Securing dynamic Web content

Tom Syroid (dwcomments@syroidmanor.com)
Contract writer
September 2002

This article details how to secure dynamic content on an Apache Web server. Topics covered include general security issues pertaining to dynamic content, securing Server Side Includes, configuring Apache's Common Gateway Interface, and *wrappering* dynamic content. The article is targeted primarily at Webmasters and system administrators responsible for maintaining and securing a Web server; however, anyone with a need or desire to server dynamic content will benefit from the topics covered. A basic understanding of Linux commands, permissions, and file structures is assumed.

Introduction

Once upon a time, the World Wide Web was a relatively static place. The Web server's sole function was to simply deliver a requested Web page, written in HTML, to a client browser. Over time, developers started looking for ways to interact with users by providing *dynamic* content -- that is, content that displayed a form or executed a script based on user input. Thus Server Side Includes (SSI) and the Common Gateway Interface (CGI) were born.

A Server Side Include page is typically an HTML page with embedded command(s) that are executed by the Web server. An SSI page is *parsed* by the server (a "normal" Web page is not), and if SSI commands are found they are executed before the resultant output is delivered to the requesting client. SSI is used in situations that demand a small amount of dynamic content be inserted in a page, such as a copyright notice or the date. SSI can also be used to call a CGI script; however, there is a performance penalty associated with SSI. The server must parse every page designated as SSI-enabled, which is not an optimal solution on a heavily loaded Web server.

The CGI is a standard for communication between a program or script, written in any one of several languages, and a Web server. The CGI specification is very simple: input from a client is passed to the program or script on `STDIN` (standard input). The program then takes that information, processes it, and returns the result on `STDOUT` (standard output) to the Web server. The Web server combines this output with the requested page and returns it to the client as HTML. CGI applications do not force the server to parse every requested page; only pages containing CGI-recognized arguments involve further processing.

This article is targeted at Webmasters and system administers responsible for securing a Web server configured to provide dynamic content to clients. It details general security issues related to SSI- and CGI-enabled content, reducing CGI risks with wrappers, and some brief language-specific caveats. This article *does not* cover the configuration steps required to enable SSI or CGI. Configuration examples provided are based on the latest 1.3 version of Apache, which at the time of this writing is 1.3.26. In addition, the following is assumed:

- Your network is secure, behind a firewall, and the server itself is in a controlled environment.
- The operating system has been properly secured and all unnecessary services are disabled.
- The Apache `user` and `group` directives are correctly set, and appropriate permissions assigned.
- The ServerRoot and log directories are protected.
- User overrides are disabled.
- Default access has been disabled, and access opened for only those system directories designated "public". For example, on a system configured to host user Web pages from */home/username/public_html*, Apache's `httpd.conf` configuration file should contain the following directives:

```
ServerName www.sitename.com
UserDir public_html

<Directory />
        Order deny,allow
        Deny from all
</Directory>

<Directory /home/*/public_html>
        Order deny,allow
        Allow from all
</Directory>
```

In other words, in order to fully absorb the material discussed in this article, the reader should have a good working knowledge of general Web server security, installing and configuring Apache, Apache modules, Apache's key configuration directives, the role of Apache's `.htaccess` file, how to read log files, UNIX file permissions, and basic system administration. Readers should also be familiar with the syntax, commands, and functions of whatever programming language they intend to use to create CGI applications.

See the Resources section of this article for a list of useful online resources.

General considerations

The very first question a Web server administrator must confront is, "Do I really want/need to provide dynamic content from my server?" While dynamic content has allowed for a diverse range of user interaction and become a de facto standard for most large Web sites, it remains one of the largest security threats on the Internet. CGI applications and SSI-enabled pages are not inherently insecure, but poorly written code can potentially open up dangerous back doors and gaping holes on what would otherwise be a well-secured system.

The following are the three most common security risks CGI applications and SSI pages create:

- Information leaks: Providing any kind of system information to a hacker could potentially provide a hacker with the ammunition they need to break into your server. The less a hacker knows about the configuration of a system, the harder it is to break into.
- Access to potentially dangerous system commands/applications: One of the most common exploits used by hackers is to "take over" a service running on the server and use it for their own purposes. For example, gaining access to a mail application via an HTML form-based script, and then harnessing the mail server to send out spam or acquire confidential user information.
- Depleting system resources: While not a direct security threat per se, a poorly written CGI application can use up a system's available resources to the point where it becomes almost completely unresponsive.

A glance at the above list shows that a high percentage of security holes is invoked or leveraged through user input. One of the most common problems with applications written in C and C++ are *buffer overflows*. When a program overflows a buffer, it crashes. A good hacker can then take advantage of the crashed program to gain access to the system. For example, look at the following snippet of C code:

```
#include <stdio.h>
#include <stdlib.h>
static char query_string[1024]

char* POST() {
int size;
size=atoi(getenv("CONTENT_LENGTH"));
fread(query_string, size,1,stdin);
return query_string;
}
```

An assumption is made that user input will be a maximum of 1024 characters in length. If the user supplies input of more than 1024 characters the above routine will break the program and allow someone to execute a system command remotely. The solution in

this case is to ensure that memory allocation for the variable `query_string` occurs dynamically by using a call to either the `malloc()` or `calloc()` function.

Another common problem involves a system call that opens a subshell to process a command. In Perl such a call could be made using any of the following functions: `system(), exec(), piped(), open(),` or `eval()`. The lesson here is to never trust user input, and ensure all your system calls are not exploitable. The first is typically achieved by establishing explicit rules (for example, by checking input with a regular expression) for what is acceptable and what is not. The process of sanitizing system calls is language-dependent. The trick is to always call external programs directly rather than going through a shell. Using Perl, this is accomplished by passing arguments to the external program as separate elements in a list rather than in one long string like so:

```
system "/usr/bin/sort","data.dat"
```

A related trick used by many hackers is to alter the PATH environment variable so it points to the program they want your script to execute, instead of the program you're expecting. This exploit can be easily subverted by invoking any programs called using full pathnames. If you have to rely on the PATH variable, get in the habit of explicitly setting it yourself at the beginning of the application. For example:

```
$ENV{'PATH'}="bin:/usr/bin:/usr/local/bin";
```

A denial of service (DOS) attack occurs when an attacker makes repeated calls to to one or more CGI applications. The Web server dutifully launches a CGI process and a child server process for each call. Eventually, if enough calls are made, the server runs out of system resources and comes to a grinding halt. Unfortunately, there's not a lot you can do to prevent a DOS attack beyond banning the host access to the server using Apache's `<Limit ...>` directive. You might also want to look into the `RLimitCPU` and `RLimitMEM` directives which limit Apache's CPU and memory usage respectively.

Later in this article we'll discuss using a *wrapper* to limit the danger inherent in running CGI applications. The next two sections of this article detail potential risks and solutions specific to Server Side Includes and CGI applications.

Securing server side includes

Many Web administrators consider Server Side Includes (SSI) on a par with CGI applications when it comes to potential security risks. As noted in the previous section, *any* program or page that uses the `exec` command to call a system file presents a huge security problem if the call is made incorrectly. On the other hand, it's a remarkably simple process to disable all exec calls from an entire Web site, or allow exec calls to be made from a specific directory only. This is accomplished with the `Options` directive.

```
<Directory>
Options IncludesNOEXEC
Order deny,allow
Deny from all
</Directory>
```

The `Options` line in the above configuration listing disables exec calls and includes for the Web site. To enable exec calls for a specific subdirectory of the Web, "scope-down" the directory container like so:

```
<Directory />
Options IncludesNOEXEC
Order deny,allow
Deny from all
</Directory>

<Directory /subdirectory>
Options Includes # (or alternately, +Includes)
Order deny,allow
Deny from all
</Directory>
```

This configuration segment allows the execution of exec commands from only the */subdirectory* directory under the site's DocumentRoot. Note that users can still execute CGI scripts from within a document, *provided* the scripts are located in a directory designated by a `ScriptAlias` directive (see the next section on Securing CGI applications for details on using the `ScriptAlias` directive).

The second consideration administrators need to be aware of concerning SSI was briefly discussed in the Introduction. It is not a good practice to allow SSI commands to be executed from pages with an .html or .htm file extension, especially on a high-traffic server. Remember, *all* SSI pages are parsed by the server. Poorly coded pages can consume system resources at an astonishing rate, and will eventually result in an unresponsive server. To avoid such a scenario, it is common practice to use a separate extension for SSI-enabled documents (typically, .shtml). This is done by adding the following lines to Apache's configuration file:

```
AddHandler server-parsed .shtml
AddType text/html .shtml
```

The first directive tells Apache to treat all files ending in .shtml as SSI pages; the second directive is sent to the browser requesting the page, and tells it to render the content the same as it would for an HTML request.

Securing CGI applications

Administrators have two options for configuring CGI under Apache. The first method uses the `ScriptAlias` directive to designate the CGI program directory. The second method uses a combination of the `Alias` and `AddHandler` directives. Each method has a place, and its own set of pros and cons.

The ScriptAlias approach

The first step in securely configuring CGI under Apache is to create a central directory to store your CGI applications in. This directory should always be separate from the DocumentRoot tree. Why? Because the less the world knows about where your CGI scripts reside, the better. It also ensures only Web administrators can access the files that reside there. So if */www/mysite/htdocs* is your DocumentRoot, */www/mysite/cgi-bin* would be a good choice for your CGI directory. Some webmasters prefer to locate their CGI directory on another filesystem completely; for example, */var/www/cgi-bin*. The next step is to inform Apache which directory contains CGI programs. This is done with the `ScriptAlias` directive.

```
ScriptAlias /cgi-bin/ /www/mysite/cgi-bin/
```

There are several points to note regarding the above directive:

- To access the script, `test.cgi`, using the example path shown, a user would enter `http://www.mysite.com/cgi-bin/test.cgi` in their browser.
- Note that both the alias and the path to the CGI directory must end with a forward slash (/).
- Apache supports multiple `ScriptAlias` directories.
- `ScriptAlias` designated directories are not able to be browsed (by default) for security reasons.
- The directory referenced by the `ScriptAlias` directive should have very strict permission settings assigned to it. Ideally, no one but the lead CGI developer and the system administrator should have full access (read, write, execute) on the files contained there.

The last point above highlights one of the main advantages to enabling CGI using the `ScriptAlias` directive. It offers the administrator a central point to administer CGI programs from (typically, servers configured with the `ScriptAlias` directive have only one CGI directory), and allows access to the CGI directory to be tightly controlled. The disadvantage to using `ScriptAlias` to designate a CGI directory is that Apache will assume *any* executable file it finds in the aliased directory is a CGI application. In other words, Apache would see no distinction between `test.cgi`, `test.pl`, and `test.bak` provided they were all in the `ScriptAlias`ed directory, and all flagged as executable.

To overcome this last problem requires the use of the `Alias` and `AddHandler` directives.

The Alias/AddHandler approach

The `AddHandler` directive is used to specify which files are to be considered CGI programs. But first you need to tell Apache where your CGI directory is, as it resides outside the document tree.

Begin by commenting out any references to the `ScriptAlias` directive from `httpd.conf`. Next, add the `Alias` directive to the configuration file:

```
Alias /cgi-bin/ /www/mysite/cgi-bin/
```

Now you must tell Apache to execute CGI programs from this directory. This involves defining a `<Directory...>` container. This is accomplished as follows:

```
<Directory /www/mysite/cgi-bin>
Options ExecCGI
AddHandler cgi-script .cgi .pl
</Directory>
```

The first line in the above configuration segment defines the full path to the CGI directory, the second line tells Apache that CGI applications can be executed there, and the third line denotes any file within the CGI directory with the extension of `.cgi` or `.pl` is considered a CGI application.

The above configuration can be expanded to provide individual users with access to their own cgi-bin directory:

```
ServerName www.company.com
UserDir public_html

<Directory ~ "/home/[a-z]+/public_html/cgi-bin
Options ExecCGI
AddHandler cgi-script .cgi .pl
</Directory>
```

When a request arrives at the server for www.company.com/~tom, it will be redirected to */home/tom/public_html* and the index page for that directory sent to the client. In a similar manner, Apache translates any requests for www.company.com/~tom/cgi-bin/ to */home/tom/public_html/cgi-bin/* and allows any CGI program with the proper extension (`.cgi` or `.pl`) to execute. Note that the above `Directory` directive requires all usernames be lowercase. If your system allows for mixed-case or alphanumeric usernames, a different regular expression would have to be used.

Before closing out this section and moving on to the topic of wrappers, it should be noted that a third CGI configuration option is available for those who like to live dangerously: an `.htaccess` file. An `.htaccess` file provides a way for administrators to set configuration directives on a per-directory basis. To use an `.htaccess` file to enable CGI access you'd need to add the `AllowOverrides Options` statement to Apache's main server configuration section, and add `Options execCGI` to the user's `.htaccess` file. What's wrong with `.htaccess` files? One, everytime Apache encounters an `.htaccess` file it has to parse and read its contents; two, if a user gains access to his or her `.htaccess` file, they could enable additional CGI options that violate system security policies. Do not rely on `.htaccess` files to control CGI access.

Reducing CGI risks with wrappers

Perfect system security is a lofty but unattainable goal. Securing any system is a dynamic process -- checking for and applying operating system updates, program fixes and patches, scanning program revisions for desirable feature additions, reviewing user security and permissions, etc. When it comes to keeping a handle on CGI-related security issues, the very best solution is to not run any CGI applications at all. Unfortunately, such a course of action is rarely left to the same people tasked with actually securing the system. Administrators charged with maintaining a CGI-enabled server need to strike a careful balance: Users demand dynamic content capabilities, the potential danger inherent in CGI applications, and protecting systems that are typically exposed 24/7 to the "big bad world of the Internet."

Ideally, in order to run a tight ship, every CGI application exposed to the public should be thoroughly checked by the system administrator or head developer for good coding practices and potentially dangerous system calls. Unfortunately, doing so often presents a two-fold problem: One, on a busy server, it severely strains administrative resources; and two, most system administrators often do not have time to stay current on a half-dozen different programming languages *and* administer their servers.

One common solution to reducing the risks inherent in CGI applications is to employ a *wrapper* program on the Web server. A wrapper allows CGI applications to be run under the user ID of the site owner -- that is, the owner of the directories and documents that comprise a Web site. How does this increase system security? Simple. In a non-wrapper environment, CGI scripts are executed by the Apache user. This means the Apache user has to be a member of the same group as the site owner. It also means that anyone with a Web account on the server has the ability to execute a script in any other site directory on the server. Wrappering CGI applications restricts the damage a user can do to the user's files alone. As an added bonus, most CGI wrappers perform additional security checks before they allow a requested application to execute.

In the following sections, two popular CGI wrappers are discussed: suEXEC and CGIWrap.

suEXEC

Apache comes bundled with its own security wrapper application called suEXEC. suEXEC allows users to run CGI and SSI programs as the owner of the site as opposed to the owner of the httpd process. Here's how suEXEC works. When a request is made for a CGI or SSI file not owned by the Apache user, the request is passed to suEXEC along with the program name and the owner's user/group ID. suEXEC then runs a series of checks to ensure the request is valid. If it is, the script is executed. If the request fails any of the checks, the script is not run and an error is logged.

For a complete list of all 20 checks performed by suEXEC, and for detailed installation/configuration instructions, see the Apache suEXEC Web site.

The most common way to use suEXEC is with the `User` and `Group` directive inside a `VirtualHost` container. For example:

```
<VirtualHost 192.168.1.5>
DocumentRoot /home/tom/public_html
ServerName insights.syroidmanor.com
ScriptAlias /cgi-bin/ /home/tom/public_html/cgi-bin/
User tom
Group tom
</VirtualHost>
```

Note that both the User and Group must be defined before suEXEC will work. Omitting one or the other will cause any request for a CGI application to fail, and will generate an error.

Two further notes regarding suEXEC. First, the owner of the Web server process must be able to 'su' to the owner/group of the script. If Apache cannot do this, the script will fail. Second, you'll know if suEXEC has been successfully installed and configured by checking Apache's error log after restarting the server. You should see a line similar to the following:

```
[Mon Aug 7 20:39:20 2002] [notice] suEXEC mechanism enabled [wrapper:
/usr/sbin/suexec]
```

CGIWrap

CGIWrap is similar to the suEXEC program in that it permits user access to CGI programs without the risk of compromising server security. It does this by running any program defined as a CGI application as the file owner rather than the Apache user. CGIWrap also performs several security checks on the CGI application; the application will not be executed if any of the checks fail.

CGIWrap is written by Nathan Neulinger and available from the Unix Tools Web site.

CGIWrap is independent of Apache and the operating system (suEXEC has some performance advantages over CGIWrap as it is compiled directly into the httpd source), and as such needs to be downloaded and compiled independently. On the plus side, CGIWrap allows you to create `allow/deny` files that can be used to restrict access to CGI applications. To install CGIWrap simply download the source tarball (current version as of this writing is 3.7.1), extract it to a directory of your choosing, and run the `configure` script.

When the program is compiled, simply copy the CGIWrap executable to the user's cgi-bin directory. Note that this directory must match the cgi-bin directory specified during the configuration process. Next, change the ownership and permission bits as follows:

```
chown root CGIWrap
chmod 4755 CGIWrap
```

Finally, create symbolic links from nph-cgiwrap, ntp-cgiwrapd, and cgiwrapd to CGIWrap:

```
ln -s CGIWrap cgiwrapd
ln -s CGIWrap nph-cgiwrap
ln -s CGIWrap nph-cgiwrapd
```

You may also have to add an extension to the CGIWrap executable (for example, `CGIWrap.cgi`) depending on what, if any, file extensions you've associated with CGI applications.

Summary

This article discussed securing dynamic content on an Apache Web server. Topics included security considerations that apply to all dynamic content in general, Server Side Includes, Apache's Common Gateway Interface, and two ways to wrapper CGI content. While an exhaustive discussion of securing all forms of dynamic content would be impossible within the confines of such a short article, we hope this document has provided a basic understanding of where the most common security holes lie, and how to address them.

As always, comments and feedback on the material presented are welcome.

Resources

For further details on securing dynamic content check out the following resources:

- The Apache Web site is the definitive online resource for all things relating to the Apache Web server. Here you'll find source code, pre-compiled binaries, FAQs, and developer links. There is also extensive documentation available for both Apache 1.3 and Apache 2.0. Unlike some Open Source projects, the Apache documentation is kept relatively up-to-date thanks in large part to the documentation coordinator Ken Coar.

- Another good online resource, especially for developers, is the O'Reilly OnLamp.com site.

- For those who prefer paper-based resources, check out *Apache Server Unleashed* by Rich Bowen and Ken Coar (SAMS). Part III contains extensive material on Dynamic Content, and Part IV is a broad overview of setting up security and auditing. Also recommended is Laurie and Laurie's *Apache: The Definitive Guide* (O'Reilly & Associates). While slightly dated as far as Apache revisions go, most material covered remains pertinent.

- No discussion of Web server security would be complete without at least a passing reference to *chroot*ing your installation. While beyond the scope of this article, a chrooted server is one of the most complete security measures available for Apache. For a detailed overview of the process involved, check out this online HOWTO put together by Denice Deatrich.

About the author

Tom Syroid is a contract writer for Studio B Productions, a literary agency based in Indianapolis, IN specializing in computer-oriented publications. Topics of interest/specialty include *NIX system security, Samba, Apache, and Web database applications based on PHP and MySQL. He has experience administering and maintaining a diverse range of operating systems including Linux (Red Hat, OpenLinux, Mandrake, Slackware, Gentoo), Windows (95, 98, NT, 2000, and XP), and AIX (4.3.3 and 5.1). He is also the co-author of *Outlook 2000 in a Nutshell* (O'Reilly & Associates) and *OpenLinux Secrets* (Hungry Minds). Tom lives in Saskatoon, Saskatchewan with his wife and two children. Hobbies include breaking perfectly good computer installations and then figuring out how to fix them, gardening, reading, and building complex structures out of Lego with his kids. Questions, comments, and errata submissions are welcome; you can either e-mail the author directly (dwcomments@syroidmanor.com.

e-mail it!

What do you think of this article?

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

Comments?

Submit feedback