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.
© Springer-Verlag Berlin Heidelberg 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

example, if we define the lattice for type qualifiers Q_1 and Q_2 as: $Q_1 \leq Q_2$, it means for any type τ, $Q_1\tau$ and $Q_2\tau$ are two new *qualified types*, and $Q_1\tau$ is a subtype of $Q_2\tau$ (written as $Q_1\tau \leq Q_2\tau$) [2, 3]. Thus, a variable of type $Q_1\tau$ can be used as a variable of type $Q_2\tau$, but not vise versa.

Manually annotating programs with type qualifiers can be expensive and error prone. Therefore, CQual only requires the user to annotate the programs at several key points and uses *type inference* to automatically infer the types of other expressions. For example, in the following code fragment, the variable `b` is not annotated with the qualifier *untrusted*, but we can infer this qualifier for `b` from the assignment statement [1].

```
int $untrusted    a;
int               b;
...
b = a;
```

To eliminate the burden of annotating programs across multiple source code files, CQual provides a `prelude` file. A user can define fully annotated function declarations in this file, and let CQual load it at run-time. This is particularly useful when the source code of certain functions is not available, e.g., the library functions and system calls. In this situation, CQual is still able to use type inference to infer the *qualified types* of expressions from the annotations in the `prelude` file. For example, in the following code fragment, after we annotate the C library function `scanf` in the `prelude` file, CQual is able to infer that the variable `a` is an *untrusted* integer variable in the program.

```
prelude:
   int scanf (char* fmt, $untrusted ...);

user_program.c:
   int    a;
   scanf ("%d", &a);
```

3 Integer Misuse Detection

This section describes how our tool detects potential integer misuse vulnerabilities in C programs. Inspired by the Biba Integrity Model [15], we propose a security check tool based on CQual to detect integer misuse. In our tool, security holes are detected by tracing dependency of variables. Integrity denotes security level. If a value of a variable is updated by an *untrusted* variable during the execution of a program, then the integrity of the variable decreases and the value is regarded as *untrusted*.

Therefore, we categorize integer variables in programs into two types: *trusted* and *untrusted*. An integer variable is *untrusted* because either its value is directly fetched from user input, or the value is propagated from user input. An integer variable is *trusted* because its value has no interaction with *untrusted* integers. In addition, we define program points that generate and propagate *untrusted* integer variables, and program points

[1] CQual requires the type qualifiers start by a $ sign. For convenience, we ignore the $ sign in our discussion except for the code fragments.

that should only accept *trusted* integer variables. For example, suppose each integer parameter of a function `read_file` is annotated as *trusted*. If there is a flow in a program that an *untrusted* integer variable is used as a parameter of function `read_file`, a security exception is generated, resulting in a alarm.

In order to speed up our efforts and develop a working prototype several assumptions are made.

3.1 Assumptions

First, we assume that a programmer does not deliberately write erroneous code. This means that we trust the integer variables prepacked in programs if these internal integer variables do not have any direct or indirect relations with user input. For example, an integer variable may be initialized statically in a program and it is used as index to access an array. There is no interaction between this integer variable and user input. The assumption is that the programmer knows the exact size of the array being accessed and the value of this integer variable is not larger than boundary of the array. We believe that this is a reasonable assumption. In fact, this kind of assumptions are often needed for many static analysis techniques.

We also assume that integer misuse only happens when *untrusted* integer variables are used to access memory. This means it is safe to use *untrusted* integer variables in many other situations. This is because, to the best of our knowledge, most integer related vulnerabilities are only associated to memory access.

To make it clear, user input integers are not limited to the integers given to an application by a command line option, or typed in by a user at a program prompt. They also include many other methods by which a program obtains data from outside the program itself, such as reading a file or receiving network packets. User input data in the context of this paper means the data that is not prepackaged within the program.

3.2 New Type Qualifiers

The first step is to define the type qualifiers for integer variables and the lattice of these type qualifiers in CQual's `lattice` file. Since there are two categories of integer variables in our method, two type qualifiers are defined: *untrusted* and *trusted*. These two qualifiers have a sub-typing relation of *trusted* $\leq$ *untrusted*. This implies that programs that accept an *untrusted* integer variable can also accept a *trusted* integer variable. However, the reverse is not true.

Our implementation is not limited to integer variables and we apply the two new qualifiers to any types of variable in C programs. This is particularly important since integers are often converted from other types of data, and we keep track of these changes. As shown in the following code fragment, the integer variable `a` will become *untrusted* after the assignment because the content of string `str` is *untrusted* [2], and the declaration of the function `atoi` in the `prelude` file specifies that an *untrusted* string has been converted to an *untrusted* integer.

[2] Different positions of a qualifier for a pointer variable have different meanings. In particular, `char untrusted *buf` defines the memory content pointed by `buf` as *untrusted*, `char* untrusted buf` defines the pointer variable `buf` itself as *untrusted*.

```
prelude:
    int $untrusted atoi (char $untrusted* string);

user_program.c:
    char $untrusted* str;
    int              a;
    ...
    a = atoi (str);
```

3.3 Annotations with Type Qualifiers

The second step is to determine the source of *untrusted* data in programs and how they propagate in the programs, and annotate the programs using the *untrusted* qualifier.

By our definition, all user inputs are *untrusted*. Therefore, we need to identify all locations that accept data from outside the programs. For programs based on standard C library and UNIX system calls, the sources of *untrusted* data include: program argument array `argv`, environment variables, standard I/O input, files and network sockets. Program argument array `argv` and environment variables accept user supplied parameters; standard I/O input is usually used to accept keyboard input from the user; files store the data from the file systems; and network sockets provide data transmitted over the network. In POSIX compatible systems, most inputs are handled in the same way as files, so it is unnecessary to distinguish them. Thus identification of user input is relatively simple: find all C library functions and system calls that are related to files, and pick those that fetch data. For example, the system call `read` and C library function `fread` both read data from files. We annotate them in the `prelude` file as illustrated in the following code fragment. In these declarations, the pointer `buf` points to a memory buffer that saves the input data. This memory buffer is annotated as *untrusted*.

```
prelude:
 int read (int fd, void untrusted* buf, int);
 int fread (void untrusted *buf, int, int, FILE*);
```

We focus on a specific type of *untrusted* data: integer variables. Thus, it is necessary to determine type conversion from *untrusted* data to *untrusted* integers. The standard C library provides a limited number of functions that can generate integers from strings. We categorize them into two groups:

1. General purpose library functions that can convert strings into integers. These functions include group of `scanf` functions, e.g., `scanf`, `fscanf`, `sscanf`, etc.. They use the "%d" format to convert a string into an integer.
2. Single purpose library functions that convert strings into integers. These functions include `atoi`, `atol`, `strtol`, `atof`, etc.

In group one, since `scanf` and `fscanf` directly read in data from user input, the integer variables fetched are immediately annotated as *untrusted*. However, since the first argument of `sscanf` can either be *trusted* or *untrusted*, the annotation of its fetched

variables will depend on the qualifier of the first argument. This difference is shown in the following code fragment [3]:

```
prelude:
    int scanf (char* fmt, untrusted ...);
    int fscanf(FILE*, char* fmt, untrusted ...);
    int sscanf(char $_1* str, char* fmt, $_1_2 ...);
```

The functions in the second group are similar to `sscanf`: the qualifier of the returned integer variable depends on the qualifier of the input string. This is shown in the code fragment below:

```
prelude:
    int $_1 atoi (char $_1* s);
    long $_1 atol (char $_1* s);
    long $_1 strtol (char $_1* s);
```

In addition to C library string functions, there are two other methods that convert different types of data into integers. One is type cast. For example, a character variable `ch` may be cast into an integer variable and be assigned to an integer variable `a`. In this case, CQual automatically propagates the type qualifiers of `ch` to `a`. In the other case integers are fetched directly into a memory location of an integer variable. For example, a program can call function `fread` to fetch data from a file into a buffer that is the memory address of an integer variable. In this case, since the content of the buffer is annotated as *untrusted*, CQual will infer the integer variable as *untrusted*.

We must consider the propagation of *untrusted* data in addition to the source of these data. CQual uses type inference to automatically infers the propagation of type qualifiers between variables through assignments. However, this is often inadequate in practice. For example, source code of library functions is often unavailable during analysis. If these functions are not annotated, propagation in libraries would be missed. Such library functions include `strcpy`, `strncpy`, `memcpy`, `memmove`, etc.. We must annotate these functions as below:

```
prelude:
    char $_1_2* strcpy(char $_1_2*, char $_1*);
    char $_1_2* strncpy(char $_1_2*, char $_1*, size_t);
    void $_1_2* memcpy(void $_1_2*, void $_1*, size_t);
    void $_1_2* memmove(void $_1_2*, void $_1*, size_t);
```

After identifying the source of *untrusted* integer variables, the next step is to determine that all expressions that must accept *trusted* integers, and make annotation as needed. To enforce memory safety, all integer variables used as direct or indirect offsets of a pointer must be *trusted* integers.

[3] `$_1` and `$_1_2` are polymorphic qualifier variables in CQual. CQual treats each pair of polymorphic variables (`A`, `B`) as if there was an assignment from `A` to `B` when `A` is a substring of `B`.

Indirect use of integers as offsets of pointers is often seen in C library functions. For example, the length parameters of functions *memcpy* and *snprintf* must be *trusted* parameters because they are implicitly used as the offset. Thus, we annotated the length parameters of these functions as *trusted*, illustrated below:

```
prelude:
  void $_1_2* memcpy(void $_1_2*, void $_1*,
  $trusted size_t);
  int snprintf(char*, $trusted size_t, char*, ...);
```

Integers are often used in pointer arithmetic operations as well, and these integers must be *trusted* to ensure memory safety. Unfortunately, CQual has not implemented the ability to annotate arithmetic operators for pointers. This significantly limits the scope of our approach. To make our tool more usable, we modify CQual's source code to check arithmetic operations on pointers. This is discussed in the next section.

With these annotations, if a program attempts to use an *untrusted* integer in an expression that only accepts *trusted* integers, CQual does not type check the program and will generate an error message. For example, in the following code fragment, the `scanf` function reads input from a user, and CQual infers *untrusted* qualifier for variable `len`. Function `memcpy` only accepts *trusted* integer as its third parameter, and CQual infers *trusted* qualifier for variable `len`. Therefore, the type check fails when *memcpy* is called with *len* as its third argument, and CQual reports an error.

```
char  buf1[BUFSIZ], buf2[BUFSIZ];
int   len;
...
scanf ("%d", &len);
memcpy (buf1, buf2, len);
```

3.4 Modifying CQual's Source Code

In a vulnerable program, user input integers are often used to manipulate pointers. In the code fragment given below, a user input integer is used as an offset to access memory, posing a potential security risk. Since there are no calls to any annotated library functions in this program fragment, CQual is not be able to catch this kind of errors. To solve this problem, we have two possible solutions: one is to enforce a rule that pointer arithmetic operators can only accept *trusted* integers; the other is to propagate type qualifiers of an integer variable, i.e., the *untrusted* in the example below, to the pointer variables and only allow dereferencing of a *trusted* pointer. We choose the first approach in our tool. This is because of the `varargs` feature of C and the way CQual handles conflicts in the second choice [4].

[4] C allows functions to take a variable number of arguments. There is no way to specify the types of the variable arguments. Thus, CQual applies the qualifier of the variable arguments to all levels of the actual arguments [17]. As we annotated the function `scanf`, if we pass a pointer, e.g., `int* ptr` to it, the pointer will becomes `int untrusted * untrusted ptr` after the call. We then cannot deference the pointer `ptr` any more.

```
int   off;
int   ptr[BUFSIZ];
...
scanf ("%d", &off);
ptr[off] = 20;
```

CQual has an infrastructure for annotating C operators, such as pointer deference. However, annotation of certain operators such $+$, $-$, $+=$ and $-=$ are not implemented [5]. These operators are often used in pointer arithmetic operations. To ensure safety of pointer operations, we require that pointer arithmetic operators only accept trusted integers. This necessitates modification to CQual's source code.

Our modification to CQual's source code is small. In particular, we make CQual record the new qualifiers we introduced when it parses the `lattice` file. This is similar to CQual handles several other type qualifiers. Though hard-coding these type qualifiers in CQual is not an ideal solution, we expect that it will not be needed in the future release of CQual. We then add a new constraint to each pointer arithmetic operation such that the integer in the operation must not be *untrusted*. CQual, like other type theory based static program analysis tools, is in general a constraint based static program analysis approach. Each type qualifier associated with a variable (or expression) is a constraint applied on the variable (or the expression). Multiple constraints can be specified for the same variable (or expression). CQual then tries to solve these constraints through static analysis. If some constraints on a variable (or an expression) conflict with each other in the analysis, it usually means a potential error in the program. Thus, our modification to CQual's source is to add a new constraint to each pointer arithmetic operation, and let CQual solve these constraints in the analysis.

4 Experiments

We have performed three kinds of experiments to test the effectiveness of our extension to CQual. First, we created several simple programs to test the functionality of our instrumentation. Second, we tested the tool against applications that have known integer related vulnerabilities reported in the CERT Advisory and Bugtraq to validate our instrumentation. These vulnerabilities in applications can empower an attacker to remotely or locally exploit integer misuse and lead to execution of arbitrary code or denial of service. Our tool is able to find most of the known vulnerabilities in these applications. Finally, we picked several popular open-source applications and executed the tool on their latest version. To our surprise, some of these applications still contain trivial integer misuse vulnerabilities.

4.1 Metrics

In the world of bug finding tools, developers produce metrics to measure the effectiveness of their tool [3].

[5] We tried to modify CQual's source code to let it support annotation of the arithmetic operators. However, our approach requires polymorphic definitions of the operators, which is not supported by CQual to the date of our experiments.

One of the most important metrics are *false positives* and *false negatives*. Usually it is relatively easy to determine the *false positives* by manually auditing the source code of the problematic programs against the warnings reported by the tools. On the other side, it is difficult to determine the *false negatives* since we cannot know the exact number of vulnerabilities in the programs, but only the number of vulnerabilities already discovered. To mitigate this problem we obtained older unfixed versions of the applications to see how many vulnerabilities were in the code. We compared these vulnerabilities to those our tool discovered in the fixed versions and, on average, the false positive ranges from 5% to 15% depending on the program size.

The following are our metrics measure for each program:

1. How many false positives are reported?
2. How many previously unknown vulnerabilities are reported?
3. How easy is it to prepare programs and run the tool?
4. What is the performance of the tool on average programs?
5. How much additional work is required, e.g., annotating source code, header files, etc. ?
6. How easy is it to analyze error reports?

4.2 Test Environment

The testing platform is a single-processor Intel Pentium IV 3.2 GHz PC with 1GB RAM and Linux kernel 2.6.7. The following tools were used during testing: gcc, version 3.3.4; emacs, version 21.3.2; and PAM (Program Analysis Mode for emacs) version 3.01; and GNU Make, version 3.80.

We chose several real-world open-source applications that are all written in C. Some of the applications have known integer related vulnerabilities like integer overflows. We tested not only the versions that have known vulnerabilities, but also the latest versions that have no known integer related vulnerabilities. We noticed that many integer related vulnerabilities were found in the image libraries. This might be because image files often contain a header structure that specifies the parameters of the images like dimension and color depths. Sanitization of these parameters is often inadequate in these applications. Therefore, our test largely focused on these libraries.

4.3 Results

The following section illustrates the types of tests performed to quantify the effectiveness of our tool.

Simple Programs. We have created several toy programs to test the effectiveness of annotations on C library functions, UNIX system calls and pointer arithmetic operations. We have shown several of these examples of our simple programs in the previous sections. Our tool reliably caught errors in these simple programs.

In addition, on some occasions our tool could not reveal some known vulnerabilities in real-world applications. We created simple programs that mimic the vulnerabilities in the applications. These experiments revealed some interesting results. For example, we found a bug in CQual's handling of the `malloc` function.

Table 1. Results of experiment. The Reported Warnings counts the numbers of warning reported by the tool. The Known Bugs is the number of real bugs found by the tool that were reported in public domain. The Unknown Bugs is the reported number of real bugs found that we were not aware of.

Name	Version	Description	Reported Warnings	Known Bugs	Unknown Bugs
gd	2.0.28	A library for dynamically creating images	4	1	1
gdk-pixbuf	0.22.0	Image handling library	4	1	1
rsync	2.5.6	A utility for file transfer	5	1	0
libtiff	3.6.1	A library parsing TIFF files	1	1	0
libunif	4.1.2	A library parsing GIF files	7	0	0
libexif	0.5.12/0.6.11	A library parsing exif files	2/2	0	0
libpng	1.0.18/1.2.8	A library parsing PNG image files	0/0	0	0
libmng	1.0.5	A library parsing MNG files	0	0	0
libwmf	0.2.5/0.2.8.2	A library parsing WMF files	4/4	0	0
libidn	0.1.4/0.5.8	A library implementing Stringprep	1/3	0	0
mpeg_lib	1.3.1	A library decoding MPEG-1 video streams	1	0	0
netpbm	10.18.12	A toolkit manipulating graphic images	8	0	0

Real-World Applications. Table 1 summarizes the experimental results in applying our tool to several popular open source applications.

Below is a brief description of some of the experiments.

GD Graphics Library [18]: There is a known integer overflow vulnerability in version 2.0.28 [19]. In particular, the library reads in the dimension parameters from the image file without careful sanitization and calls the `malloc` function with the parameters. We have annotated the `malloc` function such that it only accepts trusted integer variables as the size parameter.

Initially the tool was not able to detect this rather trivial integer overflow vulnerability. It could not detect similar problems in our simple program that contains the simplified code of the vulnerability in GD library. This turned out to be a bug of CQual, which did not correctly handle the type qualifiers for the parameters of the `malloc` function. This was confirmed by CQual's developers.

After fixing CQual, our approach was able to detect a vulnerability in the GD library (see Fig. 1 for the output of the tool). Interestingly, we found it is different from the known one, and it turned out to be an unknown vulnerability that even exists in the latest release of GD, version 2.0.33. This was confirmed by the maintainers of GD, and they also closed a similar potential vulnerability based on our discovery.

Since our tool did not report the known vulnerability, we examined the GD source. It turned out that the *untrusted* integer variable in the vulnerability is not generated by standard C library functions, but by an unannotated function *png_get_rowbytes* exported by the PNG library. Thus, the integer variables returned by the function are considered to be *trusted*. After we annotated the function, the tool was able to detect this bug.

Rsync [20]: There is a known integer overflow vulnerability in version 2.5.6 [21]. As show in Fig 2, the rsync program reads in an integer from the input and uses it to allocate memory without careful sanitization. There is a potential integer overflow vulnerability here. Our tool, however, did not issue a warning about this trivial vulnerability. After

```
gd.c:2464 type of actual argument 1
doesn't match type of formal
sx: $noninit $trusted $untrusted
$tainted
/cqual/config/prelude.cq:35 $untrusted
== *fgets_ret@2410
  gd.c:2410        == s[]
  gd.c:2414        == *sp
  gd.c:2429        == atoi_ret@2429
  gd.c:2429        == w
  gd.c:2464        == sx
  gd.c:88          == im->sx
  gd.c:1830        == x1
  gd.c:1838        == x
  gd.c:1840        == x
  gd.c:747         == $trusted
```

Fig. 1. Warning of GD Vulnerability

```
s->count = read_int(f);
...
if (s->count == 0)
    return(s);
s->sums = (struct sum_buf *)malloc
    (sizeof(s->sums[0])*s->count);
...
for (i=0; i < (int) s->count;i++) {
   s->sums[i].sum1 = read_int(f);
   read_buf(f,s->sums[i].sum2,
      csum_length);
...
```

Fig. 2. Vulnerable Rsync Program

examining the program, we found it is due to the error reporting mechanism used by CQual. In particular, CQual clusters warnings that have the same root cause and only reports one. The vulnerability relies on the return variable of function `read_int`, which itself contains a warning and masks the real interesting one. Our solution is to annotate the return variable of `read_int` as *untrusted* and execute the tool on the single file that contains the vulnerable code but not the code of `read_int`.

gdk-pixbuf version 0.22.0 [22]: There are two known integer overflow vulnerabilities in *gdk-pixbuf* [23, 24]. Our tool detected two real bugs. One of them is known [23], the other is a new bug in the XBM image handler. We missed one known vulnerability because the CQual's reporting mechanism masked it out.

libtiff version 3.6.1: There are two known integer overflow vulnerabilities [25] in this program. Our tool reported one of the bugs and missed the other. Again, CQual's reporting mechanism masked it out.

4.4 Evaluation

Our tool is capable of finding known and unknown security bugs in real-world programs with a few false positives generated. Our examination of the false positives shows that most of them are due to the lack of flow-sensitive analysis. Another reason is the lack of precision in our analysis. Specifically, type qualifiers like *trusted* and *untrusted* do not record the range of possible values of an integer variable. Arithmetic operations of such an integer variable without appropriate check can turn a *trusted* integer into an *untrusted*. Our tools is unable to detect this type of transitions. This is a more serious limitation of our approach. In addition, our tool also produces false negatives. They are due to the lack of required annotation for functions in programs and CQual's warning reporting mechanism that suppress alerts with the same root cause. We will discuss these issues with more details later, suggesting fixes to the tool.

Preparation of the program for the automated analysis takes from several seconds to a few minutes depending on the size of the program. The first step of preparation involves running the *configure* script where necessary. We then compile the program files with `make CC="gcc -save-temps"` option to create the intermediary files, and execute our tool on them with the command `cqual *.i`.

The runtime performance of the tool is good, thanks to CQual's efficient type inference and type checking algorithms. Usually it takes only a few seconds to finish the analysis. For larger applications of say 20K lines of code, it takes about two minutes.

CQual's output of warnings is mostly clear, since it prints the flow path of qualifier propagation. However, from our experiments, the flow path is not always clear. Because CQual tends to choose the shortest path in the constraint graph, it is not necessarily the path of unsafe sequence of execution. In addition, we sometimes get a path spreading across multiple files that looks irrelevant. This takes a great deal of time in our analysis. We are investigating the reason behind this.

In summary, we have successfully evaluated the effectiveness of our approach on a number of real-world applications and have discovered integer misuse bugs that were unknown prior to our approach. We strongly feel that these results illustrate the potential of our approach in detecting integer misuses that were overlooked. Through the success of our experiments we have provided another example of how CQual can be extended to catch new kinds of vulnerabilities.

5 Discussions

Our analysis currently is data flow-insensitive. This means that the *trusted* or *untrusted* property of an integer variable never changes in the program. However, often this is not true. For example, many programs often fetch an *untrusted* integer from the user input and then correctly sanitize it. Thus, the *untrusted* integer variable can be converted into a *trusted* integer variable. Due to the limitation of CQual in flow-sensitive analysis, our tool cannot handle this kind of cases. In addition, an integer variable may be used for two conflicting purposes in the program. For example, an integer variable may be used as a *untrusted* variable in the first part of a function, and is reused as a *trusted* variable in the second part of the function for a different purpose. In this case, our tool cannot distinguish the two cases correctly, which may result in false alarms. An intuitive fix

for these problems is to introduce flow-sensitive analysis, which does the conversion after sanitization. However, the problem is that the definition of *correct sanitization* usually is closely related to program logic. There is no universal way for sanitization. Therefore, to determine the program point that a sanitization can be done at is difficult. Although we are able to look for sanitization code in an *ad hoc* way, CQual's capability to handle flow-sensitive qualifiers is not general. It requires us to extensively revise CQual's source code.

The precision of our analysis is limited even if it is flow-sensitive. For example, in a case that two *trusted* integer variables are computed by integer arithmetic operations, the result can be bad, e.g., an integer overflow may occur if the programmer is not sufficiently careful. Our analysis cannot detect this kind of error because the type qualifiers do not contain the range information of each integer variable. We also treat all types of integers equally regardless of their size. But in programs, the size of an integer variable can determine its value range and thus eliminate certain problems like integer overflow if they are assigned to an integer variable of larger size. A constraint based analysis with integer range solver [26] can possibly solve this problem.

Our approach missed several known integer overflow vulnerabilities in the applications. The primary reason is that the *untrusted* integers are fetched from functions that were not annotated. For example, many image processing applications often obtain the image dimensions by calling image library functions. Without annotating the library functions, the integer variables returned by these functions are not considered *untrusted*. The other reason is related to CQual's warning reporting mechanism. Though the number of warnings is reduced by clustering, real bugs can be masked. Further study is needed to eliminate this problem.

Since we focus on the use of integer variables to access memory, our tool cannot detect errors out of this situation. For example, if a user input integer variable is used as the amount of a bank account balance, without a proper check our tool will miss the misuse because the variable is not used to access memory.

Recent advances in program analysis have provided several powerful tools like CQual for automatically analyzing legacy code and discovering security bugs. However, for these tools, there is often a gap of usability between the state-of-art and user requirements. For example, in our approach, simply introducing new type qualifiers is not sufficient, especially in the case of pointer arithmetic, even though similar approaches [3, 16] have existed based on the same tool. Therefore, CQual's source code has to be modified. In addition, the changes we have made are not a generic solution that can be used to address similar problems.

6 Related Work

Static analysis is important in eliminating security bugs in the programs. Lexical tools [27, 28, 29] can only find misuse of dangerous function calls in non-preprocessed source files. But for the purpose of our work where the program uses legitimate functions, they are not effective because they do not understand the language semantics. LCLint [30] also uses annotation like qualifiers to specify additional properties to programs. However, it does not apply type inference and requires the programmer to annotate the

source code extensively. Meta-level compilation [31] allows the programmer to specify flow-sensitive property as a finite state automaton and uses the automaton to check the property of the program. However, unlike CQual, it is not designed to be sound or complete.

There have been some studies to detect and prevent integer overflows. Horovitz developed a tool for protecting applications from integer overflows that occur from big loops [11]. His tool, *big loop integer protection* or *blip*, is a gcc extension that detects and flags integer vulnerabilities at run-time. Chinchani et al. [10] propose an approach, named ARCHERR, to automatically insert safety checks against possible integer overflows in the program. This approach also detects integer overflows at execution time. Leblanc proposed to use a safe integer class SafeInt in C++ to avoid integer overflows [14]. However, it only works for C++. In addition, to revise existing C++ programs using the SafeInt class may be a considerable workload. Howard studies integer overflows and proposes several ways to write secure code against integer overflow [12, 13]. However, an automatic tool to analyze C programs and to detect potential integer overflows is not provided.

Out approach is similar to the approach proposed by Shankar et al. [3] to detect format string vulnerabilities. Their approach also is using CQual. Johnson and Wagner have also extended CQual in order to detect user/kernel pointer bugs [16]. These approaches and ours all apply a similar idea by categorizing the data into *trusted* and *untrusted* and detecting misuse of *untrusted* data. Our contribution is to address the specific issues in analyzing integer variables in C programs and propose the solutions for solving these problems.

7 Conclusion and Future Work

We extended CQual to detect integer misuse vulnerabilities in real-world applications through static analysis. We distinguished between *trusted* integer variables and *untrusted* user input integer variables, a vision inspired by the classical Biba integrity model. Our extension to CQual is not limited to integer overflow vulnerabilities but can be applied to any type of integer misuse detection.

Our implementation is not flow-sensitive, thus it generates false positives. In our experiments, false positives range from 5% to 15% depending on the size (lines of code) of the program. In addition, since the type qualifiers do not contain any range information of the value of integer variables, our analysis is not precise. We plan to add flow-sensitive and integer range analysis into our tool to further reduce false positive rates. Our experiments largely focused on open-source image processing libraries as many vulnerabilities are found in these libraries. We also plan to use the tool to check other programs like file archiving and network packets transmission.

Acknowledgment

We thank Rob Johnson and Jeff Foster for developing CQual and helping us to use and understand CQual. We would also like to thank Hao Chen and Zhendong Su for their

suggestions and comments on this project. We thank Tye Stallard, Marcus Tytlutki and Senthilkumar Cheetancheri for proof reading the draft.

References

1. The ICAT team: Icat vulnerability statistics. http://icat.nist.gov/icat.cfm?function=statistics (2005)
2. Foster, J.S., Fhndrich, M., Aiken, A.: A theory of type qualifiers. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99), Atlanta, Georgia. (1999)
3. Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting format string vulnerabilities with type qualifiers. In: Proceedings of the 10th Usenix Security Symposium, Washington, D.C. (2001)
4. Blexim: Basic integer overflows. Phrack Issue 0x3c, Phile 0x0a of 0x10 (2002)
5. CERT: Apache web server chunk handling vulnerability. Advisory CA-2002-17 (2002)
6. CERT: Openssh vulnerabilities in challenge response. Advisory CA-2002-18 (2002)
7. CERT: Integer overflow in sun rpc xdr library routines. Advisory CA-2003-10 (2003)
8. CERT: Apple quicktime contains an integer overflow in the "quicktime.qts" extension. Vulnerability Note VU#782958 (2004)
9. X-Force: Sendmail debugging function signed integer overflow. Vulnerability DB Entry 7016 (2001)
10. Chinchani, R., Iyer, A., Jayaraman, B., Upadhyaya, S.: Archerr: Runtime environment driven program safety. In: Proceedings of 9th European Symposium on Research in Computer Security. (1999)
11. Horovitz, O.: Big loop integer protection. Phrack Issue 0x3c, Phile 0x09 of 0x10 (2002)
12. Howard, M.: An overlooked construct and an integer overflow redux. http://msdn.microsoft.com/library/en-us/dncode/html/secure09112003.asp (2003)
13. Howard, M.: Reviewing code for integer manipulation vulnerabilities. http://msdn.microsoft.com/library/en-us/dncode/html/secure04102003.asp (2003)
14. LeBlanc, D.: Integer handling with the c++ safeint class. http://msdn.microsoft.com/library/en-us/dncode/html/secure01142004.asp (2004)
15. Biba, K.J.: Integrity considerations for secure computer system. Technical Report ESD-TR-76-372, MTR-3153, The MITRE Corporation, USAF Electronic Systems Division, Bedford, MA (1977)
16. Johnson, R., Wagner, D.: Finding user/kernel pointer bugs with type inference. In: Proceedings of the 13th USENIX Security Symposium, San Diego, CA. (2004)
17. Foster, J.S.: Type Qualifiers: Lightweight Specifications to Improve Software Quality. PhD thesis, University of California, Berkeley (2002)
18. Boutell.com: Gd graphics library. http://www.boutell.com/gd/ (2004)
19. Gentoo Linux: Gd: Integer overflow. Security Advisory GLSA 200411-08 (2004)
20. The rsync project: News for rsync 2.5.7. http://rsync.samba.org (2003)
21. Sirainen, T.: Possible security hole. http://www.mail-archive.com/rsync@lists.samba.org/msg08271.html (2003)
22. The GNOME Project: Gnome imaging model - gdkpixbuf. http://developer.gnome.org/arch/imaging/gdkpixbuf.html (2003)
23. CERT: Gdkpixbuf xpm parser contains a heap overflow vulnerability. Vulnerability Note VU#729894 (2004)

24. CERT: Gdkpixbuf ico parser contains a integer overflow vulnerability. Vulnerability Note VU#577654 (2004)
25. CERT: Libtiff contains multiple heap-based buffer overflows. Vulnerability Note VU#948752 (2004)
26. Su, Z., Wagner, D.: A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In: Proceedings of Tenth Internal Conference on Tools and Algorithms for the Construction and Analysis of Systems. (2004)
27. Viega, J., Bloch, J.T., Kohno, T., McGraw, G.: ITS4: A static vulnerability scanner for C and C++ code. ACM Transactions on Information and System Security **5** (2002)
28. Secure Software Inc.: Rats: Rough auditing tool for security. `http://www.securesw.com/rats.php` (2002)
29. Wheeler, D.A.: Flawfinder. `http://www.dwheeler.com/flawfinder/` (2001)
30. Evans, D.: Static detection of dynamic memory errors. In: Proceedings of the 1996 ACM Conference on Programming Language Design and Implementation (SIGPLAN). (1996) 44–53
31. Ashcraft, K., Engler, D.R.: Using programmer-written compiler extensions to catch security holes. In: Proceedings of IEEE Symposium on Security and Privacy. (2002) 143–159