Securing web-based applications with hole-in-the-chroot

Anne Bennett <anne@alcor.concordia.ca>
Instructional and Information Technology Services
Concordia University
2000/08/25

Abstract

The central IT department at our university was given the task of making certain applications, formerly available only using interactive login to a sensitive system, available over the web. Some of these applications were not designed for use in a "hostile environment", and were found to be fragile; in many cases, we had binary-only distributions and could not fix the applications. Of course, some of those applications required access to a sensitive database. Finally, it had to be possible for relatively inexperienced programmers to put those applications on the web.

We wished to satisfy these requirements without appreciably dimishing the security of the sensitive database, or that of the web server installation (Apache in a chroot). We succeeded in doing this using a mechanism whereby a tiny hole is poked through the wall of the "chroot prison" to implement a highly controlled channel of communication between CGI programs in the chroot and fragile applications running on the system. We packaged up this system such that our inexperienced programmers supply only the web form and some very rudimentary processing of query data for submission to the system-side application; everything else is handled by our package.

Hole-in-the-chroot version 1 has been operational since mid-1999, and while some limitations have been encountered, it has fulfilled the original requirements very successfully. Version 2, which permits more complex (but still controlled) interactions between the CGI programs and the system-side applications, is in progress.

Web server security and chroot

Many attacks are possible on a web server, from simple denial-of-service all the way to attacks whereby the system is subverted to give unauthorized access to data, to alter data, or to run programs in an unauthorized manner. Vulnerabilities can occur in the web server implementation, in the web server configuration, in the web server's associated software (such as CGI programs or plug-in modules), and in independent applications used by the CGI and plug-in programs, with the latter categories being particularly ripe for attack by user-supplied hostile data. We won't repeat here the principles and practice of securing web servers, as they have been covered thorougly and competently elsewhere. In particular, excellent coverage of web security issues can be found in [Kossakowski 2000], a compact but complete practical guide to the configuration and operation of a web server, and in [Rubin 1997], which covers browser as well as server security.

Let us then postulate that the operating system is managed properly, that the web server is kept up-to-date with patches and is configured carefully as per [Apache 1999], that the web server runs as an unprivileged user, and that `cgiwrap` ([Neulinger 1999]) is used to isolate privileges for CGI programs. Despite all these precautions, an error in a CGI program could still allow an attacker to perform arbitrary commands with the privileges of the CGI program author. Such relatively unprivileged access to the system is still dangerous: most operating systems contain vulnerabilities whereby local access can eventually be leveraged, by means of errors in privileged programs, to gain system (*root*) privileges. This is the motivation for running the web server in a chroot, or Unix changed filesystem root. Under this scheme, the web server can see only that part of the filesystem under the chroot directory. Thus, we can prevent the web server and associated software from accessing most of the filesystem in any way, by running it in a "prison" where only web pages and a small number of programs are available.

How safe is chroot? It is possible to escape a chroot environment, but such an escape usually requires *root* privilege, which is, in practice, possible to protect on a web server, especially one running in a chroot. Another attack involves signalling and taking control of a process outside the chroot; this avenue of attack depends on the presence of a system-side process running under the same UID as the subverted web-side process, and thus can be eliminated by keeping separate user databases on the two sides of the chroot.

CGI programs and the software they call are the most likely points of failure in the security of a web-based application, because they must accept and process data from untrusted sources. If this data is "hostile", and is not handled correctly, the result could be the execution of programs outside the control of the CGI program author. Unfortunately, because of the rapid growth of the web, and the consequently huge demand for "web programmers", many CGI program developers are relatively inexperienced, and are often unaware of secure programming practices, such as those outlined in [Stein 2000] .

Project requirements

The central IT department at our university was asked to make available over the world-wide web the reports generated by the University's Financial Information System (FIS), which runs on a Unix system. The group responsible for the FIS had little experience with Unix, none with Perl, and no knowledge of safe CGI programming practices. In addition, it was found that at least one of the budget reporting programs in our Financial Information System was subject to buffer overflow when presented with long command-line arguments. Since we did not have source code for this program and so could not fix it, it was clear that we'd have to protect it carefully from being subjected to unusually long command-line arguments. Also, since we had no way of knowing whether the program made any `open`, `system`, or other potentially shell-spawning system calls, we thought it wise to protect it as well from input data containing shell metacharacters.

We considered three approaches to satisfying the given requirements, i.e. to connecting the FIS applications with a CGI front end:

- Forget the chroot, and let the CGI programs just invoke the report programs directly. For our purposes, moving the entire Financial Information System software and database under the chroot was considered almost equivalent to not using a chroot at all.
- Run the web server on a separate system, and use the network to communicate between the CGI program (on the web host) and the application (on the FIS host).
- Use a chrooted web server on the FIS host, but poke a small hole in the chroot to permit the execution of FIS applications on behalf of a CGI program.

For obvious reasons, we were reluctant to relax our stringent requirements with respect to the security of our web servers and their applications; hence, the first option was immediately ruled out. When we compared the second and third options, we found that they were functionally very similar, insofar as in either case, a special communication protocol would have to be developed to connect the CGI program to its system-side application, and in either case, the web server and the FIS were kept isolated from each other. However, we concluded that it would be easier to write and easier to secure an application that used only the filesystem for communication, not the network.

Hence, we undertook to develop a mechanism by which a small hole would be poked through the wall of the "chroot prison", while still minimizing the risk introduced to the rest of the system thereby. Our mechanism would therefore have to work with our web server (Apache) running as an unprivileged user and under a chroot, and with all CGI programs are invoked using `cgiwrap`. Communication between the system and chroot sides would have to be very carefully controlled, and preferably that "hole" would be used as a filter to protect our fragile report programs from hostile data. Finally, our technique would have to be accessible to inexperienced programmers -- or at least, to programmers unfamiliar with Unix, Perl, and CGI: our programmers are quite competent with COBOL on a VMS system!

The hole-in-the-chroot package

We met the requirements with a package we call "hole-in-the-chroot", which works as follows. Bearing in mind that files under the chroot are visible to the rest of the system, but that the reverse is of course not true, we assign some files under the chroot as the communication channel, and we create a program that runs outside the chroot, and that manages communication between these files and the rest of the system; this program is the *listener daemon*. In addition to the listener daemon, we supply a code library that does the bulk of the work on the chroot side, as well as a stub CGI program to which the programmer need only add the "meat" of two subroutines: one to display a form to the user, and one to take the information submitted by the user and prepare it for use by the report program. (We should note that this mechanism is for use in a cooperative manner by programs the webmaster has written or vetted; it cannot be shared with undisciplined programs.)

Our communication mechanism: three files

Our communication channel consists of:

- Meta-communication:
  - *pipe to daemon*:
    There is exactly one communication channel to the daemon, and it is a Unix named pipe created by the webmaster during the installation of hole-in-the-chroot. This daemon pipe is written by the various instances of CGI programs, using a lock file to guarantee exclusive access to the pipe. The pipe is written only, not read, from the web (chroot) side; this is ensured using file permissions. A CGI program writes two values to the pipe: the name of a job to be run, and the location of the data communication files. The listener daemon monitors this named pipe, and takes action whenever a request is written to it.
- Per-CGI-invocation communication:
  There is one pair of data communication files per CGI program instance, so care must be taken to ensure unique filenames for these temporary files. This pair of files is created by the CGI program before kicking the daemon (writing to the daemon pipe).
  - *file to daemon child*:
    This file contains the input for the system-side application; this input is derived from query data supplied by the end user through the web form.
  - *pipe from daemon child*:
    This pipe is used to receive results from the system-side application.

File permissions are used to help restrict what can happen to the data while it is going through the communication pipes. In particular, we create two dedicated and generally unprivileged users to implement our scheme: an "under the chroot" user (*web-side*), and an "outside the chroot" user (*sys-side*). The outside user may have special privileges in order to accomplish its particular task (for example, it may need to access a database). Note that the use of separate users for the system and chroot sides has the beneficial side effect of preventing signal-based chroot-escape attacks.

Thus, we have the following files and directories:

- A named pipe (daemon pipe) for job requests, writable only by *web-side*, and readable only to *sys-side*.
- The directory in which the named pipe resides, writable to *web-side*, since we use a lockfile in that directory to manage contention for the named pipe.
- A temporary directory writable only by *web-side*, but executable by *sys-side*, for passing data to and reading results from the programs on the system side.
- A pair of per-CGI-invocation communication files; *web-side* is responsible for creating these files and for cleaning them up when it is done. It must also ensure that the permissions are correct to permit itself and *sys-side* to read or write those files as appropriate.

Named pipes are used instead of regular files because they greatly simplify synchronization of the two halves of the communication. If we didn’t use named pipes, the web-side process would have to somehow communicate to the system-side process that there was data ready, and the system-side process would have to communicate to the

web-side process that results were available. With named pipes, we avoid the difficulties of interprocess communication, and the special difficulties of such communication across what is supposed to be a strong barrier. The daemon simply blocks reading the daemon pipe until there is a message for it. The CGI program can simply start to read its results pipe, knowing that it will block until the results arrive.

The listener daemon and its child

The listener daemon gets the actual work done. It must be started at system boot time, and must be robust, since its death would prevent the further handling of any hole-in-the-chroot CGI programs. To avoid having an excessively long-running program delay the processing of further requests, the daemon parent simply spawns a child to deal with each request, and immediately goes back to listening on the pipe.

As mentioned above, two values are written to the pipe to represent each job request: a job name, and the location of the communications files for that job.

The job name is mapped by the listener daemon child to a sequence of programs; an unknown job name results in an error. Thus, no matter what the web side asks for, the system side will not run arbitrary programs, but only those programs pre-coded into the listener daemon for known job names. Each job name is mapped to a sequence of programs, where each program in the sequence is connected to the next with a shell "and" ("&&"), to stop the execution of the sequence as soon as one program returns a non-zero exit status. Thus, we can put in an authorization program whose exit status will determine whether or not any further programs in the sequence will run, for example.

Since each program in the sequence may need its own input, it is necessary for both sides, web (CGI) and system (daemon) to agree on a separator which will mark the boundary between input for the first program in the sequence, the second, and so on. In the configuration of the listener daemon, where the job name to program sequence mapping is performed, there is syntax which permits the specification of which programs use which input files (or sections of the input file). For example, here is an extract from this mapping:

```
%programs = (
  'b12' => [ 'DB01', [ 'run', 'check_open', '<',
                       $insert_input_file ],
                     [ 'run', 'authorize', '<',
                       $insert_input_file ],
                     [ 'fis', 'budget_12' ],
           ],
);
```

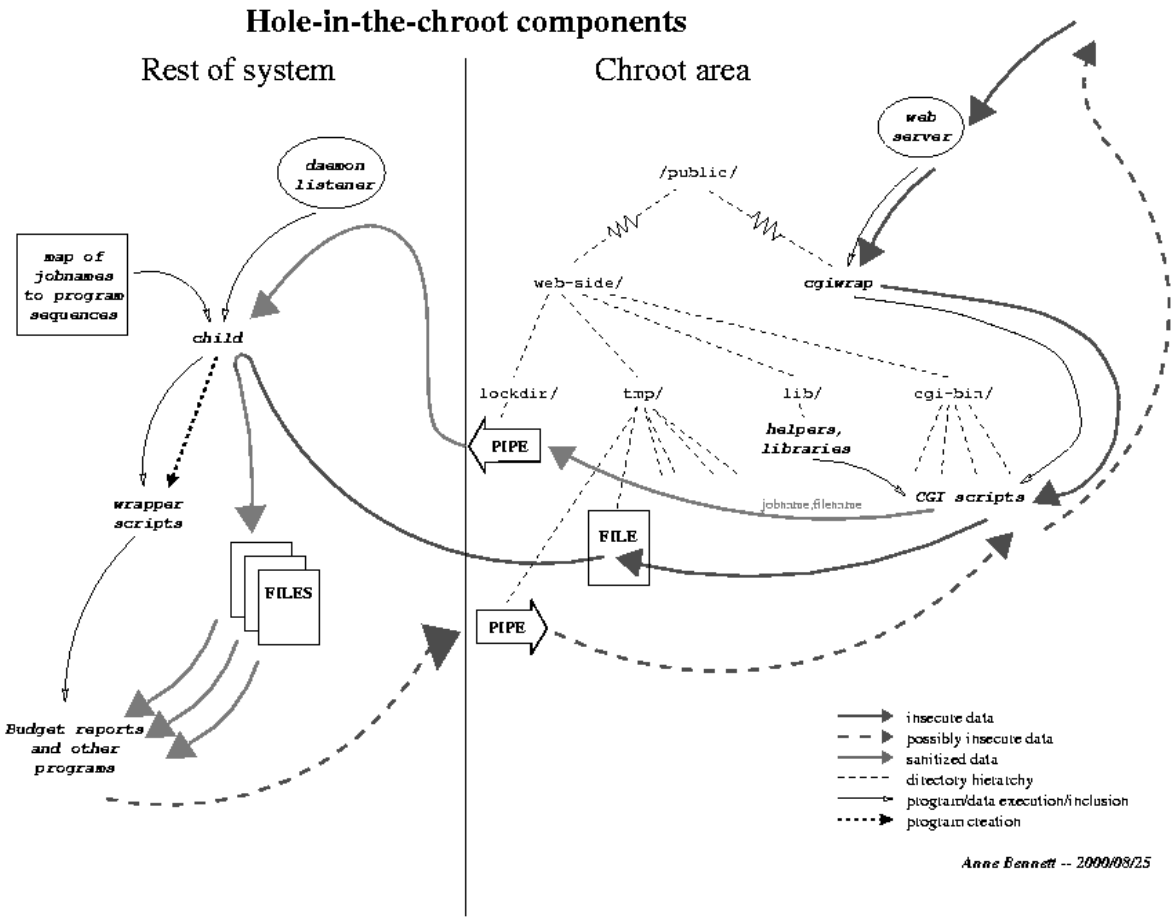
In the above extract, jobname "b12" maps to a sequence of three programs connected to database instance "DB01": first, a program to check that the database is available is run using the first part of the input file; then, a program to check the user-supplied authentication data is run using the second part of the input file, and finally, if all is well, the "budget_12" report program is executed.

Since the data passed from the web side must be treated as tainted (despite the fact that we hope the CGI program has performed some checks on it), this data should be sanitized (checked for length and invalid characters, at least) by the daemon child before the sequence of programs is invoked. This is especially important if some of the programs invoked are outside the webmaster's control and are known or suspected to be insecure (as was the case with some of our report programs).

The sequence of events

Here is the sequence of events (flow of data and control) which occurs to process a web query on a program using the described scheme. Note that we are using the common CGI programming technique of having the same program issue the form (if it is invoked with no query parameters) and process the request (if POST or GET data are supplied in the request).

- The web server is called with the URL of the form/CGI program (via `cgiwrap`, which changes identity to *web-side* before starting the program); on the first invocation, generally no query parameters (GET or POST data) are supplied, so our CGI program simply issues a web form. *(The form is supplied by the FIS programmers.)*
- The web server is called with the URL of the form/CGI program (via `cgiwrap`, which changes identity to *web-side* before starting the program); on the second invocation, usually the end user has supplied query data, which are collected by our CGI program.
- The CGI program runs as *web-side*. Its algorithm is as follows, with most of the code supplied in a library module:
  - Start the web output stream by issuing a content header.
  - Check that we were invoked via https (encrypted data stream), otherwise refer the user to the correct URL with an explanation.
  - Set up our temporary files for communication with the other side; note the filenames.
  - Open the web-to-system temporary file for writing.
  - Process the data from the form, and write whatever we need to write to that temporary file. This may include several subfiles indicated by a separator line in the file, if there will be several programs expecting input on the other side. *(This code is supplied by the FIS programmers.)*
  - Close that temporary file.
  - Set an alarm: if the job takes too long, issue an error message, clean up, and exit. We don't want these things to stick around forever if there's a problem.
  - Kick the daemon (write to its communications pipe) to tell the other side we have a job for it; check return status. This basically means writing to the pipe a line containing the requested job name and the basename of our temporary files.
  - Open the system-to-web named pipe for reading (this will block until there is a writer on the other side); this puts us to sleep until the other side has results for us.
  - Loop reading the output from the named pipe; process it as necessary, presumably sending it to standard output for the web server to transmit to the user, possibly formatting it on the way through. (Of course, the named pipe is being supplied from the other side with the results of running the application.)
  - Close the named pipe.
  - Clean up the temporary files and exit.
- The listener, which started at boot time, is listening on the daemon pipe. Note that we must re-open the pipe every time we see EOF, since there may not always be a writer on the other side. The listener runs as *sys-side*. Its algorithm is as follows, leaving aside the initial reading of parameters and daemonization:
  - Loop on reading the pipe. Every time we get a line, we fork, the parent goes back to listening, and the child closes the pipe and sets up and executes the job, using the temporary files for input and output.
    - Parent:
      - Go back to listening on the pipe.
    - Child:
      - First fork again to avoid zombies.
      - Parse the message, extracting the job name and the temporary file location. Check that we recognize that job name, and that our temporary files exist and have the right permissions.
      - Split our input file into multiple files if it contains separators, and keep track of the names of the subfiles.
      - Set up the commands we need to run by writing a shell script. Assign the input files to the various programs as specified in the daemon configuration. Sanitize the data in each input file as it is encountered.
      - Run the shell script, sending the output to the named pipe temporary file.
      - Any problems should be written to this output file, since nothing will check the shell script's exit status.



Using the hole-in-the-chroot package

Facilities supplied by the package

The hole-in-the-chroot package contains the following elements:

- The listener daemon.
- The boot-time start-up script for the daemon.
- The CGI-side Perl library, which includes the main program (subroutine `mainprog`) for the CGI scripts.
- A sample Perl CGI script.

Facilities provided by the FIS programmers

The FIS group must supply the "meat" of the implementation: code to actually print the form, code to gather the input and format it for the system-side applications, any support programs, and the application programs themselves.

The CGI scripts, which run under `cgiwrap` as *web-side*, are based on the stub supplied in the package. The FIS group must write or modify these items in the stub CGI script:

- *configuration*: this sets the job name, the web page title, and a few other parameters.
- *print_form*: this subroutine outputs fields of a web form for the end user to fill out.
- *process_data*: this subroutine picks up the user input and prepares it for use by the application(s).

Adding a new application

In order to add a new application, three things must happen:

- A *CGI script* must be written and put into place by the FIS group.
- A corresponding system-side *application* must be put into place by the FIS group.
- The `%programs` hash entry in the *listener daemon* must be updated to add the new job name and its description; this description is supplied by the FIS group, but the listener daemon is modified and restarted by the webmaster.

Operational Experience with Hole-in-the-chroot

Version 1 of hole-in-the-chroot was been in use at our university since mid-1999, and has met with great success. End users are glad to have web-based access to their budget reports, and find the response time of the system to be very satisfactory. FIS programmers find it quite easy to add new reports, and indeed additional reports have been added much more quickly than originally anticipated; in fact, the popularity of this system is generating requirements for additional features as people see its potential. The sysadmin is happy to have satisfied the users' functionality requirements without jeopardizing the security of the system.

However, the system as described above can handle only one interaction (one application call) at a time: there is no concept of session. One of the first requests we received from users concerned the possibility of authenticating only once with their username and password, and to have a persistent session thereafter (for some amount of time). Also, there is no provision for interactions between applications; for example, it is not possible to use one report program to generate data for use in a form from which another program would then be selected. Finally, there is no way to run programs as a run-time-determined user, as would be necessary, for example, to manipulate user files in a mail-forwarding configuration application.

Since the implementation of hole-in-the-chroot version 1, the university has developed requirements for more complex applications which require not only persistent authentication sessions and complex interaction between multiple forms and applications, but also the need in some applications to support multiple languages. Version 2 of hole-in-the-chroot, which supplies this additional functionality, is currently being tested.

References

- **[Apache 1999]**
  The Apache Software Foundation. Security Tips for Server Configuration. 1999.
- **[Kossakowski 2000]**
  Kossakowski, Klaus-Peter, et al. Securing Public Web Servers (CMU/SEI-SIM-011). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, 2000.
- **[Neulinger 1999]**
  Neulinger, Nathan. CGIWrap: User CGI Access. 1999.
- **[Rubin 1997]**
  Rubin, Aviel D.; Geer, Daniel; & Ranum, Marcus J. *Web Security Sourcebook* New York: John Wiley & Sons, 1997.
- **[Stein 2000]**
  Stein, Lincoln D. The World Wide Web Security FAQ. 2000.