Visualizing Type Qualifier Inference with Eclipse

David Greenfieldboyce Jeffrey S. Foster
University of Maryland, College Park
{dgreenfi,jfoster}@cs.umd.edu

Abstract

Type qualifiers are a lightweight, practical mechanism for specifying and checking program properties. In previous work, we have developed CQUAL, a tool for adding type qualifiers to C. In this short article, we describe an Eclipse plug-in for CQUAL that allows programmers to visualize the results of CQUAL's type qualifier inference and thereby quickly understand and resolve potential programming errors.

1 Introduction

CQUAL is a tool that enables programmers to define and verify program properties using *type qualifiers* in a simple, scalable fashion [2]. A type qualifier is an atomic property that "qualifies" the standard types. Type qualifiers have been used to check for format-string and user/kernel vulnerabilities (see below), as well as to infer `const` annotations [3], to check for deadlocks [4], Y2K bugs, and improper use of init data in the Linux kernel, to find missing security checks in the Linux kernel [9], and for redacting crash information [1].

In this paper, we describe the CQUAL plug-in for Eclipse. The plug-in allows a programmer to review the type qualifiers that CQUAL infers for each identifier in a program and explore the sequence of inferences that indicate that a program property has been violated. From a user's perspective, such an interface is one of the most important and visible feature of this or any other program analysis tool.

2 Background: CQUAL

We begin by describing CQUAL via two example applications: detecting format string vulnerabilities in C [8] and finding user/kernel pointer errors in the Linux kernel [6]. Both of these are security vulnerabilities. We begin by looking at format string vulnerabilities.

```
int main(void)
{
  char *s, *t;
  s = getenv("LD_LIBRARY_PATH");
  t = s;
  printf(t);
}
```

Figure 1: Program with format string vulnerability

The standard C libraries include a number of functions, such as `printf`, `scanf`, and `syslog`, that take as parameters a format string followed by a varying number of arguments. A format string vulnerability arises if an untrusted string—one that is supplied by a malicious user—is used as a format string. For example, consider the program in Figure 1. A malicious user invoking this program could first set the value of the environment variable LD_LIBRARY_PATH so that it contains format specifiers for arguments that are not present. If the malicious user injects `%s` format specifiers into `s` this way, the call to `printf(t)` will read garbage pointers off the stack and most likely crash. Even worse, the attacker can inject the `%n` argument to write to memory and thereby completely compromise security.

In this case, the vulnerability is simple to fix: the last line in the program should be changed to `printf("%s", t)`. Note that in general, however, variable format string arguments must be allowed in order to support, for example, programs that are internationalized. And although format string vulnerabilities are increasingly uncommon, the techniques CQUAL uses to find format string vulnerabilities can also be used to check for other security problems.

```
partial order {
  $untainted [level=value, sign=neg]
  $tainted   [level=value, sign=pos]

  $untainted < $tainted
}
```

Figure 2: Sample configuration file

```
$t char *getenv(const char *name);
int printf(const char $u * format, ...);
```

Figure 3: Sample prelude file

To find format string vulnerabilities with CQUAL, we begin by defining two qualifiers: `$tainted` to mark data under adversary control and `$untainted` to mark data that must be trusted. In CQUAL, all qualifiers except `const` begin with a dollar sign to make parsing easy.

The qualifiers are defined in a CQUAL configuration file, as in Figure 2. The first two lines declare the qualifiers, and the last line introduces a subtyping relationship between them that allows untainted data to be used where tainted data is required, but not the opposite. This is safe because `$untainted` data is always trusted, and only `$tainted` data has restrictions on it. The Eclipse plug-in uses this configuration file to be backwards-compatible with the command-line version of CQUAL.

The next step in finding format string vulnerabilities, or any other similar trusted/untrusted mismatch vulnerabilities, is to annotate the signatures of the relevant functions with the correct qualifiers. This can be done either directly in the code or in a separate *prelude* file. For example, Figure 3 shows a portion of the prelude file used in the tainted/untainted analysis. In this figure, `$t` abbreviates `$tainted` and `$u` abbreviates `$untainted`. These prototypes indicate that data read using `getenv` will be tainted, while the format string for `printf` must be untainted—and CQUAL will therefore require that the result of `getenv` is never passed to `printf`, directly or indirectly.

CQUAL simplifies the process of annotating code with qualifiers by applying *type qualifier inference* to determine the appropriate qualifiers for unmarked variables and functions. For example, in the code in Figure 1, we need not supply qualifiers for `s` and `t`, because CQUAL will determine them automatically. As a result, typically only a small number of annotations is required even for a large program.

Type qualifier inference involves traversing each statement in the program and determining dependencies among the types of various identifiers. For example, given the assignment $e_1 = e_2$, CQUAL requires that the type of e_2 be a subtype of the type of e_1. Each expression that specifies a relationship between two qualifiers generates an appropriate constraint. CQUAL solves this system of constraints using context-free language (CFL) reachability [7].

When CQUAL infers two different, contradictory qualifiers for the same variable or function, this indicates that there may be a violation of the properties specified in the partial order and prelude files, e.g., a format string vulnerability. However, this does not guarantee that there is a vulnerability, because the analysis is necessarily conservative.

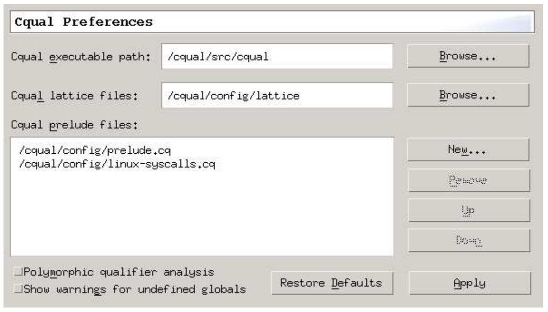
The CQUAL visualization interface, described in Section 3, is critical in determining which warnings correspond to actual vulnerabilities, where the vulnerabilities are, and, often, how to fix them.

Another application that CQUAL has been used for is detecting unchecked uses of user-space pointers by the Linux kernel [6]. In handling system calls, the Linux kernel often copies data from user-space to kernel-space. However, pointers to user-space should not be dereferenced without first verifying that they are valid user-space memory addresses; without this check, subtle security holes may arise. To perform these checks, all accesses to user-space should be performed using the function `copy_from_user()`.

CQUAL can enforce this requirement by using `$user` and `$kernel` qualifiers to mark data from user and kernel space, respectively, and requiring that only `$kernel` data is explicitly dereferenced. Due to a lack of space, we omit further discussion; Section 3 presents one example user/kernel error.

3 The CQUAL Plug-in Interface

Previous versions of CQUAL included an interface for presenting type qualifier inference results via Program Analysis Mode (PAM) for Emacs [5]. The CQUAL plug-in is based on PAM. However,



Cqual Preferences

Cqual executable path: /cqual/src/cqual

Cqual lattice files: /cqual/config/lattice

Cqual prelude files:
/cqual/config/prelude.cq
/cqual/config/linux-syscalls.cq

☐ Polymorphic qualifier analysis
☐ Show warnings for undefined globals

Figure 4: The preferences dialog

the flexibility and extensibility of the Eclipse plug-in framework has provided us more control over the presentation of the analysis results and allowed tighter integration of CQUAL with the development process.

To use CQUAL with a project in Eclipse, the user first sets preferences using the dialog shown in Figure 4. Here the user locates the CQUAL executable, the partial order configuration file, and any prelude files that define the analysis. The user also specifies whether to use polymorphic qualifier inference and whether to generate warnings for undefined global identifiers that are not given types in a prelude file.

To run CQUAL on a set of files or folders in a project, the user first selects the items in the Navigator view, and then chooses "Run Cqual" from the pop-up context menu. The plug-in opens a CQUAL view and launches the analysis in a separate process. The results of the analysis are communicated back to the Eclipse plug-in for display. The results include the set of files that were analyzed, any undefined global variables, and any errors that were detected. If the interface needs to display more detailed information, the plug-in retrieves it by communicating with CQUAL, which continues to run in the background.

Previously, the CQUAL process was responsible for generating the details of the results visualization, with PAM acting as a thin presentation layer. In order to take advantage of the programmability of the Eclipse interface, we modified the CQUAL process to send the elements of the analysis as discrete data elements encoded as S-expressions (we could also use XML), which are then unpacked into corresponding Java objects. This allows greater flexibility to the plug-in programmer in determining how to best use the Eclipse interface for presentation.

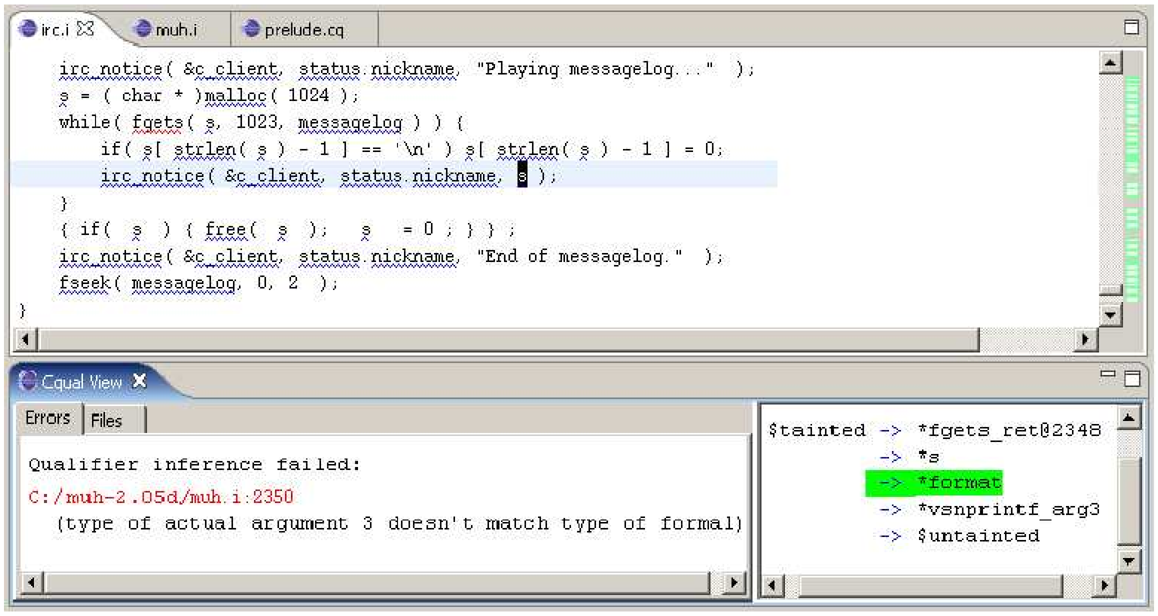
Figure 5 shows the results of analyzing `muh 2.05d`, an IRC proxy program with a format string vulnerability. The lower window shows the CQUAL view, which provides a list of errors that have been found and, on another tab, the list of analyzed files. In this case, CQUAL has found one potential format string vulnerability and identified a location in the code that appears to be the source of the error.

When the programmer clicks on the error message, as we have done here, the plug-in displays in the lower right pane the path of inference that led to the apparent error (see below). The plug-in also opens a modified text editor showing the file in which the error was detected. In this editor, identifiers on the path of the current error are annotated with red hyperlinks, while the rest of the identifiers are annotated with blue hyperlinks. As the programmer selects different error messages, the set of red and blue hyperlinks is changed appropriately. Clicking on these links displays the type qualifiers that have been inferred for the identifier—in this case, showing for each identifier whether CQUAL inferred that it was `$tainted` or `$untainted`.

In an earlier version of the interface, we colored the hyperlinks according to the inferred qualifiers: `$untainted` identifiers would be green, and `$tainted` identifiers would be red. However, this additional information in the interface turned out to be unhelpful. In our experience, it is only the identifiers involved in errors that we wish to isolate, and adding extra colors for other information simply makes it harder to read the source code to little benefit; instead, programmers click on the hyperlinks to retrieve this little-used information.

As mentioned above, clicking on an error displays an inference path determined by CQUAL. This path is a sequence of constraints that implies a contradiction in the qualifiers—in this example, a path on which `$tainted` data may reach a position that must be `$untainted`. Thus, in our interface, such a path serves as an explanation of why CQUAL concluded that an error exists in the program.

In general there are many different paths that could be used to explain an error. Based on previous experience, CQUAL selects the shortest one, which we have often found the most useful to look at. The CQUAL plug-in uses a heuristic to choose a good position in this path to begin looking for the error. When the path is first presented, this position



```
    irc_notice( &c_client, status.nickname, "Playing messagelog..."  );
    s = ( char * )malloc( 1024 );
    while( fgets( s, 1023, messagelog ) ) {
        if( s[ strlen( s ) - 1 ] == '\n' ) s[ strlen( s ) - 1 ] = 0;
        irc_notice( &c_client, status.nickname, s );
    }
    { if(  s  ) { free(  s  );   s   = 0 ; } } ;
    irc_notice( &c_client, status.nickname, "End of messagelog."  );
    fseek( messagelog, 0, 2  );
}
```

```
Qualifier inference failed:
C:/muh-2.05d/muh.i:2350
  (type of actual argument 3 doesn't match type of formal)
```

```
$tainted -> *fgets_ret@2348
         -> *s
         -> *format
         -> *vsnprintf_arg3
         -> $untainted
```

Figure 5: An example format string vulnerability from muh 2.05d

(for our example, *format) is highlighted, and the location where that constraint was generated is selected in the text editor. The user can click on positions in the path to navigate through the program and verify the error.

To illustrate the use of the plug-in, we present the process for tracking down the format string vulnerability detected in Figure 5. This figure shows the state of the plug-in after clicking on the error message, with *format highlighted in the path and the call to irc_notice highlighted in the program listing, where the constraint between *format and *s is inferred.

Clicking on the next link in the path, *vsnprintf_arg3, highlights the line containing the call to vsnprintf() (last line shown in figure), in which format is passed as the third argument. Clicking on the final link in the path shows us the line in the prelude file declaring vsnprintf(), in which the third argument is declared as $untainted.

At this point it is clear why the third argument to irc_notice() is required to be $untainted. Now we follow the path in the other direction, to find out how a $tainted value reaches this position. Clicking on *format in the path, we jump to a statement (not shown) in which $tainted data is passed as the third argument irc_notice:

```
irc_notice(&c_client,status.nickname,s);
```

Clicking on the preceding link, we then find the statement (also not shown)

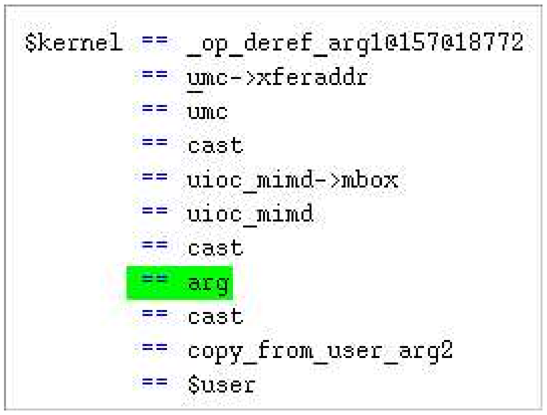
```
while(fgets(s, 1023, messagelog)) \{...
```

Since data read from the file may not be trusted, the signature of fgets() in the prelude file declares this argument as $tainted. Thus by tracing the path backwards, we see that the value of s is being read from a file via fgets(), and then passed to irc_notice, and so this is a true format string vulnerability. The fix is to rewrite the statement calling irc_notice() as

```
irc_notice(&c_client,status.nickname,
  "%s", s);
```

Although this example took several paragraphs to describe, it should be clear that, at least in this case, tracking down the vulnerability does not take much time.

We have found the interface to be similarly effective for investigating user/kernel vulnerabilities. We briefly describe the process of exploring an error path from analyzing megaraid.c from a scsi device driver in the 2.6.7-rc3 version of the Linux kernel. The error path is shown in Figure 6. In this path, the relationship between adjacent qualifiers is indicated by ''==''. Since no subtyping relationship exists between $userand $kernel, contraints between qualifiers indicate that the qualifiers must be identical.



```
$kernel == _op_deref_arg1@157@18772
        == umc->xferaddr
        == umc
        == cast
        == uioc_mimd->mbox
        == uioc_mimd
        == cast
        == arg
        == cast
        == copy_from_user_arg2
        == $user
```

Figure 6: Error path from a User/Kernel error

The section of the program where this error occurs includes the following statements:

```
copy_from_user(signature,(char *)arg, 7)
uioc_mimd = (struct uioctl_t *)arg;
umc = (megacmd_t *)uioc_mimd->mbox;
upthru = (mega_passthru *)umc->xferaddr;
```

CQUAL reports the warning "Incompatible qualifiers at cast". The pointer `arg` is copied from user space. It is then assigned to `uioc_mimd`. Since this is now a pointer to a user-space structure, any pointers within the structure are also user-space pointers, so `uioc_mimd->mbox` is a user-space pointer. In the next statement, we see that `umc` will also be a user-space pointer. And in the final statement this pointer is dereferenced without verification, which produces a vulnerability. (The astute reader may see how this error could be presented via a more compact path of inference. This is an issue in the CQUAL analysis that we are pursuing.)

In the CQUAL plug-in interface, clicking on the error brings up the statement in which `arg` is assigned to `uioc_mimd`, and the corresponding edge is selected in the error path. By examining the path edges above and below this edge and the associated statements, the user can quickly come to understand the source of the error.

In an example such as this one, an experienced user may often be able to understand the nature of the error from a cursory perusal of the error path presented by CQUAL. While we have only tested the interface on a small number of examples, we are encouraged that it will prove to be an effective means for visualizing qualifier inference and related analysis techniques.

4 Future Work

We are continuing to improve the plug-in interface to CQUAL. We are also in the process of developing JQUAL, a tool for adding type qualifiers to Java. JQUAL uses Eclipse's Java parser and links to CQUAL's type qualifier constraint solver via the Java Native Interface.

About the Authors

David Greenfieldboyce is a graduate student at the University of Maryland in College Park.

Jeffrey S. Foster is an assistant professor at the University of Maryland in College Park. His research focuses on programming languages with applications to software engineering, and includes advanced static type systems, scalable constraint-based analysis, and alias analysis.

References

[1] P. Broadwell, M. Harren, and N. Sastry. Scrash: A System for Generating Secure Crash Information. In *Usenix Security 2003*, Washington, DC, Aug. 2003.

[2] J. S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, Dec. 2002.

[3] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *PLDI'99*, pages 192–203, Atlanta, Georgia, May 1999.

[4] J. S. Foster, T. Terauchi, and A. Aiken. Flow-Sensitive Type Qualifiers. In *PLDI'02*, pages 1–12, Berlin, Germany, June 2002.

[5] C. Harrelson. Program Analysis Mode, 2001. `http://www.cs.berkeley.edu/~chrishtr/pam`.

[6] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs with Type Inference. In *Usenix Security 2004*, San Diego, CA, Aug. 2004.

[7] J. Rehof and M. Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *POPL'01*, pages 54–66, London, United Kingdom, Jan. 2001.

[8] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Usenix Security 2001*, Washington, D.C., Aug. 2001.

[9] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Usenix Security 2002*, San Francisco, CA, Aug. 2002.