Using Greedy Hamiltonian Call Paths to Detect Stack Smashing Attacks

Mark Foster, Joseph N. Wilson, and Shigang Chen

Department of Computer and Information Sciences and Engineering
University of Florida
Gainesville, Florida 32611
mfoster@cise.ufl.edu

Abstract. The ICAT statistics over the past few years have shown at least one out of every five CVE and CVE candidate vulnerabilities have been due to buffer overflows. This constitutes a significant portion of today's computer related security concerns. In this paper we introduce a novel method for detecting stack smashing and buffer overflow attacks. Our runtime method extracts return addresses from the program call stack and uses these return addresses to extract their corresponding invoked addresses from memory. We demonstrate how these return and invoked addresses can be represented as a directed weighted graph and used in the detection of stack smashing attacks. We introduce the concept of a Greedy Hamiltonian Call Path and show how the lack of such a path can be used to detect stack-smashing attacks.

1 Introduction

The term buffer overflow refers to copying more data into a buffer than the buffer was designed to hold. A buffer overflow attack takes places when a malicious individual purposely overflows a buffer to alter a program's intended behavior. The most common form of this attack deals with the attacker intentionally overflowing a buffer on the stack so that the excess data overwrites the return address that resides just below the buffer on the stack. Thus when the current function returns, control flow is transferred to an address chosen by the attacker. Commonly, this address is a location on the stack where the attacker has injected his/her own malicious code inside the same buffer. This type of buffer overflow attack is also referred to as a stack smashing attack since the buffer resides on the stack. Stack smashing attacks are one of the most common forms of buffer overflow attacks due to their simplicity of implementation.

Buffer overflow attacks have been a major security issue for a number of years. Wagner et al. [1] extracted statistics from CERT advisories showing that between 1988 and 1999 buffer overflows accounted for up to 50% of the vulnerabilities reported by CERT. Wagner cites several other statistics showing where buffer overflows were at a minimum of 23% of the vulnerabilities in different databases. A more recent look at the ICAT statistics shows that a significant number of the CVE and CVE candidate vulnerabilities were due to buffer overflows. For the years 2001, 2002 and 2003 buffer overflows accounted for 21%, 22%, and 23% of the vulnerabilities respectively [2]. In addition, SecurityTracker.com released statistics for the

K. Zhang and Y. Zheng (Eds.): ISC 2004, LNCS 3225, pp. 183–194, 2004.

time period between April 2001 and March 2002. These statistics show that buffer overflows were the cause behind 20% of the vulnerabilities reported by Security-Tracker [3]. These more recent statistics reinforce the case made by Wagner et al. Buffer overflows are a significant issue for system security.

The purpose of this paper is to introduce a new method for detecting stack smashing and buffer overflow attacks. While much work has been focused on detecting stack-smashing attacks, few approaches use the program call stack to detect such attacks. We propose a new method of detecting stack smashing attacks that relies solely on intercepting system calls and information that can be extracted from the program call stack and process image. Upon intercepting a system call, our method traces the program call stack to extract return addresses. These return addresses are used to extract what we refer to as *invoked addresses*. In the process image, return addresses are preceded by *call* instructions. These *call* instructions are what placed the return addresses on the stack and then transferred control flow to another location. An address that was invoked by a *call* instruction is referred to as an *invoked address*. We use the return and invoked addresses to create a weighted directed graph. We have found that the graph constructed from an uncompromised process always contains a Greedy Hamiltonian Call Path (GHCP). This allows us to use the lack of a GHCP to indicate the presence of a buffer overflow or stack smashing attack.

The rest of this paper is organized in the following manner. In Section 2 we discuss related work. In Section 3 we introduce our new method and prove its correctness using induction. Sections 4 and 5 discuss the limitations and benefits of our proposed method. Our conclusions from this study are stated in Section 6.

2 Related Work

One of the most notable approaches to detecting and preventing buffer overflow attacks is referred to as StackGuard [4]. Cowan et al. created a compiler technique that involves placing a *canary* word on the stack next to the return address. When a function returns, if the canary word has been modified, it implies that the return address has also been modified. The only downfall of Stackguard is that programs are only protected if they have been recompiled with a specially enhanced compiler.

Baratloo et al. proposed two new methods referred to as Libsafe and Libverify [5]. Libsafe uses the saved frame pointers on the stack to act as upper bounds when writing to a buffer. Libverify uses a similar approach to Stackguard in that a return address is verified before a function is allowed to return. Libverify does this by copying each function into the heap and overwriting the original beginning and end of each function with a call to a wrapper function. One downfall of this method is that the amount of space in memory required for each function is double that of what the process would require if not using Libverify.

One approach proposed by Feng et al. is VtPath [6]. VtPath is designed to detect anomalous behavior but would also work well in detecting buffer overflow attacks. VtPath is unique in that it uses information from a program's call stack to perform anomaly detection. At each system call VtPath takes a snapshot of the return addresses on the stack. The sequence of return addresses found between two system calls creates what is referred to as a virtual path. A training phase is used to learn a