Advances in Computer Science and Information Technology (ACSIT)
Print ISSN: 2393-9907; Online ISSN: 2393-9915; Volume 2, Number 6; April-June, 2015 pp. 569-572
© Krishi Sanskriti Publications
http://www.krishisanskriti.org/acsit.html

An Architectural Approach to Forestalling Code Injection Attacks

Ayushi Chaudhary[1] and Gaurav Rajuara[2]

[1,2]*Dept. of Computer Science and Engineering Galgotias College of Engineering and Technology Greater Noida, India*
E-mail: [1]ayushichaudhary24@yahoo.com, [2]rajaura.gaurav@gmail.com

Abstract—*Network security policies offers no protection against attacks which do not rely on executing code injected by the attacker. The existing system follows von Neumann architecture, in which the memory cannot split into several segments. To forestall the code injection attack, the memory architecture is changed by virtually Splitting it into two segments i.e. code segment and data segment. The change in architecture does not allow the intruder to take charge of the injected code, as the injected code remains no executable. The split memory technique follows Harvard Architecture. Also, Address space layout randomization is followed, where the data are stored in various locations and not as whole in a single memory location. The intruder or an attacker can be tracked by knowing their location, IP address, date and time of the attack etc, that are not available in the existing system. In this paper we introduce the code Injection technique for displaying the user content in the memory according to the content split into number of intruder's information. Our proposed technique also implements URL based attacks in the memory content of the users.*

Keywords: *URL based attacks, Memory split, Code injection Attack, Randomization.*

1. INTRODUCTION

Code injection method can be used by an attacker to describe code into a computer program to change the course of code injection of execution. The results of a code injection attack can be disastrous.

For instance, code injection is used by some computer worms to propagate.

Remote File Inclusion (RFI) is a type of vulnerability most often found on websites. It allows an attacker to include a remote file, usually through a script on the web server. The vulnerability occurs due to the use of user-supplied input without proper validation. This can lead to something as minimal as outputting the contents of the file, but depending on the severity. The existing system follows von Neumann architecture. Where the memory cannot split into several segments. This type of technique allows the intruder to inject the code in the single segment and executes it. The intruder takes control of the entire code running in the system and it grants access to modify the data and perform activities without the knowledge of the authorized users. Also address space layout randomization is not possible, the entire data are stored in the single address space, and it allows the third party member to obtain all the valuable information, which is being stored in the database.[1]

Most of the web applications are addicted towards code injection attacks. Data is injected by an intruder or an attacker and that third person takes control of the entire system thus leading to loss of secured data and also malfunctioning of the entire system. If the system is attacked, the attacker is not known by the administrator and the person remains invincible. This leads to many disorders in the web applications.

Code injection attacks can be prevented by virtual splitting of memory i.e. code segment and the data segment. It is based on Harvard Architecture. The memory space is allocated in such a way that the code and data segment of the system are stored separately. The injected code remains in the data segment and it will not be executed as it makes unavailable for the processor during the instruction fetch from the memory. Also, the tracking facility enables the administrator to detect the intruder with their IP address, system name, path, location etc. It allows the administrator to take necessary action on the intruder.[2-3]

Code injection can be prevented with Address space layout randomization phase, preventing code injection phase. In this intruders attack by means of URL is prevented by ASLR phase. The intruders can't guess by means of the URL displayed in the Address bar. In the code injection prevention, the system will run code only when it is inserted from the Administrators IP address and host name. Thus intruder's code is not executed and prevented from huge disaster to the website.

2. RELATED WORK

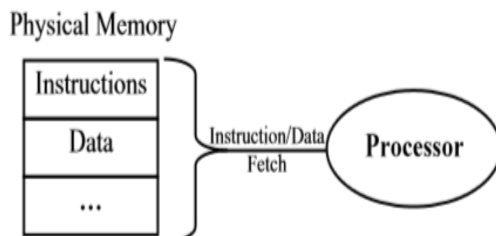
Research on code injection attacks has been ongoing for a number of years now, and a large number of protection methods have been researched and tested. There are two classes of techniques that have become widely supported in modern hardware and operating systems; one is concerned with preventing the execution of malicious code after control

flow hijacking, while the other is concerned with preventing an attacker from hijacking control flow. The first class of technique is concerned with preventing an attacker from executing injected code using no executable memory pages, but does not prevent the attacker from impacting program control flow.

This protection comes in the form of hardware support or a software only patch. Hardware support has been put forth by both Intel and AMD that extends the page-level protections of the virtual memory subsystem to allow for non-executable pages. (Intel refers to this as the “execute-disable bit”.[4] The usage of this technique is fairly simple: Program information is separated into code pages and data pages. The data pages (stack, heap, bss, etc) are all marked no executable. At the same time, code pages are all marked read-only. In the event an attacker exploits a vulnerability to inject code, it is guaranteed to be injected on a page that is nonexecutable and therefore the injected code is n ever run. Microsoft makes use of this protection mechanism in its latest operating systems, calling the feature Data Execution Protection (DEP). This mediation method is very effective for traditional code injection attacks, however it requires hardware support in order to be of use. Legacy x86 hardware does not support this feature. This technique is also available as a software-only patch to the operating system that allows it to simulate the execute-disable bit through careful mediation of certain memory accesses. PAX PAGEEXEC is an open source implementation of this technique that is applied to the Linux kernel. It functions identically to the hardware supported version, however it also supports legacy x86 hardware due to being a software only patch.

3. EXISTING SYSTEM

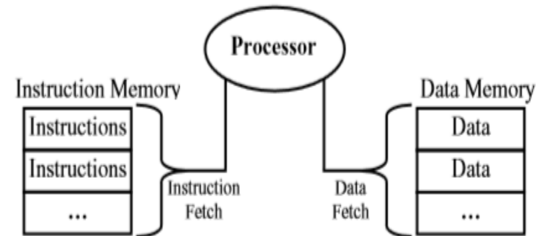
To forestall the code injection attack, the memory architecture is changed by virtually Splitting it into two segments i.e. code segment and data segment. The change in architecture does not allow the intruder to take charge of the injected code, as the injected code remains no executable. The split technique follows Harvard Architecture[9,10]. Also, Address space layout randomization is followed, where the data are stored in various locations and not as whole in a single memory location. The intruder or an attacker can be tracked by knowing their location, IP address, date and time of the attack etc, that are not available in the existing system



Von Neumann architecture

4. PROPOSED SYSTEM

To understand how the most basic shell injection might work, imagine a simple case. A custom script is needed to display file contents to users, but the development team doesn't want to spend time writing a procedure to read the files. Instead, they decide to allow users to specify a file, then use the Unix command cat to display the results.[5-8]



Harvard architecture

Algorithm:
Split Memory Page Fault Handler

Input: Faulting Address (addr), CPU instruction pointer (EIP), Pagetable Entry for addr (pte)

```
1  if addr == EIP then          /* Code Access */
2      pte = the_code_page;
3      unrestrict(pte);
4      enable_single_step();
5      return;
6  else                         /* Data Access */
7      pte = the_data_page;
8      unrestrict(pte);
9      read_byte(addr);
10     restrict(pte);
11     return;
12 end
```

5. AUTHENTICATION PHASE

This is the first module of all applications which contains the user registration and login and administrator’s login. In the previous stages, an unknown user also can block the valid user account without knowing the password of the account holder. This is one type of intruder. In the first phase if the user wrongly types the password simultaneously (more than 3 times) then the login will be transferred to a temporary (fake) account page. The intruder does not know that he is in a fake page as it resembles original page.[9]

6. ADDRESS SPACE INTRUSION AVOIDANCE PHASE

Address space layout randomization could be combined with this phase to prevent the URL based attacks. Even if the

intruder attacks the system through URL the control will not be granted to the intruder. If the intruder wants to move to next page after the authentication through URL the user remains in the same address, but the page that is being displayed will be different.[10]

7. PREVENTING CODE INJECTION PHASE

When the intruder tries to modify any data or create any malicious event, the intruder is not permitted to perform the activities since intrusion is done with unauthorized user name and password. If the changes are done with unauthorized access then the information of the intruder are gathered and it is being sent to the administrator in the secure manner.

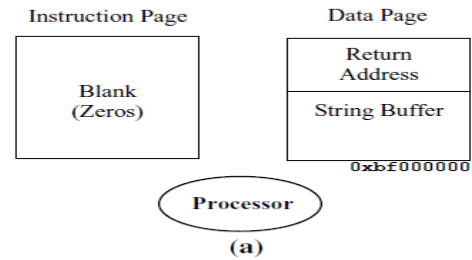
EMPERICAL RESULT

In this section we describe the two consecutive terms of URL based attack detection. Input design is the process of converting the user oriented input to the computer oriented format. Authentication module is used to log in to the system and perform the operations. Split memory module is used to separate the code and the data segment. Preventing code injection module helps the administrator to know the details of the intruder. The details are collected and it is stored in the database. Output design generally refers to the results and information that are generated by the system for many end-users; output is the main reason for developing the system and the basis on which they evaluate the usefulness of the application. In this system, with the authenticated user name and password, the user can perform the operations without any restrictions. If the users want to update the data or transfer the amount, the action can be done successfully, where as if the intruder logs in without knowing the password or user name there by giving false details more than three times, the intruder is redirected towards a fake page where the details of the intruder can be tracked. Also when an attacker wants to update any data, the updation is done only temporarily and it is not stored or updated in the database.

To the third person the transaction is restricted and on clicking the 'view details' only fake details are displayed. Thus any attack performed by the third person is blocked or restricted. The attacker may probably corrupt various parts of a program's memory space. Due to the fact that the operating system doesn't understand the working of the running program, it would be infeasible for it to attempt any sort of recovery that would permit the application to continue running. It may be much more feasible, for the application itself to register a call-back function or a special signal handler that the operating system could transfer execution to in the event an attack is detected. In our approach above mentioned problems can be discussed in the malicious attackers.

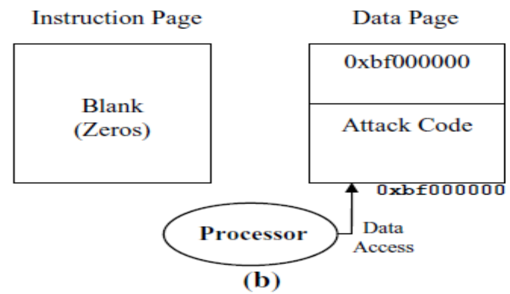
Assume for a moment that you have found the previous examples page, which takes as an argument a filename as input and executes the shell command "cat" against that file. In the previous example, a semicolon was used to separate out one command form another, to indicate that after the cat command completed, another function should be called in the same line. It is reasonable to assume that a more advanced developer might have filtered out some forms of shell injection, such as by removing semicolons, rendering the previous attack ineffective. There are a number of ways to string shell commands together to create new commands. Here are the common operators you can use, as well as examples of how they might be used in an attack:

1- before the attacker injects the code



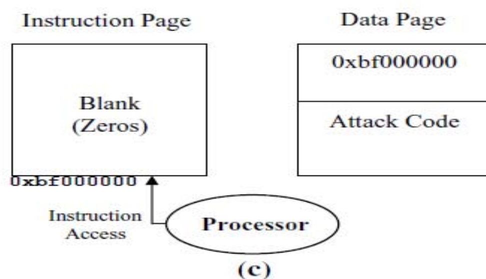
(a)

2- injection to the data page



(b)

3-the execution get routed to instruction page



(c)

LIMITATIONS

There are a few limitations to our approach. First, when an attack is stopped by our system the process involved will crash. We offer no attempt at any sort of recovery. This means an attacker can still exploit flaws to mount denial-of service

attacks. Second, as shown in other work[15], a split memory architecture does not lend itself well to handling self-modifying code. As such, self-modifying programs cannot be protected using our technique. Next, this protection scheme offers no protection against attacks which do not rely on executing code injected by the attacker. For example, modifying a function's return address to point to a different part of the original code pages will not be stopped by this scheme. Fortunately, address space layout randomization [11,12,13,14] could be combined with our technique to help prevent this kind of attack. Along those same lines, non control- data attacks, wherein an attacker modifies a program's data in order to alter program flow, are also not protected by this system. We have also not analyzed the system's functionality on programs that include dynamically loadable modules (such as DLL files on windows) but do not anticipate that such programs would be difficult to support.

8. CONCLUSION

In this paper, we present an architectural approach to prevent code injection attacks. Instead of maintaining the traditional single memory space containing both code and data ,which is often exploited by code injection attacks, our approach creates a split memory that separates code and data into different memory spaces. Consequently, in a system protected by our approach, code injection attacks may result in the injection of attack code into the data space. However, the attack code in the data space can not be fetched for execution as instructions are only retrieved from the code space.

9. FUTURE ENHANCEMENT

One common problem with interceding during an attack is that while the attacker has not successfully executed his malicious code, the attacker may probably corrupt various parts of a program's memory space. Due to the fact that the operating system doesn't understand the working of the running program, it would be in feasible for it to attempt an y sort of recovery that would permit the application to continue running. It may be much more feasible, for the application itself to register a call-back function or a special signal handler that the operating system could transfer execution to in the event an attack is detected. This require changes to the existing applications and would need to investigate in future work.

REFRENCES

[1] https://www.golemtechnologies.com/articles/shell-injection

[2] Technet.microsoft.com/emus/library/cc723564.aspx,Feb 1, 2001

[3] https://www.golemtechnologies.com/articles/shell-injection Oct 9, 2011.

[4] Doi.ieeecomputersociety.org/10.1109/TDSC.2010.1 by R Riley- 2010 - Cited by 26 -Related articles Dec 15, 2010.

[5] Ryan Riley, Xuxian Jiang, Dongyan Xu,"An Architectural Approach to PreventingCode Injection Attacks", Nov 2010.

[6] A detailed description of the data execution prevention(dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003. http: //support.microsoft.com/kb/875352. Last accessed Dec 2006.

[7] Pax pageexec documentation.http://pax. grsecurity.net/docs/pageexec.txt. Last accessed Dec 2006.

[8] I. Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Intel Corp., 2006. Publication number 253668

[9] H. H. Aiken. Proposed automatic calculating machine.1937. Reprinted in *The Origins of Digital Computers Selected Papers*, Second Edition, pages 191–198, 1975.

[10] H. H. Aiken and G. M. Hopper. The automatic sequence controlled calculator. 1946. Reprinted in *The Origins of Digital Computers Selected Papers*, Second Edition, pages 199–218, 1975.

[11] Pax aslr documentation. http://pax.grsecurity.net/docs/aslr.txt. Last accessed Dec 2006.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. *12th USENIX Security*, 2003.

[13] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *14th USENIX Security*, 2005.

[14] J.Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent Runtime Randomization for Security. *In Proc. of 22nd Symposium on Reliable and Distributed Systems (SRDS) , Florence,Italy*,Oct.2003.

[15] J.Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, pages 18–27, Tucson, AZ,USA,Dec.2005.AppliedComputerAssociates,IEEE