Wooster: Mandatory Access Control for Rails

Julian Bangert, Alexander Lin, Julia Huang

Abstract

We introduce Wooster, a system for enforcing flexible mandatory access control policies in Ruby on Rails' ActiveRecord database wrapper. Currently, Rails applications maintain access control in the controllers and views, requiring programmers to implement access control checking in many locations throughout their code. Wooster allows programmers to specify a single policy that controls access to all data in a web application, removing this burden and leading to more comprehensive policies.

1. Introduction and Motivation

Access control is a major security threat and even slightly improper implementation can cause enormous security leaks. For example, we discussed in class a flaw in Citibank's website that allowed users to view each other's balances by changing the user id in the URL. Furthermore, these bugs can be rather subtle and thus hard to detect; three of the OWASP Top Ten vulnerability classes (Insecure Direct References, Sensitive Data Exposure and Missing Function Level Access Control) are related to improper access control.

As we will discuss, existing access control solutions for Ruby on Rails provide only incomplete protection. Our mandatory access control scheme provides complete coverage without requiring extensive developer work by transparently performing access control checks whenever data is accessed.

2. Design

Wooster consists of two components: a domain-specific language to express policies and a reference monitor that hooks ActiveRecord to enforce this policy. Programmers write a Wooster policy for all data models in their application in one file and include the Wooster gem, which will automatically load the reference monitor.

Our policy allows access control entries for both entire records and individual fields within that record. For records, reading, writing, creating and deleting are covered by individual rules, and individual fields support different rules for reading and writing. For simplicity and efficiency, these policies are only enforced at a database layer - code may modify fields and create records at will, but Wooster will guarantee that any attempt to save invalid changes will result in an error.

In order to access a field in a record, the policy checks for both the record and the field must succeed. Because we expect most applications to initially only use record-level security, the default policy for fields is to allow access, whereas the entire record is denied by default. Therefore, while there is no access by default, a programmer needs to write just a single rule to allow access to all columns.

As opposed to traditional ACL rules, which just match on an ‘actor’ - e.g. the logged-in user - our rules match on arbitrary ruby expressions, which allows very flexible policies, such as granting access to any photos posted by a friend in a social network setting.

An example policy for a users table follows:

```
   permissions Client {
1    read (client)-> {current_user.admin or
2                     client == current_user.client}
3    is_admin = (x)->{current_user.admin}
4    employed_by_client = (x)->{x == current_user.client}
5    create is_admin
6    write any(is_admin, employed_by_client)
7    field_readwrite :taxnumber, employed_by_client
8    field_read :taxnumber, (x)->{[false, x.taxnumber.anonymize]}
   }
```

The first line within the block specifies what conditions have to be met in order to see the Client record - the user either has to be an administrator, or the current user has to belong to this client. Conditions can also be stored in variables (line 2 and 3). We use this to implement a form of role-based access control (RBAC), and provide various convenience helpers to compose conditions, such as `any`, `all`, `deny` and `allow`. These helpers are used in line 5 to repeat the same condition as in line 1 more concisely. As shown in the full policy in Appendix 1, read and write statements can be specified together in one line (`record :read, :write, any(is_admin, employed_by_client)`).

Lines 7 and 8 show rules governing access to a single sensitive fields. Wooster allows the policy to create a substitute value to be shown to unprivileged users instead of a sensitive value. Currently, checks for fields can either return true (in which case the access is granted) or false and an optional fake value (which will be exposed to the model). When no fake value is returned, the default value for that column is inserted by Wooster. Different contexts can therefore receive different anonymization results. When more than one rule for an action is specified, at least one of the checks has to succeed for the access to be allowed.

We implemented Wooster by hooking ActiveRecord. This enforced the policy transparently without changing any application code. We use the provided hooks to inspect updates, creates and deletes before they are committed to the database. We also insert a after_find hook that applies the field checks, transparently replacing the values in the model. Finally, to support restrictions on queries, we override the find_by_sql method which all ActiveRecord helpers ultimately invoke.

3. Use Cases

We tested Wooster by implementing a policy for RailsGoat.

RailsGoat is a Rails 3 application maintained by OWASP which demonstrates web vulnerabilities, among them many of the common access control problems endemic to rails applications. It demonstrates each of OWASP's Top 10 most common web vulnerabilities; as mentioned before, three of these ten issues are related to access control.

We added the Wooster gem and wrote the 35 line policy shown in Appendix 1, which successfully prevented the (intentional) vulnerabilities in RailsGoat without any application changes.

4. Prior Work

Traditionally, Rails programmers had to manually check for permissions on every request, e.g. with before_filter. Recently, a few libraries for providing access control, such as restful_acl and cancan, have emerged.

Restful ACL (https://github.com/mdarby/restful_acl/) allows the programmer to specify access control lists on individual actions. Any allowed page could however still leak sensitive information; for example, a page could query other database objects based on a form parameter and thus potentially violate access control policies.

CanCan (https://github.com/ryanb/cancan) provides a very expressive DSL for writing policies and provides convenient shorthand for checking whether a certain access is ok.

A policy might state “if user.admin? can :destroy, :post end”, but it is the programmers responsibility to check that a given user is indeed an administrator before allowing users to destroy posts. This both requires programmers to rewrite significant amounts of their code and potentially creates holes in an application if a developer forgets to add a check to any of the myriad pages that might require one.

Our work was also inspired by Jeeves (https://projects.csail.mit.edu/jeeves/about.php), a programming language developed at MIT for providing anonymity through faceted execution. However, we found faceted execution hard to implement in an imperative language like Ruby. The full expressive power of faceted execution is also not necessary in a web application, where each request goes only to one destination and is relatively short lived. Therefore, we can implement policies similar to Jeeves with Wooster.

Appendix 1: Access Control Policy for RailsGoat

```
Wooster::Policy.build  do
  admins = ->(x){ current_user.admin }
  current = ->(x){ current_user.user_id == x.user_id}
  admins_and_current = ->(x){ current_user.admin or x.user_id ==
current_user.user_id}

  permissions User do
    read allow
    write admins_and_current
    delete admins
    create admins
    field_write :admin, admins
  end
  permissions WorkInfo do
    record admins_and_current
    field_readwrite :ssn, current
  end
  record Schedule,admins_and_current
  record Retirement,admins_and_current
  record Performance, admins_and_current
  record Pay, admins_and_current
  record PaidTimeOff,admins_and_current
  record KeyManagement, admins_and_current
  permissions Analytics do
    create allow
    read admins
  end
  permissions Message do
    receiver = ->(x) {current_user.user_id == x.receiver_id}
    sender = ->(x) {current_user.user_id == x.creator_id}
    read(any receiver, sender)
    create sender
    delete receiver
  end
end
```