An Enhanced Dynamic Information Flow Tracking Method with Reverse Stack Execution

Anna Trikalinou, Intel Corporation, Portland, OR, USA
Nikolaos Bourbakis, Center of Assistive Research Technologies (CART), Wright State University, Dayton, OH, USA

ABSTRACT

Memory errors have long been a critical security issue primarily for C/C++ programming languages and are still considered one of the top three most dangerous software errors according to the MITRE ranking. In this paper the authors focus on their exploitation via control-flow hijacking and data-only attacks (stack, and partially heap (G. Novarck & E. Berger, 2010)) by proposing a synergistic security methodology, which can accurately detect and thwart them. Their methodology is based on the Dynamic Information Flow Tracking (DIFT) technique and improves its data-only attack detection by utilizing features from the Reverse Stack Execution (RSE) security technique. Thus, the authors can significantly lower the resource consumption of the latter methodology, while increasing the former’s accuracy. Their proof-of-concept compiler implementation verifies their assumptions and is able to protect vulnerable C programs against various real-world attack scenarios.

Keywords: Application Security, Buffer Overflow, Control-Flow Hijacking Attack, Data-Only Attack, Dynamic Information Flow Tainting (DIFT), Memory Error

1. INTRODUCTION

Although memory errors, primarily in C and C++ programming languages, have been among the oldest classes of software vulnerabilities and a research topic for more than 40 years, such flaws still impose a major threat to our systems. Examples of these attacks date from the infamous Morris worm in 1988, which performed a stack-based buffer overflow, to the recent zero-day exploit in Internet Explorer, which exploited a double free vulnerability and caused the German government to urge the public to temporarily stop using this browser.

Generally, by memory errors we refer to exploitable programming bugs, such as a buffer overflow/underflow, an unchecked allocation failure or an indexing bug, which allow an attacker to alter a program’s behavior, take full control over it or expose critical information. In the article DOI: 10.4018/IJMSTR.2015010103
by L. Szekeres, M. Payer, et al. (2012), the authors have published an extensive study on how a programming bug can facilitate an attack to the vulnerable application and, depending on the process an attacker follows, have classified the resulting attacks into 4 overlapping categories: control-flow hijacking attacks, data-only attacks, code corruption attacks and information leaks.

More specifically, in a control-flow hijacking attack, an attacker exploits a memory error to corrupt the program’s control data in order to alter its control-flow graph. Control data is defined as the sensitive information (i.e. return address, function pointer, etc.) that directly influence the control-flow of a program by being used along with a return or branch instruction to transfer execution into another place in memory. By maliciously modifying this information, an attacker can force the target computer system to execute the attacker’s code (code injection) or execute a special sequence of gadgets (Return Oriented Programming) chained together to perform a malicious operation (S. Checkoway, L. Davi, et al., 2010). An example of the control flow during the attack is illustrated in Figure 1.

In this case, after the beginning of the execution of the callee() function, the attacker exploits i.e. a buffer overflow and is able to corrupt the return address of the current activation record. Thus, when the callee() function ends, instead of returning execution to the caller() function stored in the original return address, the execution is transferred to the attacker’s code.

In the data-only attack, however, the attacker targets other parts of the memory, which does not alter the control-flow graph of the program, but it forces it to follow certain “incorrect” paths. Thus, the intruder can successfully modify the program’s logic, gain more privileges or leak sensitive information. Consider, for instance, the example shown in Figure 3 as code and in Figure 2 as a control-flow graph, is similar to a real-world zero-day attack. Assuming that an attacker logs into the system with no administration privileges, he corrupts the isAdmin variable via a buffer overflow. Then, although the control data do_privileged_operation address is unaffected by the attacker, he is able to indirectly influence the execution path and perform unauthorized operations.

Finally, in a code corruption attack the intruder overwrites the vulnerable application’s code with his own, while in the information leakage memory contents, which would otherwise be excluded from the output, become exposed. However, in this work we will mainly focus on the former two attacks, since the steps required in order for the attacker to leak information greatly resemble the ones required for the aforementioned two types of attacks. Regarding code corrup-

![Figure 1. Example of a control flow hijacking attack. The intruder by corrupting the return address is able to transfer execution to his malicious code](image-url)
Related Content

Medical Image Mining Using Fuzzy Connectedness Image Segmentation: Efficient Retrieval of Patients’ Stored Images
www.igi-global.com/chapter/medical-image-mining-using-fuzzy-connectedness-image-segmentation/164606?camid=4v1a

Extended Automata for Temporal Planning of Interacting Agents
www.igi-global.com/article/extended-automata-for-temporal-planning-of-interacting-agents/182505?camid=4v1a

WBAN Based Long Term ECG Monitoring
www.igi-global.com/article/wban-based-long-term-ecg-monitoring/97699?camid=4v1a
Curve Fitting Methods: A Survey

www.igi-global.com/article/curve-fitting-methods/180665?camid=4v1a