Summary

Even though memory corruption vulnerabilities are inherent to C, this language is not likely to be replaced by strongly typed languages with memory safety guarantees [76; 200; 30]. However, the lack of safety causes serious security problems. Memory corruption vulnerabilities are reported daily [180; 150; 137], and we regularly witness attacks compromising popular software or critical networks [123; 198].

The research community has long recognised the problem, and has proposed multiple solutions. However, the existing proposals that are practical for production use prove inefficient, while the more comprehensive ones are either inapplicable to legacy software, or incur a high performance overhead.

In this thesis, we address the problem of protecting legacy C binaries against memory corruption attacks. We focus on techniques employing data flow tracking, since they are applicable to existing software, and at the same time offer a mechanism to monitor and accurately reason about a program execution. Because such monitoring is often prohibitively expensive, current systems employing information flow tracking are mainly limited to non production machines, such as malware analysis engines or honeypots. In our work, we seek solutions that would let us benefit from the wealth of information available during a run of the program, but at the same time be efficient and applicable in a timely fashion.

We divide memory corruption attacks into two classes: (1) control-diverting, that divert the flow of execution of a program to code injected or chosen by an attacker, and (2) non-control-diverting, that do not directly divert a program's control flow, but might modify a value in memory that represents e.g., a user's privilege level, or a server configuration string.

The research community has widely applied information flow tracking to protect against both types of memory corruptions. A popular branch of the technique, known as Dynamic Taint Analysis [62; 149], has been successfully employed to detect control-diverting attacks. As the consequences of non-control-diverting attacks can be as serious as the control-diverting one's, multiple projects have tried to apply an incarnation of DTA, known as limited pointer tainting, to detect both. An extended version of this technique, known as full pointer tainting, has been also employed to track the propagation of keystrokes, and detect keyloggers.

Part I The first part of this dissertation is dedicated to dynamic taint analysis. First, we evaluate the ability of both branches of pointer tainting to detect non-control-diverting attacks and keyloggers. Next, we further extend the basic dynamic taint analysis to perform attack analysis, when a control-diverting attack is detected.

In Chapter 3, we carry out an in-depth analysis of the problems of pointer tainting on real systems, which shows that the method does not work against malware spying on users' behaviour, and is problematic in other forms also. It is an open challenge to use an incarnation of pointer tainting to detect non-control-diverting attacks on the most popular PC architecture (`x86`) and the most popular OS (Windows).

In Chapter 4, we develop Prospector, an emulator capable of tracking which bytes contributed to a buffer overflow attack on the heap or stack. Whenever we recognise the protocol governing the malicious network message, we use the information discovered by Prospector to generate signatures for polymorphic attacks. We look at the length of protocol fields, rather than the actual contents. In practice, the number of false positives is negligible, and the number of false negatives is also low. At the same time, the signatures allow for efficient filtering.

Further, in Chapter 5, we propose Hassle, a honeypot that is capable of generating signatures for attacks over both encrypted and non-encrypted channels. Since the techniques we describe work by interposing encryption routines, they are applicable to most types of encryption, and require no modification of the applications that need protection.

Part II Since the methods discussed above cannot handle non-control-diverting attacks, in the second part of the thesis we go beyond dynamic taint analysis. We introduce BodyArmour, a completely new method of protecting legacy binaries against buffer overflow attacks, also the non-control-diverting ones. In a nutshell, we monitor the execution of a protected binary to make sure that once a pointer is assigned to a certain buffer, it never accesses memory beyond the buffer's boundaries. This basic rule precludes both control-diverting and non-control-diverting attacks. Even though the requirement sounds straightforward, it assumes lots of knowledge about the binary being protected, e.g., about buffer locations together with pointers associated with them, or instructions accessing these buffers. Since we aim at protecting legacy binaries for which we do not have symbol tables, we first investigate how to employ information flow tracking to gather the necessary information.

In Chapter 6, we introduce Howard, a framework which extracts data structures from a stripped binary. The analysis is performed by dynamically observing the program's memory access patterns. The main goal of Howard is to provide data structures that allow BodyArmour to retrofit security onto existing binaries. We demonstrate two other applications of data gathered by Howard: (1) we use Howard to furnish existing disassemblers and debuggers with information about data structures and types to ease reverse engineering, (2) we show how Howard can be used to further reverse engineer a binary, and recover high-level pointer structures, e.g., singly- and doubly-linked lists or trees.

Chapter 7 presents BodyArmour, an approach to harden a legacy binary without access to source code or even original symbol tables. Using our approach, we can protect binaries against buffer overflows pro-actively, before we know they are vulnerable. Besides attacks that divert the control flow of a program, we also detect and stop attacks against non-control data. BodyArmour can operate in two modes: the BA-fields mode and the BA-objects mode. In the BA-fields mode, by protecting individual fields inside a structure rather than aggregates, BodyArmour is significantly finer-grained than other solutions. Even though we cannot guarantee no false positives in the BA-fields mode, they are very unlikely, and we never encountered them. No false positives are possible when the protection is limited to structures (the BA-objects mode). Our solution stops a variety of exploits, and works in a timely fashion.