Chapter XII
Memory Corruption Attacks, Defenses, and Evasions

Carlo Belletini
Università degli Studi di Milano, Italy

Julian L. Rrushi
Università degli Studi di Milano, Italy

ABSTRACT

The chapter introduces and describes representative defense mechanisms to protect from both basic and advanced exploitation of low-level coding vulnerabilities. Exploitation of low-level coding vulnerabilities has evolved from a basic stack-based buffer overflow with code injection to highly sophisticated attack techniques. In addition, pure-data attacks were demonstrated to be as efficient as control-data attacks and quite realistic. On the other hand research on assessment of the robustness of proposed mitigation techniques revealed various weaknesses in them leading to design and implementation of evasion techniques. Most of the defensive techniques protect only from a limited set of attack techniques, thus a defense employment requires multiple complementary mitigation techniques. Furthermore, there are few mitigation techniques designed to counter pure-data attacks. In response to these limitations, current research proposes better defensive mechanisms such as pointer taintedness detection and attack data burning capable of countering any kind of control-data or pure-data attack.

INTRODUCTION

Low-level coding vulnerabilities have been widespread in modern operating systems. These vulnerabilities account for the majority of CERT advisories released in recent years. Attack techniques which exploit low-level coding vulnerabilities evolved from a basic stack-based buffer overflow to actual highly sophisticated forms. These attack techniques have demonstrated to be quite devastating and fully applicable to real world target programs, and are considered to be among the main attack techniques employable to get intruded into target operating systems or take destructive actions against them. Besides being used directly by attackers or being embedded in automatic attack tools, attack code which exploits low-level coding vulnerabilities performs as well when incorporated in autonomous attack agents. It was the case of the historical Morris worm, Code Red, Code Red II, or Blaster for instance. The efficiency of such attack techniques is derived directly from a high control on sensitive data in the memory of a running program acquirable from the nature of vulnerabilities they exploit.

The overall objective of this chapter is to provide thorough defense intelligence to protect computer systems from exploitation of low-level coding vulnerabilities. This chapter covers fundamental defensive approaches result of significant defensive research on
the field. For each one of these defensive approaches this chapter describes the defense coverage, limitations, particular requirements, computational cost and applicability to real-world computer systems. In addition, this chapter gives an overview of both basic and advanced attack techniques derived from significant research on offensive attack capabilities carried out mainly by hackers underground, and provided in the form of nontraditional publications in highly technical hacker journals and hacker mailing lists. This chapter also provides information about evasion techniques built upon research on assessments of the robustness of operational defensive approaches.

WHAT ARE LOW-LEVEL CODING VULNERABILITIES AND HOW ARE THEY EXPLOITED?

These vulnerabilities may be defined as programming errors which open the way to an attacker to corrupt the memory of a program. Exploitation of such vulnerabilities generally takes the form of control-data or pure-data attacks. Control-data attacks corrupt memory management data for the purpose of transferring control to binary code inserted into the address space of a target process, or to existing arbitrary instructions which usually are forced to take attacker supplied data as arguments. Pure-data attacks (Chen, Xu, Sezer, Gauriar, & Iyer, 2005a; Pincus, & Baker, 2004) are built upon corruption of noncontrol data (i.e., computational data usually held by global or local variables in a program, such as, for example, user identification data, configuration data, decision making data, file descriptors, RPC procedure numbers, and so on).

Array Overflows

An array overflow is a programming error which occurs when no range checks are performed on a value which is used to index an array. The danger rises when such a value may be directly or indirectly affected by an attacker, and the array is filled with user-supplied data.

Buffer Overflows

A buffer overflow vulnerability is a programming error which allows data to be stored beyond the boundaries of a destination buffer, therefore overwriting adjacent memory locations and possibly further away. Buffer overflows may be caused by instructions which do not perform any bounds checking on a destination buffer when storing data into it. Some functions such as `strncpy()` allow a programmer to explicitly specify the number of bytes to copy to a destination buffer, but do not null-terminate the destination buffer. These apparently safe functions may lead to the creation of adjacent not null-terminated buffers. Such a situation in conjunction with a vulnerable function may cause an excessive amount of data to be copied to a destination buffer, thus overflowing it. In fact, the intention to copy one of these buffers to a destination buffer may copy the intended buffer along with one or more adjacent buffers causing an overflow of the destination buffer. A stack-based buffer overflow attack in one of its very first forms consists in injecting binary code and overwriting the saved instruction pointer stored on stack with the address of the injected code (Aleph1, 1996).

If executable memory areas where an attacker could inject binary code is not available or the available buffers are too small to hold the entire injected binary code, the attacker may overwrite the saved instruction pointer on stack with the address of existing instructions. The attacker may specify possible arguments by injecting them on stack along with the corrupting address. A common approach is to overwrite the saved instruction pointer with the address of the `system()` function of the libc dynamic library along with injecting on stack the address of the string that represents a command the attacker aims at executing on a target system. This attack technique is referred to as return-into-library (Nergal, 2001; Solar Designer, 1997). As a result of errors in handling the index of arrays in looping and iteration, a destination buffer may be overflowed by just a few bytes, more commonly by one byte or by five bytes.

Although such a buffer overflow is limited, it may be sufficient for an attacker to reach and corrupt the least significant byte of the saved frame pointer in a Little Endian architecture, and consequently dictate the address where the operating system pops a saved instruction pointer (Klog, 1999). Heap overflow attacks (Anonymous, 2001; Conover, 1999; Kaempf, 2001) are built upon the fact that most of the memory allocator algorithms such as System V in Solaris, Doug Lea’s Malloc used by the GNU C Library, RtlHeap in Windows, and so on, store heap management data in band on the heap itself. By overflowing a buffer on