Applying Flow-sensitive CQUAL to Verify MINIX Authorization Check Placement

Timothy Fraser Nick L. Petroni Jr. William A. Arbaugh
University of Maryland Department of Computer Science
{tfraser,npetroni,waa}@cs.umd.edu

Abstract

We present the first use of flow-sensitive CQUAL to verify the placement of operating system authorization checks. Our analysis of MINIX 3 system servers and discovery of a non-exploitable Time-Of-Check/Time-Of-Use bug demonstrate the effectiveness of flow-sensitive CQUAL and its advantage over earlier flow-insensitive versions. We also identify and suggest alternatives to current CQUAL usability features that encourage analysts to make omissions that cause the otherwise sound tool to produce false-negative results.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.4.6 [*Operating Systems*]: Security and Protection—Access controls

General Terms Security

Keywords CQUAL, MINIX, static analysis, type qualifiers

1. Introduction

We present the first use of flow-sensitive CQUAL to verifying the placement of authorization checks in an operating system. We used CQUAL to verify the placement of authorization checks in the MINIX 3.1.1 Data Store (DS), Process Manager (PM), and File System (FS) system servers. These system servers are privileged user-space processes running on top of the MINIX microkernel and are responsible for making most of the authorization checks commonly expected in a UNIX system. The purpose of the analysis was to verify that all the checks were in place and could not be bypassed by an adversary exploiting an unexpected control flow. Our effort discovered a non-exploitable Time-Of-Check/Time-Of-Use (TOCTOU) bug in PM.

Our experiment had three primary results: First, the new flow-sensitive CQUAL accomplished its task without the complex workarounds required in similar previous efforts using flow-insensitive versions. Second, the structure of MINIX authorization checks caused our task to require more manual annotation than earlier efforts to verify different security properties with CQUAL. Finally, the fact that CQUAL permits unqualified types makes it "insecure by default", encouraging analysts to omit annotations, causing the otherwise sound tool to report false-negatives.

PLAS'06 June 10, 2006, Ottawa, Ontario, Canada.

We describe our experiment in section 2, present our results in section 3, and propose a new "secure by default" CQUAL mode in section 4 to address some of the problems found. We discuss related work in section 5 and present our conclusions in section 6.

2. Experiment

The goal of verifying authorization check placement in a server is to verify the absence of control-flow paths where a client can ask the server to perform some security-relevant operation (for example, write a file) without that request first passing an authorization check (for example, checking to see if the client is permitted to write the file according to the file's access control list).

For example, in the following annotated C code the function `service` operates on `object *` handles on behalf of clients. The `use` function encapsulates some security-relevant operation on handles. The `check` function encapsulates an authorization check intended to guard the `use` function: `check` returns a handle in cases where `use` is authorized and `NULL` when it is not.

```
01 int use(object * $checked);
02 object * $checked check(object *);
03
04 int service(object * $unchecked h, int skip) {
05   if(!skip)
06     if(!(h = check(h))) return -1;
07   return use(h);
08 }
```

Note that along the control flow path where `skip` is false (lines 05, 06, 07 sequentially), control must pass `check`, ensuring that an adversarial client cannot misuse the `use` function. However, along the control flow path where `skip` is true (jumps from line 05 to 07), there is no call to `check`. Consequently, an adversarial client can misuse the `use` function simply by ensuring that the program always takes the `skip` path. We used CQUAL to discover such paths.

CQUAL is a static analysis tool that uses type qualifier inference to detect inconsistencies in a large program based on a small number of type qualifier annotations placed by an analyst [5, 6]. Analysts state facts about a C program by annotating its types with CQUAL type qualifiers that are similar to C's `const` qualifier.

We applied one of two qualifiers to handles like `object *`'s in the above example: `$checked` or `$unchecked`. In our approach, the `object * $checked` type is a handle that has passed an authorization check. The `object * $unchecked` type is one that has not. For example, the `$checked` annotation on line 01 states that `use` function's input handle must be `$checked`. Similarly, the `$checked` annotation on line 02 states that handles returned by `check` are `$checked`. Finally, line 04's `$unchecked` annotation on the `service` function's `h` parameter states that `h` is initially `$unchecked`.

When CQUAL analyzes this code statically, it determines that line 06's call to `check` along the `!skip` path changes `h`'s type to `object * $checked`. However, CQUAL also notes that the `skip` path avoids this call, allowing `h`'s type to remain `object * $unchecked` upon reaching line 07. Being unable to predict the value of `skip` statically, CQUAL conservatively decides that `h` may be either `$checked` or `$unchecked` at line 07. The call to `use` states that `h` must definitely be `$checked`, contradicting the "either `$checked` or `$unchecked`" fact inferred from previous lines. This contradiction indicates the `check` authorization check is misplaced because a least one path misses it. Moving it out of the `if(!skip)` body would correct this problem.

Section 4 describes previous efforts to verify security properties using earlier flow-insensitive versions of CQUAL. Due to their lack of flow-sensitivity, these versions could not permit functions like `check` to change the type of a variable—a variable's type could be set only once. Changing types from `$unchecked` to `$checked` is essential to representing the effects of authorization checks; the lack of this feature was a significant limitation on previous efforts to verify authorization check placement with CQUAL.

To prevent CQUAL from reporting contradictions in certain harmless cases, we defined `$checked` to be a subtype of `$unchecked`, allowing a `$checked` handle to be used in places where an `$unchecked` one was expected. We took information from the System Interfaces Volume of the Single Unix Specification version 3 [15] and man pages to form our list of checks to verify .

3. Results

MINIX *checks required more annotation.* We analyzed approximately 25,000 lines of preprocessed C code, adding 873 annotations by automatic string-replacement and 198 annotations manually. Total CQUAL runtime was less than 3 seconds on a typical laptop. During the experiment, it became apparent that the authorization check structure of some services permitted better uses of CQUAL than others in terms of manual annotation labor saved. Table 1 provides code fragments demonstrating various cases.

Section 2's code fragment represents an imaginary ideal situation for saving annotation labor. Both authorization checks and security-relevant operations are neatly encapsulated in functions. Once the analyst has annotated their prototypes, CQUAL can reason about their calls throughout the program without further help from the analyst. This ideal situation is similar to that encountered by some earlier efforts to apply CQUAL to different security properties described in section 5. Unfortunately, the actual MINIX code never had such convenient encapsulation.

The example in table 1A contains a highly-simplified representation of the PM code for sending a signal to one or more processes on behalf of an authorized client. Because a client can signal many processes with a single request to PM, PM must iterate through its array of process structures (`procs`, line 17), check if the client wants to signal a given entry (line 19) and authorize the send to that entry's process individually (line 20). CQUAL is of considerable help in verifying authorization check placement in complicated control flow structures such as this signal-sending loop.

Unlike the previous example, the authorization check on line 20 is not encapsulated in a function that returns a handle. Instead, it is implemented with a C comparison operator. Finding no authorization check function to annotate, the analyst must manually locate the authorization check and place a CQUAL `change_type` operator to change the formerly `$unchecked` handle's type to `$checked` on the lines following a positive authorization outcome (line 21). CQUAL operators like `change_type` affect CQUAL's analysis but do not result in executable code.

Placing `change_type` operators requires a much larger amount of manual code examination than was required to simply annotate

A: simplified PM/signal.c:check_sig()

```
11 void sig_proc(proc * $checked, int);
12
13 void check_sig(proc *sender, pid_t who,
14                 int sig) {
15   proc * $unchecked p;
16
17   for(p = &procs[0]; p < &procs[LAST]; p++) {
18     change_type(p, proc * $unchecked);
19     if(who != p->pid) continue;
20     if(sender->euid != p->euid) continue;
21     change_type(p, proc * $checked);
22     sig_proc(p, sig);
23   }
24 }
```

B: simplified FS/protect.c:do_chmod()

```
31 int do_chmod(proc *caller, char *path,
32               mode_t mode) {
33   inode * $unchecked i = namei(path);
34
35   if(caller->euid != i->uid) return -1;
36   change_type(i, inode * $checked);
37   assert_type(i, inode * $checked);
38   i->mode = mode;
39   return 0;
40 }
```

C: imaginary annotation omission example

```
51 void use(object * $checked);
52
53 object * $unchecked h;
54 h = 42;
55 use(h);
```

Table 1. Example annotation cases

authorization check function parameters in the previous example. Fortunately, in this example the security-relevant operation is still encapsulated in a function—`sig_proc`. Although this case is still more work than the previous imaginary optimal example, the analyst can still save considerable labor by annotating `sig_proc`'s prototype (line 11). Four out of 23 services annotated in the experiment were of this sub-optimal but relatively convenient form.

The example in table 1B contains a highly-simplified representation of the FS code for changing inode access control settings on behalf of authorized users. In this case, neither the authorization check nor the security relevant operation are encapsulated in functions. Instead, they are implemented with C comparison and assignment operators (lines 35 and 38). In this case, the analyst must insert both a CQUAL `change_type` operator after the authorization check to reflect a positive authorization (line 36) and a CQUAL `assert_type` operator before the security-relevant operation to demand that the handle be checked at that point (line 37).

This no-encapsulation case represents the worst-case scenario for manual annotation effort and was unfortunately common in the experiment, representing 19 out of 23 annotated services. In simple functions like table 1B's example, the two CQUAL operators wound up on adjacent lines, suggesting that the function's correctness might be just as easily verified by hand. However, in more complex functions, these operators were separated by intervening instructions, putting CQUAL to better use. In 11 of the 19 worst-case services, the annotations fell in simple functions resulting in `change_type` and `assert_type` appearing on adjacent lines.

CQUAL *is insecure by default.* Table 1C contains an imaginary example code fragment designed to explain how the fact that CQUAL permits unqualified types encourages analysts to mistakenly omit annotations, leading the otherwise sound CQUAL tool to report false-negatives.

In the 1C code fragment, the analyst has already annotated all `object *` handles as `$unchecked` using automatic string-replacement, including the local variable `h` on line 53. The analyst has also manually annotated the `use` function—a function that encapsulated a security-relevant operation—indicating that it demands a `$checked` input handle as a parameter (line 51). As he has not yet used the CQUAL `change_type` operator to annotate any checks, upon running CQUAL the analyst may expect to find all calls to `use`—since all actual parameters to `use` will be `$unchecked`, CQUAL should report type errors on every call.

This expectation would be wrong, however. In response to the assignment of an unqualified value on line 54, CQUAL will consider the handle `h` to have no qualifiers from that point in the program on, forgetting its earlier `$unchecked` annotation. Upon reaching line 55, CQUAL will see that `use` states the constraint that `h` must be `$checked` and, finding no contradicting qualifier on `h` at that point, will infer that `h` is indeed `$checked`. As a result, CQUAL will not report the call to `use`, the analyst will not examine it for authorization checks, and a false-negative due to analyst error will result if an authorization check is indeed missing. If CQUAL insisted that the analyst cast the literal 42 to an annotated type like `object * $unchecked` or automatically applied such a default annotation itself, this problem would be avoided.

During the early part of the experiment, we missed the call to `sig_proc` in table 1A's check_sig example for precisely this reason, later discovering it only through manual analysis. Note that to avoid such mistakes, an analyst must take great care to search through the source for intervening assignments such as those in the `for` statement on line 17 of table 1A's check_sig example.

We characterize the fact that CQUAL permits unannotated types rather than insisting on or providing default annotations chosen to avoid false-negatives as "insecure by default" in comparison to the "secure by default" alternative we propose in section 4. However, this aspect of CQUAL stems from a deliberate and reasonable decision to support usability by reducing typical manual annotation burdens. The usability of our potential alternative remains an open question.

PM had a TOCTOU bug. The analysis revealed a classic UNIX filesystem TOCTOU bug of the kind described by Bishop and Dilger [2] in the MINIX 3.1.1 PM. Note that on MINIX, PM is a user process like any other outside the microkernel.

```
91 /* make my effective uid that of my client */
92 if(access("filename", rx_mask)) return EACCES;
93 /* return effective uid to superuser status */
94 open("filename", flags);
```

On line 91 the PM impersonates its client, dropping privileges so that it may call `access` on line 92 to determine whether or not the client is authorized to open the file named "filename" for execution. This done, on line 93 the PM regains its full privileges and proceeds to open the file on line 94. With its full privileges restored, PM can open any file.

Unfortunately, on a traditional UNIX system, "filename" does not necessarily refer to the same file at the time of the `open` as it did at the time of the `check`. An adversarial user can ask PM to execute an allowable file with a symbolic link in its "filename" path. Once this file passes the `access` check, a quick adversary can then change the symbolic link so that the same "filename" points to a prohibited file by the time PM reaches `open`. PM will then open the prohibited file and allow the adversary to execute it.

Fortunately, MINIX 3.1.1 does not yet implement symbolic links, so this TOCTOU bug is not currently exploitable. The MINIX developers noted that the bug's discovery was timely, however, because symbolic links are scheduled for implementation in the next release [7].

If the PM's authorization check was structured like the imaginary optimal example in section 2, CQUAL might have detected the TOCTOU bug and directed our attention to it. However, this was not the case. Instead, we discovered the bug manually while trying to decide where to place CQUAL operators as in the example in table 1B.

4. Future work

It is not hard to imagine a new CQUAL mode specialized for verifying authorization check placement in which a mistakenly omitted annotation could not produce a false-negative result. The current version of CQUAL will report no type errors when given an unannotated but otherwise correct input. Following the "secure by default" principle of Saltzer and Schroder [12], the specialized mode might report a maximum number of type errors when given an unannotated input.

An analyst uses the current CQUAL by first adding annotations to alert CQUAL to potential problems ("this handle is unchecked and that operation demands a checked handle") and then adds further annotations to resolve these problems ("this statement makes the handle checked"). Omitting a potential-problem annotation can lead to false-negatives. With the specialized mode, CQUAL would assume problems were everywhere, and the analyst would add annotations only to resolve them. Omissions would cause CQUAL to err on the side of caution with false-positives.

This specialized mode might be implemented by modifying CQUAL's parser to give un-annotated variables, literals, and function signatures special treatment: It might automatically annotate them in a manner designed to indicate potential problems, freeing the analyst from this critical error-prone task. Alternately, a high-level annotation-placement language similar to the one used in MECA [17] might be developed to achieve the same end.

5. Related work

There have been a number of prior static analyses of properties similar to authorization check placement. Zhang and others used an earlier flow-insensitive version of CQUAL to verify the placement of Linux Security Module (LSM) [16] authorization hooks in the Linux kernel, finding one exploitable bug [18]. Because authorization check placement is a flow-sensitive property and Zhang's version of CQUAL was flow-insensitive, Zhang was unable to `change_type` a variable from `$unchecked` to `$checked` on successful completion of an authorization check as done in this work. Instead, whenever she desired to change the type of a variable `foo` from `$unchecked` to `$checked`, she inserted a new `$checked` variable `bar` into the code, added a statement to assign it the value of `foo`, and replaced subsequent uses of `foo` with `bar`. Zhang relied on an argument outside of CQUAL to show that there were no flows bypassing checks inside functions.

Ashcraft and Engler developed a technique called Metacompilation in which an analyst applies knowledge of the purpose and meaning of the constructs of a given program (function semantics, naming conventions, patterns) to extend a version of the GNU Compiler Collection with new static analysis functionality designed to verify a specific security property [1]. They used this technique to verify that untrusted input is validated before being used by security-relevant functions—a property similar to authorization check placement—finding 125 bugs in a then-current version of Linux and 12 in OpenBSD. Although Ashcraft and Engler

cite their approach as an alternative to type annotation schemes where omitted annotations can lead to false-negatives, the soundness of Metacompilation results are not guaranteed.

Schwarz and others have applied the MOPS model-checker to checking an entire GNU/Linux distribution for security violations, including the same kind of TOCTOU bugs discovered by our own work [13]. Although they did not verify the placement of authorization checks, their method of expressing desired security properties as finite state automata seems capable of doing so and deserves investigation. The use of function pointers and signals can cause MOPS to report false-negatives. Nevertheless, their analysis found 108 exploitable bugs.

Jensen and others developed a model-checking approach to verifying whether or not a Java 1.2 program's security checks are sufficient to support some program-wide security property—an analysis capable of both verifying authorization check placement and detecting redundant checks [8]. Their approach to modeling programs involved eliminating all details except function calls and checks. To apply it to MINIX, one would have to adjust it both to allow for the C language and to avoid discarding the many security-relevant operations not encapsulated in functions.

Prior uses of* CQUAL *for security properties other than authorization check placement have generally required less annotation. Johnson and Wagner used a flow-insensitive version of CQUAL to find 17 exploitable User/Kernel bugs in the Linux kernel [9]. These bugs included cases where the kernel functions `copy_to_user` and `copy_from_user` were incorrectly used with user-space addresses when kernel-space addresses were expected, or with kernel-space addresses when user-space addresses were expected.

Shankar and others used a flow-insensitive version of CQUAL to detect 3 format-string vulnerabilities in a collection of popular daemons written in C [14]. These vulnerabilities occur in cases where an adversarial user can control the format strings used by C library functions like `sprintf`.

In contrast to the verification of authorization check placement, the security properties investigated by both of these efforts did not require variables to change their type annotations (as in `$unchecked` to `$checked`) once initially set. In further contrast, all annotations were on operators and function prototypes, allowing CQUAL's type qualifier inference to eliminate the kind of extensive manual search and annotation effort described in section 3.

CQUAL *compares well with process-oriented alternatives.* Manual software engineering processes such as code reviews mandated by criteria schemes [10, 3] can be applied to authorization check placement, but unlike CQUAL's automated analysis, they cannot be easily repeated whenever a program changes due to maintenance. Cumbersome historical approaches like the Hierarchical Development Methodology (HDM) could produce formal specifications of programs with properties proven by non-interactive automated theorem provers [11]. However, CQUAL is sufficient to produce our desired result with much less effort. On the other hand, CQUAL bears some resemblance to the historical Gypsy Verification Environment (GVE)—in both systems, the analyst encodes both the program and logical assertions about its behavior in the same source code and then debugs the result with the help of a compiler that is essentially a highly-specialized interactive automated theorem prover [4].

6. Conclusion

We have described the first use of flow-sensitive CQUAL to verify the placement of operating system authorization checks—an analysis of the MINIX 3.1.1 DS, PM, and FS system servers. Our analysis uncovered a non-exploitable TOCTOU bug in PM. We demonstrated that CQUAL's new flow-sensitive features avoided the complex workarounds employed by similar efforts with an earlier flow-insensitive version of CQUAL. We described how the structure of MINIX authorization checks required a larger manual annotation burden than earlier CQUAL efforts to verify different properties. Finally, we identified cases in which CQUAL's provision for unannotated types led to "insecure by default" behavior in which an analyst could easily cause CQUAL to produce false-negative results by mistakenly omitting annotations and suggested improvements.

Acknowledgments

The authors would like to thank Professors Jeffrey S. Foster and Michael W. Hicks for their invaluable advice and review.

References

[1] Ken Ashcraft and Dawson Engler Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[2] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. In *USENIX Computing Systems*, vol. 2, no. 2, 1996.

[3] Common Criteria for Information Technology Security Evaluation v2.3 ISO/IEC 15408:2005, August, 2005.

[4] M. Cheheyl and M. Gasser and G. Huff and J. Millen. Verifying Security. In *Computing Surveys* vol. 13, no. 3, September 1981.

[5] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[6] Jeffrey S. Foster and Tachio Terauchi and Alex Aiken. Flow-Sensitive Type Qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, 2002.

[7] Ben Gras. In personal E-mail communication, 23 February 2006.

[8] T. Jensen and D. Le Metayer and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[9] Rob Johnson and David Wagner. Finding User/Kernel Bugs With Type Inference. In *Proceedings of the 13th Usenix Security Symposium*, 2004.

[10] National Computer Security Center. Department of Defense Trusted Computer System Evaluation Criteria. Dod 5200.28-STD, 1985.

[11] P. Neumann and R. Feiertag and L. Robinson and K. Levitt. Software Development and Proofs of Multi-Level Security In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[12] J. H. Saltzer and M. D. Schroder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE Vol. 63(9)*, 1975.

[13] Benjamin Schwarz and Hao Chen and David Wagner and Jeremy Lin and Wei Tu. Model Checking An Entire Linux Distribution for Security Violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[14] Umesh Shankar and Kunal Talwar and Jeffrey S. Foster and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[15] The Open Group. Single UNIX Standard: System Interfaces Volume. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004.

[16] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th Annual USENIX Security Symposium*, 2002.

[17] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, 2003.

[18] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th Usenix Security Symposium*, 2002.