Information Flow Using CQual

Morgan Kleene

July 18, 2008

1 Introduction

The RIFLE [2] Architecture defines an architecture where memory addresses are augmented with information flow information and flows are tracked at runtime by the architecture. This, in conjunction with binary rewriting, allows all information flows unrelated to timing and premature program termination to be caught at runtime. Its weakness is that it is an architectural approach. There is no efficient way to implement the dynamic checks required by the architecture in software and it appears unlikely that such a system will be made available in the near future.

2 Static Systems and Dynamic Systems

Dynamic systems have a few big advantages over static ones. The most important one is that there are things that a program *may* do, according to what is known statically, but never, or almost never does. If we rely on purely static methods then we will reject many useful programs. For example, a webserver does nothing but read from the filesystem and write to the network. Since, in general, we do not know the names of the files being read and then written out to the network, it is impossible to verify that such a program does not compromise sensitive information without making use of somewhat heavyweight theoretical tools, such as schemes where the sanitization of file names becomes part of the static analysis.

For most applications that manipulate data in nontrivial ways, there will certainly be static information flows that may, depending on which information is accessed, compromise policy.

3 Our Approach

Motivated by the flexibility of dynamic systems and the efficiency of static ones we make the following observation: *there is no need for full-scale tracking as long as we track the security relevant flows.* In the webserver example, it is clear how a user may enforce his privacy; he may simply look at the code, figure

out where the reads are, and not allow any reads of sensitive data. If we desire a bit more flexibility, we could figure out which reads are dangerous, meaning that the information returned from such reads might flow to a dangerous output channel, and only monitor those. If we desire implement a scheme as general as that of RIFLE we need to track the state of all information flows and then, when we reach a program point where output is written to a potentially dangerous channel, decide, based on the state of the channel, whether or not we should allow this flow. It is the goal of this paper to assess whether such an approach is feasible.

4 Subtyping and Information Flow

In a subtyping system, the notation $A \leq B$ denotes that anywhere a value of type B is expected, it is acceptable to use a value of type A. This has a straightforward interpretation in terms of information flow. If $A \leq B$ then if it is acceptable to write B to a particular channel then it is also acceptable for us to write A to that same output channel. So, we would expect that $public \leq secret$.

All of the judgements which CQUAL [1] enforces are relevant and needed for information flow but there were a few subtypting rules for control flow which needed to be added. For an if statement, any variable assigned to on the if or else arm needed to be a subtype of anything used as an r-value in the condition. Loops were similar, everything assigned to in the body needed to be made a subtype of everything used as an r-value in the condition statement.

5 Policy Specification

There are two parts of policy specification corresponding to the dynamic and static mechanisms we describe.

5.1 Static Portions of Policies

Static policies conservatively describe the set of dangerous flows. We are conservative because there are many statically identifiable operations which may carry out dangerous actions but often do not.

In general, a static policy consists on user-defined constraints on system calls. In Linux we would annotate `write()`, `read()`, `close()`, `open()`, etc. System calls define both the ways that sensitive data may enter our system and the ways that sensitive data may leave the system. We use the notation of CQUAL to denote qualifier constants (they are preceded by a ´$) and show an example policy using three of the four system calls mentioned above.

- `$secret int open(const char *pathname, int flags);`
- `int read($_1 int fd, $_1 void *buff, size_t size);`

- `int write(int fd, const $public void* buff, size_t size);`

This policy reflects the flow of information from potentially sensitive information sources (the file named in the 'pathname' argument to open) to potentially dangerous output channels. The policy on read states that if we have a $public or $secret file descriptor we will get back data that has the same classification. The policy on `write` reflects our pessimism about where the data will go.

Such a policy on the three essential system calls leads to the detection of nearly all flows, given access to the full source code of a program. Consider the following program:

```
#include <fcntl.h>

int main(int argc, char **argv)
{
char buff[20];
int fd1 = open("~/my_data", O_RDONLY),
            fd2 = open(atoi(argv[1]), O_WRONLY);
        read (fd1, buff, 20);
write(fd2, buff, 20);

return 0;
}
```

The program obviously has a potential information leak, and the system we have put in place detects it. The error we get from CQUAL is:

```
buff[]: $public $secret
../test_programs/preludes/prelude.cq:230    $secret == open_ret
                       progs/test1.c:6              == fd1
                       progs/test1.c:8              == read_arg1@8
                       progs/test1.c:8              == buff[]
                       progs/test1.c:9              == *write_arg2
../test_programs/preludes/prelude.cq:245            == $public
```

In CQUAL functions have polymorphic types. In terms of constraint generation, this is modeled by giving each function call distinct qualifier variables (for the function return, and its arguments). The decorations on the arguments and return types denote relationships between the qualifiers we wish to enforce at all function call sites. For example, `open` always returns a file descriptor that is labeled with $secret, for all instantiations. The notation $_1_2 denotes a set of variables, in this case 1 and 2. Let $set(A)$ denote the set of elements which decorate a qualified type A. Then if $set(B) \subseteq set(A)$ CQUAL enforces the restriction that $B \leq A$.

CQUAL uses prelude files, where the qualified types of functions may be given statically. If we specify the type of a function in a prelude file then the analysis of the internals of the function are suppressed and the type given in

the prelude file is assumed to be correct. This is useful in the case where we wish to take into account more semantic information than can be encoded in the subtyping system. For example, suppose we defined a version of `open` which sanitized file accesses and only returned pointers to files which were deemed safe. Then we could take into account these semantics and give our new function a type consistent with its semantics; namely it would return a pointer qualified by $public.

5.2 Dynamic Portions of Policies

The policies enforced by RIFLE are a superset of those enforceable by static mechanisms because the ability to view what is actually happening during a computation is so powerful. We propose to imitate the dynamic portion of RIFLE but only do so during the security relevant portions of the program. Each potentially dangerous information flow is monitored at runtime with inserted instrumentation and if it appears that sensitive information is going to leave on an inappropriate channel, the program is halted, as occurs in RIFLE. We have the user specify a function for each security-relevant system call, the name and purpose of which depend on how the function relates to the security policy, i.e. if it is a channel whereby secret information enters the system or a channel where secret information leaves the system. Here are the functions we would need for the policy we have specified above. Each function returns true in the case that a violation has occurred. The functions have access to runtime security relevant information about the sensitive data.

- `bool check_open_return(const char *pathname, int flags);`
- `bool check_write(int fd, void *buff, size_t size);`

The `check_open` function is entirely written by the user and should take in a pathname and a set of flags and tell us whether or not the file descriptor returned by `write` represents a security risk. If it does, then we add the fact that the file descriptor returned by `write` is $secret and that any use of it in a $public context should be flagged as an error. The `check_write` function checks if we are writing $secret data to a $public channel and if we are, returns an error. This could actually be done by the system.

6 Runtime Monitoring of Flows

Every sequence of program actions resulting in a security violation occurs because of some sequence of program actions. The static portion of the policy allows us to identify the sequence of actions responsible for a particular violation.

If we were to implement RIFLE in software the thing that would make it completely impractical is the overhead required to give each memory address a security type. The important thing about identifying dangerous flows is that

we only need to monitor those instructions involved with the flows. For this approach to make sense it is important that flows are not too long and involved or we may end up implementing RIFLE in software.

Consider this version of the above program:

```
#include <fcntl.h>

int main(int argc, char **argv)
{
char buff[20];

open_ret_check("~/my_data", O_RDONLY, "fd1");
int fd1 = open("~/my_data", O_RDONLY)
open_ret_check(atoi(argv[1]), O_WRONLY, "fd2");
int fd2 = open(atoi(argv[1]), O_WRONLY);
read_arg2_check(fd1, buff, 20, "buff");
        read (fd1, buff, 20);
write_arg2_check(fd2, buff, 20, "fd2");
write(fd2, buff, 20);
return 0;
}
```

Every step of the dangerous path found above has been checked, and all decisions about policy can be encapsulated in checks of the arguments to read and write. Other statements that propogate information may be checked similarly, and casts will be dealt with in a sound manner, according to the policy in place. The statements we would need to track assigments and control flow could be inserted entirely independent of policy. The user would have no need be involved himself. It is my hope, although I have only a tiny bit of data to show it, that there are classes of applications for which such a policy will incur too much overhead. This may only be settled with experience on a wide range of applications.

7 Proposed Implementation of Runtime Monitoring

7.1 Architecture

CQUAL generates constraints and can be easily configured to print out the constraints as it goes along. Each type variable corresponds to a declaration of a data type, except for those associated with function calls. These are generated implicitly each time a function is encountered. The first step is to generate all constraints. When we generate the constraints we would dump information about the location in the source file we are generating the constraint so we may instrument the source file later.

At this point we do not have any constants involved yet. The constraints we have generated describe the set of all possible information flows throughout the program. The next step is to specify the set of flows that are interesting, by using constants to constrain the types of system call return values and arguments through greater-than and less-than relations. If we are considering a hybrid dynamic/static approach, it really only makes sense to have two levels: safe and potentially dangerous. These levels will define the sets of interesting flows that we wish to monitor.

The constraints, which now specify types for the system calls specified in our arguments are solved using IQUALS, an interface to the CQUAL constraint solver. If we deduce that all type qualifiers are either $public or $secret but not both, then there are no security violations. Otherwise, we must, for all type qualifiers that have been deduced to have two inconsistent types, add statements which check the runtime type of the qualifiers for all inconsistent derivations.

The overhead involved in such instrumentation is related to the amount of code involved in the total number of program statements in statements on sensitive paths. There are other concerns as well; if we instrument extremely well-trodden portions of the control flow graph then performance will suffer a great deal.

7.2 Memory Safety

In the absence of memory safety all bets are off. Any array access is potentially an alias of every other array. This is the major weakness of such an approach for a language like C. We could envision using separate heaps, one for data of each security level and then sandboxing accesses to these heaps. All reads from $secret heaps would taint the variables that received such values. However, the more serious problem that there would be no such thing as a local variable with a classified type would remain. It seems that this approach would not work well without array bounds checks and so forth.

8 thttpd

`thttpd` is a tiny webserver. The information leaks in such a program are obvious and flows were found without too much effort. In terms of static policies, it is clear that programs like are guaranteed to leak information, that is their purpose. Any webserver must interact with the filesystem using system calls. These system calls have a restrictive type, indicating that subject to the users judgement, sensitive data may be leaked. The files users have requested are then sent out over the network. Two example information flows found were the following:

```
prelude.cq:42     $secret == *fread_arg1
            libhttpd.c:863            == buf[]
            libhttpd.c:867            == *str
            libhttpd.c:556            == *memmove_arg2@556
```

```
../preludes/prelude.cq:94              == *memmove_ret@556
            libhttpd.c:556             == *hc->response
              thttpd.c:1737            == *iv[].iov_base
              thttpd.c:1739            == *hc->file_address
              thttpd.c:1727            == *write_arg2
../preludes/prelude.cq:239             == $public

../preludes/prelude.cq:39     $secret == *fgets_ret@1016
              thttpd.c:1016            == line[]
              thttpd.c:1023            == *cp
              thttpd.c:1024            == *strspn_arg1
            libhttpd.c:3274            == *cp
            libhttpd.c:3265            == *headers
            libhttpd.c:3304            == *buf
            libhttpd.c:4213            == *cast649
            libhttpd.c:4213            == *write_arg2 == $public
```

The lines from the preludes directory are lines that we have specified ourselves, either as part of the policy, or part of the approximation of library files for which we do not have source code. The above appears to be a genuine leak of information, that a user might wish to impose policy decisions on. The above error is typical of the errors found: a file is read, and after some transformations, sent out over the network, which is exactly what one would expect.

The use of polymorphic functions is essential. Without them, it is impossible to specify any kind of reasonable policy. `thttpd` is a small program, but the constraints generated for it took up 1.75 KB. It will be interesting to see how this number increases as the size of programs increase. I was able to ascertain that 5 truly dangerous flows were reported (probably more a sign of poor annotations than a lack of flows). This also might be a consequence of the structure of the program, there are actually very few channels leading to the network. The above flow can be annotated easily in the manner we have described above. Overall, the annotations to secure `thttpd` as described should take no more than 40 or so function calls. If we include the checks that would be included had we included the spurious annotations we would need more like 120.

9 Future Work

The next step would be to validate these results for a wider class of applications and actually insert the needed instrumentation, assessing the penalty one would pay for runtime enforcement of the policies. Hopefully the set of flows is small enough that the runtime cost is not prohibitive. It also seems possible to do the same thing on binaries, since if we only consider system calls it may be possible to map high level operations like file accesses to memory locations. The main difficulty here would be the need for some kind of memory safety. Without having at least an estimate of the type of an object we would not even know what the simplifying assumption should be.

References

[1] Jeffrey Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, September 2002.

[2] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manishi Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information flow security. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO)*, December 2004.