A General Model and Guidelines for Attack Manifestation Generation

Ulf E. Larson, Dennis K. Nilsson, and Erland Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology,
Rännvägen 6B, 412 96 Gothenburg, Sweden
{ulf.larson,dennis.nilsson,erland.jonsson}@ce.chalmers.se

Abstract. Many critical infrastructures such as health care, crisis management and financial systems are part of the Internet and exposed to the rather hostile environment found there. At the same time it is recognized that traditional defensive mechanisms provide some protection, but has to be complemented with supervisory features, such as intrusion detection. Intrusion detection systems (IDS) monitor the network and the host computers for signs of intrusions and intrusion attempts. However, an IDS needs training data to learn how to discriminate between intrusion attempts and benign events. In order to properly train the detection system we need data containing attack manifestations. The provision of such manifestations may pose considerable problems and effort, especially since many attacks are not successful against a particular system version. This paper suggests a general model for how to implement an automatic tool that can be used for generation of successful attacks and finding the relevant manifestations with a limited amount of effort and time delay. Those manifestations can then promptly be used for setting up the IDS and countering the attack. To illustrate the concepts we provide an implementation example for an important attack type, the stack-smashing buffer overflow attack.

Keywords: Execution monitoring, automation, mutation, model, manifestation generation.

1 Introduction

The protection of critical assets available on an open network largely relies on the ability to detect malicious and abnormal activities. Most traditional defensive mechanisms can be subverted by a clever adversary. For example, authentication schemes can be rendered useless by identity theft: key cards can be stolen, passwords can be sniffed off the network or "shoulder surfed". Firewalls can be by-passed by stealthy attack packets and scanning techniques. Therefore, it is necessary also to use supervisory features, such as intrusion detection. Intrusion detection systems (IDS) monitor the network and the host computers for signs of intrusions and intrusion attempts. However, an IDS needs training data to learn how to discriminate between intrusion attempts and benign events. To produce

J. Lopez and B. Hämmerli (Eds.): CRITIS 2007, LNCS 5141, pp. 274–286, 2008.

accurate training data, there is a need to understand the functionality of the attacks that will generate the data. When an attack enters a system, it leaves traces in various places. If a log source is deployed at the location where the attack manifests itself, the attack could potentially be detected, provided that the system knows that the manifestations are generated by the attack. Thus, in order to properly train the detection system we need data containing attack manifestations. Attack manifestations can be obtained by executing successful attacks against a system and recording the resulting data traces and extracting the manifestations. In practice this poses considerable problems and effort, especially since many attacks are not successful against a particular system version. This paper suggests a general model for how to implement a feedback driven, fully automatic tool that can be used for generation of successful attacks and finding the relevant manifestations with a limited amount of effort and time delay. Those manifestations can then promptly be used for setting up the IDS and countering the attack.

The remainder of the paper is outlined as follows: Section 2 presents related work. Section 3 presents a general model and implementation guidelines for automated attack generation. In Section 4 we describe an example implementation of the model for a buffer overflow attack. Section 5 provides possible future work directions, and Section 6 concludes the paper.

2 Related Work

Previous work by Larson et al. [1] has successfully detected and extracted attack manifestations from data in system call logs. Lundin-Barse and Jonsson [2] showed that different attacks manifest themselves in different logs and that it is needed to monitor several logs. However, this previous work has used existing attacks and the process suffered from extensive attack configuration overhead. To shorten the time spent on attack configuration, there is a need for an automated method for attack generation to support rapid extraction of manifestations.

Earlier efforts in attack automation have largely focused on creating vulnerability exploitation tools, such as metasploit [3] and bidiblah [4], to simplify the process of collecting and executing attacks. The main goal of these tools are not to produce data for manifestation extraction, but rather to assess the security of deployed systems. In these systems, automation is made through the use of imported nessus [5] and nmap [6] scanning results [3] for attack selection. Neither metasploit nor bidiblah however provide feedback on how to modify the attack.

Other efforts have approached attack automation through mutation and genetic programming. Kayacik et al. [7] use genetic programming to evolve variants of successful buffer overflow attacks. Their approach does not use feedback from the target system to improve the attacks, and in order to create new variants, the original attack needs to be successful. Vigna et al. [8] use attack mutation to produce attack variants automatically. They use an *oracle* to determine whether an

executed attack is successful, but they do not use an automated method for providing feedback to the Mutator on how to modify a failed attack to become successful.

3 A Model for Automated Attack Generation

The overall purpose of this paper is twofold. First we identify components that are needed for automated attack generation and provide a conceptual model showing the relation of the components. Secondly, we identify a set of guidelines for how to implement a tool for generating specific attack types.

As a sound basis for the identification of components and implementation guidelines we state the following requirements:

1. *The model must be general.* It must be possible to implement components regardless of the specific attack type. No system or attack information can be assumed within the model. When this requirement is met, we can guarantee the generality of the model.
2. *The implementation must be efficient and entirely automated.* During attack generation, there should be no manual inspection of neither intermediate nor final results. When this requirement is met, we can guarantee a certain performance of the implementation, which outperforms the traditional trial-and-error method [9,10].
3. *The implementation must be secure.* Since successful attacks are created, the attack site must be shielded from the rest of the Internet to prevent attacks from leaking out. The purpose is not to produce attacks that can be used for malicious intent, but to provide log data for improvement of defensive systems.
4. *The implementation should be easily extensible.* A specific attack type may have other closely related attack types or variations. To easily add new variants of the attack, the model should be extensible. When this requirement is met, we can guarantee that newly discovered attacks which only differs slightly from already implemented ones can be implemented rapidly[1].

3.1 Component Identification and Conceptual Model Creation

According to requirement 1, the components of the model must be generally implementable. Which means that there must be no restrictions made regarding neither system type nor attack type when selecting components. More, the relation of the components must not assume any specific system setup but be applicable to a general system model.

Component identification. The overall goal is to *automatically* generate working attacks. This implies that if an attack is not successful immediately, the attack needs to be modified according to a scheme that will eventually render the attack successful. Based on this fact, we introduce the concept of *feedback*. Feedback is

[1] Which is necessary to keep up with the current pace of attack development.

used to successfully improve attacks until they either become successful, or they can be considered as non-working. To generate an automation loop, we apply the feedback concept to return information to the sender of the attack, which in turn resends the attack after modification. Thus, we need a component which can modify the attack according to the feedback. In order to produce data that can be used as feedback, we also need one or more components that can observe system assets and collect data. We identify four general components as follows:

- **Attack Tool:** The component which is responsible for sending attacks against a selected target or set of targets.
- **Mutator:** The component which uses feedback information to modify the attack.
- **Monitor:** The component which extracts data from the Sensors (see below), processes data and provides feedback regarding whether the attack is successful or not. This module must know what sensor data constitutes a successful attack and what sensor data that constitutes a failed attack.
- **Sensor:** The component or set of components which collect data. This data is used as a basis for the Monitor to provide feedback. The sensors must be selected to produce the data necessary for the Monitor to make its decision regarding the outcome of the attack.

The components are related as described in the following section.

Conceptual system model. The relation between the identified components of the model are shown in Figure 1. A general system model is shown which defines two systems, the Attack System and the Target System.

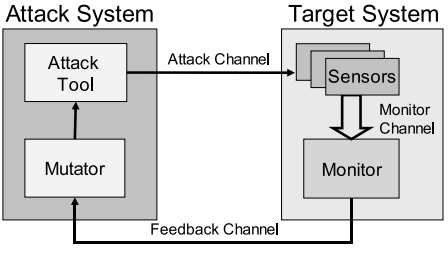

Fig. 1. The conceptual model consists of two systems. Each system contains a number of modules.

The Attack System contains the Attack Tool and the Mutator, and the Target System contains the Monitor and the Sensors. This is a natural division, since the Sensors and the Monitor need to be close to the target of the attack, while the Attack Tool and the Mutator are close to where the attack is generated.

Sequence of events. During operation, the components interact according to the sequence of events illustrated by the arrows in Figure 1.

The Attack Tool on the Attack System launches a generated attack over the Attack Channel against the Target System. The data that is generated inside the Target System is recorded by the Sensors and passed to the Monitor through the Monitor Channel. The Monitor then analyses the data to decide whether the attack is successful or not and sends feedback to the Attack System over the Feedback Channel. The Mutator uses the analysis from the Monitor and creates a modified attack. Thereafter, it instructs the Attack Tool to launch the attack against the Target System.

3.2 Implementation Guidelines

When the components are identified and the general conceptual model has been established, it is time to look further into how to use the model to implement an attack generation tool. For this purpose we propose a set of guidelines. To identify guidelines, we identified the information that is required by each component to work properly. Based on the required information, we identified the following guidelines: *Choose attack type, Determine attack characteristics, Select sensors, Design the monitor* and *Design the mutator and the attack tool*. The guidelines are described in the following listing:

- **Choose attack type.** The selection of attack type is the first item to address, since activity related to the other guidelines depends on the attack type. Also, if the attack type come in many variants, such as buffer overflow attacks, one specific variant needs to be selected. We also need to consider specific issues, such as whether the attack is applicable to several systems and whether this should be taken into consideration or not.
- **Identify attack characteristics.** Based on the attack type, the appropriate attack characteristics should be identified. This is strongly dependent on the attack type and one must carefully select the limitations of the attack before proceeding with this step. The outcome of the attack depends entirely on how each characteristic is initialized, as discussed in Section 4.2. In addition, a list of complimentary terms and definitions related to the selected attack type need to be recorded.
- **Select sensors.** When the attack characteristics are known, the appropriate sensors should be selected. The selected sensors must be able to collect events that are generated by the attack. For example, if the selected attack is remotely executed and targets a network stack, then it is reasonable to assume that a sensor should be deployed at the network level and that it should be able to capture network related data.
- **Design the monitor.** The Monitor must be designed in such a way that it can discriminate between a successful attack and a failed one. It must also be able to identify the reason for why the attack failed and to provide feedback regarding which attack characteristic that should be modified next. The identification ability must be adapted to the enabled protection

mechanisms. Only if the Monitor can discover that a specific protection mechanism is in use, it can provide the necessary feedback for the Mutator to change the attack accordingly.[2] For example, a Monitor for a buffer overflow attack must be able to correctly identify the presence of address space layout randomization protection if this is enabled. Otherwise, this information can not be communicated to the Mutator, which in turn fails to produce an attack to evade the randomization protection.

- **Design the mutator and the attack tool.** The Mutator must be designed to be able to modify the attack characteristics according to the feedback from the Monitor. The Mutator needs to know how the attack works and how to combine attack characteristics so that a potentially successful attack is generated. The Mutator also needs to be aware of possible enabled protection mechanisms so that it can successfully reflect the feedback from the Monitor. If there are no strategy for handling a certain protection mechanism, the Mutator will not be able to produce a successful attack when this protection is enabled. The Attack Tool needs to be able to launch the selected attack type.

The implementation guidelines are general, and specific attack and system knowledge has been deliberately left out of the discussion at this level. The next section will apply the general model and guidelines to a specific case, namely the implementation of the model on a stack-smashing buffer overflow attack.

4 Creating a Buffer Overflow Attack Generator Using the Model Guidelines

This section describes the implementation of the model for a remotely executed, stack-smashing, buffer overflow attack type [11]. For each guideline, the implementation decisions are described and motivated. For extra information, a detailed description of the tool, including selected attack characteristics, the list of terms and definitions, tool operation and result discussion is available in [12].

4.1 Choose Attack Type

A particularly dangerous type of attack is the buffer overflow. A successful buffer overflow allows an attacker to inject arbitrary instructions into the flow of an attacked system. This means that the attacker can, in the worst case, take control of the system. More specifically, a buffer overflow attack is interesting based on the following facts:

- A successful attack allows for injection of arbitrary code in the flow of a running process. This implies that the attacker can cause the attacked process to execute the attacker's code with the privileges of the running process.

[2] Note that the purpose of the model is to produce attacks that can be analyzed to extract manifestations. Thus, the Monitor must be aware of the current protection mechanism in order to be useful.

- Since it is possible to insert arbitrary instructions, the attacks can easily be encoded to avoid signature based detection systems.
- Again, since it is possible to insert arbitrary instructions, the attacker can observe normal behavior of the process and then add instructions which mimic the normal behavior [13,14], but that still meet the attacker's goal.
- Buffer overflow attacks use weaknesses in popular programming languages. Therefore we can expect to find buffer overflow vulnerabilities in a variety of operating systems.
- A stack-smashing buffer overflow variant is easy to implement and works well as a proof-of-concept. There are many protection mechanisms that will render this simple attack type less effective [15,16], but for illustrative purposes the attack works well.[3]

Therefore, for our example, we select the buffer overflow attack, and since there are several flavors of the attack type, we also limit our implementation to the stack-smashing buffer overflow type. In addition, we also consider a remotely executed attack, since we believe that this is more common than the corresponding local attack.

4.2 Identify Attack Characteristics

A buffer overflow attack is crafted using a combination of a no-operation (NOP) instruction sled, an executable payload and a return address pointing to the NOP sled or payload, as illustrated in Figure 2.

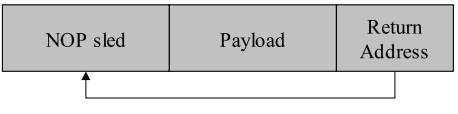

Fig. 2. A buffer overflow attack consisting of NOP sled, payload and return address

These three characteristics determine the outcome of the attack. If the NOP sled is too short or too long the return address will not be correctly aligned to overwrite the saved Extended Instruction Pointer (EIP), and the attack fails. If a payload, constructed for a different architecture than the target system, is used, the attack will fail since the payload contains instructions that are not valid for the target system. Lastly, if the return address included in the attack is incorrect, i.e., it is not pointing to the NOP sled or the payload (or any other code that one would like to execute), the application will interpret and execute the instructions that exist in the location that the return address is pointing to. The result is that the attack fails.

[3] Also, many of these protection mechanisms can in turn be evaded [17] and there are still many old systems running without protection mechanisms enabled.

4.3 Select Sensors

We selected two sensors for our tool implementation. First, since the attack is executed remotely we use a *Packet Capturer*, which captures packets of the network and stores the packets to a network packet log file. Second, since the attack targets a *Vulnerable Application*, we also need to monitor the behavior of the Vulnerable Application for the presence of a core dump, which occurs as an effect of a failed attack.

4.4 Design the Monitor

We designed an Execution Monitor to monitor the execution flow and memory state in the Vulnerable Application. Execution monitoring is vital since the buffer overflow attack targets memory buffers which after program execution are reclaimed by the operating system and thus cleared of content.

4.5 Design the Mutator and the Attack Tool

The Exploit Mutator modifies the attack by changing the NOP size, the return address and the payload according to the feedback from the Execution Monitor. The Mutator uses a *Payload Database* for selection of valid payload variants to use for the attack.[4]

The Attack Tool simply relays the information it receives from the Exploit Mutator to the Target System.

An implementation of the buffer overflow attack. The implementation of the buffer overflow attack type in line with the conceptual model is illustrated in Figure 3.

- **Exploit Tool:** Sends the newly generated attack to the Target System.
- **Exploit Mutator:** Uses the feedback information from the Target System to either modify the attack such that it eventually succeeds or to conclude that a working exploit is not possible. According to the feedback it re-initializes the attack characteristics, e.g., increases the NOP size.
- **Payload Database:** Contains a set of payloads [18,3]. The payload is selected from the database by the Exploit Mutator based on feedback from the Target System.[5]

The Target System consists of the Packet Capturer, the Packet Log, the Vulnerable Application and the Execution Monitor. The modules in the Target System are described below:

[4] In our implementation we used two types of payload, one bindshell and one reverse-shell.

[5] The metasploit framework [3] can easily be used to expand the Payload Database substantially.

- **Packet Capturer:** Listens to the network interface and dumps the incoming packets to the Packet Log.
- **Packet Log:** Contains packets captured by the Packet Capturer.
- **Vulnerable Application:** The Vulnerable Application listens to incoming data (up to 1024 bytes) on a port. The read data is then accessed by a badly written `strcpy` function [19] that can handle fewer bytes than the maximum read value (< 1024 bytes). Thus, a buffer overflow vulnerability exists [20,21,22,23,24].
- **Execution Monitor:** Examines the contents of the Vulnerable Application's stack and register values. If the attack fails it determines the NOP size of the attack by reading from the Packet Log, and calculates the return address. The Execution Monitor also determines the operating system type that is running on the Target System. After a thorough analysis the Execution Monitor provides feedback to the Attack System about which characteristic that caused the attack to fail.

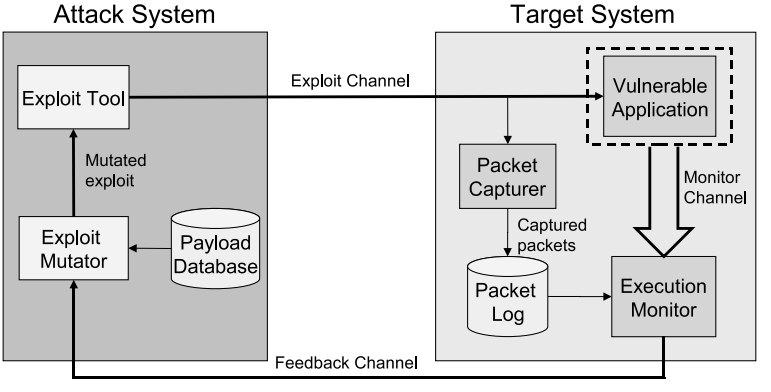

Fig. 3. The implementation of the buffer overflow attack

The implementation-specific sequence of events. The implementation-specific sequence of events is illustrated by the thick arrows in Figure 3 and is based on the event sequence model in Section 3.1. The sequence is described as follows:

First, the Exploit Tool on the Attack System launches a generated attack over the Exploit Channel against the Vulnerable Application running on the Target System. The network packets containing the attack are logged by the Packet Capturer and stored in the Packet Log.

Second, the Vulnerable Application processes the incoming data. The Execution Monitor examines the memory space of the Vulnerable Application over the Monitor Channel if it crashes. In particular, it inspects the register values to determine which characteristic that caused the attack to fail, e.g., the return address is wrong.

Third, the Execution Monitor sends feedback to the Attack System over the Feedback Channel. It points out which characteristic caused the attack to fail and also provides information on how the next attack should be modified.

Fourth, the Exploit Mutator uses the analysis from the Execution Monitor and re-initializes the attack characteristic accordingly. Thereafter, it instructs the Exploit Tool to launch the newly modified attack against the Target System over the Exploit Channel.

4.6 Results

To assess the flexibility of our buffer overflow tool implementation we conducted a series of tests. A detailed description of the tests as well as a thorough discussion of the results are found in [12].

To assess the accuracy of our tool, we used two types of payload, bindshell and reverseshell, and four operating system distributions, FreeBSD and three Linux flavors. For the three Linux distributions the tests were conducted with address space layout randomization, (VA) [25] enabled and disabled. We then observed the outcome of the attack generation for each test setup. Table 1 shows the results of generating attacks with the bindshell payload. The test setup used a fixed buffer size of 100 bytes.

Table 1. Attack outcome for the seven different operating system configurations using a buffer size of 100 bytes

Operating System	*VA*	*Result*
FreeSBIE		Success
Gentoo		Success
Knoppix		Success
Slax		Success
Gentoo	X	Success
Knoppix	X	Success
Slax	X	Success

The first column in Table 1 shows the operating system distribution, and the second column shows whether address space layout randomization (VA) protection is enabled (marked by an X) or disabled. The third column shows whether a working attack was successfully generated or not. We see that for all performed tests, it was possible to generate a successful attack.

To assess the efficiency we observed the number of feedback rounds that are needed to generate a successful attack. We count on a feedback loop time of approximately 1-3 seconds, depending on whether the Vulnerable Application crashes or not. Table 2 shows the number of feedback rounds required to successfully generate a successful attack for different buffer sizes.[6]

[6] For this test we used one type of payload (bindshell) and one operating system distribution (Slax) with VA enabled.

Table 2. Number of feedback rounds required to generate a successful attack

Buffer Size (bytes)	25	32	50	64	100	128	200	256	400	512	800
#Feedback rounds	10	10	12	12	12	14	14	16	16	18	18

These results show that the implementation worked well for generating successful attacks on a number of system configurations and also that it is efficient in generating attacks.

4.7 Applicability of the Model for Other Attack Types

We have described an implementation of the model for the buffer overflow attack and will now briefly describe a possible implementation for a denial of service [26] attack.

A denial of service attack attempts to reduce the availability of a resource by either exhausting bandwidth or depleting internal resources such as memory and CPU power. The attack may also target weaknesses in the communication protocols, such as depleting the pool of available communication sockets. A bandwidth exhausting attack could be implemented as follows: The Attack Tool sends to the Target System a mix of request packets and dummy packets. The request packet rate is held constant while the rate of dummy packets successively increases according to the instructions from the Mutator. The Sensors capture network packets of the link and the Monitor inspects the packet logs to investigate how the service, i.e., the number of received request packets and returned reply packets are affected by the increasing network load. Based on the observations, the Monitor either instructs the Mutator to increase the number of packets or concludes that the attack is successful.

5 Future Work

Future work includes creating a framework based on the general model. The framework will define common interfaces between components, which should simplify future attack implementation and allow for other attacks to be implemented by only replacing the affected components. A common, known interface would also make it easier for external parties to contribute with components, which would make development faster and allow for more attack types to be included. In particular, we intend to adjust our buffer overflow tool to generate denial of service attacks. This type of attack also poses a great threat against crisis management systems, since it may prevent legitimate users from viewing the pages of a web server or cause the web server to disconnect and crash.

Moreover, the configuration time saved by automatic attack generation could be spent on data collection with an extensive set of sensors. This would make it possible to thoroughly analyse what sensors provide the best manifestations for different attacks. This, in turn, can provide hints for sensor selection when a tool

for a specific attack type is to be implemented. Lundin-Barse and Jonsson [2] showed that different attacks manifest themselves in different logs and therefore, it is necessary to search for manifestations in various logs.

6 Conclusions

We have introduced a conceptual model and implementation guidelines for automatic generation of successful attacks within a specific attack type. Our effort is driven by the fact that a set of successful attacks are needed for attack analysis and manifestation extraction, which in turn is needed for attack detection. Attack detection is an important activity in computer systems, but particularly in crisis management systems, which may hold information that could save lives. We have also provided an example implementation of a potentially dangerous attack type.

References

1. Larson, U., Lundin-Barse, E., Jonsson, E.: METAL - a tool for extracting attack manifestations. In: Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment workshop (DIMVA), Vienna, Austria, July 7-8 (2005)
2. Barse, E.L., Jonsson, E.: Extracting attack manifestations to determine log data requirements for intrusion detection. In: Proceedings of the 20th Annual Computer Security Applications Conference, ACSAC 2004, Tucson, Arizona, USA. IEEE Computer Society, Los Alamitos (2004)
3. The metasploit framework (September 2006), `http://www.metasploit.com`
4. Bidiblah - security assessment power tools (September 2006), `http://www.sensepost.com/research/bidiblah/`
5. The nessus vulnerability scanner (September 2006), `http://www.nessus.org/documentation/index.php`
6. Nmap security scanner (September 2006), `http://insecure.org/nmap`
7. Kayacik, H.G., Heywood, M., Zincir-Heywood, N.: On evolving buffer overflow attacks using genetic programming. In: GECCO 2006 (July 2006)
8. Vigna, G., Robertson, W., Balzarotti, D.: Testing network based intrusion detection signatures using mutant exploits. In: ACM Conference on Computer Security (2004)
9. Puketza, N.J., Zhang, K., Chung, M., Mukherjee, B., Olsson, R.A.: A methodology for testing intrusion detection systems. In: 17th National Computer Security Conference, Baltimore, MD (1994)
10. Foster, J.C., Williams, A.: Sockets, Shellcode, Porting and Coding. In: Syngress, ch. 12 (March 2005)
11. Aleph One. Smashing the stack for fun and profit (1996), `http://www.theparticle.com/files/txt/hacking/phrack/p49.txt`
12. Nilsson, D.K., Larson, U., Jonsson, E.: A general model and guidelines for attack manifestation generation. Technical Report TR-2007:8, Department of Computer Science and Engineering, Chalmers University of Technology (2007)
13. Wagner, D., Soto, P.: Mimicry attacks on host based intrusion detection systems. In: Ninth ACM Conference on Computer and Communications Security (2002)

14. Tan, K.M.C., Killourhy, K.S., Maxion, R.A.: Undermining an anomaly-based intrusion detection system using common exploits. In: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (2002)
15. Cowan, C., et al.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium (January 1998)
16. Etoh, H.: GCC extension for protecting applications from stack-smashing attacks (ProPolice) (2003)
17. Richarte, G.: Four different tricks to bypass stackshield and stackguard protection. Technical Report NIST IR 7007, NIST (2002)
18. shellcode.org (June 2006), http://www.shellcode.org
19. Kelley, A., Pohl, I.: A Book on C, 4th edn., December 1997. Addisson-Wesley Professional (1997)
20. Erickson, J.: Hacking, the art of exploitation. No Starch Press, Inc. (2003)
21. Burebista. Remote automatic exploitation of stack overflows (2003), http://www.infosecwriters.com/text_resources/pdf/remote_overflows.pdf
22. contex. Exploiting x86 stack based buffer overflows (2006), http://www.milw0rm.com/papers/34
23. xgc/dx A.K.A T. Silva. Introduction to local stack overflow (2005), http://www.milw0rm.com/papers/4
24. Preddy. Buffer overflow tutorial (2006), http://www.milw0rm.com/papers/73
25. Address space layout randomization (Latest visited, July 2007), http://en.wikipedia.org/wiki/Address_space_layout_randomization
26. Denial-of-service attack (Latest visited July 2007), http://en.wikipedia.org/wiki/Denial-of-service_attack