Using Type Qualifiers to Analyze Untrusted Integers and Detecting Security Flaws in C Programs

Ebrima N. Ceesay, Jingmin Zhou, Michael Gertz, Karl Levitt, and Matt Bishop

Computer Security Laboratory
University of California at Davis
Davis, CA 95616, USA
{ceesay, zhouji, gertz, levitt, bishop}@cs.ucdavis.edu

Abstract. Incomplete or improper input validation is one of the major sources of security bugs in programs. While traditional approaches often focus on detecting string related buffer overflow vulnerabilities, we present an approach to automatically detect potential integer misuse, such as integer overflows in C programs. Our tool is based on CQual, a static analysis tool using type theory. Our techniques have been implemented and tested on several widely used open source applications. Using the tool, we found known and unknown integer related vulnerabilities in these applications.

1 Introduction

Most known security vulnerabilities are caused by incomplete or improper input validation instead of program logic errors. The ICAT vulnerability statistics [1] show for the past three years that more than 50% of known vulnerabilities in the CVE database are caused by input validation errors. This percentage is still increasing. Thus, improved means to detect input validation errors in programs is crucial for improving software security.

Traditionally, manual code inspection and runtime verification are the major approaches to check program input. However, these approaches can be very expensive and have proven ineffective. Recently, there has been increasing interest in static program analysis techniques and using them to improve software security. In this paper, we introduce a type qualifier based approach to perform analysis of user input integers and to detect potential integer misuse in C programs. Our tool is based on CQual [2], an extensible type qualifier framework for the C programming language.

An integer is mathematically defined as a real whole number that may be positive, negative, or equal to zero [3]. We need to qualify this definition to include the fact that integers are often represented by integer variables in programs. Integer variables are the same as any other variables in that they are just regions of memory set aside to store a specific type of data as interpreted by the programmer [4]. Regardless of the data type intended by the programmer, the computer interprets the data as a sequence of bits. Integer variables on various systems may have different sizes in terms of allocated bits. Without loss of generality, we assume that an integer variable is stored in a 32-bit memory location, where the first bit is used as a sign flag for the integer value.

Integer variables are widely used in programs as counters, pointer offsets and indexes to arrays in order to access memory. If the value of an integer variable comes

R. Büschkes and P. Laskov (Eds.): DIMVA 2006, LNCS 4064, pp. 1–16, 2006.

from untrusted source such as user input, it often results in security vulnerabilities. For example, recently an increasing number of integer related vulnerabilities have been discovered and exploited [5, 6, 7, 8, 9]. They are all caused by the misuse of integers input by a user. The concept of integer misuse like integer overflow has become common knowledge. Several researchers have studied the problem and proposed solutions like compiler extension, manual auditing and safe C++ integer classes [4, 10, 11, 12, 13, 14]. However, to date there is no tool that statically detects and prevents integer misuse vulnerabilities in C programs.

Inspired by the classical Biba Integrity Model [15] and Shankar and Johnson's tools [3, 16] to detect format string and user/kernel pointer bugs, we have implemented a tool to detect potential misuse of user input integers in C programs. The idea is simple: we categorize integer variables into two types: *trusted* and *untrusted*. If an *untrusted* integer variable is used to access memory, an alarm is reported. Our tool is built on top of CQual, an open source static analyzer based on the theory of type qualifiers [2]. Our experiments show that the tool can detect potential misuse of integers in C programs.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to CQual and the theory behind it. Section 3 describes the main idea of our approach and the development of our tool based on CQual. Section 4 shows the experiments we have performed and the results. In Section 5 we discuss several issues related to our approach. Section 6 discusses related work. Finally, Section 7 concludes this paper with future work.

2 CQual and Type Qualifiers

We developed our tool as an enhancement to CQual. It provides a type-based static analysis tool for specifying and checking properties of C programs.

The idea of type qualifiers is well-known to C programmers. Type qualifiers add additional constraints besides standard types to the variables in the program. For example, in ANSI C, there is a type qualifier *const* that attaches the unalteration property to C variables. However, qualifiers like *const* are built-in language features of C, which seriously restrict the scope of their potential applications. CQual allows a user to introduce new type qualifiers. These new type qualifiers specify the customized properties in which the user is interested. The user then annotates a program with new type qualifiers, and lets CQual statically check it and decide whether such properties hold throughout the program. The new type qualifiers introduced in the program are not a part of the C language, and C compilers can ignore them.

There are two key ideas in CQual: *subtyping* and *type inference*.

Subtyping is familiar to programmers who practice object-oriented programming. For example, in GUI programming, a class DialogWindow is a subclass of class Window. Then we say DialogWindow is a subtype of Window (written as DialogWindow $\leq$ Window). This means that an object of DialogWindow can appear wherever an object of Window is expected, but not vise versa. Thus, if an object of type Window is provided to a program where a DialogWindow is expected, it is a potential vulnerability and the program does not type check.

CQual requires the user to define the subtyping relation of user supplied type qualifiers. The definition appears as a lattice in CQual's `lattice` configuration file. For