HackSim: An Automation of Penetration Testing for Remote Buffer Overflow Vulnerabilities

O-Hoon Kwon[1], Seung Min Lee[1], Heejo Lee[2], Jong Kim[1], Sang Cheon Kim[3], Gun Woo Nam[3], and Joong Gil Park[3]

[1] Dept. of Computer Science & Engineering, Pohang University of Science and Technology
{dolphin, puhaha, jkim}@postech.ac.kr
[2] Dept. of Computer Science & Engineering, Korea University
heejo@korea.ac.kr
[3] National Security Research Institute
{wsound, daemon99, jgpark}@etri.re.kr

Abstract. We propose an extensible exploit framework for automation of penetration testing (or pen-testing) without loss of safety and describe possible methods for sanitizing unreliable code in each part of the framework. The proposed framework plays a key role in implementing HackSim a pen-testing tool that remotely exploits known buffer-overflow vulnerabilities. Implementing our enhanced version of HackSim for Solaris and Windows systems, we show the advantages of our sanitized pen-testing tool in terms of safety compared with existing pen-testing tools and exploit frameworks. This work is stepping toward a systematic approach for substituting difficult parts of the labor-intensive pen-testing process.

1 Introduction

Vulnerability scanning is deployed to check known vulnerabilities on a single system or a series of systems in a network. There are a number of scanning tools which are available publicly or commercially [1]. Penetration testing (or pen-testing) is a goal-oriented method similar to "catch-the-flag" that attempts to gain privileged access to a system using pre-conditional means that a potential attacker could manipulate. A tester, sometimes known as an ethical hacker, generally uses the same methods and tools used by attackers to undermine network security. Afterward, penetration testers report on the exploitable vulnerabilities they found and suggest strengthening steps needed to make their client's systems more secure [2,10,11]. Most security consulting firms provide pen-testing services by red teams or ethical hackers [3,4], and the market volume for these services is expected to grow substantially [5,19].

Vulnerability scanners provide automated scanning with user-friendly interfaces and extensible structures for updating new vulnerabilities. In addition, scanning is conducted using safe methods that do not produce unexpected impact on target systems, at the expense of false-positive results. Pen-testing is performed manually using the same methods a real attacker employs. Such a

C. Kim (Ed.): ICOIN 2005, LNCS 3391, pp. 652–661, 2005.

time consuming task as pen-testing provides visible and useful results from a deep investigation of a target system. However, pen-tesing may leave behind security holes or cause unintended damage to the system [6]. Thus, this safety problem is an obstacle in automating pen-testing procedures. Because recent pen-testing tools or exploit frameworks for pen-testing do not provide a sanitization method to deal with unreliable exploit code, safety of their pen-testing cannot be guaranteed [10,11,12,13].

Therefore, in this paper, we propose an extensible exploit framework as a foundation for automating pen-testing with safeguards and describe considerations for sanitizing unreliable code in each part of the framework. Also, we implement HackSim, an automated pen-testing tool, as the prototype system of the proposed framework. Current implementations of HackSim are able to exploit four well-known vulnerabilities in Solaris and three in Windows. Nevertheless, it is easy to add new vulnerability tests to HackSim. Also, we show two examples of sanitized exploit code that do not negate the benefits of pen-testing.

The remainder of the paper is organized as follows: We describe related works and the differences underlying our work in Section 2. Overall system architecture, the extensible exploit framework for pen-testing and the design consideration for sanitizing each part of exploit framework is presented in Section 3. We examine implementation issues and implementation results in Section 4. Finally, we summarize this paper and give concluding remarks in Section 5.

2 Related Works

Testing methodologies can be classified into two categories: blackbox testing and whitebox testing. Blackbox testing is used when the tester has no prior knowledge of a system. On the other hand, whitebox testing is used when the tester knows everything about the system – like a glass house in which everything is visible.

Blackbox testing is a very useful method for finding unpublished vulnerabilities and it can be performed quickly using automated tools such as SPIKE [9]. Using it, we can collect information that is necessary for testing vulnerabilities and we also can obtain exploit codes for the vulnerabilities that we want to check. However, such a tool terminates the system or service because it carries out random attacks in order to know whether the testing has succeeded or not. Thus, blackbox testing is inappropriate for finding potential vulnerabilities when the purpose of pen-testing is not to terminate system or service but to safely find vulnerabilities.

Several commercial pen-testing tools and open-sourced exploit frameworks using whitebox testing have been proposed. Canvas and Core Impact are commercial pen-testing tools that include a network scanner and exploit framework [10,11]. Also, open source projects such as Metasploit and LibExploit provide exploit frameworks for pen-testing [13,12]. These frameworks include libraries of common routines and tools to generate shellcode [4].

[4] In case of exploit codes for remote targets, shellcode is defined as a set of instructions injected into an exploited program and then executed on remote targets.

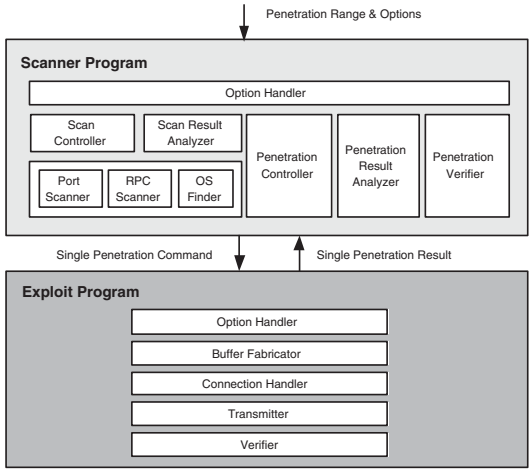

Fig. 1. HackSim Architecture Overview

Existing pen-testing tools and exploit frameworks have pros and cons. Those released as open source have not been fully verified in terms of safety. They produce lots of unreliable exploit code which causes pen-testing to fail and may lead to operating system and application crashes. Therefore, the safety of exploit code is highly required, but existing tools and frameworks are not concerned about the safety of exploit codes. They do not try to verify the safety of their pen-testing procedures.

In order to achieve appealing results for administrators and reliable pen-testing, we designed and implemented an automated pen-testing tool supporting the usability and safety of vulnerability scanners as well as the correctness of manual pen-testing.

3 HackSim Design

3.1 Architecture Overview

A top-down approach is used for describing our proposed system. First, we present the overall system architecture for automated pen-testing, which consists of two parts: "scanner program" and "exploit program". The scanner program is for processing user inputs and preparing corresponding parameters being passed to the exploit program. The exploit program is for launching the exploit codes in an extensible and safe way, which is described in the following subsections.

Components in the overall architecture are shown in Fig. 1. The lower part is for the exploit program, and the upper part is for the scanner program. When the scanner program generates penetration commands after getting user inputs such as the target range and other option values, the scanner program invokes the exploit program with commands generated by the scanner program.

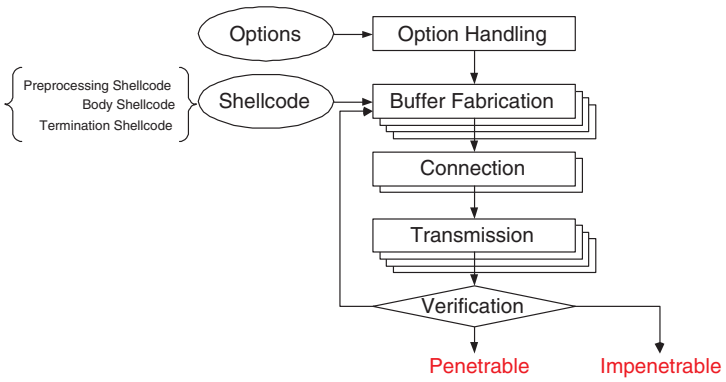

Fig. 2. Structure of the extensible exploit framework

The scanner program includes concise scanning functionalities, as shown in the dotted box in Fig. 1. The simple scanner performs OS fingerprinting and port scanning for checking the availability of targets and investigating information about them before penetration.

3.2 Extensible Framework for Exploit Codes

The exploit program of our pen-testing tool is designed as an extensible exploit framework for representing various exploit codes. Exploit codes are organized quite differently by the class of vulnerability. Among many classes of vulnerabilities, we confine our efforts to the remote buffer overflow vulnerability. The reason for this is that buffer overflows accounted for more than 50 percent of CERT advisories from 1996 to 2001 and 40 percent of the 20 most critical internet security vulnerabilities identified by the SANS Institute and the FBI [14,15].

From the analysis of public exploit codes for remote buffer overflow vulnerabilities, we can build an exploit framework that consists of five functions and two input data as illustrated in Fig. 2. This framework plays a key role as an engine for pen-testing and provides extensibility as a common platform for adding new exploit codes.

The option handling part is in charge of handling options for individual exploit codes in the framework. Different options in each exploit code are integrated into common interfaces in the option handling part.

The shellcode part is a set of instructions to be injected into an exploited program and executed on a target when the exploit works. It consists of the preprocessing shellcode, the body shellcode and the termination shellcode. The preprocessing shellcode is the preparation code to be executed before executing the body shellcode. For example, if the body shellcode contains a null character, the preprocessing shellcode should use a technique to avoid it. In Windows, if the body shellcode makes use of dynamic libraries, it also supports techniques such as PEB, SEH or TOPSTACK to retrieve function APIs(Application Programming

Interface) safely [17]. The body shellcode is a main set of instructions to be explicitly chosen by a pen-tester so that it should be executed on a target in case of successful exploitations. The termination shellcode returns the exploited hosts to their normal state. Note that this shellcode part can be reused for different exploit codes on the same OS and CPU architecture.

The buffer fabrication part adjusts the size of an input buffer, and writes an address on the buffer for returning to the prepared shellcode position. Therefore, every target service needs to be exploited using different buffer fabrication codes. The connection part is for establishing a connection to a remote service using either a socket connection or remote procedure call(RPC). The transmission part sends the fabricated buffer to the vulnerable position in the target service. To put the fabricated buffer onto the right position, the transmission part should talk to the target service with its own protocol until the overflow is caused.

The verification part returns the result after executing the shellcode. The result informs whether the penetration succeeds or not. If the penetration succeeds, the framework reports that the corresponding service is 'penetrable'. Otherwise, the penetration starts over from the buffer fabrication, as indicated by the outer arrow in Fig. 2. In each trial, the return address stored on the run-time stack is written again incrementally in order to fit the exploitable position. If the number of trials is not set by a tester, the exploit program repeats until it succeeds. However, if the penetration fails in the system that this brute force attack is not allowed, the framework reports that the corresponding service is 'impenetrable' at a trial.

3.3 Design Consideration for Sanitizing Exploit Codes

Pen-testing is performed using the same methods employed by an attacker. Consequently, it executes coded commands after exploiting vulnerable hosts. Thus, we have to make sure that this code is trustworthy and safe. In this section, we consider the sanitization of exploit codes to guarantee these needs.

Pen-testing should provide the assurance of safety. Safety can be handled by two parts: system part and service part. System safety means the safety of whole system and includes the safety of all services in the system. Service safety should meet the availability and the reliability of the service. Availability means that the service is in operation and reliability means that the service operates correctly. That is, we should verify that pen-testing against a service does not affect the safety of other services and the system by creating back doors, propagating worms, etc. In order to accomplish safe pen-testing, some considerations for sanitizing each module in Fig. 2 are described as follows.

The buffer fabrication part is very important in some cases, especially when a multi-threaded service in Windows is terminated after one of its threads generates an unhandled exception. Mis-prediction of a return address causes an application to crash and Windows does not allow brute force attacks. Some methods support relatively safe jumps to the shell code area by using the register. This method also works well in a multi-threaded environment [16].

The function of the connection and transmission part is to deliver a shellcode to the service of a remote system. In these parts, pen-testing will fail if there are problems caused by a tester or by other factors such as the network environment, service availability, etc. We take it for granted that there is no problem connecting to the service of a remote system and just concentrate on user faults to sanitize these parts. It is helpful to support libraries or modules that are divided by protocol and to contain functions making communication requests easily.

As mentioned before, shellcode can be divided into three parts: preprocessing, body and termination. Among them, the body shellcode is the one to let the tester execute arbitrary commands on the target machine. Therefore, this part should be carefully checked so that the body shellcode does not affect the integrity of the system and successfully sends results. Also, because the system call number or the address of a kernel service in system library is different in various operating systems, the body shellcode using kernel services should be checked carefully.

The termination shellcode returns the exploited service to its normal state. Most public exploit codes do not consider the importance of the termination shellcode for the safety of pen-testing. An infected thread or process rarely goes back to its normal state and polluted data makes it impossible to return a system to its normal state. The best choice is always to terminate the thread or process safely. To achieve this goal, we have to handle important tasks like releasing resources used by the thread and restoring data in shared memory, etc. It is not always a necessary task but the system might be impaired if we close our eyes to this matter. After that, we should terminate the thread by calling the corresponding function to exit it.

In some cases, not all modules of the extensible exploit framework need to be inspected, but it is highly recommended that the shellcodes are sanitized carefully, especially the body shellcode and the termination shellcode.

4 Implementation

This section describes the implementation issues of HackSim, based on previous system design. HackSim was implemented on Linux operating systems using C and Java for the exploit framework and the scanner program, respectively. It can do pen-testing against Windows and Solaris operating systems.

4.1 Modularizing Exploit Codes

To implement a prototype of the proposed exploit framework, we collected publicly available remote exploit codes for well-known vulnerabilities in Solaris and Windows. We analyzed them and selected 7 exploit codes for rapid prototyping of the proposed framework [14]. The characteristics of the seven vulnerabilities and their exploits are listed on Table 1. The number of exploit codes is small, but they cover the main exploits of Solaris and Windows. Their connection

Table 1. Exploit codes used for implementing the exploit framework

CVE Index	Target	OS	Connection	Vulnerability	Shellcode
CVE-2001-0236	snmpXdmid	Solaris	RPC	Stack Overflow	findsocket
CVE-2001-0797	telnetd	Solaris	Socket	Stack Overflow	bindsocket
CVE-2001-0803	dtspcd	Solaris	Socket	Stack Overflow	cmdshell
CVE-2002-0033	cachefsd	Solaris	RPC	Heap Overflow	findsocket
CAN-2003-0352	RPC-DCOM	Windows	RPC	Stack Overflow	bindsocket
CAN-2003-0533	LSASS	Windows	RPC	Stack Overflow	bindsocket
CAN-2003-0719	IIS-PCT	Windows	Socket	Stack Overflow	bindsocket

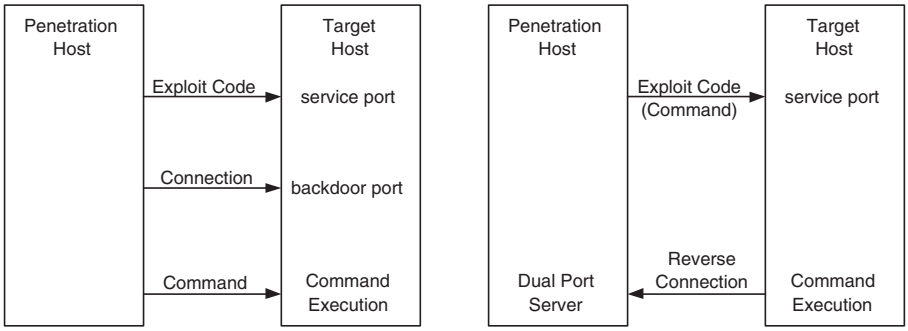

Fig. 3. Two ways for executing specified commands on a remote target host

methods included both Socket and RPC, and their vulnerabilities included both stack and heap overflow. In addition, their shellcodes manipulate commonly-used codes such as findsocket, cmdshell, or bindsocket [5].

To integrate diverse exploit codes into a single framework, the most important part is the shellcode. The proposed system uses the same shellcode for all the different exploit codes. Hence, the verification of executing the shellcode is integrated into the framework using the same code for each exploit code.

There are two ways for executing specified commands on a remote host using shellcodes as shown in Fig. 3. One is to execute commands on a root shell acquired by connecting to a backdoor port that is opened on a target host by shellcodes such as bindsocket. The other is to execute commands directly within shellcodes invoking a root shell. We selected the latter to avoid leaving any backdoors. We manually replaced the shellcodes of four exploit codes for Solaris with a common shellcode "cmdshell" executing reverse telnet commands on a target host [8]. Also, we replaced shellcodes of three exploit codes for Windows with a common shellcode "connectback", which provides the same result as executing reverse telnet commands [13].

[5] Assembly codes of findsocket, cmdshell, bindsocket are described in [7].

4.2 Sanitizing Exploit Codes

Sanitizing the Body Shellcode

If public exploit codes are used to perform penetration testing as they are, unexpected security holes may be left on target systems. For example, in Table 1, exploit codes for 'telnetd' and 'dtspcd' services may leave backdoors on target systems. The following code comes out of the exploit code for 'dtspcd' services, and this shows how the exploit code creates a backdoor.

```
cmd[] = "echo \"ingreslock stream tcp nowait root /bin/sh sh -i
\"$>$/tmp/.x; /usr/sbin/inetd -s /tmp/.x;/bin/rm -f /tmp/.x";
execl("/bin/sh","/bin/sh","-c", cmd, 0);
```

If the above code is executed on a target system, a backdoor port is opened on the system like the left figure of Fig. 3. Unless the backdoor port is closed explicitly, it will remain a severe security hole.

In order to execute specified commands without leaving any backdoor on a target system, we can use the following reverse telnet code [8].

```
cmd[] = "telnet target 1234 | /bin/sh | telnet target 5678";
execl("/bin/sh","/bin/sh","-c", cmd, 0);
```

Reverse telnet also allows the execution of commands on a compromised system even when the system is protected by a firewall. This is due to the fact that the security policy for incoming traffic is usually stricter than that of outgoing traffic. For the purpose of satisfying the requirements of safety and utilizing the benefits of reverse telnet, we used this technique in order to sanitize every exploit codes for Solaris.

But, if the above command is used in shellcode, the exploit code does not work because the second telnet session fails to write 'stdout' messages on the target hosts. Accordingly, in order to redirect 'stdout' messages of the second telnet session without printing out any messages, we added a '| sleep 1' command to *cmd*[].

Sanitizing the Termination Shellcode

If public exploit codes or the exploit framework of Metasploit is used in order to perform pen-testing on the LSASS [18] vulnerability, unintended damage on target systems occurs.

That is, they succeed in penetrating the systems, but if the command window is closed, the target host is rebooted in one minute. The reason is that a multi-threaded process in Windows is terminated when one of its threads generates an unhandled exception. This problem can be solved by adding the 'ExitThread' function to the end of the termination shellcode instead of 'ExitProcess'.

Also, we analyzed the RPCRT4 thread model because the exploit codes for the LSASS vulnerability affects the RPCRT4 thread. The analysis is focused on

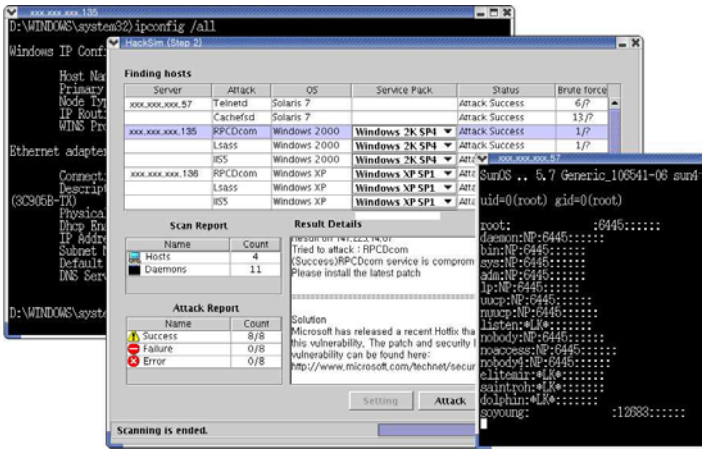

Fig. 4. A result of automated penetration testing using HackSim

whether the LSASS service works correctly after one thread is terminated using the 'ExitThread' function call. From this analysis, we got the positive result that the service is recovered and works well.

However, one problem remains. The exploit succeeds in Windows 2000 but not always in Windows XP. The reason is that one value in the data part of LSASRV.dll is changed to a wrong value during the exploit. The vulnerable function changes this value to zero when the last byte of shellcode is not a new line character, 0x0A. This zero value causes the following exploit or request to fail. This problem can be solved easily by assigning a non zero value to the data area or by adding a new line character to the last position of the shellcode.

4.3 Implementation Results

HackSim provides a high level of automation for labor-intensive pen-testing. Selectable options allow testing a wider range of targets and provide professional pen-testers with a flexible testing environment. Also, the result of penetration appears in the status window and provides collective evidence with higher accuracy than existing scanners. When the exploit works, the tool provides a privileged access on the newly created window in order to provide an evidence about exploited targets. By terminating the window, all connections are simply cleaned up without leaving behind any security hole. Fig. 4 shows pen-testing results using HackSim.

In addition, HackSim provides the extensibility for newly found vulnerabilities. This tool supports remote buffer overflow vulnerabilities that are used in the recent most worms. Also, it includes a sanitized shellcode that can be used commonly for all exploit codes. Therefore, HackSim can be easily extended to support exploit codes for newly found remote buffer overflow vulnerabilities.

5 Conclusion

In this paper, we have proposed an extensible exploit framework for an automation of pen-testing without loss of safety and described considerations for sanitizing unreliable codes in each part of the framework. Furthermore, a penetration testing tool, HackSim, is implemented on the basis of this framework. The enhanced HackSim can perform automated penetration testing without loss of safety and collect probed evidence if it succeeded in penetrating a system. Experiments against Solaris and Windows systems have shown how safely we can retrieve the most important information using automated pen-testing.

From the fact that penetrating a network is often done by exploiting a well-known weakness, this study is one step toward confirming the usefulness of automated pen-testing. The extension of HackSim to enhance the extensibility for newly found vulnerability and support the automation of the sanitization process remains for future work.

References

1. Joel Snyder, "How Vulnerable?," Information Security Magazine, Mar. 2003.
2. Pete Herzog, Open-Source Security Testing Methodology Manual(OSSTMM) 2.1, Institute for Security and Open Methodologies, 2003.
3. B. J. Wood and Ruth A. Duggan, "Red Teaming of Advanced Information Assurance Concepts," *DARPA Information Survivability Conference and Exposition (DISCEX)*, pp.112-118, 2000.
4. Chanrles C. Palmer, "Ethical Hacking," *IBM Systems Journal 3*, pp.769-780, 2001.
5. N. Wingfield, "It Takes a Hacker," *Wall Street Journal*, Mar. 11, 2002.
6. B. Skaggs, B. Blackburn, G. Manes, and S. Shenoi, "Network Vulnerability Analysis," *IEEE Midwest Symposium on Circuits and Systems (MWSCAS-2002)*, 2002.
7. UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes, The Last Stage of Delirium Research Group, 2001, http://lsd-pl.net.
8. J. Scambray, S. McClure, and G. Kurtz *Hacking Exposed. 2nd Ed.* pp. 319-321, Osborne: McGraw Hill, 2001.
9. Dave Aitel, "The Advantages of Block-Based Protocol Analysis for Security Testing", 2002, http://www.immunitysec.com/resources-papers.shtml
10. CANVAS Homepage, http://www.immunitysec.com/products-canvas.shtml.
11. CORE IMPACT Homepage, http://www.coresecurity.com.
12. LibExploit Homepage, http://www.packetfactory.net/Projects/libexploit.
13. Metasploit Homepage, http://www.metasploit.com.
14. Common Vulnerabilities and Exposures Homepage, http://www.cve.mitre.org.
15. The 20 Most Critical Internet Security Vulnerabilities, Version 4.0, October 8, 2003, http://www.sans.org/top20.
16. Jack Koziol, Dave Aitel, David Litchfield, Cris Anley, Sinan Eren, Neel Mehta, and Riley Hassell, *The Shellcoder's Handbook Discovering and Exploiting Security Holes*, pp. 49-53, Wiley Publishing, Inc., 1997.
17. Win32 Assembly Components, The Last Stage of Delirium Research Group, 2002, http://lsd-pl.net.
18. LASSS Vulnerability, http://www.microsoft.com/technet/security/bulletin/MS04-011.mspx.
19. TruSecure Homepage, http://www.trusecure.com.