

Mutation Resistant Runtime Code using Kernel Attestation

Kirti Chawla
Department of Computer Science
University of Virginia, Charlottesville
kirti@cs.virginia.edu

Taniya Siddiqua
Department of Computer Science
University of Virginia, Charlottesville
taniya@cs.virginia.edu

ABSTRACT

Viruses, Trojan horses and other malicious code have become increasingly prevalent in commodity applications. Malicious code attacks such as code injection exploit new and unknown vulnerabilities to reduce the overall availability of the applications and affect user productivity. These attacks insert a payload into the running application, mutating its runtime code, altering its runtime execution behavior, and potentially causing system-wide spread of the malicious code. In order to safeguard against mal-content and prevent mutation, routine inspection of the application runtime code is required. To mitigate such attacks, we propose a modified kernel-mode attestation technique that enables the end-user to verify the “*genuineness*” of the application runtime code. Our proposed kernel-mode attester verifies the code at runtime, augments static analyzers, and requires no modification to the source code of the application. Finally, our proposed method can be implemented economically with low in-memory space (~30 Kbytes).

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection – Invasive Software

General Terms

Design, Security

Keywords

Attestation, Code injection, Kernel

1. INTRODUCTION

Commodity applications are increasingly getting bigger and complex, and potentially have vast amount of “*fallible*” code embedded within them. Moreover, certain facilities provided by the underlying operating system can also be misused. These shortcomings manifest as vulnerabilities that are exploited by malicious code e.g. Viruses, and Trojan horses to mutate the runtime code, disrupt the execution, and reduce the avail-

ability of the application to the end-user. Code injection is a class of malicious attacks that inserts a payload¹ in an application at runtime. The “*tainted*” application then acts as a host to malicious code and helps in spreading the mal-content. Some of the most common implications of code injection attacks include remote command execution, process hijack, and application disruption. To guard against such threats, it is important that routine inspection of the application runtime code is performed. Our contributions towards this end are as follows:

- We classify code injection attacks based on 2 payload types and 2 delivery methods. We focus on code injection attacks based on the “*ptrace()*” system call and the dynamically linkable and relocatable (or DLR) objects in Linux operating system. Furthermore, we discuss arbitrary and self-modifying payload types, which can be used by the above payload delivery methods to mount code injection attacks. Also, we enumerate the implications of such attacks.
- We provide detail design of the kernel-mode attester that detects the mutation of application runtime code and enables the end-user to take appropriate actions. Our attester does not require any modification to the application source code and complements the static analyzers to mitigate the code injection attacks.
- We evaluate the proposed technique by measuring time and amount of address space traversal required to perform trustworthy attestation. Our proposed method is implemented using (~1200) lines of code and occupies (~30Kbytes) in memory.

¹ In this paper, we use “*payload*” to define the malicious part of the Virus, Trojan horse etc. and separate it from their propagation methods. Thus, every instance of malicious code can be seen as a carrier, whose purpose is to deliver a payload to an exploitable host application. By this definition, the purpose of payload is to further perpetuate the carrier and payload cycle via a host application.

This paper is organized as follows. In section 2, we present a motivating example that shows the use of operating system facilities to mount code injection attacks. We classify code injection attacks based on payload types and delivery methods in section 3. In section 4, we present the design of kernel-mode attester and its related modules. We evaluate the proposed technique in section 5, while section 6 highlights its limitation. We relate the proposed technique to other approaches in section 7, present directions of future work in section 8 and conclude in section 9.

2. MOTIVATING EXAMPLE

In this section we present an insight on the application code vulnerability due to the use of facilities provided by the underlying operating system. Although applications may themselves have vulnerable code, we show that by using a chosen set of operating system facilities, a malicious code can exploit almost any target application. Figure 1 depicts 2 examples of the code injection using the Windows and Linux operating system facilities. In Windows operating system, the following process and thread manipulation APIs are provided:

- **OpenProcess():** This API enables a process to open an existing local process object.
- **VirtualAllocEx():** This API allocates a specified amount of memory within the virtual address space of a given process. Initialized memory is set to zero.
- **WriteProcessMemory():** This API enables a process to write data to a specified memory area in a given process. The memory area should be accessible for the operation to succeed.

- **CreateRemoteThread():** This API enables a process to create a thread and run in the virtual address space of a given process.

These APIs enable writing code that can take advantage of multiple processors or cores in the hardware e.g. a weather forecasting application can have one parent process per processor dedicated for predicting weather at a site. Each such process then spawns remote threads and interacts with other processes to generate correlations between weather at various sites. We note that there can be numerous instances where the above APIs can be applied to good effect.

In figure 1(a), the same set of APIs can be used to create a malicious code that can inject a dynamic link library (or DLL) into a victim process. As the DLL becomes part of the address space of the victim process, it can overwrite portions of code segment, invoke process-specific functionality, and more to mutate runtime code and cause disruption in normal execution behavior. A similar but restricted set of APIs and facilities that are provided in Linux operating system are given below:

- **ptrace():** This API enables a parent process to observe and control the execution of a given process. It also allows the parent process to examine the core image and registers of the given process.
- **/proc/<pid>/maps:** This entry point in virtual file-system enables a process to know the memory-map of a given process having process-id “*pid*”.
- **ELF header:** The header of this executable file format maintains a log containing details of type of file, architecture, section for text, data, bss segments, and more. Linux uses this format for binary objects.

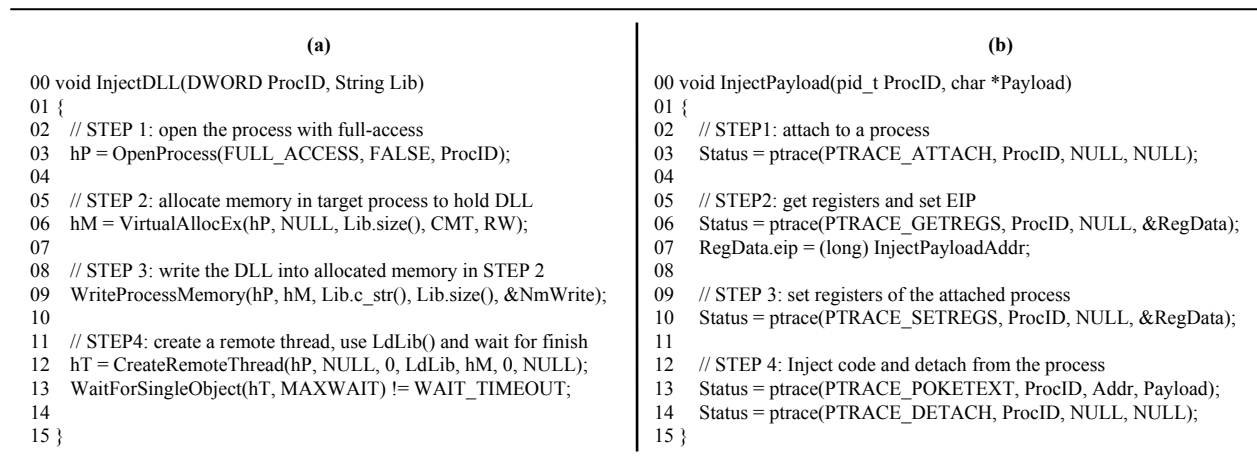


Figure 1: Code injection attack code based on the operating system facilities – (a) on Windows, and (b) on Linux

3.2 Payload Type

A malicious code expands its sphere of influence via its payload. A basic payload enables the malicious code to copy itself and spread, while a more sophisticated payload can cause specific actions such as file deletions, BIOS reset and more. It is a non-trivial task to design rule-based system that can intercept current and new payload types. We handle this problem by introducing a classification mechanism that segregates the payload into 2 types.

3.2.1 Payload Type I: Arbitrary Code

An arbitrary code as payload type handles all the combinations of the payload contents. It can be seen a sequence of bytes and has a definite size. Formally, we define the arbitrary code as below:

$$\forall P; \quad \langle B_1, B_2, B_3 \dots B_{|P|} \rangle \quad \dots \quad (1)$$

Where, P is a payload, B_I is byte at location I, and $\langle \cdot, \dots \cdot \rangle$ is an arbitrary code. Note that for an arbitrary code of size |P| bytes, there are $\{|P|!\}$ payloads. Also, if the size of arbitrary code is not constrained then there are $2^{|P|} - 1$ arbitrary codes of various sizes up to |P| bytes. Payload types such polymorphic code, encrypting code and more can be seen as a permutation or translation of an arbitrary code. Hence, by definition they are in the closed set of arbitrary codes.

3.2.2 Payload Type II: Self Modifying Code

A self-modifying code as payload type handles a special case of arbitrary code, wherein the code can modify itself. It can be seen as a payload that executes and then rewrites itself with another payload and continues the execution. Formally, we define the self-modifying code as below:

$$\forall P, I, J; \quad \langle P_I, P_I \leftarrow P_J, \text{repeat}_K \rangle \quad \dots \quad (2)$$

Where, P is a payload, I and J are instances of payload, $\{X \leftarrow Y\}$ is the rewrite of the payload X by payload Y, and $\langle \cdot, \dots \cdot \rangle$ is a K-self-modifying code. The definition allows potentially infinite rewrite operations. Note that the case of 1 rewrite is handled in this definition. This payload is distinguished from the other types of payload, due to its dominant prevalence in current state-of-the-art malicious code. Payload types such as metamorphic code can be seen as a variant of self-modifying code.

We have classified the payload into 2 types and now focus on the payload delivery methods.

3.3 Payload Delivery Method

A malicious code uses the payload delivery method to attach a payload to a victim process. To do so, the method must have access to the payload and the location of the delivery. We discuss 2 payload delivery methods that can deliver the arbitrary and self-modifying code to a victim process.

3.3.1 Payload Delivery Method I: ptrace() Call

Using the “*ptrace()*” system call the malicious code can get and set values in the registers, deliver a payload and resume the execution of the victim process. Furthermore, it enables creating and controlling of the child processes within the malicious code. In figure 3, the general prototype of the “*ptrace()*” system call is presented:

```
00 long ptrace(
01     enum ptrace_request req,
02     pid_t pid,
03     void *addr,
04     void *data
05     );
```

Figure 3: API prototype for “*ptrace()*” system call

Where, “*req*” determines the action to be performed, the code is injected to the process with process-id “*pid*”, “*addr*” is the read or write location of the code, and “*data*” is the value to be read or written at that location. EIP re-positioning can be done explicitly via this method.

3.3.2 Payload Delivery Mechanism II: DLR Objects

On 32-bit architecture, DLR objects are 32-bit ELF LSB relocatable object files. By using the DLR objects with the “*gdb*” debugger, the malicious code can determine victim process binary-specific symbols and use them within the payload. This enables the malicious code to alter the victim process execution behavior. Figure 4 depicts construction of DLR objects:

```
00 gcc -Wall -c CodeInjection.c -o CodeInjectio.o
```

Figure 4: “*gcc*” flag required to generate DLR object

“*gdb*” is used to load the DLR objects in the memory space of the victim process. When the victim process invokes the “*tainted*” symbol, the payload delivered via DLR object gets executed.

We now present the design of modified kernel-mode attester and its related modules as a technique for the mitigation of such attacks.

4. KERNEL ATTESTATION

In the previous sections, we have highlighted the payload types and methods that a malicious code can use to mount code injection attacks. We now present detail design of the proposed approach for mitigating such attacks.

4.1 Attack Assumptions and Design Principles

In order to consider scope and setting of the malicious code and the code injection attacks, we make the following assumptions:

- Code injection attacks are limited to user-space and kernel is not compromised with such attacks.
- Processor, memory controller and system memory is not controlled by malicious code.
- Malicious code may inherit root privileges for short time-bursts.

The proposed approach based on kernel attestation must obey the following design principles:

- Simple design based on the pluggable interface.
- Small source and in-memory code size.
- Reasonably “good” coverage for code attestation guarantees.

Our assumptions for malicious code act as “hidden” design principles. Given these assumptions of the malicious code and the design principles for the proposed approach, we now focus on the design of the kernel-mode attester.

4.2 Design of Kernel-Mode Attester

In figure 5, the kernel-mode attester contains 3 sub-systems, each having a specific operation to perform during the course of attestation. The description of these sub-systems is given below:

4.2.1 Sub-system I: Attester Kernel Module

The role of attester kernel module is to perform the attestation on a given process. This module is written as a kernel module that enables or disables the attester on the basis of insertion or deletion of the module into kernel. To perform the attestation on a given process, the attester generates an ensemble of random addresses as below:

$$\forall A, I; \quad \langle A_1, A_2, A_3 \dots A_I \rangle \quad \dots (3)$$

Where, A is the address, I is instance, and $\langle \cdot, \dots \cdot \rangle$ is the address ensemble. Using a fixed size ensemble of randomly generated addresses, the attester reads the contents of above addresses in the virtual memory area of the given process. The reads from the contents from above addresses are given as ensemble below:

$$\forall C, A, I; \quad \langle C_A^1, C_A^2, C_A^3 \dots C_A^I \rangle \quad \dots (4)$$

Where, C is content of address A having an instance I, and $\langle \cdot, \dots \cdot \rangle$ is the content ensemble. The attester provides the above content ensemble to a hash module to generate a snapshot. These snapshots are saved in a list. Formally, we define a snapshot and the snapshot list as below:

$$\forall H, S, C, A, I; \quad H(\langle C_A^1, C_A^2, C_A^3 \dots C_A^I \rangle) \Rightarrow S_I \quad \langle S_1, S_2, S_3 \dots S_I \rangle \quad \dots (5)$$

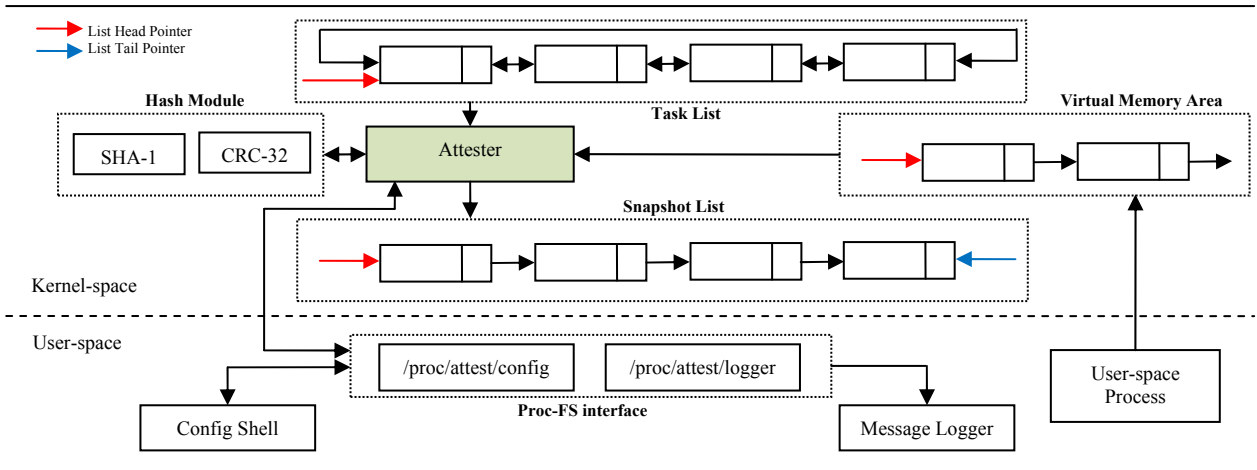


Figure 5: Kernel-Mode Attester interaction with different sub-systems and Linux Kernel

Where, H is a hash function, S is a snapshot, I is a snapshot instance, and $\langle \cdot, \cdot \dots \cdot \rangle$ is the snapshot list. The attester accesses the task list within the kernel to get the “*task_struct*” structure of the given process. It uses this structure to access the virtual memory area of the given process. The virtual memory area for any given process can be accessed using “*mm_struct*” structure within the “*task_struct*” structure. Each virtual memory area as defined by the “*mm_struct*” structure has a start and an end marker. Also, access to each virtual memory area is carefully controlled by a set of flags. These flags may designate a virtual memory area as “*read*”, “*write*”, “*execute*”, and “*shared*”. The virtual memory area of the text segment of a process has an “*exec*” flag.

If the malicious code needs to perform EIP-reposition after the code injection, then the target virtual memory area needs to have “*exec*” flag. Hence, by only attesting address ensembles within an “*exec*” virtual memory area, the attester can prevent EIP-reposition implication of the code injection attacks. The attester in the present design is fully configurable via a user-space configuration shell and reports activities through a message logger.

4.2.2 Sub-system II: Configuration Shell

A user can configure the kernel-mode attester via a user-space configuration shell. The configuration shell is an interactive console-mode shell such as “*bash*”-shell and provides 10 commands to interact, configure and manage the kernel mode attester. The commands and their description are listed below:

- **pid**: This command enables the user to specify the process-id of the process to be attested.
- **ssc**: This command provides a means to the user to specify the snapshot count. This count controls the length of snapshot list or the instance I in equation (5) within the kernel-mode attester.
- **spp**: This command provides the user a variable to control snapshot period. This period controls the frequency of snapshot per unit time by the kernel-mode attester.
- **hash**: This command enables the user to specify the choice of hash function.
- **clb**: This command provides the user a mechanism to clear that buffer.
- **get**: This command enables the user to read the current active configuration of the kernel-mode attester.

- **set**: This command provides a mechanism to the user to set the configuration stored in local buffer of the configuration shell into the kernel-mode attester.

- **list**: This command enables the user to list all the supported commands of the configuration shell.

- **swb**: This command provides a mechanism to the user to see the local buffer of the configuration shell.

- **quit**: This command enables the user to quit the configuration shell.

The configuration shell interacts with the kernel-mode attester via the “*/proc/attest/config*” read/write interface.

4.2.3 Sub-system III: Message Logger

The kernel-mode attester reports events that occur during the attestation operation via a user-space message logger. Each event reported by the kernel-mode attester during the attestation operation can be formally defined as below:

$$\forall T, M, I; \quad \langle T, M, I \rangle \quad \dots \quad (6)$$

Where, T is the timestamp, M is the message, and I is instance of the message. The logger retrieves the events through the read-only “*/proc/attest/config*” interface and shows the messages on the user-console.

4.3 Choice of Hash Function

The choice of hash function is an important design and implementation criterion. If the chosen hash function is not collision-resistant, then a malicious code can generate payloads that can be enable indistinguishable hash outputs.

Alternatively, if the size of each hash is large then it impacts the size of snapshot list. Also, the time required to perform hash affects the time required to perform the attestation operation. Moreover, the use of different hash functions for performing snapshot tradeoffs collision resistance and hash output size in attestation operation.

In the current design of the kernel-mode attester, we have chosen 2 hash functions, which enable us to tradeoff collision resistance, and output size of a given hash function. We use “*SHA-1*” (or Secure Hash Algorithm) and “*CRC-32*” (or Cyclic Redundancy Check) as the hash functions. Each of these hash functions have a specific property that acts as a guiding reason for their selection. We describe the properties of these hash functions as follows:

Hash Function	Choice	
	“SHA-1”	“CRC-32”
CR (Msgs/Collision)	2 ⁶³	NCR
Output Size (Bits)	160	32
Throughput (Gbps)	~2	~10

* CR – Collision Resistance, NCR – Not Collision Resistant

Table 1: Comparison of “SHA-1” and “CRC-32”

It is evident from the aforementioned table that “SHA-1” hash function provides greater collision resistance in exchange to output size and throughput, while “CRC-32” provides almost minimal collision resistance but retains space efficiency and throughput as its advantages. It should be noted here that the choice of providing only 2 hash functions reflects current scope of the implementation and does not restrict any future revisions.

We now focus on the evaluation of the proposed attestation technique.

5. EVALUATION

Evaluation of the proposed attestation technique enables us to measure its various aspects such as source and in-memory code size, stress-testing of the applications against code inject attacks, and address coverage required for “good” attestation guarantees.

5.1 System Configuration

We have performed the experiments to evaluate the proposed attestation technique on 1 computer having the following configuration:

Hardware		Software	
CPU ^A	1500.033 MHz	Compiler	GCC 4.3.2
Hard Disk	120 GB		
RAM	1 GB		
V-RAM	64 MB	Debugger	GDB 6.8-29
Cache	256 KB		
Flags	TSC		
Serial Port	2	OS ^L	Kernel 2.6.27.21- 170
Parallel Port	1		
Floppy	3.5 In, 1.44 MB		

* ^A = AMD Athlon XP 2400+, ^L = Fedora Core 10

Table 2: System configuration of experimental setup

While generating code for configuration shell and message logger, we have used “gcc” with “-o” flag (with other flags turned off). For generating code for synthetic code injection module, we have used “gcc” with flags as depicted in figure 4. For generating code kernel-mode attester, we have used standard module build procedure.

5.2 Methodology

We have adopted a consistent set of rules and methods across all the experiments to reduce errors based on the methodology. For measuring time, we have used “rdtsc()” instruction with less than 1µs of resolution and 0.0413 µs of overhead. Also, to reduce the time-measurement complexity related to “rdtsc()” instruction, we have used 1-core machine. We normalize intra-experiment variations by taking average of 1000 iterations across all experiments. Figure 6 depicts our execution time measurement approach:

```

00 for (i = 0; i < ITER_COUNT; i++)
01 {
02 // start timer
03 Begin = Timer_Begin();
04
05 // operation to be measured
06 Operation();
07
08 // end timer
09 End = Timer_End();
10
11 // Aggregate elapsed time
12 TimerBuffer[i] = End - Begin;
13 }

```

Figure 6: Approach for execution-time measurement

We use “TimerBuffer” to compute average of 1000 iterations after the measurement is complete. This method is repeated across all the experiments.

5.3 Experiments and Results

In order to determine the efficacy of the proposed attestation technique, we have measured it across 5 experiments that are chosen carefully to highlight key strengths and weaknesses of the proposed technique. The experiments are described below:

5.3.1 Experiment I: Source and In-memory Code Size

The goal of this experiment is to adhere to one of the design principles of having small source and in-memory code size.

Code Size	Sub-system		
	Kernel Mode Attester	Configuration Shell	Message Logger
Source Code (LOC)	623	521	62
In-memory (Bytes)	14,160	10,458	5,703

* LOC = Lines of Code

Table 3: Source code and In-memory code size

It is evident from the above table that kernel-mode attester, configuration shell, and message logger are in-memory space efficient (~30 Kbytes) and consists an aggregate of (~1200) lines of code. We have taken care to exclude any architecture-specific routines or instructions except “*rdtsc()*” to enable ease-of-porting to different versions of Linux like operating systems.

5.3.2 Experiment II: Synthetic Code Injection

In this experiment, we measure the rate of code injection by synthetically injecting code into 5 text-mode applications. This experiment is required to understand the impact of code injection rate on application breakdown. We injected code with payload size up to 64 Kbytes and monitored the target application execution progress. MC and NC-FTP crashed at 64 Kbytes payload size due segmentation fault. Figure 7 shows the rate of code injection across 5 applications.

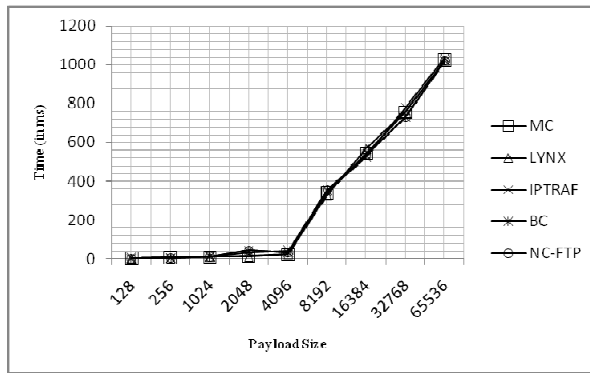


Figure 7: Code injection rate for 5 applications

It is evident from the above illustration that code injection rate remains independent of the application type. Also, certain applications such as MC and NC-FTP are susceptible to 64 Kbytes payload size. Furthermore by injecting code at the above mentioned rate, we can test if an application can withstand such attacks. This enables us to stress test the applications against the size of attack payloads.

6. CONCLUSION

Malicious code attacks such as code injection exploit new and unknown vulnerabilities to reduce the overall availability of the applications and affect user productivity. To mitigate such attacks, we have proposed a modified kernel-mode attestation technique to detect mutations in the application runtime code. Our proposed kernel-mode attester verifies the code at runtime and can be implemented economically with low in-memory space (~30 Kbytes).

REFERENCES

- [1] Arvind Seshadri, Adrian Perrig, Leendert van Doorn and Pradeep Khosla. *SWATT: Software-based attestation for embedded devices*. In Proceedings of the IEEE Symposium on Security and Privacy, 2004
- [2] Gaurav S. Kc, Angelos D. Keromytis and Vassilis Prevelakis. *Countering Code-Injection Attacks with Instruction-Set Randomization*. In Proceedings of the ACM Computer and Communication Security, 2003
- [3] D. S. Peterson, M. Bishop, and R. Pandey. *A Flexible Containment Mechanism for Executing Untrusted Code*. In Proceedings of the 11th USENIX Security Symposium, 2002
- [4] H. Chen and D. Wagner. *MOPS: an Infrastructure for Examining Security Properties of Software*. In Proceedings of the ACM Computer and Communications Security Conference, 2002
- [5] G. C. Necula, S. McPeak, and W. Weimer. *CCured: Type-Safe Retrofitting of Legacy Code*. In Proceedings of the Principles of Programming Languages, 2002
- [6] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. *Bouncer: Securing Software by Blocking Bad Input*. In Proceedings of the Symposium of the Operating Systems Principles, 2007
- [7] Leonard Adleman. *An Abstract Theory of Computer Viruses*. In Proceedings of the Advances in Cryptology - CRYPTO, 1988
- [8] Yves Younan, Wouter Joosen, and Frank Piessens. *Code injection in C and C++ : A survey of vulnerabilities and countermeasures*. Technical Report, Department of Computer Science, Katholieke Universiteit Leuven, 2004
- [9] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. *Nozzle: A Defense Against Heap-spraying Code Injection Attacks*. Microsoft Research Technical Report, MSR-TR-2008-176, 2008
- [10] Xuxian Jiang, Helen J. Wang, Dongyan Xu, and Yi-Min Wang. *RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization*. CERIAS Technical Report, CERIAS Tech Report 2007-31, Purdue University, 2007
- [11] Skape, and Jarkko Turkulainen. *Remote Library Injection*. www.nologin.org, 2004
- [12] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. *Secure and Practical Defense Against Code-injection Attacks using Software Dynamic Translation*. In Proceedings of the 2nd International Conference on Virtual Execution Environments, 2006
- [13] Bryan Burns, Jennifer Stisa Granick, Steve Manzuik, Paul Guersch, Dave Killion, Nicolas Beauchesne, Eric Moret, Julien Sobrier, Michael Lynn, Eric Markham, Chris Iezzoni, and Philippe Biondi. *Security Power Tools*. O'Reilly Media, 2007
- [14] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The Linux Kernel Module Programming Guide*. www.tldp.org, 2007

- [15] Shaun Clowes. *InjectSO - Inject shared libraries into running processes under Solaris and Linux*. www.secureality.com.au, 2000
- [16] SecurityTube. *Bytecode Injection into a Running Process using Ptrace()*. [www.securitytube.net/Bytecode-Injection-into-a-Running-Process-using-Ptrace\(\)-video.aspx](http://www.securitytube.net/Bytecode-Injection-into-a-Running-Process-using-Ptrace()-video.aspx)
- [17] Pradeep Padala. *Playing with ptrace, Part I and II*. www.linuxjournal.com/article/6100
- [18] Bruce Schneier. *Interview with an Adware Developer*. www.schneier.com/blog/archives/2009/01/interview_with_10.html
- [19] M. Tim Jones. *Anatomy of Linux Loadable Kernel Modules*. www.ibm.com/developerworks/linux/library/l-lkm/index.html