International Journal of Security and Its Applications
Vol.7, No.5 (2013), pp.101-112
http://dx.doi.org/10.14257/ijsia.2013.7.5.08

A Robust Behavior Modeling for Detecting Hard-coded Address Contained Shellcodes

Javad Khodaverdi[1] and Farnaz Amin[2]

[1]*Dept. of Computer Engineering and IT, Amirkabir University of Technology, Iran*
[2]*Dept. of Electrical and Computer Engineering, Yazd University, Iran*
[1]*j.khodaverdi@aut.ac.ir*, [2]*Farnaz.amin@stu.yazduni.ac.ir*

Abstract

Nowadays, code injection is one of the most dangerous cyber attacks. Shellcode is a malicious code which is used in this type of attack. Processor emulation at network level is one of the best proposed methods against code injection attacks. Multiple run-time heuristics have been discussed in previous researches. However, none of them can detect those shellcodes in which hard-coded addresses are used. This type of shellcode cannot be used against ASLR-enabled Windows. Howbeit, older versions of Windows have still too many users. In addition, there are several hard-coded address contained shellcodes in public shellcode repositories which can be used easily by dummy hackers. In this paper, we propose a robust run-time heuristic for detecting this type of shellcode. Our objective is to augment the collection of the existing run-time heuristics. The experimental results show that our new heuristic can effectively detect every shellcode in which hard-coded addresses are used.

Keywords: *Shellcode, Emulation, Dynamic Analysis, Detection Accuracy*

1. Introduction

Traditional signature-based detection systems can be easily bypassed by a polymorphic shellcode. In addition, static analysis based detection systems can be bypassed taking advantage of self-modification technique. Static analysis based detection systems rely on the control flow and the data flow derived from the input stream, while a hacker can make a fake control and data flow for his code since the original payload will be revealed using self-modification at the target. The effectiveness of emulation-based approach is that it does not matter whether self-modification has been used. In this approach the input stream is executed and the emulation-based detection system concentrates on the observed behavior of the code and verifies the existence of a malicious behavior during the execution.

The architecture of an emulation-based detection system consists of an emulator to execute the input stream and a virtual memory to load the input stream within it and to perform read/write instructions made by it. The virtual memory also is used for loading the common modules of a sample process. This memory plays an important role because a shellcode may point to a loaded module. So we prefer to continue the execution in such circumstance.

Since the concentration of an emulation-based detector is on the behavior of the input stream, it should have some heuristics to be matched with the observed behavior. Therefore, there are one heuristic for each class of shellcode in this architecture. Every heuristic has multiple conditions that all should be satisfied for triggering an alert. Every condition is related to one step of the intended behavior of the shellcode. For example, if we know that there are three unavoidable steps within the behavior of a shellcode, we should consider three conditions in the corresponding heuristic in order to detect such a shellcode. Our detection system

ISSN: 1738-9976 IJSIA

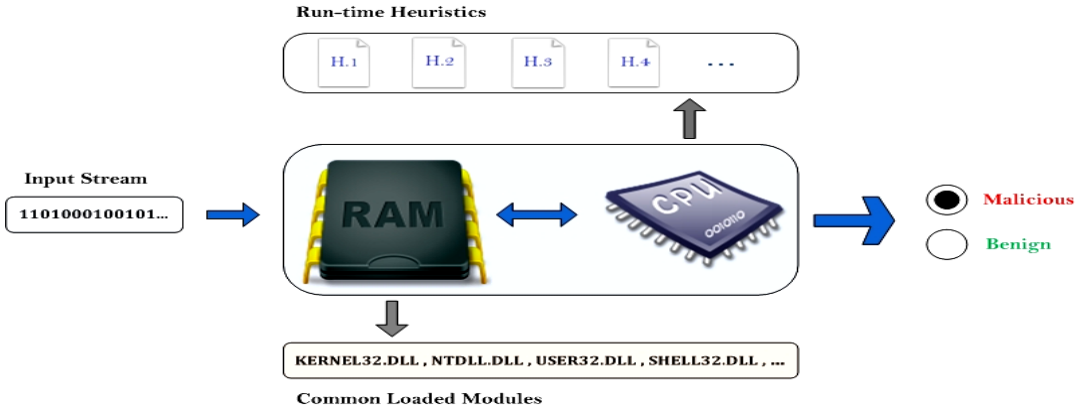

Figure 1. Architecture of an Emulation-based Shellcode Detector

checks these conditions simultaneously for all heuristics during the execution. It is possible for a shellcode that triggers more than one heuristic but whenever any detection heuristic is triggered, an alert will be generated. According to these characteristics, the architecture of an emulation-based shellcode detector can be similar to one illustrated in Figure 1.

In recent years, most of the proposed emulation-based detection methods are to identify polymorphic shellcode and their focus is on detecting a decryption routine in the input stream [1-4]. However, they cannot detect plain shellcodes which have no self-decrypting or self-modification behavior. Obviously, a plain shellcode is the decrypted version of a polymorphic shellcode. Thus, if an intrusion detection system can detect plain shellcodes, it will be able to detect polymorphic shellcodes as well because the decrypted body of the shellcode is revealed after the execution of the decryption routine. In another word, after the execution of the decryption routine, a jump should be made to the beginning of the main payload otherwise the shellcode has no functionality.

In one of the previous researches [5], four behaviors of shellcodes were elaborated and implemented in Gene, a code injection attack detector based on passive traffic monitoring. First two behaviors are about those shellcodes in which the base address of KERNEL32.DLL is resolved and second two behaviors are about those shellcodes which are referred to as egg-hunter shellcode. This type of shellcode should be used when there is not enough space for injecting the main shellcode, referred to as egg. The role of an egg-hunter is to find the egg in the address space of the attacked process and to jump to it. Our contribution in this paper is to augment this collection. Here, a brief description of each of four mentioned behaviors is provided.

PEB Windows API functions are divided into several dynamic load libraries (DLLs). First operation of a shellcode is to load an intended DLL into memory and to look within it for the intended API functions. KERNEL32.DLL is an important module that does not need to be loaded by shellcode since this DLL is always loaded into the address space of any Windows process. There are two useful API functions within this DLL: (i) *LoadLibrary* for loading an arbitrary DLL into the address space of the current process, and (ii) *GetProcAddress* which returns the address of an arbitrary API function. After resolving these two APIs, any other API in any other DLL can be loaded directly by using these functions. A reliable method for locating the base address of KERNEL32.DLL is through PEB data structure [5], [6]. PEB is a user-level data structure that holds extensive process-specific information. This structure is always accessible by reading FS: [0x30]. FS is a segment register pointing to a segment in which a pointer to PEB is placed. There is a pointer to PEB_LDR_DATA data

structure in the offset 0xC of PEB data structure. Also there are three pointers in this data structure to three linked lists of the loaded modules. One of the linked list's elements is always related to KERNEL32.DLL in which the base address of this DLL has been stored. As such, a shellcode can exploit PEB data structure to find the base address of any loaded module such as KERNEL32.DLL. The corresponding heuristic for detecting this behavior of shellcode had been designed according to the above steps, and had been called PEB [5].

BACKWD In addition to above method, there is another way to resolve the base address of KERNEL32.DLL. Taking advantage of the structured exception handler (SEH) mechanism of Windows, a shellcode can read FS: [0] to obtain the beginning of SEH chain and refer to the last, default SEH frame in this chain and obtain a pointer to a location in the address range of KERNEL32.DLL by reading the *Handler* field of the frame. The *Handler* field of any SEH frame is a pointer to the corresponding exception handler routine. Also we know that the last frame in the SEH chain is registered by the system whose *Handler* field points to a routine in the address range of KERNEL32.DLL. So after reading this field, shellcode can move backward towards the first byte of the KERNEL32.DLL and find the base address of it. This method is called backward search because shellcode find an address in KERNEL32.DLL and then looking backward for 'MZ' [7]. Every DLL in Windows has this signature as its first two bytes. The corresponding heuristic for detecting this type of shellcode had been designed according to the above steps, and had been called BACKWD [5]. Furthermore, a similar method has been elaborated by Javad Khodaverdi [27].

SYSCALL Some Windows system calls take a pointer to an input parameter as an argument. If the pointer is invalid, the value 0xC0000005 is returned after executing "*int 0x2E*" instruction. Egg-hunter shellcodes can use this to check the availability of a memory address. The most famous system calls which can be used by egg-hunter shellcodes are *NtAddAtom* (0x8), *NtAccessCheckAndAuditAlarm* (0x2) and *NtDisplayString*. There is an issue in using the system calls. In various versions of Windows, the number assigned to a system call may be changed. Although this number has no change for *NtAccessCheckAndAuditAlarm* and *NtAddAtom*, *NtDisplayString* is assigned various numbers in different versions of Windows. For example, it is 0x43 in Windows XP, 0x46 in Windows 2003 Server and 0x7f in Windows Vista. The corresponding heuristic for detecting this type of shellcode had been designed based on system call invocation mechanism of Windows, and had been called SYSCALL [5].

SHE Another technique used by egg-hunter shellcode to locate the egg is the registration of a custom SEH frame [8], [9]. In another word, it takes advantage of SEH (Structured Exception Handling) mechanism of Windows. Every SEH frame contains two fields. The first is a pointer to the next frame and the second is a pointer to the exception handler routine. The second field is called *Handler*. Note that the first field of the last frame, registered by the system, contains the value 0xFFFFFFFF. When a hacker is forced to use egg-hunter, he should write an egg-hunter in a way that it can handle the probable exceptions during the search process. If it does not handle the exception, an unexpected event may be occurred. Therefore, egg-hunter shellcode first register a custom SEH frame so that when an exception occurred, the arbitrary handler routine is executed. To this end, the hacker can uses one of the two possible ways as follows. The first is that the egg-hunter create a new SEH frame and overwrite the location FS: [0] with the address of this new frame. Since FS: [0] is always referred by Windows to handle an exception, the arbitrary handler routine will be executed after throwing an exception. The second one is that egg-hunter shellcode read FS:[0] and after locating the first SEH frame, modify the *Handler* field of the frame to point to the location where his arbitrary handler routine has been placed. In this case his handler routine will be executed when an exception is thrown. Note that in the first case a new frame is created but in the latter an existing frame is exploited. The corresponding heuristic for detecting this type of shellcode

had been designed according to the above steps and had been called SEH [5]. This heuristic has a third condition that is very useful for preventing false positive. After the satisfaction of the second condition, it is validated that a custom SEH frame has been registered correctly or not. In another words, the integrity of SEH chain is verified.

The remaining sections are organized as follows. Section 2 reviews the previous works on shellcode detection. Main contribution of this paper, describing a general behavior of shellcodes, is described in Section 3. This section also provides a dynamic taint algorithm. Section 4 is about the implementation of our prototype and Section 5 provides the experimental results. Finally Section 6 and 7 provide a discussion and conclusion respectively.

2. Related Work

After the manifestation of polymorphic shellcodes, the early signature-based detection systems could not identify these shellcodes. At the first, static analysis approach was introduced for identifying the presence of shellcode in network streams. In this approach, first the network stream is disassembled and a detection process is then performed according to the derived control and data flow. In [10] Toth and Kruegel tried to identify the NOP-sled in the network streams by using the code disassembly. However, the NOP-sled may not be used in the shellcode. Because of the use of "*int 0x80*" instruction in the Linux shellcodes, Anderson et al. proposed a detection method based on the multiple occurrences of the instruction sequences ending with this instruction [11]. A few methods were proposed for detecting the previously unknown polymorphic shellcodes, which are based on the identification of structural similarities between several different worm instances [12]. Particularly [13, 14] used the data flow and control flow analysis on several instances of polymorphic shellcodes, aiming to obtain the structural similarities and [15] used neural networks to do it.

There may be exists some obfuscation methods that are not known for us. However, by using the known obfuscation methods such as self-modification and indirect jump [1], the static analysis based detection methods can be bypassed easily. Although [14] tried to detect self-modification behavior by static taint and initialization analysis, it can be evaded by assembly tricks such as using the sequence of "*mov eax, esp*" and "*mov ebx, [ss:eax]*" instead of "*mov ebx, [ss:esp]*". This leads to introduction of emulation approach, a dynamic analysis method, to execute the derived instructions of the network streams and focus on the observed behavior. The early proposed emulation-based methods were based on the observing a mandatory behavior of a polymorphic shellcode. For example, a shellcode has to obtain its absolute address since IA32 architecture does not support the relative addressing mode. There are a few methods to obtain the value of the program counter. Most of the previous detection methods are based on the observing one of these behaviors during the execution [1]. However, there are some particular cases in which the shellcode uses a register pointing to its base address after the injection [2]. Zhang et al. combined network level execution and static data flow analysis for enhancing the runtime execution performance [4]. Finally, since self-decrypting behavior is never seen in the plain shellcodes, Polychronakis et al. proposed a heuristic-based detection system [5]. This detection system uses four different runtime heuristics according to four different behaviors of plain shellcodes. If anyone of the runtime heuristics is matched with the observed behavior during the execution, the presence of shellcode is detected.

Emulation-based shellcode detection can be used for detecting drive-by download attacks and malicious websites too. Such a research is accomplished by Egele et al. proposing a technique in which an embedded CPU emulator is used in browser to identify malicious javascript string buffers [16].

Symbolic execution is another approach much similar to emulation. Symbolic execution can be used to extract the real structure of a shellcode in run-time. For instance, Spector [17]

uses this approach to extract the sequence of library calls made by the shellcode including their arguments and to generate a low-level execution trace of the shellcode. Shellzer [18] is a dynamic shellcode analyzer that generates a complete list of the API functions called by the shellcode. This tool is able to modify both the arguments and the returned value of any API function on the fly. Its concentration is on malicious PDF files and malicious web pages. In order to execute the contained shellcode correctly, it simulates a specific execution context such as Adobe Reader because the shellcode may assumes that it is running inside an instance of this program. This tool can be evaded by some assembly tricks such as indirect API call.

3. Shellcode Behavior Modeling

It is not imaginable for a shellcode that calls no API function. In another word, the functionality of a shellcode depends completely upon its API function calls. Previous proposed run-time heuristics were designed based on the early indispensable operations of shellcode, such as resolving the base address of KERNEL32.DLL. After resolving KERNEL32.DLL base address, shellcode refers to this DLL's export table and find the relative virtual address (RVA) of the intended API function. Then, it adds the DLL's base address to the obtained RVA to calculate the absolute address of the intended API function. Finally, it can call this address. However, when a hacker knows his target, it may be possible to use the hard-coded addresses of the intended API functions within the shellcode. In such circumstances, there is no behavior in the shellcode based on which the previous heuristics have been designed. So we need a new heuristic to detect this type of shellcode. Our heuristic (*i.e.*, *HC Shellcode*) should be designed based on the overall behavior of shellcode.

Overall behavior of a functional shellcode could be modeled as follows: (i) pushing some values in stack as the input parameters of the intended API function, and (ii) calling the known address of the function. These conditions are so simple and we need to consider some shellcode-specific features which could prevent false positive effectively.

One of the most common instructions in random code (*i.e.*, normal traffic) is *push*. In another word, when a random byte sequence is disassembled, *push* is a prevalent instruction in the disassembly. So it cannot be enough to consider only the *push* instruction as the first part the above behavior. Furthermore, it is likely to encounter a *call* instruction during an execution chain. Thus, if we design our heuristic only based on these simple conditions, false positive rate may not be admissibly low.

Since this behavior can be seen in random code, we need an additional condition to discriminate between normal traffic and malicious code (*i.e.*, shellcode). A discriminating feature is that a shellcode pushes the input parameters of the intended function and also calls the hard-coded address consciously. To identify this consciousness and perform the above discrimination correctly, we should track the hard-coded values during the execution. To this end, we need a dynamic taint algorithm.

3.1. Dynamic Taint Algorithm

The emulator used for shellcode detection should be able to parse the operands of instructions. For example, it should be identifiable whether an instruction has operand. Also the emulator should be able to identify the number of operands and determine source and destination operand. During the execution of an instruction, first we determine whether the source operand is a hard-coded value and if so, we taint the destination operand. It does not matter which instruction do this. For example, after executing "*mov eax, 0x12345678*" or the sequence of "*push 0x12345678*" and "*pop eax*" we taint *eax*. In addition, after parsing the operands and identifying source and destination operands, if the source operand is a tainted register, the

destination operand will be tainted too. This operation takes into account the memory addresses as well. In another word, if the source operand is a hard-coded value or a tainted register and the destination operand is a pointer to a memory address, that memory address will be tainted too. Finally, we may have multiple tainted places (*i.e.*, registers or memory addresses) during every execution chain. If no shellcode is detected during an execution chain, tainted places are cleared before starting the next execution chain.

3.2. Detection Heuristic

As mentioned earlier, an obvious behavior of shellcodes is pushing some values in stack as input parameters of the intended API function and calling their addresses. *LoadLibrary* is a common function that is called in too many shellcodes. Since this function has only one input parameter, we can use this fact to determine the number of hard-coded values that should be pushed onto the stack.

When the emulator encounters a *push* instruction, its operand is verified by the above algorithm to determine whether it is tainted. If so, the first part of our detection heuristic is satisfied. We can define the first condition of our heuristic (*i.e.*, *HC Shellcode*) as follows.

Condition H1: a hard-coded value is pushed onto the stack consciously. The consciousness is determined by dynamic taint algorithm presented in Section 3.1.

Then, after encountering a *call* instruction, its operand is parsed. If one of the following holds true, the second part of our detection heuristic is satisfied.

- the called address is an immediate operand (such as *call 0x7c820742*).
- the called address is stored in a tainted place, either a register or a memory address.

Thus, the second condition of our heuristic can be defined as follows.

Condition H2: a hard-coded address is called consciously.

Two above conditions should be satisfied more than one time in every functional shellcode. Even in a shellcode that only opens a message box, two API functions should be called: *MessageBox* and *ExitProcess*. A wised hacker usually uses *ExitProcess* to prevent the creation of core dump files and to exit the vulnerable process cleanly. Thus, we can use a threshold equal to 2 for the repetition of the above behavior and define the last condition of our heuristic. Note that most of functional shellcodes such as reverse shell and bind shell use multiple API functions and so the above behavior is seen multiple times.

Condition H3: this can be defined as a meta-condition: H1 and H2 hold true two times.

Although the above conditions are quite constraining, we can consider additional conditions to preserve false positive rate as low as possible. First, it is possible that a loop is constructed in random code in which all of the above conditions are satisfied. In such circumstances, our heuristic identifies it as a shellcode but this is not the case. To address this issue, we add this condition to our heuristic that two occurrences of the mentioned behavior should be seen with different hard-coded addresses. In another word, two different hard-coded addresses should be called during an execution chain. Second, we know the address range of the loaded modules in Windows. Thus, we can add this condition that the called addresses should be in a specified range.

We used the threshold equal to 2 for the repetition of the above mentioned behavior in order to preventing false positive. However, there is a special shellcode that may not be detected by the above heuristic. This type of shellcode is discussed in the following section.

3.3. WinExec Shellcode

A wised hacker normally calls *ExitProcess* or any similar function at the end of his shellcode to prevent the creation of core dump files and to exit the attacked process cleanly. However, *ExitProcess* may not be used by a dummy hacker. It seems that this type of shellcode cannot be dangerous as it calls only one API function, but we should take a special shellcode into account in which merely WinExec API function is called. This function has two input parameters and allows a hacker to execute an arbitrary command on the attacked machine remotely. For example, if the hacker executes the command *shutdown -s*, he can turn off the target system. Since this type of shellcode is considered as a dangerous shellcode and cannot be detected by the previous heuristic, we should take it into account separately and define an additional heuristic. We should collect the addresses of WinExec in all versions of Windows. Note that this address is not fixed and also is not predictable, neither for us nor a hacker, in ASLR-enabled Windows. *WinExec* heuristic has three conditions as follows.

Condition W1: two hard-coded values are pushed onto the stack consciously. Consciousness is determined by dynamic taint algorithm presented in Section 3.1.

Condition W2: one of the known fixed addresses of WinExec is called consciously.

Condition W3: Although the above conditions are quite constraining, we can consider an additional condition. As mentioned earlier, WinExec allows a hacker to execute an arbitrary command on the attacked machine. It is obvious that the command must contain merely ASCII printable characters. Thus, whenever the condition W2 is satisfied, we refer to top of the stack and obtain the address at which the intended command is stored. Then, the command is verified as a string (i.e. terminated by null byte) to determine whether it contains merely valid characters. If so, last condition of *WinExec* heuristic is satisfied as well.

4. Implementation

A prototype of our detection system has been implemented using C++ programming language. We have used a custom CPU emulator [19] which supports the execution of IA32 instructions. Since we do not know the exact location of the shellcode in the input stream, we should repeat the execution multiple times starting from each location of the input stream but before it we find all possible executable sequences of instructions within the input stream and then execute them. The performance of this detector can be enhanced by using some optimizations such as aero-delimited chunk optimization [1]. To emulate the memory access instructions correctly, we load 10 common Windows modules and PEB/TIB data structures in the virtual memory of the detection system.

When FS register is used during the execution, we keep this state and after reading any memory address, we can recognize which type of shellcode is running. However, our detection system does not produce any alert before the satisfaction of all conditions of a heuristic. When an "*int 0x2E*" instruction is encountered, we do not emulate the actual functionality of the system call and just a value is set in EAX according to the accessibility of the supplied address. This cause the memory scanning loop can continue normally.

Although 10 common modules are loaded into the virtual memory of our detector, it is possible for another API function to be called which is not within these modules. Thus, when a *call* instruction is encountered and the emulator cannot parse the next instruction at that address, the *call* instruction is skipped and the next one (*i.e.*, after *call*) will be executed.

The push operation is not identified by any known instruction such as "*push eax*". In another words, whenever the register ESP is decreased, it is considered as a *push* operation; unless a relative call is executed.

5. Experimental Result

Our experiments can be divided into two major parts. On the one hand, we measured the effectiveness of our heuristics. On the other hand, we measured the resistant of them against false positive. For the first case, we used several shellcodes extracted from two known repositories: Packet Storm [20] and Shell-Storm [21]. For the first part of our experiments, detection effectiveness of our prototype has been compared in contrast to Gene [5]. For the second case, we used two different dataset: (i) a collection of 10 million random binary files in different sizes, and (ii) a collection of 400 Windows executable binary files selected randomly from different formats such as EXE, DLL, MSI, etc.

5.1. Detection Effectiveness

First evaluation of our prototype was about the detection accuracy (i.e. false negative rate). For Windows targets, there are 34 functional shellcodes in Packet Storm repository. Abundance of any type of shellcode, according to the behaviors described in this paper, has been illustrated in Figure 2. As it was expected, PEB is the most reliable method for shellcode construction. However, there are multiple functional shellcodes which cannot be detected by Gene, because of the lack of our new heuristics described in this paper. Detection effectiveness comparison between our prototype and Gene has been depicted in Figure 3. Only one shellcode could not be detected by our prototype in which return oriented programming [22] is used. A shellcode constructed by this method contains no executable code and so cannot be detected easily by an emulation-based shellcode detection system. Second dataset of functional shellcodes is a collection of 54 shellcodes extracted from Shell-Storm repository. Abundance of any type of shellcode, according to the behaviors described in this paper, has been illustrated in Figure 4. There is an ROP-based shellcode in this collection too and so cannot be detected by our emulation-based detector. Most of these shellcodes uses PEB method and it shows the reliability of this method. However, there are still several shellcodes which cannot be detected by Gene. Detection effectiveness comparison between our prototype and Gene has been depicted in Figure 5.

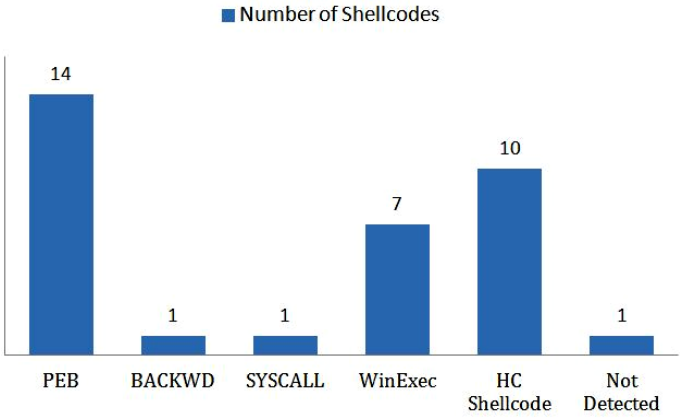

Figure 2. Abundance of any Type of Shellcode in Packet Storm Shellcode Collection

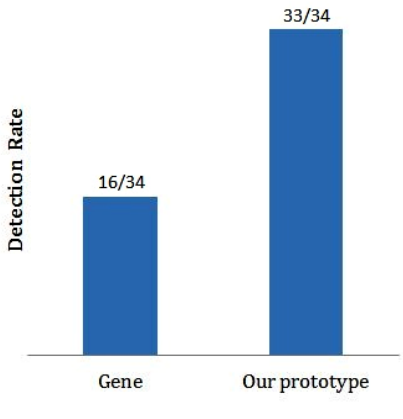

Figure 3. Detection Effectiveness Comparison between our Prototype and Gene with Respect to Packet Storm Shellcode Collection

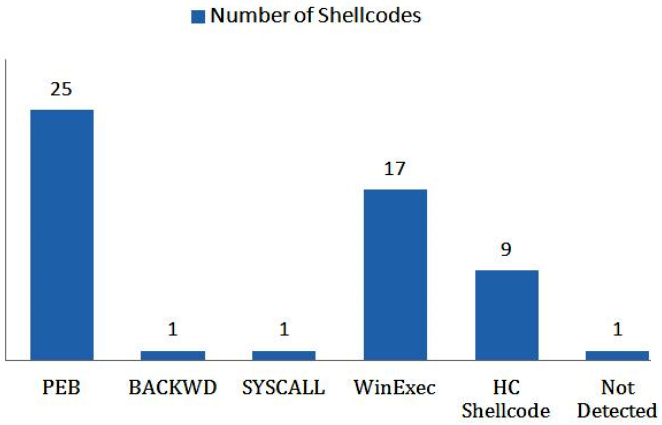

Figure 4. Abundance of any Type of Shellcode in Shell-Storm Shellcode Collection

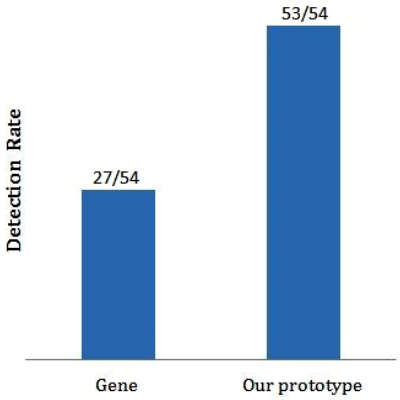

Figure 5. Detection Effectiveness Comparison between our Prototype and Gene with Respect to Shell-Storm Shellcode Collection

5.2. Heuristic Robustness

Evaluation of the robustness of our heuristics has been done against two different dataset: (i) a set of 10 million random binary files, and (ii) 400 different Windows executable files. In our experiments, the resistance of the heuristics was measured and also the reaction of every heuristic against shellcode-free data was explored. In Figure 6 it has been shown that how many times the first or the second condition of every heuristic is satisfied. For instance, the first condition of SEH heuristic is satisfied just for 6 executable files. There was no false positive for this dataset. Similar result has been shown in Figure 7 for the second dataset (*i.e.*, random binary files). As you can see, there are 18 false positives for this dataset. Based on the abundance of this dataset, false positive rate is 0.00018% which is acceptably low. Note that only 4 out of these 18 false positives are related to *HC Shellcode* heuristics. It indicates the robustness of our heuristic against false positive in contrast to other previously proposed heuristics.

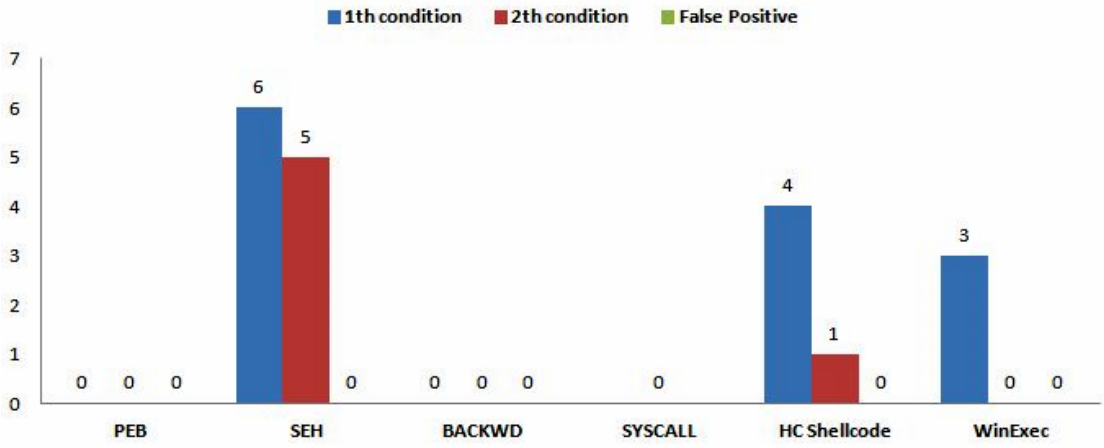

Figure 6. Resistance of our Shellcode Detection System against False Positive based on 400 different Windows Executable Files

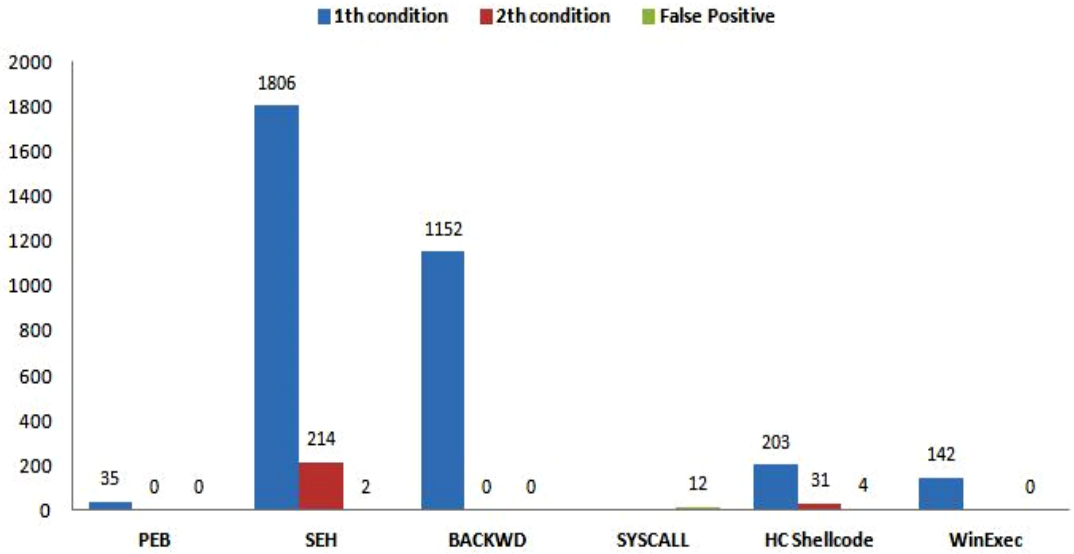

Figure 7. Resistance of our Shellcode Detection System against False Positive based on 10 Million Random Binary Files

6. Discussion

Although the emulation approach is a time-consuming method for shellcode detection, it can be equipped with multiple optimizations. Furthermore, by using a quick emulator such as [23], this approach can be used in a host-level detector as the input traffic rate of a host is significantly lower than a network.

Every code injection attack in which there is no executable code such as swarm attack [24] or those attack vectors constructed by return oriented programming (ROP) cannot be detected by an emulator easily. Some research efforts [25, 26] have been done for detecting ROP-based but they cannot be used at network layer in an environment-independent manner. Some of them need the source code of the protected application and some of them simulate an environment such as the process of Adobe Reader to detect only a special type of code injection attacks. However, a full-ROP-based shellcode construction is so hard and sometimes impossible since there may not be exists needed gadgets. In reality, ROP is an exploitation (*i.e.*, not shellcode writing) technique but it may be possible to construct a full-ROP-based shellcode.

7. Conclusion

With respect to the increased professionalism of cyber attacks and the vast number of code injection attacks, it is necessary to enhance the effectiveness of code injection attack detection. There are several publicly available shellcodes in different repositories which can be used by everybody. A shellcode detection system can be considered as a good defense against this type of cyber attacks only when it can detect at least every public shellcode. One of the most common types of shellcode which cannot be detected by existing shellcode detection systems is those shellcodes in which the hard-coded addresses of Windows API function are used. These shellcodes can be obtained easily in the wild. Thus, we decided to design a robust heuristic for detecting them in order to augment the collection of the previously proposed heuristics.

The experimental evaluation in this paper shows that the proposed heuristic can effectively detect those shellcodes in which the hard-coded address of WinExec or at least two other API functions has been used. There are 24 *WinExec* shellcodes and 19 *HC Shellcodes* in two selected shellcode collections which all are detected by our proposed heuristics. Since the proposed heuristic, called *HC Shellcode*, is designed based on the overall behavior of shellcodes, it may be a good idea for designing a general run-time heuristic for detecting every known and unknown shellcode.

References

[1] M. Polychronakis, K. G. Anagnostakis and E. P. Markatos, "Network-level polymorphic shellcode detection using emulation," *J. Comput. Virol.*, vol. 2, no. 4, **(2007)**, pp. 257-274.
[2] M. Polychronakis, K. Anagnostakis and E. Markatos, "Emulation-based detection of non-self-contained polymorphic shellcode", Recent Advances in Intrusion Detection, **(2007)**, pp. 87-106.
[3] M. Polychronakis, K. G. Anagnostakis and E. P. Markatos, "Real-world polymorphic attack detection using network-level emulation", Proceedings of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead, **(2008)**, pp. 21.
[4] Q. Zhang, D. S. Reeves, P. Ning and S. P. Iyer, "Analyzing network traffic to detect self-decrypting exploit code", Proceedings of the 2nd ACM symposium on Information, computer and communications security, **(2007)**, pp. 4-12.
[5] M. Polychronakis, K. G. Anagnostakis and E. P. Markatos, "Comprehensive shellcode detection using runtime heuristics", Proceedings of the 26th Annual Computer Security Applications Conference, **(2010)**, pp. 287-296.
[6] D. Zimmer. Understanding the PEB_LDR_DATA Structure. Available: http://sandsprite.com/CodeStuff/Understanding_the_Peb_Loader_Data_List.html.
[7] P. Van Eeckhoutte. Exploit writing tutorial part 9: Introduction to Win32 shellcoding, **(2010)**. Available: https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32-shellcoding/.
[8] P. Van Eeckhoutte. Exploit writing tutorial part 8: Win32 Egg Hunting, **(2010)**. Available: https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/.
[9] M. Miller. Understanding Windows Shellcode, **(2003)**. Available: http://www.hick.org/code/skape/ papers/win32-shellcode.pdf.
[10] T. Toth and C. Kruegel, "Accurate buffer overflow detection via abstract pay load execution", Recent Advances in Intrusion Detection, **(2002)**, pp. 274-291.
[11] S. Andersson, A. Clar and G. Mohay, "Network based buffer overflow detection by exploit code analysis", **(2004)**.
[12] C. Kruegel, E. Kirda, D. Mutz, W. Robertson and G. Vigna, "Polymorphic worm detection using structural information of executables", Recent Advances in Intrusion Detection, **(2006)**, pp. 207-226.
[13] R. Chinchani and E. Van Den Berg, "A fast static analysis approach to detect exploit code inside network flows", Recent Advances in Intrusion Detection, **(2006)**, pp. 284-308.
[14] X. Wang, Y. C. Jhi, S. Zhu and P. Liu, "Still: Exploit code detection via static taint and initialization analyses", Computer Security Applications Conference, 2008. ACSAC 2008. Annual, **(2008)**, pp. 289-298.
[15] U. Payer, P. Teufl and M. Lamberger, "Hybrid engine for polymorphic shellcode detection", Detection of Intrusions and Malware, and Vulnerability Assessment, **(2005)**, pp. 565-565.
[16] M. Egele, P. Wurzinger, C. Kruegel and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks", Detection of Intrusions and Malware, and Vulnerability Assessment, **(2009)**, pp. 88-106.
[17] K. Borders, A. Prakash and M. Zielinski, "Spector: Automatically analyzing shell code", Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, **(2007)**, pp. 501-514.
[18] Y. Fratantonio, C. Kruegel and G. Vigna, "Shellzer: a tool for the dynamic analysis of malicious shellcode", Recent Advances in Intrusion Detection, **(2011)**, pp. 61-80.
[19] P. Baecher and M. Koetter, "libemu – x86 Shellcode Emulation", **(2007)**. Available: http://libemu.carnivore.it/.
[20] Packet Storm. Available: http://packetstormsecurity.com/.
[21] Shell-Storm | Shellcode | Windows. Available: http://www.shell-storm.org/shellcode/shellcode-windows.php.

[22] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)", Proceedings of the 14th ACM conference on Computer and communications security, **(2007)**, pp. 552–561.
[23] QEMU, open source processor emulator. Available: http://www.qemu.org/.
[24] S. Chung and A. Mok, "Swarm attacks against network-level emulation/analysis", Recent Advances in Intrusion Detection, **(2008)**, pp. 175-190.
[25] V. Pappas, M. Polychronakis and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization", Security and Privacy (SP), 2012 IEEE Symposium on, **(2012)**, pp. 601-615.
[26] M. Polychronakis and A. D. Keromytis, "ROP payload detection using speculative code execution", Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on, **(2011)**, pp. 58-65.
[27] J. Khodaverdi, "Enhancing the Effectiveness of Shellcode Detection by New Run-time Heuristics", Int. J. Comput. Sci., vol. 3, no. 02, **(2013)**, pp. 02-11.

Authors

Javad Khodaverdi is a master of science in Information Security graduated from Amirkabir University of Technology – Tehran Polytechnic. As his undergraduate thesis, he focused on bot and botnet activities. So his research interest was about botnet detection but after graduating from Amirkabir University, as his thesis was about enhancing the performance of shellcode detection using emulation approach, now he is working on Shellcode Detection either in network or host level and also Web Application Vulnerabilities. Nowadays, he works as a researcher in ECIS Lab. in Amirkabir University of Technology. His technical experiences include Security Operation Center (SOC) development and management, Website designing and developing, and network security management.

Farnaz Amin is a master of science in Computer Networks graduated from Yazd University, Iran. Her undergraduate thesis was in the scope of botnet, particularly botmaster traceback. After graduating from Yazd University, as her thesis was about queuing management based on self-similarity in network traffic, and with respect to her previous interest, she is working on both scopes now. The most current research of Ms. Amin is in the scope of information security, particularly shellcode detection through dynamic analysis. She has the technical experience of working as a network manager.

