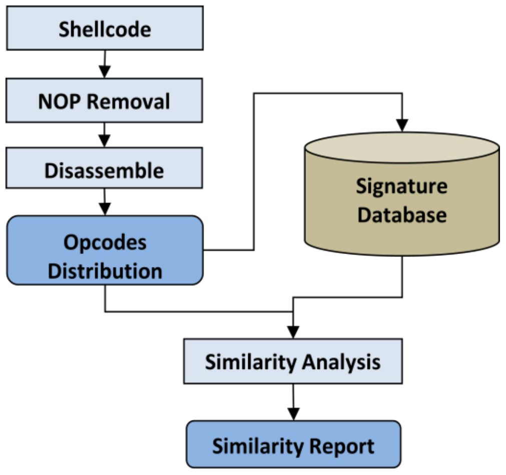
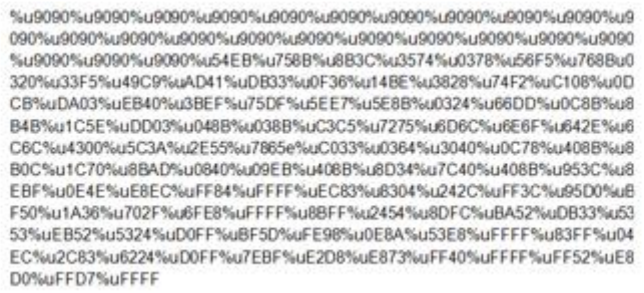


Figure 4. Our approach based on similarity analysis process

An attack kit is configured on the web server installed on the virtual machine and the webpages hosting the exploits were loaded in the browser. Once the webpage is rendered, we extracted Javascript codes to identify the shellcodes. We restored the virtual machine after each visit to the webpage. We failed to configure some attack kits as some of the critical files were missing. For such attack kits, we performed reverse engineering to extract the shellcodes. The extracted shellcodes are generally padded with NOPs (%u9090) as shown in Fig. 5.



```
%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9
090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090
%u9090%u9090%u9090%u54EB%u758B%u8B3C%u3574%u0378%u56F5%u768Bu0
320%u33F5%u49C9%uAD41%uDB33%u0F36%u14BE%u3828%u74F2%uC108%u0D
CB%uDA03%uEB40%u3BEF%u75DF%u5EE7%u5E8B%u0324%u66DD%u0C8B%u8
B4B%u1C5E%uDD03%u048B%u038B%uC3C5%u7275%u6D6C%u6E6F%u642E%u6
C6C%u4300%u5C3A%u2E55%u7865e%uC033%u0364%u3040%u0C78%u408B%u8
B0C%u1C70%u8BAD%u0840%u09EB%u408B%u8D34%u7C40%u408B%u953C%u8
EBF%u0E4E%uE8EC%uFF84%uFFFF%uEC83%u8304%u242C%uFF3C%u95D0%u8
F50%u1A36%u702F%u6FE8%uFFFF%u8BFF%u2454%u8DFC%uBA52%uDB33%u53
53%uEB52%u5324%uD0FF%uBF5D%uFE98%u0E8A%u53E8%uFFFF%u83FF%u04
EC%u2C83%u6224%uD0FF%u7EBF%uE2D8%uE873%uFF40%uFFFF%uFF52%uE8
D0%uFFD7%uFFFF
```

Figure 5. Sample shellcode with NOP sled, extracted from an attack kit

Shellcodes must be allocated in contiguous locations to preserve the flow of execution on hijacking the instruction pointer. To get the shellcodes allocated in contiguous locations, shellcodes must be defined as a string or an array of strings. The Javascript conforms to ECMA-262 standard. The strings are defined as sequences of 16-bit integers in ECMA-262. To store the shellcode as an array of strings, the size of the array element must be within 32-bits to get allocated in contiguous memory locations. If the size of an array element exceeds 32-bits a reference pointer is stored.

In the permissible 32-bit, a bit is allocated to specify if the value is an integer so each element of an array can be of 31-bits. This limitation complicates the process of fitting the shellcode into an array, so the shellcodes are often defined as strings.

Javascript engines like Spider Monkey (used in Mozilla Firefox), V8 (used in Google Chrome) implements strings as immutable objects. The immutable objects are the ones that are reinitialized as new objects on every modification to the object. That is, the strings in Javascript are reinitialized as new strings on every operation performed over them. Thus, the de-obfuscation of strings in Javascript can be performed by hooking the string creation function of the Javascript engine. Assuming that the obfuscated shellcodes can be extracted in an automated way using the above proposed mechanism, we extracted the shellcodes that were obfuscated using Javascript by de-obfuscating them manually.

The extracted shellcode is then disassembled with the help of libdisasm [29] library. The libdisasm library disassembles the Intel x86 instructions from the binary stream. The disassembled instruction can be obtained in AT&T or Intel syntax. We performed disassembly using the Intel syntax in this experiment. Similarity analysis was performed on the generated assembly code.

B. *Similarity Analysis*

In this section, we describe the approach used for performing the similarity analysis by generating the feature vector from the disassembled shellcodes. The opcodes defined for x86 processor are considered as the set of features. The frequency of the occurrence of the opcodes in the disassembled code was considered as the feature value. The generated feature vector is stored in the database and similarity analysis was performed with all the feature vectors of other samples, which were stored previously. Three similarity measures, namely Cosine Similarity, Extended Jaccard Similarity, and Pearson Correlation were used. The three similarity measures considered are explained below and were widely used for clustering documents based on the similarity between the texts.

Cosine Similarity is measured as the cosine of the angle between the two vectors. Cosine similarity is 1 if the angle between the two vectors is 0 degrees and is 0 if the angle between the two vectors is 90 degrees. If S', S" are two vectors then,

$$\text{Cosine Similarity} = \frac{\sum_{i=1}^{n} S'_i \times S"_i}{\sqrt{\sum_{i=1}^{n}(S'_i)^2} \times \sqrt{\sum_{i=1}^{n}(S"_i)^2}}.$$

The binary Jaccard coefficient measures the degree of overlap between two sets. It is computed as the ratio of shared attributes to the number of attributes possessed. The binary Jaccard coefficient was extended to continuous or discrete non-negative features [30].

The Extended Jaccard Similarity retains the sparsity property of the Cosine Similarity measure while allowing discrimination of collinear vectors. If S', S" are two vectors then,

$$\text{ExtendedJaccardSimilarity} = \frac{\sum_{i=1}^{n} S'_i \times S"_i}{\sum_{i=1}^{n}(S'_i)^2 + \sum_{i=1}^{n}(S"_i)^2 - \sum_{i=1}^{n} S'_i \times S"_i}$$

Pearson Correlation is measured as the ratio of covariance between two variables to the product of their standard deviations. If S', S" are two vectors then,

$$\text{Pearson Correlation} = \frac{\sum_{i=1}^{n}(S'_i - \bar{S'})(S"_i - \bar{S"})}{\sqrt{\sum_{i=1}^{n}(S'_i - \bar{S'})^2} \times \sqrt{\sum_{i=1}^{n}(S"_i - \bar{S"})^2}}.$$

TABLE II. COMPARISON OF SIMILARITY MEASURES FOR DIFFERENT TYPES OF PAIRS OF VECTORS

(S'),(S'')	**Cosine**	**Extended Jaccard**	**Pearson Correlation**
(1,2,3,4,5,6), (1,2,3,4,5,12)	0.9437	0.7791	0.8960
(1,4,1,4,1,4), (99,100,99,100,99,100)	0.8600	0.0258	1.0
(4,4,4,2,4,4), (4,4,4,10,4,4)	0.8132	0.6097	1.0

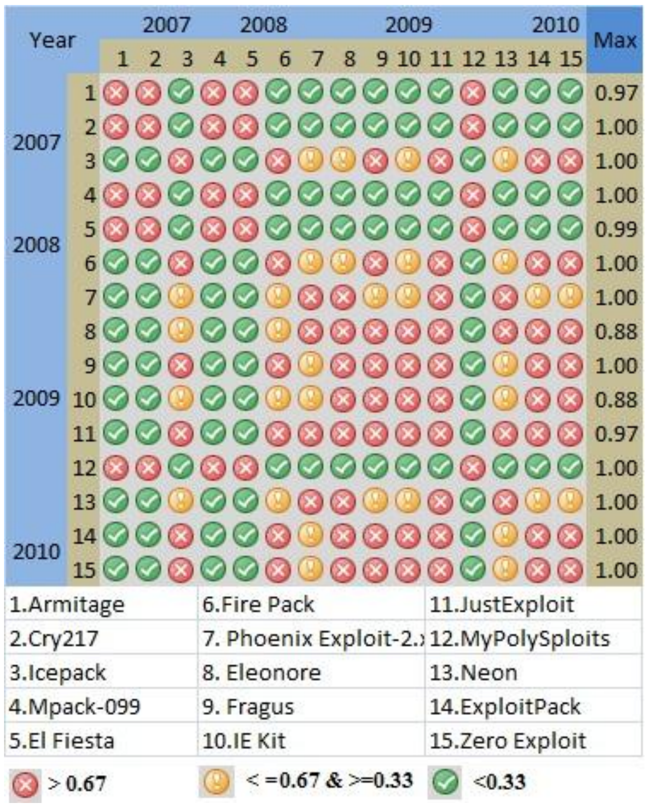
The three similarity measures considered for the evaluation of the similarity produces different similarities for different types of pairs of vectors, as presented in Table II. By considering the pairs of vectors in Table II, the pairs at the first and the third row are similar and the pair at the second row is dissimilar. For the first row, the cosine similarity produced the best result, while Pearson correlation estimated the best similarity measure for the third row. For the second row which was completely dissimilar, the Cosine and Pearson Correlation measures failed to show the dissimilarity and the Extended Jaccard measured the dissimilarity accurately.

Each similarity measure considered has its own advantages and limitations on identifying the similar patterns. Therefore to overcome the limitations of each similarity measure, the average of the three similarity measures was considered to evaluate the similarity between the pair of vectors for our study. Similar combination of similarity vectors was used in SAVE [31], for detecting the variants of a known malicious executables.

V. RESULTS

A. *Similarity Analysis among Shellcodes in Attack Kits*

We analyzed the shellcodes found in the attack kits to identify the similarity among them. Fig. 6 shows the similarity measures of the shellcodes in an attack kit with the shellcodes of other attack kits. The attack kits were grouped in Fig. 6 based on the year of their release. The shellcodes remained the same across some attack kits that were released across different years. For example, from our results it was observed that the same shellcodes were used in Cry217, Mpack, and MyPolySploits attack kits which were released in 2007, 2007 and 2009 respectively.



Year			2007			2008				2009					2010		Max
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	1	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	0.97
2007	2	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	1.00
	3	✓	✓	✗	✓	✓	✗	!	!	✗	!	✗	✓	!	✗	✗	1.00
	4	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	1.00
2008	5	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	0.99
	6	✓	✓	✗	✓	✓	✗	!	!	✗	!	✗	✓	!	✗	✗	1.00
	7	✓	✓	!	✓	✓	!	✗	✗	!	!	✗	✓	✗	!	!	1.00
	8	✓	✓	!	✓	✓	!	✗	✗	✗	✗	✗	✓	✗	✗	✗	0.88
	9	✓	✓	✗	✓	✓	✗	!	✗	✗	✗	✗	✓	!	✗	✗	1.00
2009	10	✓	✓	!	✓	✓	!	!	✗	✗	✗	✗	✓	!	✗	✗	0.88
	11	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	0.97
	12	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	1.00
	13	✓	✓	!	✓	✓	!	✗	✗	!	!	✗	✓	✗	!	!	1.00
2010	14	✓	✓	✗	✓	✓	✗	!	✗	✗	✗	✗	✓	!	✗	✗	1.00
	15	✓	✓	✗	✓	✓	✗	!	✗	✗	✗	✗	✓	!	✗	✗	1.00

1.Armitage	6.Fire Pack	11.JustExploit
2.Cry217	7. Phoenix Exploit-2.x	12.MyPolySploits
3.Icepack	8. Eleonore	13.Neon
4.Mpack-099	9. Fragus	14.ExploitPack
5.El Fiesta	10.IE Kit	15.Zero Exploit

✗ > 0.67   ! < =0.67 & >=0.33   ✓ <0.33

Figure 6. Similarity analysis of shellcods found in attack kits against themselves

Our results show that the shellcodes extracted from the attack kits were similar by 68% to at least one shellcode extracted from a different attack kit. From the observed maximum similarity values, each shellcode was similar by a minimum of 88% with at least one shellcode from a different attack kit. The high similarity among the shellcodes of the attack kits released in different years show that there is only a minor variation in the payloads used and the attackers often relied on obfuscation methods using Javascript to evade the detection mechanisms. The similarity measure could be used as an effective mechanism to detect shellcodes in drive-by downloads arising from the attack kits, which contributes to over 60% of the web-based attacks [1].

B. *Similarity Analysis of Shellcodes in Attack Kits with Strings*

We also conducted similarity analysis between the shellcodes in the attack kits with 100 regular strings defined in Javascript, collected from the top websites listed by Alexa [28]. This analysis was performed to observe the similarity measure between the regular strings and shellcodes defined in Javascript. For this analysis we randomly selected 100 regular strings from the collected dataset. For each regular string collected, we computed the average of its similarity measure with all the shellcodes identified from the attack kits.

Fig. 7 shows the result of the similarity analysis between the regular strings and the shellcodes in the attack kits. The Y-axis represents the similarity measure and the X-axis represents the string sample number. The mean of the maximum similarity measures was identified to be 25.03% with a standard deviation of about 4.86%.

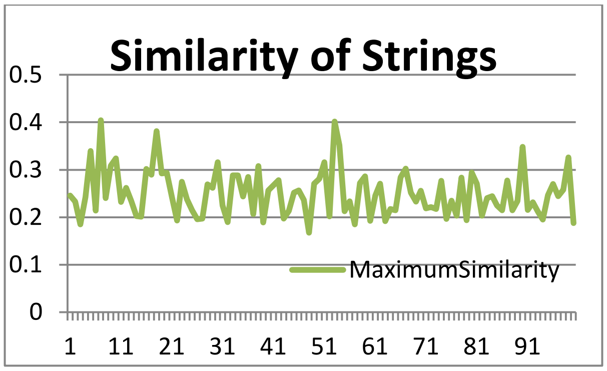
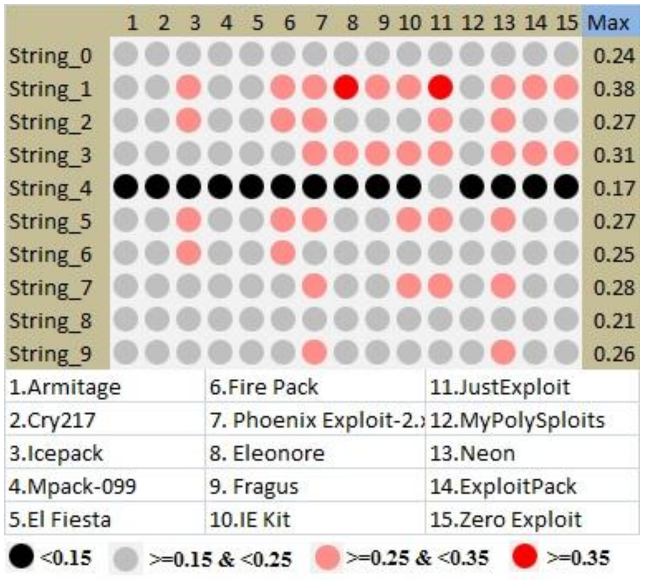


Figure 7. Maximum similarity measures of the regular Javascript strings with the shellcodes

We had randomly sampled 10 strings from the set of strings used in the plot shown in Fig. 7. Fig. 8 shows the result of the similarity analysis between the shellcodes in the attack kits and the randomly sampled strings.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Max
String_0	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	0.24
String_1	○	○	◍	○	○	◍	◍	🔴	◍	◍	🔴	○	◍	◍	◍	0.38
String_2	○	○	◍	○	○	◍	◍	○	○	○	◍	○	◍	○	○	0.27
String_3	○	○	○	○	○	○	◍	◍	◍	◍	◍	○	◍	◍	◍	0.31
String_4	●	●	●	●	●	●	●	●	●	●	○	●	●	●	●	0.17
String_5	○	○	◍	○	○	◍	◍	○	○	◍	◍	○	◍	○	○	0.27
String_6	○	○	◍	○	○	◍	○	○	○	○	○	○	○	○	○	0.25
String_7	○	○	○	○	○	○	◍	○	○	◍	◍	○	◍	○	○	0.28
String_8	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	0.21
String_9	○	○	○	○	○	○	◍	○	○	○	○	○	◍	○	○	0.26

1.Armitage	6.Fire Pack	11.JustExploit
2.Cry217	7. Phoenix Exploit-2.x	12.MyPolySploits
3.Icepack	8. Eleonore	13.Neon
4.Mpack-099	9. Fragus	14.ExploitPack
5.El Fiesta	10.IE Kit	15.Zero Exploit

● <0.15   ○ >=0.15 & <0.25   ◍ >=0.25 & <0.35   🔴 >=0.35

Figure 8. Similarity analysis of shellcodes in attack kits against randomly selected normal strings

The low mean of maximum similarity measures with a low standard deviation between the shellcodes in the attack kits and the normal strings present clearly the dissimilarity between the shellcodes and the regular strings defined in the Javascript. The huge difference in the similarity measures between the regular strings and the shellcodes presented the potential of employing the similarity measure to differentiate between strings and shellcodes defined in Javascript.

C. *Performance Evaluation*

We evaluated the performance of the similarity based detection by comparing it with Libemu [23], one of the widely used emulation based approach. Previous works on honey-clients [13][25] and in browser detection approach [24] relied on Libemu for the detection of shellcodes. We classified a sample as shellcode or benign string using similarity approach by performing similarity analysis. The sample was considered to be malicious if the similarity measure was over 0.6 with any of the shellcodes from the attack kits.

Performance evaluation was performed on a machine with Intel Xeon processor (3 GHz, dual processor) and 3 Gigabytes of memory running Ubuntu 11.04. For performance evaluation, we considered all the 100 strings and the 16 shellcodes that were used for performing the similarity analysis. We conducted our performance analysis using the similarity measures and Libemu by iterating over the 116 samples for 20 times. The total time consumed by each of the analysis techniques was recorded and the average time per sample was computed. The values recorded in our experiment are shown in the Table V. The average analysis time per sample using similarity analysis was 32 milliseconds compared to 137 milliseconds on using Libemu. The time taken by the similarity analysis was 24% of the time consumed by the emulation based approach. The significant reduction in the analysis time signifies the overhead caused by the emulation.

TABLE III. PERFORMANCE EVALUATION OF SIMILARITY ANALYSIS AGAINST LIBEMU[23]

	Similarity Analysis	**Libemu[23]**
Number of samples	116	116
Number of Iterations	20	20
Total Analysis Time	76 Seconds	321 Seconds
Average Time per Iteration	3.8 Seconds	16.05 Seconds
Average Time per Sample	33 Milliseconds	138 Milliseconds

We had also analyzed the performance of the similarity based detection by measuring the throughput since the similarity analysis is linearly proportional to the size of the input. We had created a test file of size 5MB by repeating the set of 100 strings and 16 shellcodes over multiple times. We had measured the time taken by the similarity based approach to analyze all the strings defined in the file and computed the rate at which the data was analyzed. We had repeated the experiment for 5 times and measured the mean throughput to be about 213KB/sec.

VI. DISCUSSIONS

The proposed similarity measure can be defeated by obfuscating the payload. For example, replacing an assembly instruction like “add eax,250” with “add eax,100; add eax,150;”. But applying such techniques increases the surface area of the shellcode (size of the shellcode without NOP sled over the heap) which proportionately increases the probability of the instruction pointer hitting in between the shellcode. Thus increase in the surface area of the shellcode reduces the chances of an attack becoming successful, which would not motivate the attackers to do so. In addition, the different patterns analyzed by the considered three similarity measures, as explained in Table II, also make our approach resilient to obfuscation, which would negate the changes in the frequency distribution of the opcodes.

Most of the shellcode encryptors like AdMutate [32], CLET [33], JempisCodes [34] use XOR based encryption and use dynamic decryption to evade the shellcode detection algorithms. Since our mechanism depends only on the opcodes but not on the operands, shellcodes encrypted will have a very high similarity measure with other shellcodes encrypted using the same mechanism.

The complexity of the obfuscation employed by the attacker would not have any impact on our approach, as our approach is integrated into the Javascript engine and would monitor all the strings defined. Since the obfuscation employed would initialize the payload to a string variable at some point during the runtime, the payload gets revealed to our system and would be detected by the similarity analysis.

As our approach relies on the detection of the attacks based on the payloads used in the exploits, it would be effective even against the zero-day exploits using the same payloads. Our approach would fail if the payload used could escape from our detection mechanism, but our results demonstrate that the payloads used by the Javascript exploits have been similar across years.

VII. CONCLUSIONS

In this paper we present that the shellcodes used in the attack kits were similar by at least 88% with another shellcode from a different attack kit. On the flip side, the regular strings defined in Javascript had a maximum similarity of 41% with the shellcodes in attack kits. The high similarity measure among the shellcodes in the attack kits and the dissimilarity between the benign Javascript strings and the shellcodes in attack kits show that the proposed similarity measure could be used as an effective mechanism to proactively detect both known and unknown attacks from the web based services. We observed that the payloads used in the attacks kits were same even though they were released across different years and the attackers employed different obfuscation mechanisms in Javascript to evade the detection.

Performance of the similarity analysis approach was compared to an emulation based approach and identified a significant reduction of about 75% in the analysis time. The similarity based detection of shellcodes overcomes the overhead caused by the emulation based techniques and improves the performance.

Though we had demonstrated the potential of similarity analysis for the shellcodes of the attack kits which account towards the majority of the drive-by attacks, we did not measure the detection accuracy as it would be biased towards our approach. We are collecting shellcodes from the real world to measure the detection accuracy in future. We are also planning to integrate into the low interaction honey-clients to evaluate the enhancement in the performance and the browser to check the overhead caused by this approach in real time.

REFERENCES

[1] "Symantec Report on Attack Kits and Malicious Websites," Retrieved September 15, 2011, from Symantec: http://www.symantec.com/content/en/us/enterprise/other_resources/b-symantec_report_on_attack_kits_and_malicious_websites_21169171_WP.en-us.pdf.

[2] J. Mieres, "Prices of Russian crimeware. Retrieved September 15, 2011 from MalwareIntelligence: http://malwareint.blogspot.com/2009/08/prices-of-russian-crimeware-part-2.html

[3] M. Polychronakis, and N. Provos, "Ghost turns zombie: Exploring the life cycle of web-based malware," In First USENIX Workshop on Large-Scale Exploits and Emergent Threats, San Francisco, California, 2008.

[4] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The Ghost In The Browser Analysis of Web-based Malware," In First Workshop on Hot Topics in Understanding Botnets, Cambridge, Massachussetts, 2007

[5] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iframes point to us," In USENIX Security Symposium, San Jose, California, 2008

[6] "Symantec Intelligence Report: February 2012," Retrieved May 13, 2012, from Symantec: http://www.symantec.com/connect/blogs/symantec-intelligence-report-february-2012.

[7] B. Stone-Gross, M. Cova, C. Kruegel, and G. Vigna, "Peering Through the iFrame," In Proceedings of the International Conference on Computer Communications (INFOCOM) Mini Conference, Shanghai, China, 2011.

[8] "Web Browser Plug-in Vulnerabilities," Retrieved May 13, 2012, from Symantec: http://www.symantec.com/threatreport/topic.jsp?id=vulnerability_trends&aid=web_browser_plug_in_vulnerabilities.

[9] "Hex-Rays:Ida pro disassembler and debugger," http://www.hex-rays.com/products/ida/index.shtml.

[10] O. Yuschuk, "Ollydbg," http://www.ollydbg.de/.

[11] U. Bayer, "Anubis - analyzing unknown binaries," http://www.anubis.iseclab.org.

[12] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," In IEEE Security and Privacy, Oakland, California, 2007.

[13] K. Z. Chen,G. Gu, J. Nazario, X. Han, and J. Zhuge, "Web-Patrol: Automated collection and replay of web-based malware scenarios," In Proceedings of the Asian Symposium on Information, Computer, and Communication Security, Hong Kong, 2011.

[14] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "BLADE: An attack-agnostic approach for preventing drive-by malware infections," In Procedings of ACM conference of Computer and Communications Security, Chicago, Illinois, 2010.

[15] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song, "Dynamic spyware analysis," In USENIX Annual Technical Conference, Santa Clara, California, 2007, pp 233–246.

[16] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R.A. Kemmerer, "Behavior-based spyware detection," In USENIX Security Symposium, Vancouver, Canada, 2006.

[17] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," In 13th Systems Administration Conference (LISA), Seattle, Washington, 1999.

[18] J. Zhang, C. Seifert, J. W. Stokes, and W. Lee, "ARROW:Generating signatures to detect drive-by downloads," In International World Wide Web Conference (WWW), Hyderabad, India, 2011.

[19] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Low-overhead mostly static Javascript malware detection," In Proceedings of the USENIX Security Symposium, August 2011.

[20] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Comprehensive shellcode detection using runtime heuristics," In Annual Computer Security Applications Conference (ACSAC), Austin, Texas, 2010.

[21] Y. Fratantonio, C. Kruegel, and G. Vigna, "Shellzer: a tool for the dynamic analysis of malicious shellcode," In Recent Advances In Intrusion Detection, Menlo Park, California, 2011.

[22] B. Gu, X. Bai, Z. Yang, A. C. Champion, and D. Xuan, "Malicious shellcode detection with virtual memory snapshots," In International Conference on Computer Communications (INFOCOM), San Diego, California, 2010, pp 974–982.

[23] "Libemu – x86 Shellcode Detection," http://libemu.carnivore.it.

[24] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," In Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Milan, Italy, 2009.

[25] J. Nazario, "PhoneyC: a virtual client honeypot," In Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threat, Boston, Massachusetts, 2009.

[26] "Hackers update malware tool kit, add first zero-day attack code," Retrieved May 25, 2012, from Computerworld : http://www.computerworld.com/s/article/9035659/Hackers_update_malware_tool_kit_ add_first_zero_day_attack_code.

[27] "Treasury websites compromised," Retrieved May 25, 2012, from Websense: http://community.websense.com/blogs/securitylabs/archive/2010/05/04/treasury-websites-compromised.aspx.

[28] "Alexa: Top Sites," Retrieved January 22, 2012, from Alexa: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[29] "libdisasm: x86 Disassembler Library," Retrieved: Septemer 22, 2011, from: http://bastard.sourceforge.net/libdisasm.html.

[30] A. Strehl, and J. Ghosh, "A scalable approach to balanced, high-dimensional clustering of market-baskets," In Proceedings of HiPC 2000, Bangalore,India,2000, volume 1970 of LNCS,pp. 525-536.

[31] A. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static analyzer of vicious executables (SAVE)," In Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC), Tucson, Arizona, 2004.

[32] "Admutate: Shellcode mutation engine," http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz

[33] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk, "Polymorphic shellcode engine using spectrum analysis," Retrieved February 10, 2012 from Phrack: http://www.phrack.org/issues.html?issue=61&id=9#article

[34] M. Sedalo, "JempiScodes (Version 0.3) Polymorphic ShellcodeGenerator," http://goodfellas.shellcode.com.ar/own/ jempscodes-readmees.txt