Rule-based access control

Improve security and make programming easier with an authorization framework

Barry Brachman

Although Web servers can perform user authentication and coarse-grained authorization checking for applications, developers of Web services and service-oriented architectures (SOAs) often must write custom code to restrict access to certain features of their system, or customize the behavior or appearance, based on the identity of a user. Embedding authorization checking within an application is inflexible, prone to error, and increases its complexity. What if authorization checking were data-driven instead of implemented by program logic? By reusing an authorization framework, scripts and compiled programs can be smaller, simpler, and more secure, and application development time and effort can be reduced.

Introduction

You are probably familiar with access control functionality provided by an operating system. It typically performs authorization checks on system calls and operation requests on resources named by a file system. Programmers don't usually have to test whether the user running the program is entitled to read a data file or set the system time. The operating system takes care of it and lets the program know through a return value whether the operation is granted or denied. For example, UNIX® enforces the requirements established by a file's owner ID, group ID, and permissions relating to the real or effective identity of the user. Although this design is simple, it's can be inadequate, leading some UNIX-type systems to extend it through access control lists and other mechanisms. Other operating systems use more elaborate security models. Still, the security model usually applies to file objects and user accounts known to the system (those that appear in / etc/passwd or NIS).

For some server-based applications this model is adequate. The server initially executes with enhanced privileges so it can perform any operation. When acting on behalf of a particular user, it effectively becomes that user. Alternatively, the server can use system calls to build a security model that restricts operations to those permitted to that user. Before executing that operation, the program must test for authorization: is the current user allowed to do the operation?
Authorization checking for Web services

Programs must sometimes enforce their own access control requirements on resources they are responsible for and on user accounts not recognized by the operating system. A prime example is the Apache Web server, which can be configured to grant or deny an HTTP request for one of its resources. Entries in the private password file create Web server accounts that are totally separate from those known to the operating system. Likewise, group membership lists are created specifically for use by the Web server. An Apache administrator can use these names with the server’s Require, Allow, and Deny directives to describe who can (or cannot) access a resource. Because the operating system is of little help in this respect, Apache programmers have implemented their own authorization subsystem.

Although the user list for the Web server and the operating system could be unified, usually they are kept separate for security, performance, and implementation purposes.

The kind of authorization provided by a Web server like Apache might be called coarse-grained access control because it provides only an outer layer of security. The outcome of the authorization check determines whether the request should be granted or not. If access is denied to a Web service, for instance, the Web service is not executed by the Web server, and therefore the request is never even seen by the Web service.

However, many server-based applications, including those that are Web based, need authorization-testing capabilities that go far beyond this. Consider a Web service application executable by anybody but that recognizes that users have different capabilities or privileges. Perhaps the simplest example is an application in which only certain users can access administrative functions -- all other users not only can’t execute these functions, they should not even be able see them (they should not be listed in a menu, or at least not be selectable). Another common example is an application with profiles for each user that only its owner and perhaps the application’s administrator can modify. Often an application must restrict access to its data by particular users. This type of authorization testing might be called fine-grained access control because it is applied by a running program to virtually any type of resource that the program works with.

In some cases, a creative programmer might apply some clever tricks to work around these problems. For instance, an application might use URIs to name its resources, configure a Web server’s access controls of those URIs, and then issue an HTTP request to determine if a user is authorized. This approach would leverage the Web server’s authorization-checking mechanisms, but would not be practical.

Most programming languages, including Perl, do not help much in this regard. They supply a simple layer on top of what the underlying system provides. The Java™ language includes an authorization framework, but of course that’s no help to the Perl or C/C++ programmer.

The challenge to the programmer is two-fold: First, users of these applications don’t need to have an account on the underlying system and second, the types of objects and how they are accessed can be quite different from what the operating system’s security model was designed
for. Consequently, programmers are forced to write a lot of code to support application-specific authorization testing, sometimes having to reimplement essentially the same framework in different languages. Database systems, such as Oracle and MySQL, manage their own user accounts, roles, and privileges for system and object-level operations.

**Fine-grained authorization checking**

Fine-grained access control goes beyond granting or denying the right to execute a program or read a data file. Consider a Wiki server, implemented as a CGI program. It maintains a collection of Web pages, one per user. Anyone has read-access to any Web page. Any user can add content to or manage their own page, but not others' pages. Any user designated as an administrator can update and manage any page (for example, to remove offensive or illegal content). When displaying a Web page to an owner or an administrator, all operations should be selectable by a menu or links; others should not see those operations or they should not be selectable. Furthermore, the application must ensure that anyone can see a Web page, but only its owner or an administrator can execute a restricted operation, such as updating the page's content.

The current practice is to embed authorization logic in the application. At appropriate places throughout the code, the programmer writes program logic to test whether the current user is authorized to perform an operation.

When constructing a menu, code similar to Listing 1 might appear.

**Listing 1. Building a menu of allowed operations -- the incorrect way**

```c
for (op = 0; op < n_operations; op++) {
    if (op == CREATE_OP && is_admin(current_user))
        add_operation_to_menu(menu, op);
    else if ((op == UPDATE_OP || op == DELETE_OP)
        && (is_owner(current_page, current_user) || is_admin(current_user)))
        add_operation_to_menu(menu, op);
    else
        add_operation_to_menu(menu, op);
}
```

Although at first glance Listing 1 might appear to be correct, it's buggy. That code made it past several revisions of this document before the problem was spotted, showing how easy it is to make careless mistakes when writing this type of test. Listing 2 is correct:

**Listing 2. Building a menu of allowed operations -- the correct way**

```c
for (op = 0; op < n_operations; op++) {
    if (op == CREATE_OP && is_admin(current_user))
        add_operation_to_menu(menu, op);
    else if ((op == UPDATE_OP || op == DELETE_OP)
        && (is_owner(current_page, current_user) || is_admin(current_user)))
        add_operation_to_menu(menu, op);
    else if (op != CREATE_OP && op != UPDATE_OP && op != DELETE_OP)
        add_operation_to_menu(menu, op);
}
```

At a different point in the code, perhaps where operations are dispatched or at the start of each operation, a similar test is made.
Listing 3. Testing for restricted operations

```java
if (op == CREATE_OP && !is_admin(current_user))
    raise_exception("Operation not permitted")
else if ((op == UPDATE_OP || op == DELETE_OP) && !(is_owner(current_page, current_user) || is_admin(current_user)))
    raise_exception("Operation not permitted")
```

The authorization checks that the two code fragments performed are closely related, though they might appear in different parts of the program. In more sophisticated cases, the authorization tests could be considerably more complicated. For example, if the application is later enhanced to allow a group member (defined by the page's owner) to perform previously restricted operations, the programmer will need to locate, review, and possibly modify all the code fragments where these checks are made (obviously there could be many more than two). Failure to make the appropriate changes could result in accidental or malicious misuse of the application.

**Rule-based authorization checking**

To help ease the programmer's burden and improve application security, we propose using a framework specifically designed for authorization checking. Just as programmers use libraries of math, cryptographic, networking, and database functions instead of reimplementing routines every time, why not leverage a package that provides most of the solution to problems that are only incidentally related to the task at hand?

We suggest a rule-based (data-driven) authorization framework that is capable of the following:

- Is callable from the command line so it can be invoked by virtually any script or executed by any program
- Provides a simple C-language API so it can be called directly and integrated as an extension to many scripting languages
- Uses a generic user-naming syntax so it can interoperate with most authentication methods
- Supports rules that can be applied to any resource
- Expresses access-control rules as simple XML documents
- Supplies variables, expressions, and control flow to facilitate more complicated decision making
- Includes a rich set of functions to test access control requirements, such as the user's IP address, time and date, or whether the user's name appears in a given list

Besides the obvious advantage of not having to implement functions provided by the framework, there are a host of others:

- The rules used by an application can be changed by anyone with permission, without changing or even recompiling the application. Administration of security policies can be delegated to a nonprogrammer or to someone not familiar with an application's implementation language.
- If a rule is modified, all occurrences throughout an application or suite of related applications automatically use the revised rule -- no changes to the code are necessary and all occurrences are synchronized.
• Because rules are independent of any programming language, the same rules can be used by different applications.
• Improvements and bug fixes applied to the framework can benefit applications that use it -- without modifications or recompiling.

Of course there are some disadvantages. Apart from the learning overhead, a bug in the framework may be inherited by all applications that use it. Authorization checking performed in this way will be somewhat slower than inline code, particularly when invoked through a separate process.

**DACScheck: an authorization framework**

It would be nice if these observations drove the design and implementation of dacscheck, but that is not its genesis. Rather, a feature-rich replacement for the Apache authorization module lead to the development of **DACS**, a lightweight single sign-on system. The realization that the authorization framework could be generalized and adopted by nearly any application, Web-based or not, came much later.

To illustrate how to use dacscheck, let's revisit the pseudo-code that builds a menu, presented in Listing 2, this time in Perl.

**Listing 4. Using DACScheck from Perl**

```perl
use DACScheck.pm;

# Tell dacscheck which rules to use
dacscheck_rules("/usr/local/wiki_app/acls");

# Build a list of operations for the menu.
for ($op = 0; $op < $n_operations; $op++) {
    # The object of interest, which is used to locate the appropriate rule,
    # is arbitrarily named "/<username>/menu".
    my $object = "/$ENV{'DOCUMENT_ROOT'}/menu";
    my $result = dacscheck_cgi($object);
    # Access to the object is granted only if the returned value is one.
    if ($result == 1) {
        add_operation_to_menu($menu, $op);
    }
}
```

The Perl module DACScheck.pm provides a simplified interface to dacscheck that includes dacscheck_cgi(). Besides naming the object, the function tells dacscheck that it's running within a CGI context and should use the value of **REMOTE_USER** as the identity of the user making the request. Apache automatically sets **REMOTE_USER** if the user has been authenticated.

Each resource managed by dacscheck has a corresponding rule. Although rules can be stored and accessed various ways, each rule is typically stored in an ordinary text file, and rules for related resources are usually kept under a common root directory. A rule is expressed as an XML document that can include free-formatted, C-like expressions. Expressions can reference variables that are instantiated from the execution context, including environment variables and Web service arguments, and a wide variety of built-in functions oriented around string manipulation and high-
level identity tests. An existing extension language, such as Tool Command Language (Tcl), was avoided to keep this capability uncomplicated. A more complicated test can be performed through a separate program, even one invoked via HTTP. A rule can also point to another rule, a feature that allows an administrator to delegate responsibility for rules.

Let's now take a look at what a rule might look like. The rule tells dacscheck how to decide whether the request should be granted or denied. We assume that the user has already logged in, and therefore $REMOTE_USER$ is set. We also assume that $OP$ is an argument that selects an operation, and that $PAGE$ is an argument that identifies the Wiki page's owner.

**Listing 5. Example dacscheck rule file**

```xml
<acl_rule>
  <services>
    <service url_expr="/${Args::PAGE}/menu/">
  </services>

  <rule order="allow,deny">
    <precondition>
      <predicate>
        ${Args::OP} eq "CREATE_OP"
      </predicate>
    </precondition>
    <allow>
      dacs_admin()
    </allow>
  </rule>

  <rule order="allow,deny">
    <precondition>
      <predicate>
        ${Args::OP} eq "UPDATE_OP" or ${Args::OP} eq "DELETE_OP"
      </predicate>
    </precondition>
    <allow>
      dacs_admin() or ${Args::PAGE} eq ${Env::REMOTE_USER}
    </allow>
  </rule>

  <rule order="allow,deny">
    <allow>
      user("any")
    </allow>
  </rule>
</acl_rule>
```

This rule is verbose to make it easier to explain, but also because it expresses the security policy in such a way that, once you're familiar with the syntax, it's easy to understand and change.

The **service** element tells dacscheck which resource the rule applies to. The argument to dacscheck_cgi() is matched against this string.

This access control rule consists of three **rule** elements: the first is selected only if the $OP$ argument is $CREATE_OP$; the second is selected only if the OP argument is $UPDATE_OP$ or $DELETE_OP$; and the third is selected only if neither of the other two are. The **order** attribute serves the same
purpose as Apache's `order` directive: it establishes the default behavior and the order in which `allow` and `deny` elements are evaluated.

In the example, the first rule grants access only if the `dacs_admin()` function returns "True". That function looks for the user's identity. The second rule grants access to an administrator and the owner of the Wiki page. The third rule grants access to anyone, since a restricted operation has not been requested.

Should the access control policy for the Wiki need to be changed, you only need to change a rule. The effect is immediate and applies to all references to the rule. If an administrator needed to deny access to a page for requests coming from a particular IP address range, something like the following could be added:

**Listing 6. An additional rule clause to deny access**

```xml
<rule order="allow,deny">
  <precondition>
    <predicate>
      from("10.0.0/24")
    </predicate>
  </precondition>
</rule>
```

The `allow,deny` ordering denies access by default, so any request coming from an IP address in the range will be denied.

There are many situations when this form of authorization checking might be applied. The ftp daemon (`ftpd`), for example, could benefit from this capability. It could be extended to employ rules to grant or deny operations (`put`, `get`, `cd`, and so on) based on a user's identity or contextual information that can be tested by rules. For example, uploads might be allowed only during certain hours, or access to a directory might be restricted to specific users or users connecting from certain IP addresses.

**Conclusion**

A flexible authorization framework can increase security and do much of the "heavy lifting" for programmers. Scripts and compiled programs can be smaller and simpler, and application development time and effort can be reduced. While there are still some loose ends, `dacscheck` has already proven to be a useful tool. Although little attention has been directed to its efficiency, in typical use `dacscheck` has not impacted performance significantly.

dacscheck has a sister program, `dacsauth`, that serves much the same purpose for authentication as `dacscheck` does for authorization. It lets programmers leverage existing authentication methods instead of having to reimplement them each time a new application needs its own list of users.

Another related program is `dacstransform`, a utility for rule-based document transformation. Using the same rules as `dacscheck`, it can redact, insert, and replace text in a marked-up document. Each transformation depends on a rule that's evaluated at run time, allowing a new document to be generated from the marked-up one that is tailored for a specific user or context.
Related topics

- Take a look at the manual page for `dacscheck(1)`.
- Read about Authentication, Authorization and Access Control for the Apache HTTP Server.
- Learn all about CGI from RFC 3875: The Common Gateway Interface (CGI) Version 1.1.
- The IBM SOA Web site offers an overview of SOA and how IBM can help you to get there.