Detecting Heap-Spray Attacks in Drive-by Downloads: Giving Attackers a Hand

Van Lam Le, Ian Welch, Xiaoying Gao
School of Engineering and Computer Science
Victoria University of Wellington
P.O. Box 600, Wellington 6140, New Zealand
{van.lam.le, ian.welch, xiaoying.gao}@ecs.vuw.ac.nz

Peter Komisarczuk
School of Computing and Technology
University of West London
St Mary's Road, Ealing, London W5 5RF
peter.komisarczuk@uwl.ac.uk

***Abstract*—In the anatomy of drive-by download attacks, one of the key steps is to place malicious code (shellcode) in the memory of the browser process in order to carry out a drive-by download attack. There are two common techniques to carry out this task: stack-based and heap-based injections. However, introduction of stack protection makes the stack-based injection harder to carry out successfully. The heap-based injections become common methods to deliver shellcode to the heap memory of the web browsers. This paper presents the role of heap-spray in drive-by download attacks. We propose a new detection mechanism which makes shellcode in heap-spray executed in order to detect drive-by download attack. The solution not only benefits detection of drive-by download attacks but also analysis of malware behavior.**

***Keywords*-Internet Security; Drive-by-download; malicious web page;**

I. Introduction

When an Internet user visits a malicious web page, a malicious web server delivers a HTML document including malicious content to the user's computer system. The malicious content then exploits vulnerabilities on the system. The exploitation leads to executing malicious code (shellcode) in memory that takes control over the system, and install malware on it. This process happens without the Intenet user's consents or notices. This type of attacks is called drive-by download attack [6] [10].

In our previous work [8], we presented the anatomy of drive-by download attack which consists of four stages. In the third stage, the malicious contents are executed by web browsers at the visitors' computer system. There are two steps in the execution as follows:

1) The malicious contents exploit vulnerabilities on OS, web browsers, and plug-ins.
2) A successful exploit will let the malicious contents control the EIP register (Extended Instruction Pointer) to point to malicious shellcode in memory which is set up by attackers.

The third stage - executing malicious contents - is critical to the success of a drive-by download attack. Two conditions must hold for the success of this stage: available vulnerabilities on the visitors' computer system, and shellcode in the memory. The availability of vulnerabilities is out of control of attackers. Further, the attackers can not influence the availability of vulnerabilities on the user's computer. However, the attackers can control the layout of shellcode in the memory during visitation to increase the chances of executing the shellcode. One of the common techniques is a heap-spray attack which sprays a huge number of heap objects (containing shellcode) in the memory. When shellcode exists in many areas in the memory, there are more chances for attackers to control the EIP register to land in one of these areas to trigger an execution of shellcode.

In terms of detecting a drive-by download using heap-spray attacks, there are two following issues which prevent the attack happen:

1) There is no targeted vulnerability on the visitors' computer system. Any drive-by download attack usually targets a specific vulnerability. Without the vulnerability, the exploit can not happen. Therefore, shellcode in the memory can not be reached and executed.
2) The locations of heap-objects (containing shellcode) in memory are usually hard to predict. Therefore, the attackers usually estimate a "good" location of heap-objects that they can use to let EIP land after a successful exploit. However, the estimation is not always correct. Wrong estimation will make EIP point to an invalid address or invalid instruction and the system will crash.

In both cases, the shellcode is not executed and the drive-by download attacks never happen. Therefore, there is no malicious activity which can take place. Any detector based on malicious activities will miss the attack. This prevents researchers from collecting malware for their studies even though their detection tool has visited websites containing malware.

In this paper, we propose a solution to make shellcode in heap-spray attacks being executed even when there is no available vulnerability, or when the attackers mis-estimate the wrong locations of heap-objects containing the shellcode.

This paper makes the following contributions:

- It improves the detection rate for heap-spray attacks as well as drive-by download attacks. By making shellcode be executed, we make a drive-by download attack perform malicious activities that are monitored by some types of detectors to identify malicious web pages.

978-1-4799-0537-9/13/$31.00 ©2013 IEEE

- It provides a very useful method for collecting web-based malware and malware behaviour in realtime environment where the context of being infected and affected is not altered.

II. Related Work

The mostly related work to ours are studies [6], [7], [11] that focus on detecting malicious code (shellcode) written to memory by exploits for later execution. These studies are based on the fact that memory corruption like heap-spray is one of the common methods to exploit web browsers. Non of these work has a capability to execute shellcode in a real environment in order to detect drive-by download attacks. Another work that proposed a detector to execute heap objects in real environment is from Choi et al. [3]. The authors created a detector called HERAD (HEap spRAying Detector). Its Tracker saved potential heap objects into a log file, and then its Scanner checked for valid instructions in the heap objects. Its Executor loaded the log file containing the heap objects with valid instruction into memory for execution. However, the context of the attack and malware executions might be changed because the heap objects were stored in a log file and then reloaded into memory for execution. Moreover, it took time for its Scanner to scan for valid instruction in heap objects before executing them. In this paper, we propose a detection mechanism to keep heap objects in the locations where they should be. We use a high interaction client honeypot to visit a web page while we monitor allocations of the heap objects created during the visit of the web page. By analysing the way attackers usually setup shellcode in JavaScript code and the characteristics of heap-spray attacks, we identify potential malicious heap objects and execute them. To handle exception thrown by invalid instruction or invalid address (from our executions or from attacks' executions), we create a exception handler.

III. A Detection Mechanism

The main concept of the detection mechanism is to execute shellcode in heap-spray attacks in order to let drive-by download attacks happen successfully. To do this, we firstly track the process of heap allocations during visitation to identify potential malicious heap objects. At the end of visitation process, we control the EIP register of browser process to point to the potential malicious heap objects and execute them. This approach lets shellcode in heap objects executed in the way that attackers plan to do. Therefore, the context of the attack is not altered.

We proposed a detection mechanism which consists of four components: High Interaction Client Honeypot, Memory Monitor, Exception Handler, and Shellcode Executor.

We implemented our detection mechanism in Windows XP operating system which has Internet Explorer 6.0 and service pack 2. The instance of Windows XP SP2 is installed in a VMWare Server environment which runs on Fedora 10 (Figure 1). We implemented its components as described below.

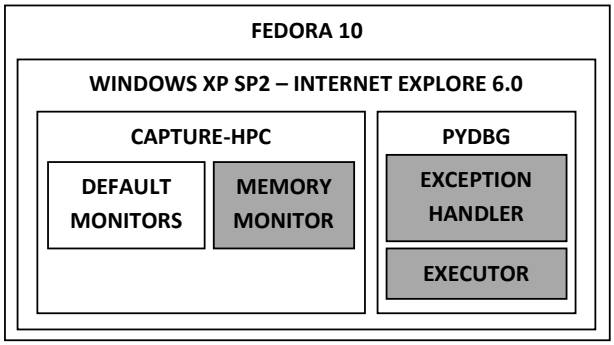

Fig. 1. Layout of Implementation

A. High Interaction Client Honeypot

We used our Capture-HPC v3.0 [12] for visiting web pages, monitoring changes in the system in order to determine whether a drive-by download attack happens or not. In addition, we limit our heap-spray carried out by JavaScript code only. The code written in other script languages is not considered in this paper.

B. Memory Monitor

To implement Memory Monitor we created a new module to add to Capture-HPC in order to monitor heap allocations. In order to track locations of heap objects, we hooked the API calls for allocating them in memory. In our initial implementation, we hooked the memory API calls in MSVCRT.DLL, OLEAUT32.DLL, KERNEL32.DLL, and JSCRIPT.DLL. However, hooking all of memory API calls in these libraries can make the number of system events increase. Capture-HPC spends more time to process the system events from the Memory Monitor and it can slow down the system. Therefore we aimed to reduce the number of hooked APIs in order to reduce the slowdown by identifying the minimal set that intercepts all API calls relevant to a heap-spraying attack. When analysing a heap-spray attack and monitoring the called API during the attack, we found the following interesting characteristics:

- Our previous analysis shows that shellcode is usually stored in a string. In order to avoid the shellcode string being changed to Unicode during memory allocation, attackers usually use *unescape* fuction in JavaScript to assign shellcode to a string object in memory.
- When we follow up the *unescape* function, it will call *PvarAllocBstrByteLen* function in JSCRIPT.DLL library. The *PvarAllocBstrByteLen* function, in its turn, will call *SysAllocStringByteLen* function in OLEAUT32.DLL library. The interface of the *SysAllocStringByteLen* function is **BSTR SysAllocStringByteLen(char FAR* psz, unsigned int len)**. We can get the location of a heap object from the pointer *psz* and the size of the heap object from the *len* argument. Based on these arguments, we can measure the size of heap-spray areas in memory.

From this analysis, we can see that a heap object from JavaScript code is allocated by a sequence of {PvarAllocBstrByteLen, SysAllocStringByteLen} calls. To track allocations of heap objects, we need to hook these API calls.

In addition, a heap-spray attack usually sprays a number of the-same-size heap objects into memory. The question is whether we have to execute all of them? To answer these questions, we classified heap-spray attacks into two types: traditional heap-spray and optimized heap-spray. The heap-spray discussed in the previous section is considered as a traditional heap-spray because typically there are many NOP instructions placed in front of the shellcode. An optimized heap-spray attack [5] attempts to reduce the number of NOP instructions in the sled before the shellcode to evade the detection tools [1] [11]. They achieve this by analyzing memory allocation and manage to reduce the number of NOPs required for the sled. While a traditional heap-spray has full shellcode in each heap object, an optimized heap-spray, in some cases, divides shellcode into smaller parts and put them in different heap objects. Therefore, choosing the first heap object to execute is a good choice in order to execute the whole shellcode in both cases.

During monitoring heap object allocation, we may discover more than one heap-spray area. The question is which one is more likely to contain malicious shellcode? In fact, a heap object from a heap-spray attack is usually larger in terms of size and much bigger in terms of quantity in comparison to a normal heap object. Therefore, the heap area with larger size gets higher priority for us to execute.

C. Exception Handler and Shellcode Executor

These two components are responsible for handling exception from invalid addresses or instructions, and executing shellcode in a potential malicious heap object. To implement these components, we used PyDbg debugger written in Python. The following tasks were implemented based on PyDbg debugger:

1) We set up a callback function to handle the EXCEPTION_ACCESS_VIOLATION event to handle exception thrown when the EIP register points to an invalid address due to an incorrect guess by the attackers of the location of the heap objects containing the shellcode.
2) After the process of visiting a web page finishes, we change the EIP register to point to a potential malicious heap object in order to execute it.

IV. Evaluation

To evaluate our detection mechanism, we implemented it on a PC running Fedora 10 with Intel®Core™2 Duo Processor 3.00 GHz and 4GB of RAM. VMWARE server was installed on this PC and Windows XP SP2 with Internet Explorer 6.0 was installed on VMWARE environment. The detection mechanism was setup on the Windows XP station. We evaluate it by using three criteria: False positive rate, false negative rate, and overhead time.

False positive: There is no false positive caused by this model itself. When this detection mechanism visit a benign web page, there are two possible cases. In the first case, the model does not find any potential malicious heap object from the web page. The model does not take any further action so the web page will be classified as benign (as Capture-HPC job). In the second case, the detection mechanism finds a potential malicious heap object from the benign web page. It will execute the potential malicious heap object. Because the benign web page does not have any malicious shellcode on its heap objects, the result of execution of heap object is to trigger an EXCEPTION_ACCESS_VIOLATION event which is handled by our model. As a result, there is no false positive in both cases.

False negative: To evaluate false negative rate, we need a set of drive-by download attacks with involved heap-spray. We used Metasploit [9] for generating a number of drive-by download attacks with involved heap-spray. In order to evaluate our detection mechanism, we need to choose a heap-spray attack that can not be successful at the Windows XP workstation. When analysing heap-spray from Metasploit, we found that "'Internet Explorer Tabular Data Control ActiveX"' attack can not execute in our implemented Windows XP workstation. We chose it as a base of drive-by download attack. Further, Capture-HPC v3.0 monitors four events in order to determine the success of a drive-by download attack: file, process, registry, and network connection events. In terms of choosing shellcode for our evaluation, we have to choose the shellcode that makes anomalous activities being monitored by Capture-HPC in order to know the success of our detection mechanism. We choose "windows/meterpreter/reverse_tcp" shellcode because it has anomalous activities including network connections and storing a file (given by us) in the system. We would like to have the various layouts of heap objects in memory randomly. To do this, for each test, we adjust two factors as follows:

1) Number of NOP strings: By the default, the number of NOP strings in the front of heap objects in the "'Internet Explorer Tabular Data Control ActiveX"' attack is 16384 (0x4000). We randomly choose this number from 1 to 16384.
2) Number of heap objects: The default number of heap objects in this attack is 150. We randomly choose this number of 1 to 150.

By adjusting these factors, the size of heap objects and heap areas, and their locations are unpredictable. We generate 100 URLs containing the "'Internet Explorer Tabular Data Control ActiveX"' exploit with various heap-spray objects. We controlled our detection mechanism to visit these URLs. The result shown that our detection mechanism handled exception thrown and made the shellcode in heap objects being executed. The anomalous activities in network connections and file system from the result of the executing the shellcode were detected. There were 100 given URLs successfully carrying out drive-by download attacks.

Overhead time: To evaluate the overhead time of our detection mechanism, we compared our detection mechanism

to the original Capture-HPC. We controlled our detection mechanism and the original Capture-HPC to visit the top 250 websites from Alexa [2]. The result shown that the original Capture-HPC spent 11.33 seconds in average for scanning an URL while our detection mechanism needed 20.31 seconds in average to scan an URL. Its overhead time is nearly 9 seconds. To analyze the reason for quite large overhead time, we tracked down the performance of our detection mechanism. While following up the performance of Memory Monitor, we found that it got busy with logging information of allocations of heap objects. We did not set the threshold (in the size of heap objects) in the hooking API functions. We leave this work as future work to improve the overhead time.

V. Discussion

The technique presented in this paper is to improve the detection of drive-by-download attacks by increasing the chances of a heap-spray attack succeeding. There are some limitations of this concept we would like to discuss in this section as future work. Firstly, this work focuses on heap-spray on drive-by download attack carried out by JavaScript code only (Although JavaScript code is commonly used to carry out drive-by download attack [4]). Secondly, this work assumes that the same-size heap objects appearing continuously belong to the same heap-spray attack. Attackers can change their mechanism to make each heap object in different size in order to evade the detection mechanism. Thirdly, the Exception Handler and Executor were implemented by using Pydbg and not convenient for setting up. Integrating them into Capture-HPC should be a good improvement in terms of setting up and operation.

VI. Conclusion

The paper presents a new detection method to detect drive-by download with involved heap-spray attacks. The main concept of this work is to improve the chances of a heap-spray attack being successful and leading to successful drive-by-download attacks. The benefit from this work is the avoidance of system crashes or lack of local vulnerabilities leading to false negatives. It allows high-interaction honeyclients to collect more malware and observe their behaviours in a real system.

The detection mechanism has successfully been implemented and added to Capture-HPC - a common high interaction client honeypots used in research. Based on our evaluation, the detection mechanism shows its effectiveness on carrying out heap-spray attacks and detecting drive-by download attacks. Although the detection mechanism has been implemented and evaluated in Windows XP SP2, it can be implemented in other Windows systems by adapting API hooking techniques and investigating appropriate API calls in specific Windows operating system.

References

[1] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis, "Stride: Polymorphic sled detection through instruction sequence analysis," in *In 20th IFIP International Information Security Conference*, 2005.

[2] Alexa, "Alexa top 500 global sites," 2012, available from http://www.alexa.com/topsites; accessed on 8 November 2012.

[3] Y. Choi, H. Kim, and D. Lee, "Detecting heap-spraying code injection attacks in malicious web pages using runtime execution," *IEICE Transactions*, vol. 95-B, no. 5, pp. 1711–1721, 2012.

[4] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *WWW2010*, Raleigh NC, USA, 2010.

[5] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou, "Heap taichi: exploiting memory allocation granularity in heap-spraying attacks," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 327–336.

[6] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 88–106.

[7] F. Gadaleta, Y. Younan, and W. Joosen, "Bubble: a javascript engine level countermeasure against heap-spraying attacks," in *Proceedings of the Second international conference on Engineering Secure Software and Systems*, ser. ESSoS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–17.

[8] V. L. Le, I. Welch, X. Gao, and P. Komisarczuk, "Anatomy of drive-by download attack," in *Australasian Information Security Conference 2013 (ACSW-AISC 2013)*, Adelaide, Australia, 2013.

[9] Metasploit, "Meatasploit: Penatration testing software," 2012, available from http://www.metasploit.com; accessed on 22 February 2012.

[10] J. Narvaez, C. Seifert, B. Endicott-Popovsky, I. Welch, and P. Komisarczuk, "Drive-by-download," Victoria University of Wellington, Wellington, Tech. Rep., 2008.

[11] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: a defense against heap-spraying code injection attacks," in *Proceedings of the 18th conference on USENIX security symposium*. Montreal, Canada: USENIX Association, 2009.

[12] C. Seifert, R. Steenson, and V. L. Le, "Capture-hpc v3.0 beta," 2009, available from https://projects.honeynet.org/capture-hpc/wiki/Releases; accessed on 22 Feburary 2010.