SBOX: Put CGI Scripts in a Box

Lincoln D. Stein
Cold Spring Harbor Laboratory
Cold Spring Harbor, NY, 11724
lstein@cshl.org, http://stein.cshl.org

Abstract

sbox is a CGI wrapper script that allows Web sites to safely grant CGI authoring privileges to untrusted or naive authors. The script increases security in several ways. It changes the process privileges of CGI scripts to match their owners, preventing one script from interfering with another's data files or operations. It establishes configurable ceilings on script resource usage, avoiding intentional or unintentional denial of service attacks. Most importantly, *sbox* can also be used to run untrusted CGI scripts within a *chroot()*-ed directory, thereby preventing CGI scripts from accessing sensitive portions of the file system.

sbox can be used and redistributed freely. The complete package is available for download at

http://stein.cshl.org/WWW/software/sbox/

1 Introduction

Common Gateway Interface (CGI) scripts were among the first techniques for creating interactive Web pages and probably remain the most popular [Stein97]. Perhaps the main reason for the enduring popularity of CGI scripts is their simplicity. To create a dynamic Web page, a Web author writes a program that prints a short HTTP header followed by the contents of the desired Web page. The author then moves the program into a specially designated "CGI directory" on the Web server host. When the program's URL is requested, its output is displayed on the Web page.

A CGI script can be written in any language, compiled or interpreted. A fully functional CGI script can be written in just three lines of Bourne shell scripting code (including the #! line):

```
#!/bin/sh
echo -e "Content-type: text/plain\n"
echo "Hello world!"
```

The full CGI protocol [Coar] provides mechanisms for scripts to accept input from web forms, get information about the current operation of the server, learn the name and IP address of the remote browser, and pass back status information to the Web server. Communication between script and server is accomplished via environment variables and standard input/output. Essentially, the CGI protocol is a transient coshell system [Fowler] in which the Web server delegates the responsibility of producing the page content onto an external program, the script.

1.1 Security Problems with CGI Scripts

The simplicity and ease with which CGI scripts can be created is also the protocol's Achille's heel [Garfinkle, Rubin, Stein98a]. It is so simple to write CGI scripts that programming novices who have no prior experience in network server software development can readily create interactive Web pages. And here's where the problem lies. Novices, and sometimes even experienced programmers, are prone to errors that expose the Web server host to attack by unscrupulous individuals.

As an example of the problems beginners run into, consider the following CGI script written in Perl. Its intent is to recover an e-mail address from a submitted fill-out form and then mail a message to that address using the *mail* program.

```
#!/usr/local/bin/perl
use CGI qw(:standard);

$mailto   = param('mailto');
$subject  = param('subject');
$contents = param('contents');

open (MAIL,"|mail -s $subject $mailto");
print MAIL $contents";
print MAIL ".\n";
close MAIL;
print "Content-type: text/plain\n\n"
print "Mail sent to $address.";
```

This script, which is intended to be typical of a beginner's program rather than illustrative of good style, begins by using the *param()* function of the Perl CGI module [Stein98b] to recover the contents of three HTML form fields named "mailto," "subject," and "contents." The values of "mailto" and "subject" are used to open up a pipe to the Unix *mail* command. The value of "contents" is then printed to the *mail* process, which is then closed. The script ends by printing out a short confirmation message.

This script has a number of problems, including a reliance on the PATH environment variable to resolve the **mail** command, a failure to examine the "contents" field for a line beginning with a dot, which would terminate the mail message prematurely, and a failure to check for errors after the *open* and *close()* calls. However the most egregious flaw is in the call to *open()*, whre the programmer passes the contents of "subject" and "mail" to the shell without having first checked them for metacharacters. Consider what happens when the wiley hacker provides the following text as the value of the "mailto" field:

```
hackers_r_us@hackers.com</etc/passwd
```

The piped *open()* transforms this into the following call:

```
mail hackers_r_us@hackers.com</etc/passwd
```

with the result that the system password file is inadvertently mailed out to a potential attacker.

Other common problems in CGI scripts include the failure to check the length of strings before copying them into static buffers, failure to check for the existence of temporary files before clobbering them, and the failure to check user-provided pathnames for the ".." characters before opening files. Possible CGI script exploits include a variety of denial of service attacks. For example, a CGI script that reads user-provided input and spools it to a disk file is vulnerable to the mischievious hacker who uses a Web robot to transmit an endless stream of random bits to the script. Eventually the server's file system will fill up, causing the host system to stall.

Experience has shown that CGI scripts are a major source of vulnerability on the Web. Over the past five years, dozens well-known and widely distributed CGI scripts have been found to contain exploitable security holes [Stein98c, CERT]. Even experienced developers get burned from time to time. Offenders include freeware/public domain scripts, such as *count.cgi* as well as commercial products from such respected developers as Microsoft and Silicon Graphics. Understandably, most Webmasters are extremely cautious about installing new and untested CGI scripts on their servers.

One way to limit the harm that poorly-written CGI scripts can do is to run the Web server with as few privileges as possible. Most Web servers run as an unprivileged user without login privileges, such as "nobody." CGI script processes spawned by the server will ordinarily run under the same privileges as their parent, and by carefully controlling file and directory permissions, the Webmaster can limit the scope of any potential damage that errant CGI scripts can inflict on the server host. To increase safety even further, the Webmaster could place the entire server into a restricted directory using the **chroot** command. Now any CGI scripts it spawns will be limited to the portion of the file system that the server runs in.

1.2 User-Maintained CGI Scripts

Now consider a Web server run in an academic environment or by an Internet service provider (ISP). Such a system generally supports multiple Web authors of varying levels of experience and aptitude. In an academic environment, the authors are students, faculty members, and support staff who are granted personal Web pages. In the case of an ISP, the users are customers who have paid for Web space, and can range from individuals who maintain personal "vanity" pages to large co-hosted corporations. If users are allowed to write and install their own CGI scripts, then the risk from user-maintained scripts is magnified several fold.

First of all, a malicious author might seek to break into the Web server host by writing a CGI script that deliberately probes the host for holes. In many Web hosting environments, authors are not given a login shell. Instead they are constrained to uploading new and modified HTML pages via FTP or a Web publishing package such as FrontPage [Microsoft]. If authors are allowed to upload Perl scripts and compiled binaries for use as CGI scripts, this policy is easily circumvented.

Second, even if the host is protected by running the Web server as an unprivileged user and in a change-root directory, there is nothing to protect authors from each others' CGI scripts. Because all CGI scripts run under the same user account and execute in the same change-root directory, there is nothing protecting one author's data from another author's script. A student could write a CGI script to peek at the answers to a faculty member's online quiz, kill other students' CGI processes, or fill a user's guestbook file with obscene messages. In an ISP environment, one corporate customer could write a CGI script to spy on another customer's order entry system and client database.

Third, even if there is no active intent to do evil on the part of an author, a single poorly-written CGI script can still be used by Internet intruders to compromise the security of all authors on the system. For example, a guestbook script that doesn't check for the presence of ".." directories in the path to its data file can easily be exploited to view or overwrite files maintained by other authors.

Fourth, user-maintained CGI scripts are an invitation to denial of service (DoS) attacks. A malicious script writer can launch a DoS attack on the Web server host with a Perl script like the one shown below, which forks itself forever until the server host runs out of slots in its process table. It is possible that the system administrator will be unable to log in to kill the runaway process and may be forced to reboot the machine:

```
#!/usr/local/bin/perl
fork() while 1;
```

Finally, it is difficult to trace an attack from a user-maintained CGI script back to its owner. Since all scripts execute with the identity and privileges of the Web server, there is no easy way to determine whose script is, for example, leaving 40 megabyte scratch files in */tmp*.

1.3 Wrappers

There are a number of approaches to the problem of user-maintained CGI scripts. One approach is to outlaw them completely. The site's administrators can preinstall a number of standard CGI scripts for users to link to and configure the server so that no additional scripts can be added. Another solution is to submit all user-written scripts to an exacting code review.

Neither of these approaches is particularly appealing. The first solution is unlikely to be popular in the competitive Web hosting market where customers migrate to the service that offers the most features for the least cost. The second solution is only practical for sites that have unusually generous administrative resources or an unusually small number of users who want to install custom CGI scripts.

A more practical solution is to use a *wrapper* script. Instead of invoking user-maintained CGI scripts directly, the web server runs then indirectly via a wrapper program. The wrapper modifies the environment in some way that make the execution of the user-maintained script safer. The wrapper is also a good place to enforce security policy decisions. For example, the wrapper can keep a log of the scripts it has run and can refuse to run scripts whose permissions are insecure.

The first and still most widely-used wrapper program was *cgiwrap*, written by Nathan Neulinger [Neulinger]. *cgiwrap* performs several useful functions. Its main feature is that it uses the Unix *setuid()* call to run user-maintained CGI scripts under the user and group ID of the script's owner rather than the shared Web server account. This prevents one user's scripts from writing to data files maintained by another, and makes it easier to track down problems caused by poorly written scripts. *cgiwrap* also allows the Webmaster to place resource limitations on user-maintained scripts using the Berkeley *setrlimit()* call. This prevents a number of deliberate and inadvertent DoS attacks.

The *cgiwrap* program is straightforward to use. Once *cgiwrap* is installed in the system CGI directory, URLs used to invoke user-maintained scripts like this one:

http://www.site.com/~fred/guestbook.cgi

are replaced by URLs that invoke *cgiwrap*:

http://www.site.com/cgi-bin/cgiwrap/fred/guestbook.cgi

More recently, the popular Apache Web server has shipped with a built-in wrapper program called *suEXEC* [Apache Group]. The operation of *suEXEC* is similar to *cgiwrap*, but it is more tightly integrated into the Web server, making it unecessary to change any URLs in order to use it. In addition to changing its user ID to match that of the owner of the script, *suEXEC* logs each script it executes along with the user and group ID that it runs under. It also performs a series of consistency checks in order to detect unsafe practices. For example, *suEXEC* will refuse to run a script that is world writable or which is contained within a world writable directory.

The main limitation of both *cgiwrap* and *suEXEC* is that neither truly insulates scripts written by one user from those written by another. Naive users who store confidential information in world readable files and directories can still be attacked when another user's CGI script is used to peek at that data. In fact, although these scripts increase the security of the Web hosting service as a whole, they decrease the security of the individual user. Because the wrapped script runs with the same privileges as the user, it has free access to all the user's files. A poorly written script can be tricked into changing the user's HTML documents or recursively deleting his home directory. It can also impersonate the user, for example by sending e-mail from the user's account.

2 The *sbox* Wrapper

The *sbox* program is a CGI wrapper that goes beyond *cgiwrap* and *suEXEC* to offer the following features:

1. *sbox* calls *suid()* to run the requested script with the privileges of the owner of the script or the script's containing directory.
2. It calls *sgid()* to run the requested script under the privileges of the group that owns the script or the script's containing directory.
3. It performs consistency checks on the script file and directory ownerships to catch insecure situations such as world-writable scripts.
4. It establishes limits on the script's use of CPU, memory, processes, files and other resources.
5. It calls *chroot()* to run the target CGI script in a restricted change-root directory locatated within the user's home directory.
6. It cleanses the environment of information that is not germaine to CGI scripts.
7. It logs its actions and executes the target script.

These features can be used together, or can be switched on and off selectively to implement a variety of security policies.

Once installed, *sbox* is straightforward to use. To run an untrusted CGI script, create a composite URL consisting of the path to *sbox* followed by the path to the target CGI script. A typical URL for invoking a user-supported script looks like this:

http://www.site.com/cgi-bin/sbox/~fred/guestbook.cgi

sbox can also be used in conjunction with the virtual hosts feature provided by Apache and other servers. With some servers, it is even possible to make *sbox* transparent, so that its name doesn't appear in the path. A scheme to do this using the Apache *mod_rewrite* module is presented later in this paper.

The next sections describe each of *sbox*'s features in more detail and shows how they can be used to increase the security of the Web site.

2.1 suid()/sgid() Features

Before *sbox* launches a user-supported CGI script, it can be configured to change its UID and/or GID to match the script's owner. There are two possible variants of this feature. In the first variant, *sbox* uses the script file to determine which user and group to run as. This functionality is similar to the scheme implemented by *cgiwrap*. In the second variant, the ownership of the script is ignored; instead the ownership of the directory that contains it is used to determine the user and/or group.

Allowing *sbox* to take on the identity of the enclosing directory might seem a bit obscure, but the rationale is that it gives the Webmaster more flexibility than just using the script ownership does. For example, the Webmaster could use this technique to create a common *cgi-bin* directory for use by a particular group of developers. The directory would be owned by a pseudo-user and be group writable by each of the developers, allowing any user in the group to create and edit CGI scripts. When the script runs, it executes under the permissions of the common pseudo-user account, preventing it from modifying any of the author's files or databases unless he explicitly gives it permission to do so by setting the group writable bit.

Another strategy that the Webmaster might want to adopt is to configure *sbox* so that it performs an *sgid()* only. This will cause the target script to be executed with the group permissions of the script or enclosing directory, but with the user permissions of the Web server. By adopting a system-wide user-private group strategy in which each user is assigned a unique primary group, the script's author can exactly control what resources the script does and does not have access to. This strategy also makes it possible to create scripts that cannot modify their own source code file or binary, a risk both *cgiwrap* and *suEXEC* are subject to.

2.2 Consistency Checks

When *sbox* launches, it checks its environment for signs that it has been tampered with or that it is being run in an unsafe fashion. If any of the checks fail, *sbox* aborts with an error message.

The following checks are performed:

1. *sbox* checks that it was launched by the unprivileged user and group that the Web server runs as, for example **nobody** and **nogroup**. This check is to avoid the possibility that some user or group is trying to exploit the script's set-user-id features from the command line.

2. It checks whether it was launched by the root user, and aborts if so. This is often a sign that the Web server is misconfigured.

3. It checks the target CGI script for set-user-id and set-group-id bits and refuses to run if so. Untrusted users shouldn't be allowed to write suid or sgid scripts.

4. It checks that the target CGI script is executable by other, and aborts if not, assuming that the script's author had some reason for turning off the world execute bit.

5. It checks that neither the target CGI script nor its enclosing directory is owned by unprivileged user and group that the web server runs as. If the target is owned by this user, it's possible that it is a trojan horse created by a file upload script.

6. It checks that neither the target file nor its enclosing directory is world writable.

7. If the *chroot()* feature is active, it checks that the target script is located within the directory that will become the new root. This is a prerequisite for launching the script after the *chroot()* call.

8. Lastly, *sbox* checks that the target and its enclosing directory are owned by users and/or groups in an approved range, usually high-numbered IDs. This prevents *sbox* from being tricked into running a script as a special user such as **bin**.

These checks, along with the environment sanitization performed later in the launch process, go a long way toward preventing many of the loopholes and configuration errors that are frequently exploited by intruders.

2.3 Resource Controls

After applying its consistency checks, *sbox* applies resource limitations to the current process using the BSD-derived *setrlimit()* system call. Limits include the size of the CGI process, its resident (virtual) size, the number of file descriptors it can open, the size of the largest single file it can create, and the number of subprocesses it can spawn.

sbox uses both "hard" resource limits and "soft" ones. The soft limits, which can be adjusted upwards by the CGI script simply by calling *setrlimit()* itself, are set at low, stringent values by default. The hard limits, which once set cannot be increased during the lifetime of the process, use more liberal values. For example, the maximum file size that the user-supported CGI script has a soft limit of 100K, and a hard limit of 2 megabytes. These values can be adjusted at *sbox* compile time. The exception to this rule is the hard ceiling on core dumps, which is set to size zero. This prevents the user's CGI script from creating core files and closes various exploits that make use of core dumps to recover confidential information or to overwrite other files.

The net result of this design is that user-supported CGI scripts will, by default, be executed in an environment with strict resource controls. If a CGI script requires more of a particular resource than the soft limits provide, it can increase the resource up to the preset hard limit by calling *setrlimit()* itself. This design limits problems caused by resource hogging scripts written by naive users without unduly restricting the options of sophisticated users who need more resources than the soft limits allow.

In addition to setting resource limits, *sbox* also nices its own process to a priority of 10. This helps keep CGI scripts from becoming too much of a drain on a loaded system. Unlike *setrlimit()* values, a priority level, once increased, can never be decreased.

The priority level and the soft and hard limits on all system resources are set at *sbox* compile time. The system administrator can change the default values, or choose not to set a particular limit at all.

2.4 Changing the Root Directory

The crux of *sbox* security is its change-root function. If configured to do so, *sbox* will use the *chroot()* system call to change its root directory to some subdirectory enclosing the target CGI script. When the target CGI script runs, it will be unable to access parts of the filesystem outside the new root directory. This closes a large number of CGI exploits, including unauthorized access to the system password file, the modification of user's `.rhosts` files, the creation of hard links to system files in `/tmp`, and many more. It also provides a way to control exactly which system binaries and other resources that user-maintained CGI scripts have access to.

Administrator-configurable options determine how *sbox* chooses which directory to make the new root. In order for the target CGI script to be executed, it must live within the subdirectory selected for the new root. However, most CGI scripts will also need access to copies of system files such as interpreters and shared libraries in order to function correctly. Because it is inconvenient for the user to intermix his CGI scripts with system files, these files are usually stored in directories parallel to the directory that contains the target script. Another consideration is the user's "document root", the directory that contains his static HTML files. A number of popular CGI scripts, including guestbook scripts and page counters, require access to the user's HTML pages. In order for these scripts to work under the *sbox* system, the user's document root, or a portion of it at least, must also be located within the new root directory.

The locations of the new root directory and the target CGI script itself are controlled by the configuration variables ROOT and CGI_BIN respectively. Both variables are pathnames relative to the user's document root. A typical configuration will use the following values:

```
ROOT      ".."
CGI_BIN   "../cgi-bin"
```

This configuration tells *sbox* to look for the target CGI script inside a directory named *cgi-bin* on the same level as the user's document root directory. The new root directory will be the parent of both the *cgi-bin* directory and the user's document root. To see how this works in practice, consider a Web site in which user-supported directories are located in `/u/username/pub/html`, where "username" is substituted with the login name of the user. In Apache, this setup could be accomplished using the configuration directive **UserDir pub/html**.

A typical listing for `/u/username/pub` might look like the example shown in Table 1.

When *sbox* starts up, it determines the user's document root by looking at the Apache settings, which reveals the directory `/u/fred/pub/html`. It applies the CGI_BIN relative path, to give `/u/fred/pub/cgi-bin` as the directory in which to search for the CGI executable, and then applies the ROOT relative path to give `/u/fred/pub` as the directory that will become the new root. When *sbox* makes the *chroot()* call, `/u/fred/pub` becomes the top of the directory tree, creating a directory hierarchy with a structure similar to a Unix root filesystem. Files and directories above *pub*, which might include the user's private files, are off limits.

A drawback to this scheme is that it makes the user's entire document tree visible to his CGI scripts, which might not always be desirable. However a slight modification improves the scheme by making only a selected portion of the user's document tree visible. In this improved scheme, the Web server is configured so that the user's document tree is found, for example, in `/u/username/public_html`, and *sbox* is configured to change its root to a directory named *sbox* that is completely outside the *public_html* document tree:

```
ROOT      "../sbox"
CGI_BIN   "../sbox/cgi-bin"
```

For this configuration to work seamlessly, the user's directory should be set up something like this:

```
% ls -F /u/fred
public_html/                          doc root
public_html/sbox/->../sbox/html/      link
sbox/bin/
sbox/cgi-bin/                         scripts
sbox/etc/
sbox/lib/
sbox/html/
sbox/tmp/
...
```

The user's CGI scripts will now execute within the restricted *sbox* subdirectory and have no access, by default, to the user's HTML document tree. However the user can grant access to selected HTML documents by placing them into `public_html/sbox/`, which is connected via a symbolic link to `sbox/html/`. This allows CGI-accessible files to be accessed directly with a URL like this one:

http://www.site.com/~fred/sbox/index.html

while *sbox*-controlled CGI scripts are accessed with a URL like this one:

http://www.site.com/cgi-bin/sbox/~fred/guestbook

and CGI scripts that need to read or manipulate static HTML files are passed the additional path information in URLs like this one:

% ls /u/username/pub									
total 10									
drwxr-xr-x	2	fred	users	1024	Oct	23	06:27	bin/	*system binaries*
drwxr-xr-x	3	fred	users	1024	Oct	19	20:44	cgi-bin/	*CGI scripts*
drwxr-xr-x	2	fred	users	1024	Oct	12	16:59	dev/	*device special files*
drwxr-xr-x	2	fred	users	1024	Oct	19	17:57	etc/	*configuration files*
drwxr-xr-x	2	fred	users	1024	Oct	22	19:14	html/	*HTML document root*
drwxr-xr-x	3	fred	users	1024	Oct	19	20:35	lib/	*shared libraries*
drwxr-xr-x	2	fred	users	1024	Oct	23	05:48	tmp/	*temporary files*

Table 1: Typical directory listing for a user-supported Web directory

http://www.site.com/cgi-bin/sbox/~fred/guestbook/html/index.html

If the Apache web server is being used, these URLs can be simplified significantly with URL rewriting rules. An example of this is shown below.

2.5 Environment Cleansing

Before executing the target CGI script, *sbox* sets up a clean environment to run the target in. Depending on how the Web server was launched, there may be residual information in the environment that is not germaine to the CGI protocol or may in fact divulge sensitive information, such as database authentication information, or private PATH directories.

sbox filters the current environment, allowing through only those environment variables that are specified by the CGI/1.1 protocol, such as REMOTE_ADDR, or which contain fields from the incoming HTTP request header, such as HTTP_USER_AGENT. In addition, *sbox* recognizes and permits a small number of common extensions to the CGI/1.1 protocol, such as the DOCUMENT_ROOT and SERVER_ADMIN variables.

Other variables are not automatically copied into the target script's environment. In particular the PATH environment variable, because of its history of exploitation is **not** passed through. Instead PATH is set up using a constant "safe path" set at compile time. By default, the safe path is `/bin:/usr/bin:/usr/local/bin`. Because the target script will be running in a change-root directory, it is likely that only */bin* will be available to the target script.

When possible, *sbox* adjusts path-related environment variables so that they correctly reflect the change-rooted filesystem seen by the user's CGI scripts. Among the environment variables that are adjusted are the DOCUMENT_ROOT variable, which should point to the top of the user's document tree and PATH_TRANSLATED, which points to the file passed to the user's CGI script as additional path information.

2.6 Logging

Before passing control to the user's CGI script, *sbox* logs its actions. It prints out a timestamp, the name of the CGI script being executed, and the UID and GID of the process that it will execute the script as. Diagnostic information is also logged when *sbox*'s consistency checks fail, or when an error occurs during the processing or execution of the target CGI script.

By default, *sbox* sends its log entries to standard error, which on most web servers becomes incorporated into the shared server error log file. However *sbox* can instead be configured to write entries into a private log file. There's there's a performance penalty in keeping a private log file, since *sbox* must open the file for appending every time it runs.

The main rationale for having a log entry for each CGI script executed is that it provides an audit trail in the case of a CGI-based attack. The time of the attack can be correlated with the *sbox* log, and possibly lead to the identification of the script that was exploited. The *sbox* log could also be used to monitor CGI script usage for patterns suggestive of probing activity.

3 Practical Considerations

Configuring the *sbox* executable and preparing user-supported directories are the most tedious parts of using the *sbox* system. In order to reduce dependencies on the external environment, *sbox* does not use a configuration file. Instead, all its operational parameters are determined at compile time via a series of preprocessor `#define`s. About three dozen defines are contained in a single include file, `sbox.h`, which the system administrator must edit before compiling the executable. Fortunately, the vast majority of the defines are boilerplate values which will not need to be changed by most sites. Only about a half dozen are truly site-specific.

System administrators used to modern configuration scripts will probably be disappointed by this primitive configuration process, even though it is simple and straightforward. For this reason, a GNU *configure* style configuration script [Friesenhalm] is currently in preparation.

A more onerous task is setting up user-supported directories so that their CGI scripts run correctly in a change-root environment. On most modern Unices, compiled programs need one or more shared libraries in order to execute. Either the user's CGI scripts must be compiled statically, or the new root directory must contain a `/lib` subdirectory (or the dialect's equivalent) containing the shared libraries the user needs.

Other system support files may needed as well. CGI scripts that require access to the DNS system for hostname resolution will need an `/etc` subdirectory containing `resolv.conf`. Scripts that perform time calculations may need access to the compiled timezone file, `/usr/lib/zoneinfo/localtime`. Programs that need access to device special files, such as `/dev/null` and `/dev/zero` will need the appropriate files created with the **mknod** program. Scripts written in interpreted languages such as Perl will require a `/bin` directory containing the interpreter executable, and any support files that the interpreter needs, such as code libraries.

Clearly there are drawbacks to replicating a good chunk of the root filesystem for each user-supported web directory. For one thing, the disk storage requirements may become prohibitive on a system with many users. One solution is to limit the type of CGI scripts that users can write to a particular development system, such as Perl. Then only those files needed to support the Perl interpreter will have to be copied into the user's scripting directory.

Another solution to this problem is to use NFS to mount a trimmed set of `/lib`, `/bin`, and `/etc` directories in each user-supported directory. Even after the *chroot()* operation, the contents of these directories will continue to remain available to the user's CGI scripts. Although this technique creates a lot of mount points, the overhead for unused NFS mounts is minimal [Stern], and an automount daemon can be further used to reduce the load [Crosby]. However if this technique is used, care must be taken not to mount directories that contain sensitive information, such as an `/etc` directory that contains a live password file. This would defeat the purpose of the change-root system.

A minor drawback to using *sbox* is that it is not completely transparent to the user. Instead of writing natural-looking CGI URLs, users have to be trained to interpose `/cgi-bin/sbox` in front of any URL that points to a CGI script. On Apache servers, an elegant solution to this problem is to use the *mod_rewrite* URL rewriting module to automatically add the `/cgi-bin/sbox` prefix to users' CGI URLs.

For example, one could use a *mod_rewrite* URL rewrite rule to transform URLs of the form:

/~fred/cgi/guestbook

into URLs of the form:

/cgi-bin/sbox/~fred/guestbook

by adding these directives to Apache's configuration file:

```
RewriteEngine On
RewriteRule ^/~(.+)/cgi/(.+) \
            /cgi-bin/sbox/~$1/$2  [PT,NS]
```

Neither the remote user nor the the script's author ever sees the longer URL. The name transformation is completely transparent. As a bonus, this rewrite expression also correctly handles additional path information appended to the end of the URL.

In order to perform its *suid()*, *sgid()*, and *chroot()* functions, *sbox* must run with superuser privileges. This means that, like *cgiwrap* and *suEXEC*, it must be installed set-user-id to root. This fact should give any cautious Unix system administrator pause. However, *sbox* consists of only 700 lines of C code, all of which are available for public scrutiny. *sbox* is careful to avoid using static buffers and string copy operations that could cause a buffer overflow. It also checks its environment at startup time to confirm that it was invoked by the web server and not some other local user.

4 Conclusions

The *sbox* wrapper increases the security of web sites that need to run untrusted CGI scripts. It prevents different users' CGI scripts from interfering with each other by running each user's program under distinct user and group IDs. It prevents user-maintained scripts from accessing sensitive parts of the file system by running each script in a change-root directory. It lessens the impact of denial of service attacks by establishing per-process resource limits, and it avoids certain common misconfigurations by checking the environment for consistency before it launches the target CGI script. Lastly, it creates an audit trail that can be used to track down malicious or poorly implemented CGI scripts.

sbox is not a panacea for CGI woes. There are a variety of CGI-based attacks that *sbox* cannot prevent. Chief among these are network-based attacks. For example, if a CGI script can be tricked into probing a firewall system from within the protected network, there is nothing that *sbox* can do to prevent this type of attack. To completely insulate the user's environment from that of the host, you need to step out of the Unix domain and use a partitioned operating system, such as Hewlett Packward's VirtualVault technology [Hewlett Packard].

Finally, it is important to remember that the *sbox* wrapper alone won't make a Web site secure. CGI script precautions are just one component of a carefully considered site security policy that includes attention to operating system security, web server configuration, operating and backup procedures, and user education. While nothing is ever going to completely eliminate the risk of running untrusted CGI scripts on a Web server, the *sbox* wrapper does go a long way towards limiting the potential damage that poorly-written or malicious scripts can inflict.

5 Acknowledgments

Many thanks to Nathan Neulinger for the original *cgiwrap* program which inspired this work. I also wish to thank the members of the Apache Project, whose web server has proven that open source projects can provide the same power and stability as conventional products – if not more so.

6 Availability

sbox is written in ANSI C and compiles on multiple flavors of Unix. It can be used and redistributed freely. The complete package is available for download at

http://stein.cshl.org/WWW/software/sbox/

References

[Apache Group] The Apache Group (1998). *Apache suEXEC Support*, *http://www.apache.org/docs/suexec.html*

[Coar] Coar K and Robinson D (1998). *The WWW Common Gateway Interface, Version 1.1* Internet Draft. *ftp://ftp.ietf.org/internet-drafts/draft-coar-cgi-v11-00.txt*.

[CERT] Computer Emergency Response Team (CERT) (1996-1998). Multiple CGI-related advisories. *ftp://ftp.cert.org/pub/cert˙advisories/*

[Crosby] Crosby M (1997). *AMD – AutoMount Daemon.* The Linux Journal 35, March 1997. *http://www.ssc.com/LJ/issue35/amd.html.*

[Fowler] Fowler G. (1993). *The Shell as a Service.* Usenix Summer 1993 Technical Conference Proceedings, Cincinnati, Ohio. *http://www.usenix.org/publications/library*

[Friesenhalm] Friesenhalm B (1997). *Autoconf Makes for Portable Software.* Byte, November 1997.

[Hewlett Packard] Hewlett Packard Inc. (1998). *HP Internet Security VirtualVault Home Page. http://www.hp.com/security/products/virtualvault/*

[Garfinkle] Garfinkle S with Spafford G (1997). *Web Security & Commerce.*, O'Reilly & Associates, Sebastopol CA.

[Microsoft] Microsoft Corporation (1998). FrontPage 98 - Overview. *http://www.microsoft.com/products/prodref/571˙ov.htm.*

[Neulinger] Neulinger N (1996-1998). *cgiwrap.* http://www.umr.edu/˜cgiwrap/

[Rubin] Rubin A, Geer D, and Ranum M (1997). Web Security Sourcebook. *John Wiley & Sons, New York.*

[Stein97] Stein L, How to Set Up and Maintain a Web Site*, Chapters 8-9. Addison-Wesley Longman, Boston.*

[Stein98a] Stein L (1998). Web Security: A Step-by-Step Reference Guide. *Addison Wesley Longman, Boston.*

[Stein98b] Stein L (1998). The Offical Guide to Programming with CGI.pm. *John Wiley & Sons, New York.*

[Stein98c] Stein L (1998). The Web Security FAQ. http://www.w3.org/Security/Faq

[Stern] Stern H (1991). Managing NFS and NIS. *O'Reilly & Associates, Sebastopol, CA.*