Writing the Introduction

Fernando Magno Quintão Pereira

The Four Parts

- Context

Modern operating systems use a protection mechanism called Address Space Layout Randomization (ASLR) [Bhatkar03,Shacham04]. This technique consists in loading the binary modules that form an executable program at different addresses each time the program is executed. This security measure protects the software from well-known attacks, such as return-to-libc [Shacham04] and return-oriented-programming (ROP) [Buchanan08,Shacham07]. Because it is effective, and easy to implement, ASLR is present in virtually every contemporary operating system. However, this kind of protection is not foolproof.

- Problem

Shacham et al. [Shacham04] have shown that address obfuscation methods are susceptible to brute force attacks; nevertheless, address obfuscation slows down the propagation rate of worms that rely on buffer overflow vulnerabilities substantially. However, an adversary can still perform a surgical attack on an ASLR protected program. In the words of the original designers of the technique[Bhatkar03(p.115)], if "the program has a bug which allows an attacker to read the memory contents", then "the attacker can craft an attack that succeeds deterministically". It is this very type of bug that we try to prevent in this paper.

- Solution

In this paper we focus on dynamic techniques to prevent address leaks. We have developed an instrumentation framework that automatically converts a program into a software that cannot contain address leaks. This transformation, which we introduce in Section 3, consists in instrumenting every program operation that propagates data. In this way, we know which information might contain address knowledge, and which information might not. If harmful information reaches an output point that an adversary can read, the program stops execution. Thus, by running the instrumented, instead of the original software, the user makes it much harder for an adversary to perform attacks that require address information to succeed.

- Results

We have implemented our instrumentation framework in the LLVM compiler [Lattner04]. As we show in Section 4, we have used this implementation to analyze a test suite that contains almost 2 million lines of code, and that includes SPEC CPU 2006. Instrumented programs can be 2% to 1,070% slower than the original programs, with an average slowdown of 76.8%. Optimizations that we have designed contribute substantially to decrease this overhead. By only instrumenting the operations that we have not been able to prove safe, we get slowdowns ranging from 0% to 22%, with an average of 1.8%. Our tool has reported 19 warnings in our test suite. Manual inspection of these warnings shows that 11 of them can be exploited. The low overhead and effective bug discovery lets us justify the use of our dynamic monitor in production code.

Context

Modern operating systems use a protection mechanism called Address Space Layout Randomization (ASLR) [Bhatkar03,Shacham04]. This technique consists in loading the binary modules that form an executable program at different addresses each time the program is executed. This security measure protects the software from well-known attacks, such as return-to-libc [Shacham04] and return-oriented-programming (ROP) [Buchanan08,Shacham07]. Because it is effective, and easy to implement, ASLR is present in virtually every contemporary operating system. However, this kind of protection is not foolproof.

- The field of research
- The key related work
- The current state of the art
- The importance of the problem

Problem

Shacham et al. [Shacham04] have shown that address obfuscation methods are susceptible to brute force attacks; nevertheless, address obfuscation slows down the propagation rate of worms that rely on buffer overflow vulnerabilities substantially. However, an adversary can still perform a surgical attack on an ASLR protected program. In the words of the original designers of the technique[Bhatkar03(p.115)], if "the program has a bug which allows an attacker to read the memory contents", then "the attacker can craft an attack that succeeds deterministically". It is this very type of bug that we try to prevent in this paper.

- The difficulty to solve the problem
- The failure of previous work

Solution

In this paper we focus on dynamic techniques to prevent address leaks. We have developed an instrumentation framework that automatically converts a program into a software that cannot contain address leaks. This transformation, which we introduce in Section 3, consists in instrumenting every program operation that propagates data. In this way, we know which information might contain address knowledge, and which information might not. If harmful information reaches an output point that an adversary can read, the program stops execution. Thus, by running the instrumented, instead of the original software, the user makes it much harder for an adversary to perform attacks that require address information to succeed.

- The goal of the paper
- The key insight
- Summary of the contributions

Results

We have implemented our instrumentation framework in the LLVM compiler [Lattner04]. As we show in Section 4, we have used this implementation to analyze a test suite that contains almost 2 million lines of code, and that includes SPEC CPU 2006. Instrumented programs can be 2% to 1,070% slower than the original programs, with an average slowdown of 76.8%. Optimizations that we have designed contribute substantially to decrease this overhead. By only instrumenting the operations that we have not been able to prove safe, we get slowdowns ranging from 0% to 22%, with an average of 1.8%. Our tool has reported 19 warnings in our test suite. Manual inspection of these warnings shows that 11 of them can be exploited. The low overhead and effective bug discovery lets us justify the use of our dynamic monitor in production code.

- Numbers!
- Key conclusions
- Implementation
- We did a lot of work!