A SECURE AND FLEXIBLE COMMON GATEWAY INTERFACE (CGI) WRAPPER

by

MINSU CHOI

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1995

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1998

A SECURE AND FLEXIBLE COMMON

GATEWAY INTERFACE (CGI)

WRAPPER

Thesis Approved:

[illegible]

Thesis Advisor

[illegible]

J Chandler

Wayne B Powell

Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my thesis advisor, Dr. K. M. George for his outstanding supervision, guidance and friendship. I would like to thank to my other committee members Dr. J. P. Chandler and Dr. G. E. Hedrick for their excellent assistant and advise. Thanks also go to my parents in Korea for their precious support and encouragement. Finally, I would like to thank the Department of Computer Science for supporting during last two years of study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The World Wide Web(WWW) makes using the Internet easy and efficient. Information providers can provide hypertext links, fill-in-forms, clickable images, and even executable programs through it [10]. Users can enjoy various forms of those services. They can also make their home page, similar to information providers. In addition to it, more people wanted to make interactive home pages to make their pages alive.

To develop interactive home pages, one needs to understand the Client-Server interaction [2] between the WWW clients (commonly known as browsers such as Mosaic, Netscape and Internet Explorer) and an Hypertext Transfer Protocol (HTTP) server called HTTPD [10]. This interaction involves two distinct but closely related issues. The first issue is the HTTP protocol used for communication with HTTP servers. This protocol has several specific communication methods (for example, GET or POST) that allow clients to request data from the server and send information to the server [10]. The second issue is the way HTTP servers handle a client's requests. If the request is for a file, the server simply locates the file and sends it, or sends an appropriate error message if the file is not available. Of interest here is the situation when the client wants to send information to the server for more complicated processing. In general, Web servers do not perform this processing themselves. The work is delegated to other programs called gateway programs. The Common Gateway Interface (CGI) specification defines the mechanism by which HTTP servers communicate with gateway programs [1].

CGI is a method for running programs on the Web server based on input from a Web browser. Gateway programs can be referenced by Universal Resource Locators (URLs) [3, 11, 12]. For example, GET is used to get a file or other resource, possibly with parameters specifying more exactly what is needed. In the case of form input, GET fully includes it in the URL, like

http://www.cs.okstate.edu/~choim/myscript.cgi?name1=value1&name2=value2

When a client accesses a URL pointing to a gateway program such as a script or executable code, the server activates the program and uses the CGI mechanisms to pass to the program data sent by the client (if any). The gateway program processes the data and sends its response back to the HTTP server, again using the CGI mechanisms. The server then forwards the data to the client such as Netscape and Mosaic that initiated the request using the HTTP protocol, completing the transaction. Figure 1 illustrates the CGI transaction in the context of World Wide Web computing.

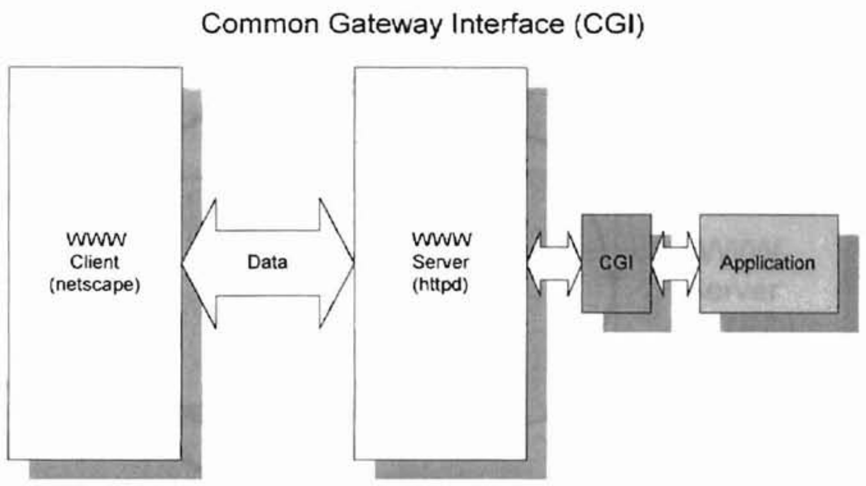


Figure 1 The Common Gateway Interface (CGI) (adopted from [1])

Since CGI scripts allow anyone on the Web to run a program on the server based on any input they choose to supply, CGI executables are probably the largest security risk for the Web sites [4, 7, 9]. The problem with CGI scripts is that each one presents yet another opportunity for exploitable bugs. CGI scripts should be written with the same care and attention given to Internet servers themselves, because, in fact, they are miniature servers which can handle specific user requests [13]. Figure 2 illustrates the CGI execution cycle.

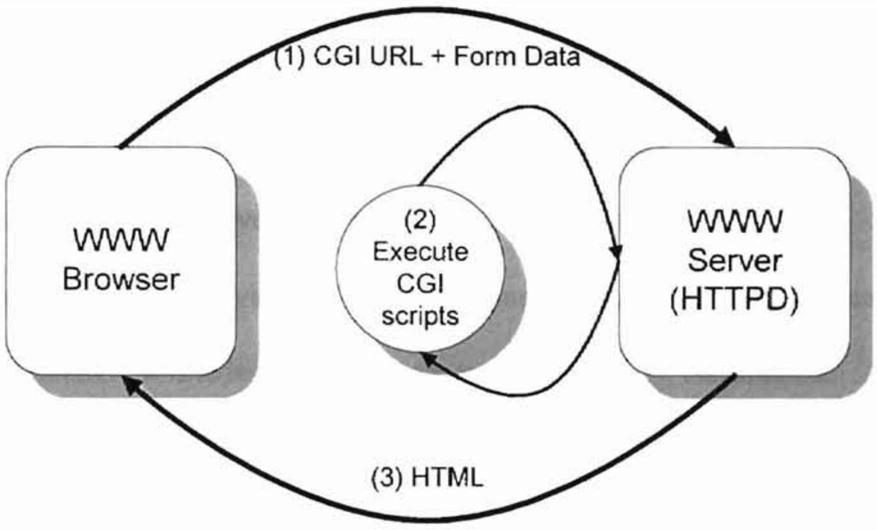


Figure 2 The CGI execution cycle (adopted from [1])

CGI scripts can present security holes in two ways [7] :

1. They may intentionally or unintentionally leak information about the host system that will help hackers break in.

2. Scripts that process remote user input, such as the contents of a form or a "searchable index" command, may be vulnerable to attacks in which the remote user tricks them into executing commands.

Some possible attacks are [9] :

1. Mailing the password file to the attacker (unless shadowed)
2. Mailing a map of the filesystem to the attacker
3. Mailing system information from /etc to the attacker
4. Starting a login server on a high port and telneting in
5. Many denial of service attacks: for example, massive file system searches, suspected infinite loops or other resource consuming commands.
6. Erasing and/or altering the server's log files

Those problems will never happen if CGI transaction is completely disabled, but powerful and convenient capabilities of CGI cannot be utilized. On the other hand, if CGI transaction is fully allowed to every remote user, security risks mentioned above cannot be avoided. Therefore, it is necessary to research a better CGI environment which ensures more security and more capability. Chapter II will discuss numerous related works which is currently used to resolve those security problems with CGI. Those problems with CGI transaction model as well as weaknesses of current solutions will be formally stated in chapter III. As a new solution to the problems, Secure and Flexible CGI Wrapper (SFW) will be introduced after that chapter.

CHAPTER II

RELATED WORKS

1. Writing secure CGI scripts

Whenever a program is interacting with a networked client, there is the possibility of that client attacking the program to gain unauthorized access to the resources of the server. Even the most innocent looking script can be very dangerous to the integrity of computer systems. With that in mind, Stein [4] and NCSA (The National Center for Supercomputing Applications) at UIUC (University of Illinois at Urbana-Champaign) [7] presented valuable guidelines to making sure that a software program does not come under attack. Following is a summary of those guidelines:

Beware of the eval statement

Languages like PERL and the Bourne shell provide an eval command which allows one to construct a string and have the interpreter execute that string [14]. This can be very dangerous. Observe the following statement in the Bourne shell:

```
eval `echo $QUERY_STRING | awk 'BEGIN{RS="&"} {printf "QS_%s\n",$1}' `
```

This script takes the query string, and converts it into a set of variable set commands. The interpreter executes those commands at the end. If the query string contains "rm -r", this short script may destroy the whole directory structure under current directory.

Do not trust the client to do anything

A well-behaved client will escape any characters which have special meaning to the Bourne shell in a query string and thus avoid problems with the CGI script misinterpreting the characters. A mischievous client may use special characters to confuse the CGI script and to gain unauthorized access.

Be careful with popen and system commands.

If any data from the client is used to construct a command line for a call to popen() or system(), be sure to place backslashes before any characters that have special meaning to the Bourne shell before calling the function.

2. CGI wrapper program

Most Hyper-text Transmission Protocol Daemons (HTTPDs) do not change user ID to a CGI script's owner. Instead they run the program as "nobody", since effective user ID of

CGI process is to be set same as executor's user ID, which is nobody [5, 13]. CGI scripts available on the Internet assume that the CGI script will be run as nobody so they require either files to be world-writable or CGIs to be Superuser ID (SUID). One, however, almost never need files to be world-writable. Usually a directory can be made world-writable so that the CGI can create files owned by nobody. Directory permission can be restored afterwards. This is not a very good idea since other users can access those world-accessible files at the same time, too. In addition to it, nobody scripts are too limited to provide advanced CGI services, since they cannot access files other then world-accessible files. Figure 3 shows how nobody CGI script works. In this case, incoming remote user tries to execute local user choim's CGI executable. Process real user ID is set to choim's but effective user ID is nobody's. It means that the CGI process is as exactly powerful as nobody who only can access world-accessible resource domain.

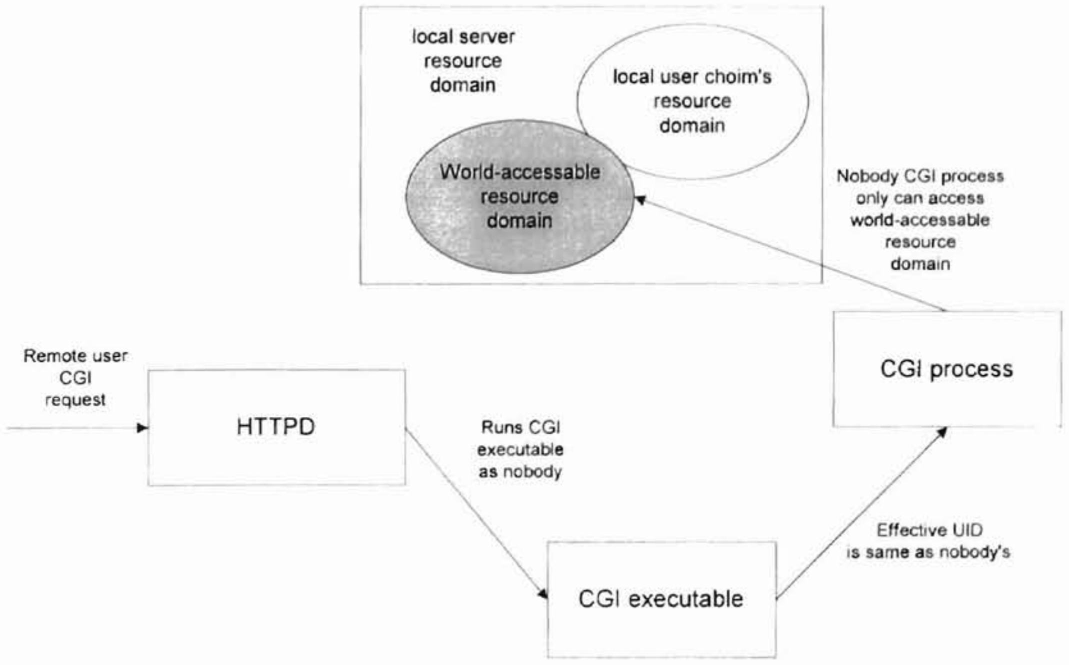

local server resource domain
local user choim's resource domain
World-accessable resource domain
Nobody CGI process only can access world-accessable resource domain
Remote user CGI request
HTTPD
Runs CGI executable as nobody
CGI process
Effective UID is same as nobody's
CGI executable

Figure 3 "nobody" CGI execution

The user with UID 0 is special and called the superuser (or root) [15]. The superuser has the power to read, write and execute all files in the system, no matter who owns them and no matter how they are protected. Making scripts SUID is very dangerous if implementers of the CGI scripts are malicious [4, 13]. For example, in a university machine with many users or in a commercial Internet service provider's machine, it is not safe to trust the other users. SUID scripts have many more potential security holes than normal CGI scripts. It is impossible to have a secure SUID shell script on some operating systems. The simplest methods for attacking SUID scripts depend on setting environment variables maliciously. Figure 4 illustrates how CGI executable can be executed on behalf of superuser. To make SUID CGI process, owner of CGI executable has to be superuser and its set-user ID bit must be on [13,15]. For example,

```
$ chown root test.cgi
$ chmod 4711 test.cgi
```

will allow SUID CGI execution.

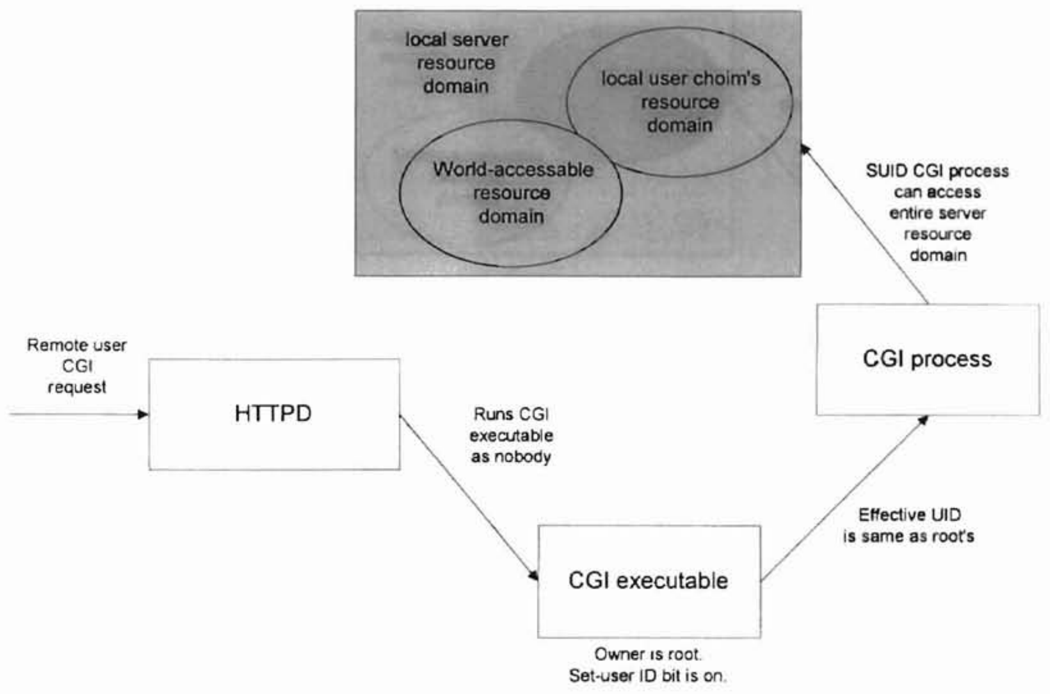

local server resource domain
local user choim's resource domain
World-accessable resource domain
SUID CGI process can access entire server resource domain
CGI process
Remote user CGI request
HTTPD
Runs CGI executable as nobody
Effective UID is same as root's
CGI executable
Owner is root. Set-user ID bit is on.

Figure 4 SUID CGI execution

The program CGIwrap which is developed by Nathan Neulinger is a good way to allow users to run CGIs under their own UID [6]. Figure 5 shows how it works. However, CGIwrap does not have capabilities to achieve more flexible and secure approach to the user directory mapping. For the duration of the CGI transaction, CGI script only can access resources with same UID of CGI script with Nathan Neulinger's CGIwrap. In addition to it, malicious remote or local user can by-pass CGIwrap and are able to directly access to the local CGI executables and resources. Thus, more flexible and secure CGI wrapper system can possibly resolve these limitations.

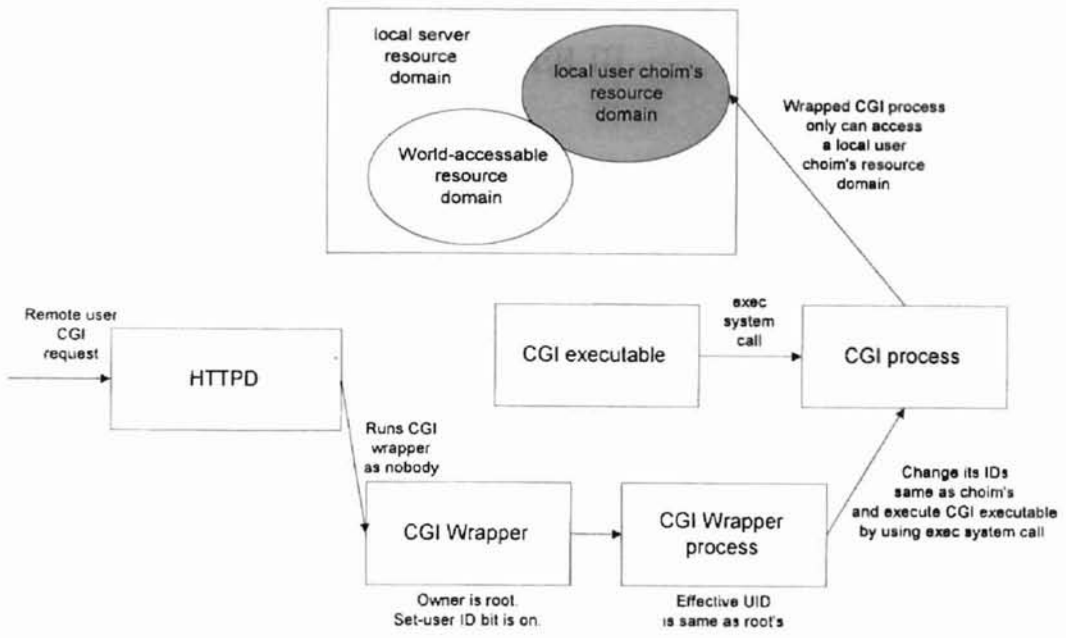

local server resource domain
local user choim's resource domain
World-accessable resource domain
Wrapped CGI process only can access a local user choim's resource domain
Remote user CGI request
HTTPD
CGI executable
exec system call
CGI process
Runs CGI wrapper as nobody
CGI Wrapper
CGI Wrapper process
Change its IDs same as choim's and execute CGI executable by using exec system call
Owner is root. Set-user ID bit is on.
Effective UID is same as root's

Figure 5 CGI wrapper CGI execution

CHAPTER III

PROBLEM STATEMENT

Writing secure CGI scripts enhance CGI security, but it is not secure enough when attackers make CGI scripts which has intentionally injected malicious features such as mailing important system files such as password file and log files of the server.

CGI wrapper program such as CGIwrap is a good way to allow users to run CGIs under their own UID. It means that wrapper partially solves SUID CGI execution problems and nobody CGI execution limitations, since it gives CGI owner's UID to users who wants to run those CGIs. It means every user who wants to run the same CGI script will have the same access privilege since the CGI script run under CGI executable owner's UID [6]. This situation is considered too restrictive and inflexible. CGIwrap can be by-passed and server system may be endangered. Different users of a CGI executable also cannot be applied by different access privileges. Thus, an advanced CGI wrapper system which is capable of better security and capability is necessary.

CHAPTER IV

SECURE AND FLEXIBLE CGI WRAPPER (SFW) ORGANIZATION

1. Apache suEXEC Support

Apache WWW server was originally based on code and ideas found in the most popular HTTP server of the time NCSA (The National Center for Supercomputing Applications) httpd 1.3 (early 1995). It has since evolved into a far superior system which can rival almost any other UNIX based HTTP server, in terms of functionality, efficiency and speed [13]. Since it began, it has been completely rewritten, and includes many new features. Apache is the most popular WWW server on the Internet [13].

The suEXEC feature, which was introduced in Apache 1.2, provides Apache users the ability to run CGI programs through dedicated CGI wrapper program [13]. Apache HTTPD is freeware. Thus, new CGI wrapper can be implemented on apache suEXEC support.

The configuration of suEXEC is a four step process: configure CGI wrapper source code to make it suitable for given Web server, compile wrapper program, place the wrapper binary in its proper location, and configure Apache for use with suEXEC and recompile it.

By default, Apache is compiled to look for the suEXEC wrapper location src/httpd.h. The following defines the path to the suEXEC wrapper:

```
#define SUEXEC_BIN "/usr/local/apache/sbin/suexec"
```

If the installation requires location of the wrapper program in a different directory, editing src/httpd.h and recompiling Apache server is required.

After CGI wrapper program is compiled, it has to be placed at the defined location for SUEXEC_BIN. In order for the wrapper to set the user ID, it must be installed as owner "root" and must the set user ID execution bit set is on. Examples are given below:

```
chown root /usr/local/apache/sbin/suexec [ENTER]
chmod 4711 /usr/local/apache/sbin/suexec [ENTER]
```

After properly installing the CGI wrapper executable, the apache server needs to be killed and restarted. Upon startup of the web-server, if Apache finds a properly configured CGI wrapper, it will print the following message to the console:

```
Configuring Apache for use with suexec wrapper.
```

If this message does not appear at server startup, the server is most likely not finding the wrapper program where it expects it, or the executable is not owned by root, or the executable's set user ID bit is off.

The primary benefit from the apache suEXEC support is that no remote user can possibly by-pass CGI wrapper program. Every CGI transaction can be security checked and log-in controlled by the properly implemented and configured CGI wrapper. It is very important since making of security CGI wrapper is not possible without this kind of Web server support.

2. suEXEC Security Model

suEXEC is based on a setuid CGI "wrapper" program that is called by the main Apache web server. This wrapper is called when an HTTP request is made for a CGI program that the administrator has designated to run with user ID other than that of the main server. The wrapper employs the following procudure [13]. The security model employs several checks to determine success or failure -- if any one of these conditions fail, the program logs the failure and exits with an error, otherwise it will continue:

1. Was the wrapper called with the proper number of arguments?

The wrapper will only execute if it is given the proper number of arguments. The proper argument format is known to the Apache web server. If the wrapper is not receiving the proper number of arguments, it is either being hacked, or there is something wrong with the suEXEC wrapper system.

2. Is the user executing this wrapper a valid user of this system?

This is to ensure that the user executing the wrapper is truly a user of the system. A special user name "nobody" or "www" is used, in general.

3. Is this valid user allowed to run the wrapper?

Is this a user allowed to run this wrapper? Only one user (the Apache user) is allowed to execute the program. Direct execution of the wrapper program must be prohibited.

4. Does the target program have an unsafe hierarchical reference?

The target program which contains a leading '/' or has a '..' back-reference is not allowed and the target program must reside within the Apache web space.

5. Is the target user name valid?

Every CGI resource has its owner. To ensure that owner's user name exists in the local system, UNIX password file (/etc/passwd) must be referenced.

6. Is the target group name valid?

The owner of CGI resource can be a member of one or more group(s) . To ensure that owner's group name exists in the local system, UNIX group file (/etc/group) must be referenced.

7. Is the target user superuser?

SUID CGI execution has potential security risk, since the CGI process will have same access privileges as superuser.

8. Is the target userid grater than the minimum ID number?

The minimum user ID number is specified during configuration. This allows to set the lowest possible user ID that will be allowed to execute CGI programs. This is useful to block out "system" accounts.

9. Is the target group the superuser group?

Members in the superuser group has same access privileges as superuser. SUID CGI execution has potential security risk, since the CGI process will have same access privileges as superuser.

10.Is the target group ID grater than the minimum ID number?

The minimum group ID number is specified during configuration. This allows to specify the lowest possible group ID that will be allowed to execute CGI programs. This is useful to block out "system" groups.

11. Can the wrapper successfully become the target user and group?

CGI program acquires the target user and group via setuid and setgid calls. The group access list is also initialized with all of the groups of which the user is a member.

12. Does the directory in which the program resides exist?

Non-existence of directory implies given file name is invalid.

13. Is the directory within the Apache webspace?

If the request is for a regular portion of the server, the requested directory has to be within the server's document root. If the request is for a user directory, usually specified by ~<user_name>, the requested directory has to be within the user's document root such as ~choim/public_html.

14. Is the directory writable by anyone else?

Only the owner user may be able to alter contents of the target directory. It means that the owner of the target directory is supposed to be the only one who can access it.

15. Does the target program exist?

Non-existence of the target CGI program implies that the file does not exist. Thus, there is no means to execute the target CGI program.

16. Is the target program writable by anyone else?

If anyone other than the owner can possibly alter the CGI program, the CGI program can be replaced by a malicious CGI program.

17. Is the target program setuid or setgid?

If the target CGI program is setuid or setgid program, it will change UID and GID of the target CGI process again. This situation is not desirable, since the target CGI program is supposed to be executed on behalf of its owner's IDs.

18. Is the target user/group the same as the program's user/group?

The target user and group name have to be same as target CGI program's user and group name.

19. Can the process environment successfully clean the process environment to ensure safe operations?

A wrapper program designated by suEXEC support cleans the process' environment variables by establishing a safe execution PATH (defined during configuration), as well as only passing through those variables whose names are listed in the safe environment list (also created during configuration).

20. Can the wrapper process successfully become the target program and execute?

The wrapper process executes target CGI program by invoking one of the exec UNIX system calls.

SFW must be compliant on this security model in order that the wrapper program takes advantage of Apache Web server's suEXEC support. CGI resources which are under the control of the SFW CGI transaction environment must be set up on this security model as well.

3. Apache user authentication and access control module

User authentication module of the apache server allows the web administrators to control access to documents on an individual user basis by utilizing user and passwords lists to provide the necessary authentication [5]. When a remote user tries to access a restricted portion of the site, the server requires the user to log in by specifying a username and a password. If the user supplies the proper information, access is granted to that user to across the site without additional login requests. (Although the user does not enter a password, the username and password get re-sent by the browser with each new request to the protected realm.) User authentication based on access control provides more selective security because access permission is validated on a per-user basis. Apache requires login and a password validation before granting access to a restricted portion of web site [13]. It is important to note that there is no correlation between the

UNIX password file (/etc/passwd) and the server's password files; it is not necessary for a remote user to have an account on the local system to be able to access protected materials on a Web server. To provide user authentication, a password file and group file have to be created and maintained properly. By also using a group file, access restrictions based on the user's group memberships can be established. For the Secure and Flexible CGI Wrapper, a built-in resource-user mapping application called sfwmapper creates, modifies and maintains a password file and a group file. This topic will be fully covered in chapter 5.

Host-based access control grants or denies access depending on the Internet Protocol (IP) address of the machine that generated the request. This system is the least intrusive to legitimate users because access is granted on the basis of the machine address. Machines matching a description are allowed or denied access to documents without requesting further information from the client. SFW utilizes user authentication based on host-based access control. More information will be provided in chapter 6.

4. Essential UNIX System Resources and Functions

Functions for user and group accounting

The UNIX system password file contains the fields shown in Table 1. These fields are members of a system defined structure called *passwd* that is defined in <pwd.h> header file [15]. Each line of password file contains the seven fields shown in Table 1, separated by colons. User name is a unique string which identifies user in the local system. There are numerous special purpose user names in UNIX system. Two of them are closely related to this research. First special system user name is *root*. It is a special user name for the super user who is able to access all system resources with no limitation at all.

Table 1 Fields in password file (adopted from [15])

Description	***struct passwd* member**
User name	*char *pw_name*
Encrypted password	*char *pw_passwd*
Numerical user ID	*uid_t pw_uid*
Numerical group ID	*gid_t pw_gid*
Comment field	*char *pw_gecos*
Initial working directory	*char *pw_dir*
Initial shell (user program)	*char *pw_shell*

Second one is *nobody*. This system user name can be used by network servers that allow users login to a system, but with a user ID and group ID that provides no privileges. The only files that users can access with this user ID and group ID are those that are readable or writable by the world.

The encrypted password field contains a copy of the user's password that has been put through a one-way encryption algorithm. The algorithm that is currently used always generates 13 printable characters form the 64-character set [a-zA-Z0-9./] [15]. Some fields in a password file entry can be empty. If the encrypted password field is empty, it usually means the user does not have a password. An empty comment field has no effect.

There are two functions, getpwuid and getpwnam, to fetch entries from the password file. These two functions allow one to look up an entry given a user's login name or user ID. Figure 6 shows prototypes of the functions.

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
struct passwd *getpwnam(const char *name);
```

Figure 6 Functions for the password file (adopted from [15])

The UNIX group file contains the fields shown in Table 2. These fields are contained in a group structure that is defined in <grp.h>.

Table 2 Fields in group file (adopted from [15])

Description	**struct *group* member**
Group name	char *gr_name
Encrypted password	char *gr_passwd
Numerical group ID	int gr_gid
Array of pointers to individual user names	char **gr_mem

The field gr_mem is an array of pointers to the user names that belong to this group. This array is terminated by a null pointer. Either a group name or a numerical group ID can be looked up by the two functions shown in Figure 7.

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrgid(gid_t gid);
struct group *getgrnam(const chat *name);
```

Figure 7 Functions for group file (adopted from [15])

File Information, Type and Permission

Given a path name, the stat function returns a structure of information about the named file. The fstat function obtains information about the file that is already open. The lstat function is similar to stat, but when the named file is a symbolic link, lstat returns information about the symbolic link. The Secure and Flexible Wrapper program utilizes lstat function to avoid following symbolic links. With stat function family, following file information is returned in a system stat structure [15]:

- File type and mode (permissions)
- i-node number
- Device number
- Device number for special files
- Number of links

- User ID of owner
- Group ID of owner
- Size in bytes
- Time information
- Best I/O block size
- Number of blocks allocated

Whenever a remote user accesses CGI program through the Secure and Flexible Wrapper, the wrapper checks target CGI program file with lstat function.

The type of a file is encoded in the st_mode number of the stat structure. The type of file can be tested by system macros such as S_ISREG(), S_ISDIR() and S_ISLNK() [15]. The S_ISREG() macro tests whether or not the given file is a regular file. The S_ISDIR() macro checks if the given file is a directory. Similarly, the S_ISLNK() macro examine if the given file is a symbolic link.

The st_mode value also encodes the access permission bits for the file. There are nine permission bits for each file, divided into three categories. These are shown in Table 3.

Table 3 Nine file access permission bits (adopted from [15])

st_mode mask	Meaning
S_IRUSR	User-read
S_IWUSR	User-write
S_IXUSR	User-execute
S_IRGRP	Group-read
S_IWGRP	Group-write

S_IXGRP	Group-execute
S_IROTH	Other-read
S_IROTH	Other-read
S_IROTH	Other-read

The chmod command and system function is typically used to modify these nine permission bits. The set-user-ID bit and set-group-ID bit of executable files also can be set by chmod command or system function. For example,

$ chmod 4711 sfw<enter>

makes the sfw CGI wrapper program a world executable set-user-ID program. It means that this program will be executed on behalf of owner's access privileges.

Changing User IDs and Group IDs

A running instance of a program is called a process. Every process has at least six IDs associated with it. The real user ID and real group ID identify whom the process really belongs to. Normally these values do not change during a login session, although there are ways for a superuser process to change them. These ways are utilized to change IDs of the requested CGI process. The effective user ID, effective group ID and supplementary group IDs determine file access permissions [15]. These effective IDs reflects who executes the program usually. However, program owner's IDs can be assigned, if the program is set-user-ID or set-group-ID [13]. For example, if the owner of the file is the super user and if the file's set-user-ID bit is set, then while that program file is running as a process, it has super user privileges. This is very important feature, since HTTPD user (normally nobody) can execute the Secure and Flexible Wrapper program associated with root's effective IDs. The saved set-user-ID and saved set-group-ID contain copies of the effective user ID and effective group ID when a program is executed. An unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID.

The real IDs and effective IDs can be changed by functions shown in Figure 8. The setuid function sets real user ID and effective user ID at the same time. Similarly group IDs can be set by setgid function. The seteuid and setegid functions only changes effective IDs.

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
int seteuid(uid_t uid);
int setegid(gid_t gid);              /* returns 0 if OK, -1 on error */
```

Figure 8 Functions for changing real and effective IDs (adopted from [15])

Environment list and CGI Environment Variables

CGI environment variables are a set of special variables that are set in the environment when a CGI program is requested [1]. These variables are accessed via a global array called environment list [15]. Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated string. The address of the array of pointers is contained in the global variable called environ [13, 15]. Following is the definition of this variable:

*extern char *environ;*

Figure 9 illustrates an environment list containing five strings.

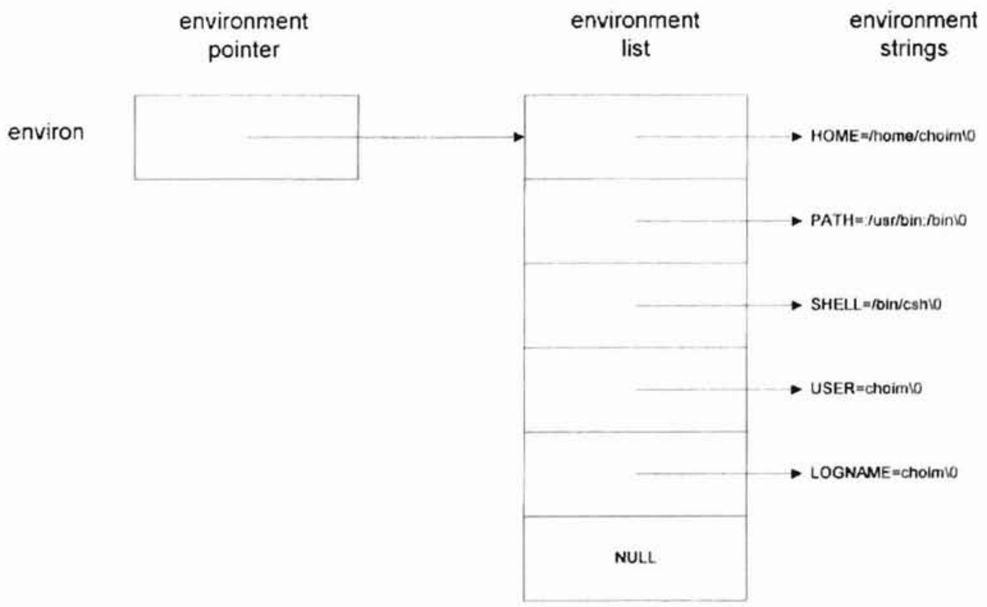


Figure 9 Environment variable list (adopted from [15])

Each string is explicitly terminated by null characters. In addition to it, the environment list is terminated by NULL. By convention the environment consists of *"name=value"* strings, as shown in Figure 9. CGI environment variables also can be obtained by referencing the list. Table 4 summarizes major CGI environment variables.

Table 4 CGI environment variables (adopted from [1])

Environment Variable	Meaning
SERVER_NAME	The host name or IP address on which the CGI program is running, as it appears in the URL.
SERVER_SOFTWARE	The type of server: for example, CERN/3.0 or NCSA/1.3.
GATEWAY_INTERFACE	The version of CGI: for example CGI/1.1
SERVER_PROTOCOL	The version of HTTP: for example HTTP/1.0
SERVER_PORT	The TCP port on which the server is running. Usually port 80 is assigned for WWW servers.
REQUEST_METHOD	POST or GET.
HTTP_ACCEPT	A list of Content-types the browser can accept directly.
HTTP_USER_AGENT	Browser information.
HTTP_REFERER	The URL of the document that this form submission came from.
PATH_INFO	Extra path information, as sent by the browser using the query method of GET in a form.
PATH_TRANSLATED	The actual system-specific path name of the path contained in PATH_INFO.
SCRIPT_NAME	The path name to the CGI executable.
QUERY_STRING	The arguments to the script or the form input (if submitted using GET). QUERY_STRING contains everything after the question mark in the URL.
REMOTE_HOST	The name of the host that submitted the executable. This value cannot be set.
REMOTE_ADDR	The IP address of the host that submitted the executable.
REMOTE_USER	The name of the user that submitted the executable. This value will be set only if server authentication is turned on.

CONTENT_TYPE	In forms submitted with POST, the value
CONTENT_LENGTH	For forms submitted with POST, the number of bytes in the standard input.

chdir, fchdir and getcwd functions

Every process has a current working directory. This directory is where the search for all relative path names starts. Improper setting of current working directory of CGI process implies possible security risk, since same relative path names may be translated into different absolute path names on different current working directory information. Current working directory of the calling process can be changed by calling the chdir or fchdir functions [15].

```
#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int filedes);                /* returns 0 if OK, -1 on error */
char *getcwd(char *buf, size_t size); /* returns buf if OK, NULL on error*/
```

Figure 10 Functions for changing current working directory (adopted from [15])

As shown in Figure 10, either a pathname or an open file descriptor specifies the new current working directory. The absolute path name of current working directory can be retrieved by the getcwd function also. Valid CGI programs are supposed to be under the user document root directory such as ~username/public_html/cgi-bin [3]. These functions mentioned above help to check the validity of the CGI path information.

Replacing current process by the new program with exec function

When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function. The process ID does not change across an exec since a new process is not created but the old process is replaced. The exec function merely replaces the current process with a new process. There are six different exec functions in UNIX system [15]. Table 5 shows the differences between the six exec functions.

Table 5 Differences between six exec functions (adopted from [15])

Function	pathname	filename	Arg list	argv[]	environ	envp[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
letter		p	l	v		e

Almost every CGI wrapper implementation utilizes this technique to replace a properly wrapped CGI wrapper process with a requested CGI program [6, 13]. In that way, the requested CGI process can be executed using the owner's access privileges. The full pathname information, argument list and environment variable list has to be passed to the target CGI program. Thus, *execv* function is considered as appropriate for implementing CGI wrapper program in this research.

5. SFW Resource-User Mapper

The user-resource mapper of the SFW consists of three components, which are SFW password file handler, SFW group file handler and SFW mapping constraint file interpreter. To utilize the user and group authentication mentioned in chapter 3, SFW has to maintain its own password file and group file (/etc/sfw_passwd and /etc/sfw_group) for user and group authentication. Each line of the SFW user-resource mapping constraint file is also interpreted into the global access configuration file (ACF) which is also described in chapter 3.

Each line of the SFW password file consists of a login ID and its password. The encrypted password field contains a copy of the user's password that has been put through a one-way encryption algorithm. The algorithm that is currently used always generates 13 printable characters form the 64-character set [a-zA-Z0-9./] [5, 15]. For example, a line of the SFW password file looks like *"choim:kadKLJREkd234"*.

This file has same structural format as UNIX system password file explained in chapter 4. However, it is used for the SFW user authentication purpose only.

Groups are simply a way of providing an alias for a set of users. Each line of the SFW group file contains a group name and the user IDs that make up that group. For example, consider the line:

mygroup: choim, kim, john

This line defines a group named mygroup and its group members choim, kim and john. It also has same structural format as UNIX group file, but only used for the SFW group authentication purpose.

SFW resource-user mapper interprets a resource-user mapping constraint file and configures the SFW section of the global access control file (ACF) [13] according to the interpretation from the mapper. The resource-user mapping constraint file has information about access controlled SFW CGI resources, users, groups and hosts. For example:

*/home/choim/public_html/cgi-bin *.bad.org kim, lee mygroup*

This line implies following information:

- Absolute pathname of CGI resource is /home/choim/public_html/cgi-bin.
- Access from the domain bad.org to this resource will be denied.

- Access by the users kim and lee is allowed. However those users must provide their user IDs and passwords in order to enter the resource space.
- Access by the group mygroup is allowed. However members of the group must provide their user IDs and passwords in order to enter the resource space.

Obviously, user names and group names have to be pre-defined in the SFW password file and group file. Finally, SFW resource-user mapper configures the global ACF based on the interpretation of the resource-user mapping constraint file. Figure 11 Illustrates SFW resource-user mapper design.

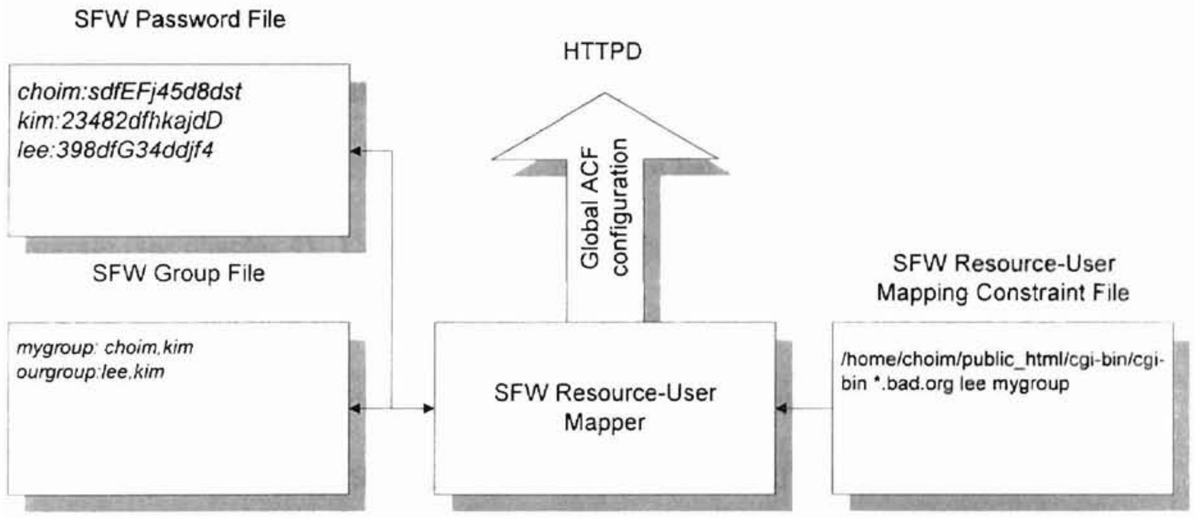


Figure 11 SFW mapper design

6. CGI Transaction Procedure on SFW

When a remote user requests CGI transaction to HTTPD, the server retrieves pathname of target CGI program, its owner's user ID, its owner's group ID and argument list for the CGI program. Then, the server executes SFW and passes these four data as arguments to the SFW process. SFW process, now, follows the security check procedure described in the suEXEC security model (see chapter 2) [13]. If any security check fails, SFW discards current CGI transaction request and logs error messages to SFW log file (/var/log/httpd/sfw_log). HTTPD also sends an error message ,such as internal server error, to the remote user's browser. If every security check has been successfully passed, SFW process changes its user IDs and group IDs same as the owner's. Then, it executes the target CGI program with exec system function. The *execv* system function is preferable since the argument list and environment list has to be passed to the target CGI program (see chapter 4). Upon execution of the target CGI program, the global ACF and the SFW resource-user mapping constraint files are referenced. If the target CGI program is access controlled resource, the remote user will be prompted by the authentication module to provide correct SFW user name and password. If the remote user entered correct user ID and correct password, the requested CGI program is executed and the user gets the result from it. Note that successful SFW CGI transactions are also logged in the SFW log file. Figure 12 Illustrates SFW CGI transaction system. Gray filled components of the system are implemented as a part of this research.

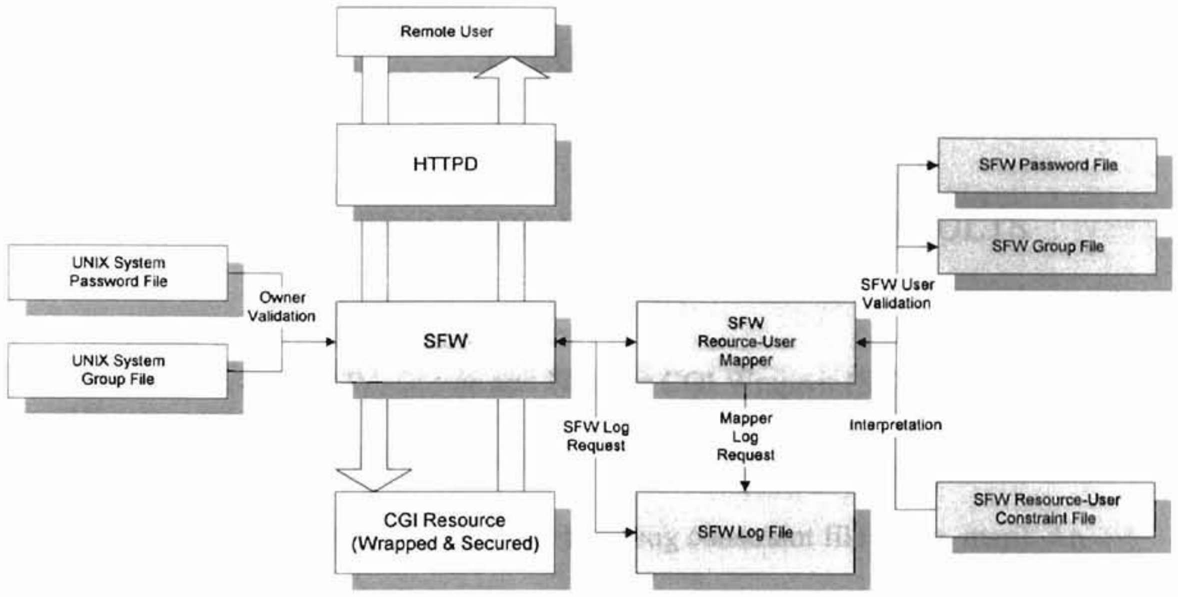


Figure 12 SFW CGI Transaction System

CHAPTER V

SECURE AND FLEXIBLE CGI WRAPPER: RESULTS

As discussed in chapter IV, Secure and Flexible CGI Wrapper (SFW) utilizes three system files, which are a SFW user password file (/etc/sfw_passwd), a SFW group file (/etc/sfw_group) and a SFW resource-user mapping constraint file (./r_u_map). An example SFW user password file is given in Figure 13. It has four users and their one-way encrypted passwords. Remote users who want to access restricted CGI resources must use those user names and corresponding passwords in order to be authenticated.

```
#
# SFW Password File
# /etc/sfw_passwd
# Format : <User_name>:<Encrypted_password>
#

kim:UQS/IE4pau4Gk
lee:IqvPLAeB.Ynks
park:dgUVxW0xwtZ6I
joe:EIWNnjno0r1M.
```

Figure 13 Example SFW user password file

An example SFW group file is shown in Figure 14. It defines three SFW user groups. Groups are simply a way of providing an alias for a set of users. Group information is not required by SFW, but users can be easily grouped with it.

```
#
# SFW Group File
# /etc/sfw_group
# Format : <Group_name>:<User_name_list>
#

team: kim, lee
party: lee, park
project: kim, lee, joe
```

Figure 14 Example SFW group file

An example SFW resource-user mapping constraint file is shown in Figure 15. It indicates that four CGI resources are under access control of SFW. The authentication behavior of SFW will be determined by these mapping constraints. For example, the first line implies that user "kim" and "lee" will be authenticated when they access to the CGI resource "/home/choim/public_html/cgi-bin".

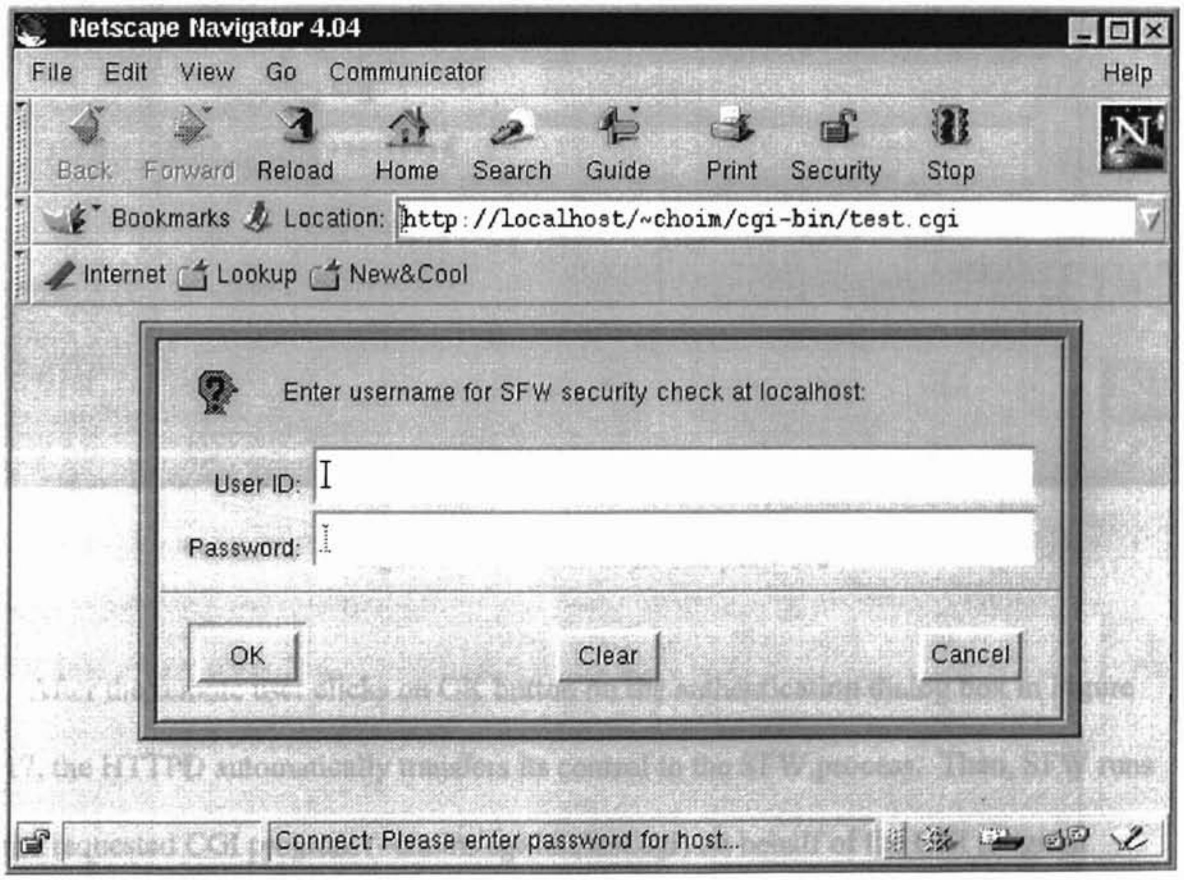
```
#
# SFW Resource-User Map File
# Format : <CGI_resource_name> <Host_name_list> <User_name_list> <Group_name_list>
# Remark : - indicates null list
#

/home/choim/public_html/cgi-bin - kim,lee -
/home/minsu/public_html/cgi-bin *.bad.org - party
/home/john/public_html/cgi-bin - - project
/home/jane/public_html/cgi-bin/local_cgi/ - - project
```

Figure 15 Example SFW resource-user map files

Note that resource names are absolute path names of directories or files of the local system. Their URLs will vary. For example, an absolute pathname /home/choim/public_html/cgi_bin is equivalent to http://localhost/~choim/cgi-bin.

When a remote user accesses a CGI resource controlled by SFW, the user will be prompted to enter his SFW user name and password as illustrated in Figure 16. If the user accesses CGI resources that do not require authentication , she/he will not be prompted to provide access authentication information.


Netscape Navigator 4.04

Location: http://localhost/~choim/cgi-bin/test.cgi

Enter username for SFW security check at localhost:

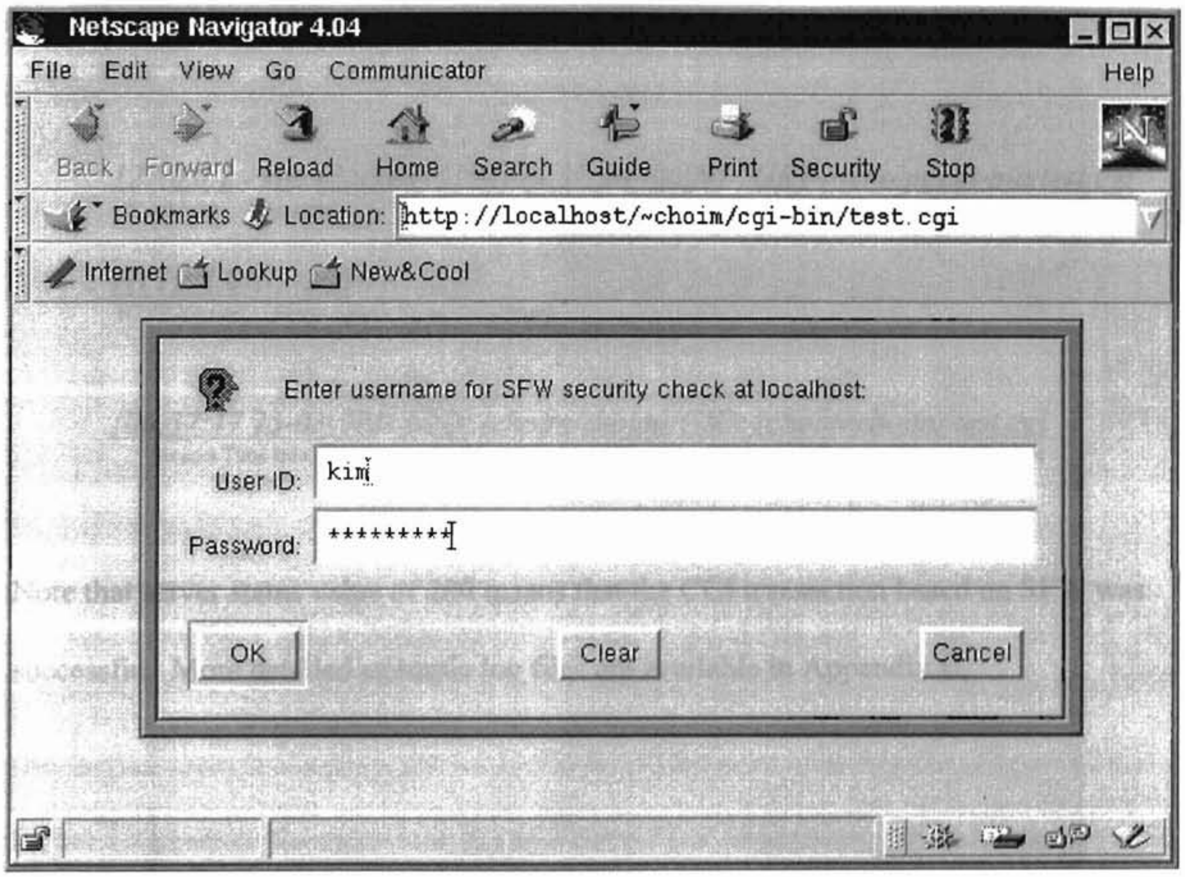
User ID:

Password:

Connect: Please enter password for host...

Figure 16 User name and password required by SFW mapper

As specified in the SFW resource-user mapping constraint file in Figure 15, SFW user "kim" and "lee" are only allowed access. For illustration, user name "kim" and its correct password are entered as shown in Figure 17.


Netscape Navigator 4.04
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: http://localhost/~choim/cgi-bin/test.cgi
Internet Lookup New&Cool
OK Clear Cancel

Enter username for SFW security check at localhost:

User ID: kim

Password: *********

Figure 17 Valid user name and password

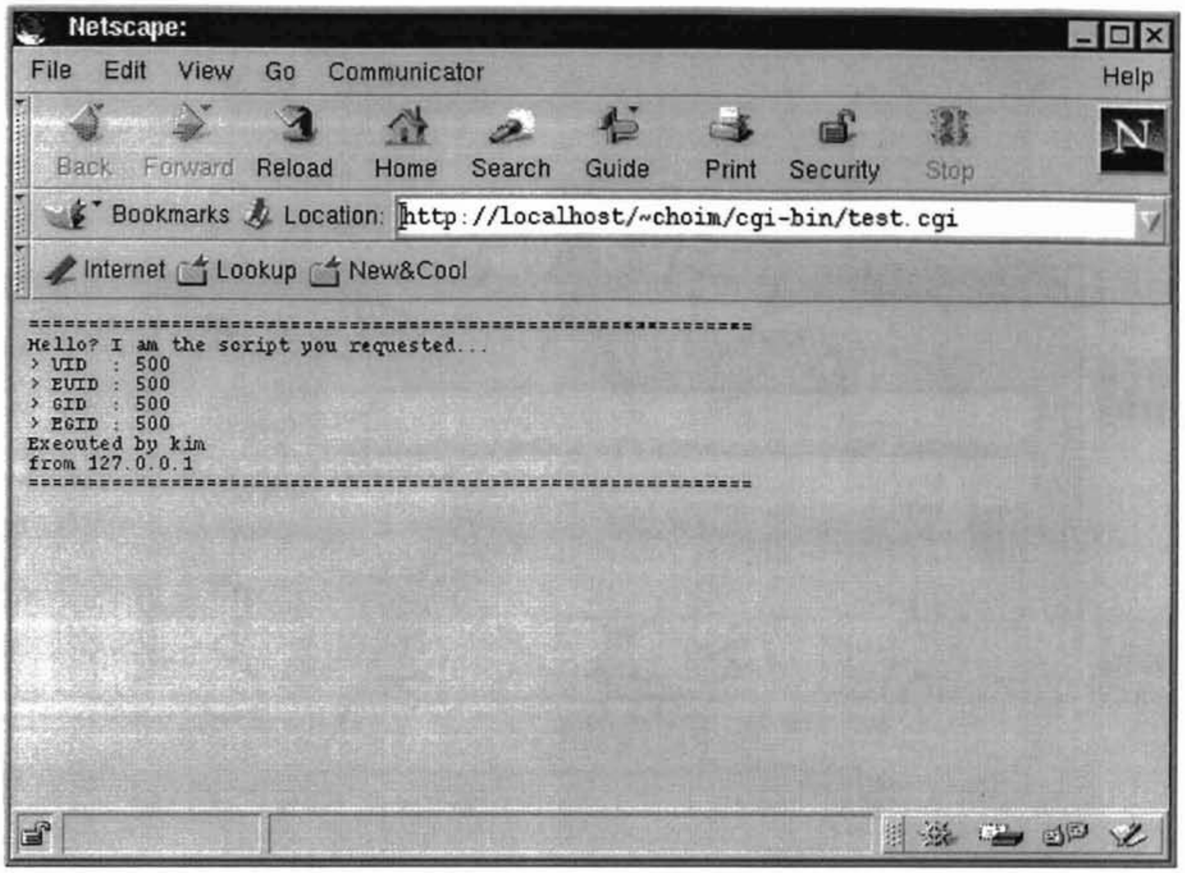
After the remote user clicks on OK button on the authentication dialog box in Figure 17, the HTTPD automatically transfers its control to the SFW process. Then, SFW runs the requested CGI program (~choim/cgi-bin/test.cgi) on behalf of the CGI program owner. The requested CGI program "test.cgi" in this example is actually a simple CGI script that shows its IDs and two CGI environment variables, REMOTE_USER and REMOTE_HOST. The information displayed is shown in Figure 18. It is very obvious that those user IDs and group IDs 500 are same as the CGI resource owner's user IDs and group IDs. Following lines of the server log file and SFW log file show that the CGI program was successfully wrapped and executed, and the result was successfully transmitted back to the remote user's browser.

```
127.0.0.1 - kim [25/Mar/1998:06:37:20 -0600] "GET /~choim/cgi-bin/test.cgi
IP Address  Username     Access Time Info      Transaction Method  CGI Location

HTTP/1.0"     200          247
HTTP Version  Server Status  Bytes Transferred


[06:37:17 25-03-98]: UID: (choim/choim) GID: (choim/choim) test.cgi
  Access Time Info     Real and Effective UID  Real and Effective GID  CGI name
```

Note that server status value of 200 means that the CGI transaction based on SFW was successful. More detailed example log files are available in Appendix C.


Netscape:
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: http://localhost/~choim/cgi-bin/test.cgi
Internet Lookup New&Cool

```
============================================================
Hello? I am the script you requested...
> UID  : 500
> EUID : 500
> GID  : 500
> EGID : 500
Executed by kim
from 127.0.0.1
============================================================
```

Figure 18 SFW CGI transaction result

What if the remote user enters an invalid user name and/or password? In that case, CGI transaction request should be denied by the SFW authentication module. Figure 19 and Figure 20 illustrate this situation. An user name "joe" is not supposed to access the CGI resource "~choim/cgi-bin/test.cgi". According to the SFW map file, only two users "kim" and "lee" can utilize the resource. Thus, SFW user "joe" will not be authorized. The remote user's access request will be denied.

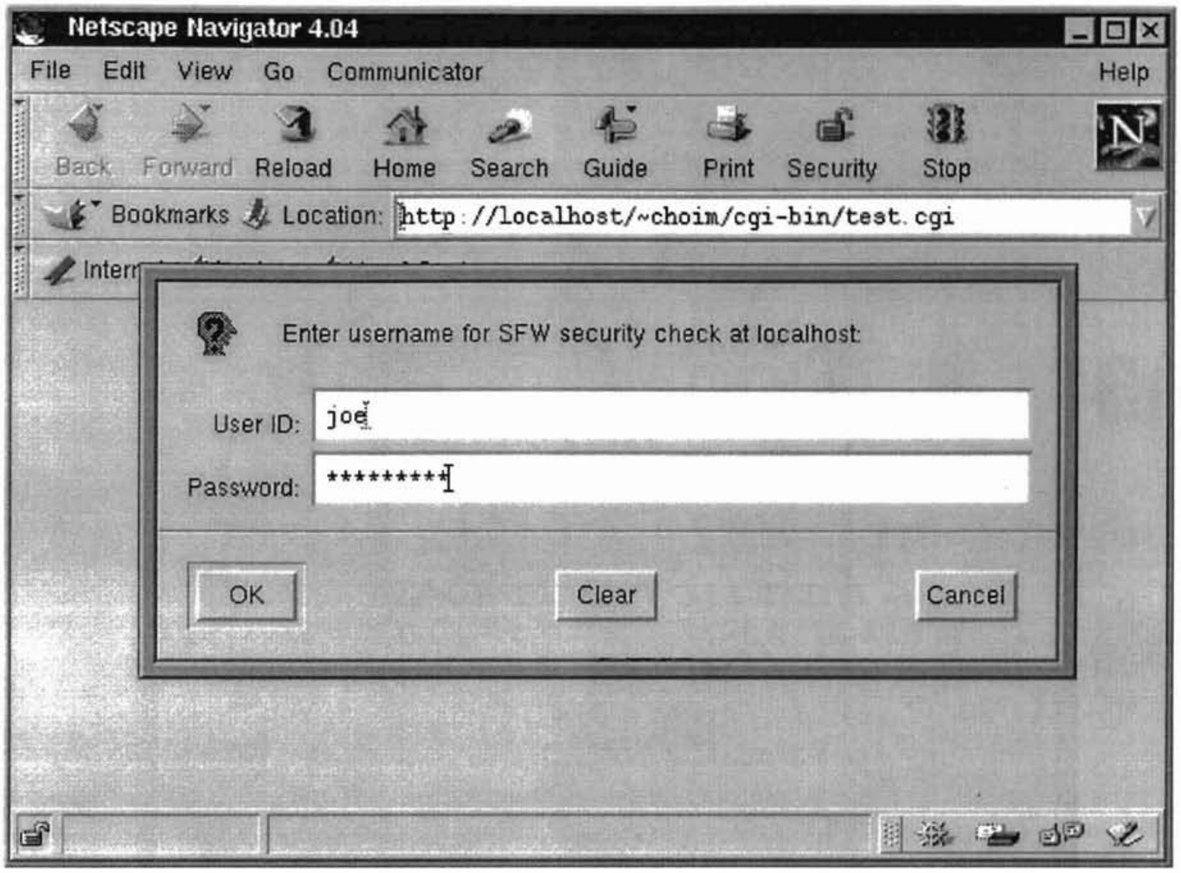

Netscape Navigator 4.04
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: http://localhost/~choim/cgi-bin/test.cgi
Enter username for SFW security check at localhost:
User ID: joe
Password: *********
OK Clear Cancel

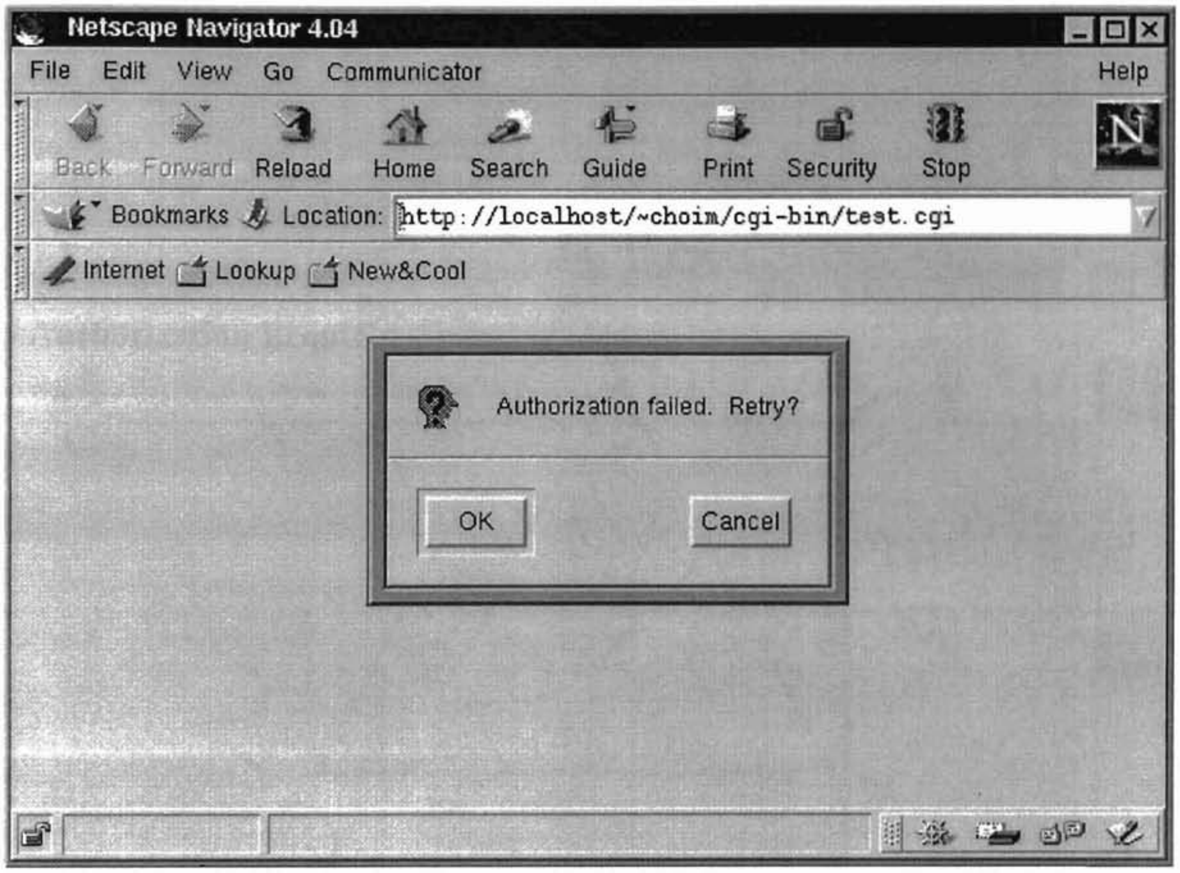
Figure 19 Invalid user name and password

The remote user may retry to enter correct user name and password as many times as he wants. If the user decides not to retry anymore, however, server error message will be transmitted back to the browser as shown in Figure 21. Following line from the server log file shows that the requested CGI transaction was not too successful.

```
127.0.0.1 - joe [25/Mar/1998:06:38:31 -0600] "GET /~choim/cgi-bin/test.cgi
IP Address  Username   Access Time Info       Transaction Method  CGI Location

HTTP/1.0"   401            350
HTTP Version  Server Status  Bytes Transferred
```

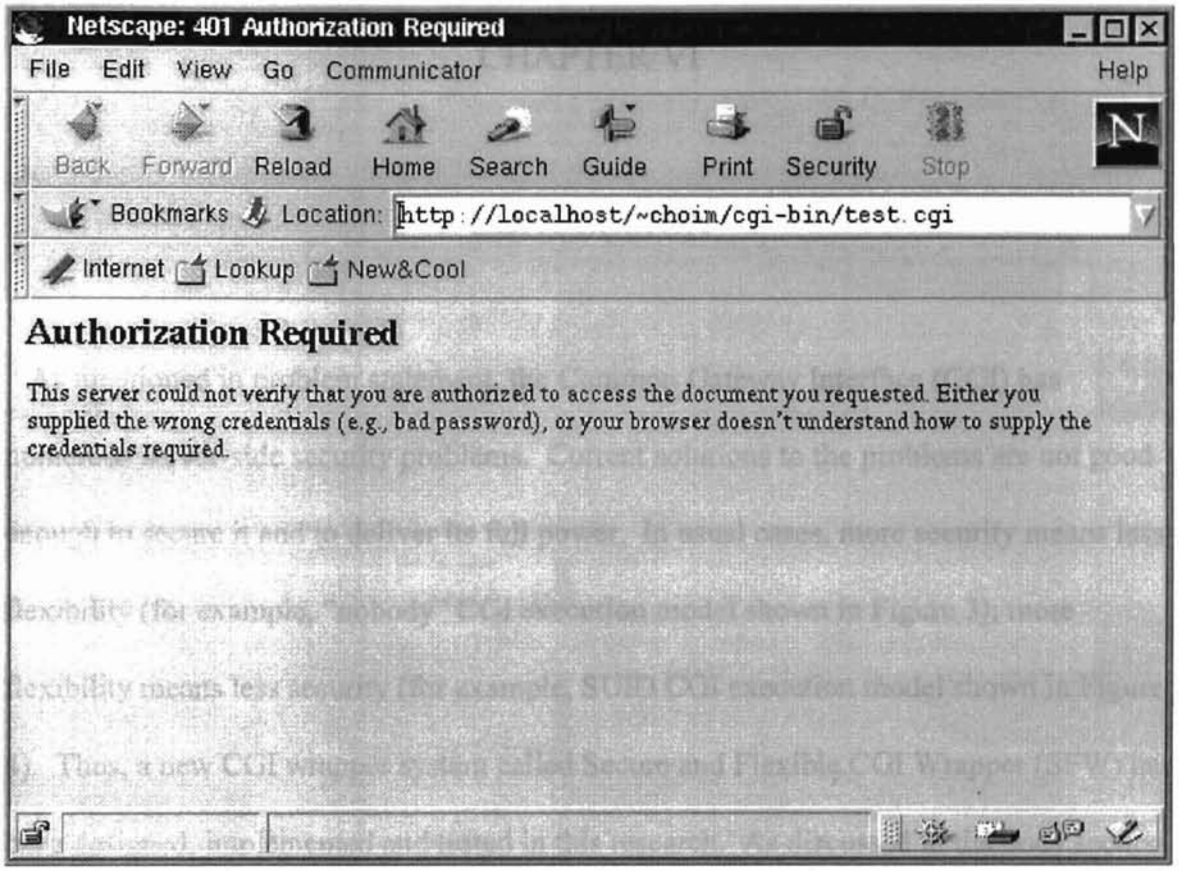
The server status code 401 means that user authentication failed.


Netscape Navigator 4.04
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location:
Internet Lookup New&Cool
OK Cancel

```
http://localhost/~choim/cgi-bin/test.cgi
```

Authorization failed. Retry?

Figure 20 Access denied


Netscape: 401 Authorization Required
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: http://localhost/~choim/cgi-bin/test.cgi
Internet Lookup New&Cool

Authorization Required

This server could not verify that you are authorized to access the document you requested. Either you supplied the wrong credentials (e.g., bad password), or your browser doesn't understand how to supply the credentials required.

Figure 21 Authorization error message

CHAPTER VI

CONCLUSION

As mentioned in problem statement, the Common Gateway Interface (CGI) has numerous server-side security problems. Current solutions to the problems are not good enough to secure it and to deliver its full power. In usual cases, more security means less flexibility (for example, "nobody" CGI execution model shown in Figure 3); more flexibility means less security (for example, SUID CGI execution model shown in Figure 4). Thus, a new CGI wrapper system called Secure and Flexible CGI Wrapper (SFW) has been designed, implemented and tested in this research. As discussed in the body section of this thesis, the new CGI wrapper system is capable of performing mandatory security checks based on the suEXEC security model. It executes a target CGI program with its owner's IDs in response to authorized remote users, based on pre-configured resource-user mapping constraints information. SFW system has numerous advantages over previous solutions discussed in the problem statement of this paper. SFW can be maintained with minimal cost and maintenance since numbers of features are effectively integrated in one system and its configuration procedures are significantly simple and straight forward. SFW is able to significantly enhance server-side security since it has a mandatory security checking routine and a dedicated user authentication module and CGI resource wrapping capability. Besides, source codes in C programming language can be conveniently ported for a number of different UNIX based machines. Thus, SFW, as a trusted gateway agent [8], easily establishes more secure and flexible CGI transaction environment for sure. Figure 22 illustrates a properly configured CGI transaction environment based on the SFW suite; gray filled portion of the figure was implemented as a part of this research. Source codes in C are available in Appendix A; example configuration files are listed in Appendix B; example SFW log files are placed in Appendix C.

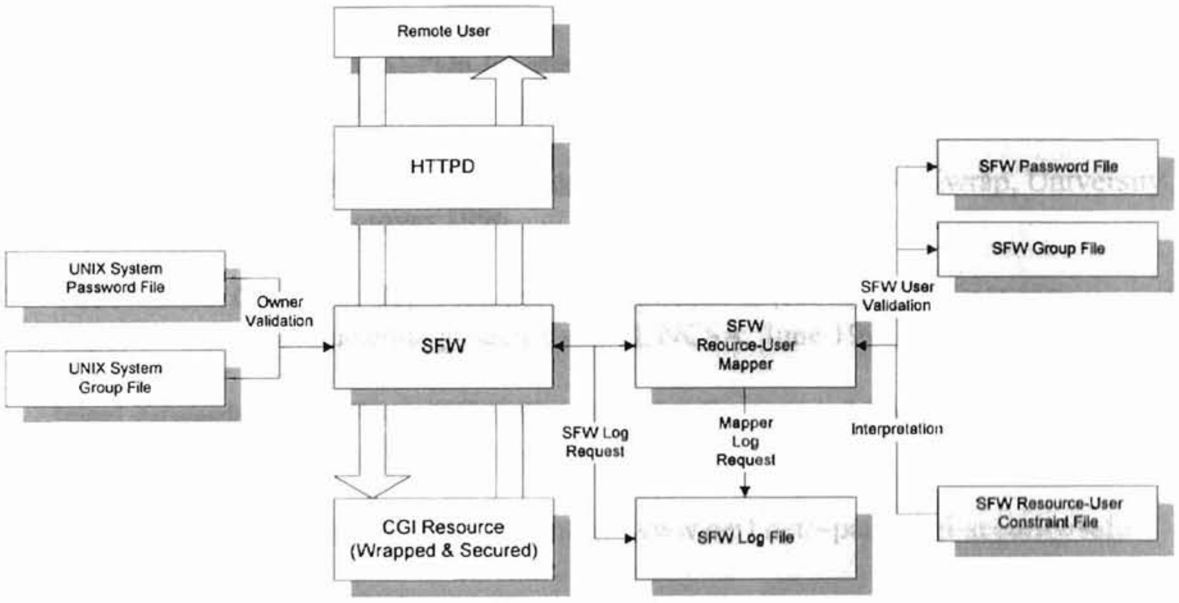


Figure 22 SFW CGI transaction environment

References

1. D. R. T Robinson, "The Common Gateway Interface Version 1.1", Internet Draft, University of Cambridge, February 1996.

2. Douglas E. Comer, Internetworking with TCP/IP Volume 1, Prentice Hall, April 1995

3. Fielding, R., "Relative Uniform Resource Locators", Request for comments(RFC) 1808, June 1995

4. Lincoln D. Stein, The World Wide Web Security FAQ, http://www.genome.wi.mit.edu/WWW/faqs/www-security-faq.html, Massachusetts Institution of Technology (MIT) , February 1997

5. N. Haller and R. Atkinson , "On Internet Authentication", RFC 1704, Bell Communications Research and Naval Research Laboratory, Octover 1994

6. Nathan Neulinger, CGIWrap Home Page, http://www.umr.edu/~cgiwrap, University of Missouri at Rolla, Octover 1996

7. The National Center for Supercomputing Applications (NCSA) , CGI security, http://hoohoo.ncsa.uiuc.edu/cgi/security.html, NCSA, June 1996

8. Nigel Edwards, Owen Rees, "High Security Web Servers and Gateways", The 6th International WWW conference proceedings, June 1996

9. Paul Phillips, The CGI security FAQ, http://www.cerf.net/~paulp/cgi-security/safe-cgi.txt, Octover 1996

10. T. Berners-Lee and D. Connolly, "Hypertext Markup Language - 2.0", MIT/W3C, November 1995

11. T. Berners-Lee, "Universal Resource Identifiers (URI) in WWW", RFC 1630, CERN, June 1994

12. T. Berners-Lee, L. Masinter, and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, CERN, Xerox PARC and University of Minnesota, December 1994.

13. The Apache Group, Apache HTTP Server Version 1.3 User's Guide, The Apache Group, January 1998

14. Tom Christiansen and Shishir Gundavaram, Perl CGI FAQ, http://www.perl.com/perl/faq/perl-cgi-faq.html, June 1996

15. W.Richard Stevens, Advanced Programming in the UNIX Environment, Addison-Wesley, June 1992

APPENDIX A

SFW SOURCE CODES IN C

```
#
# Make file for secure & flexible wrapper
#
# To compile, type "make".
# To clean up, type "make clean".
#
# By Minsu Choi
# Graduate Student
# Computer Science Dept.
# Oklahoma State University
# Stillwater, Oklahoma
#

all : sfw sfwmapper
.PHONY : all

sfw : sfw.o
      cc -o sfw sfw.o

sfwmapper : sfwmapper.o
      cc -o sfwmapper sfwmapper.o

sfw.o : sfw.c sfw.h
      cc -c sfw.c

sfwmapper.o : sfwmapper.c sfwmapper.h
      cc -c sfwmapper.c

clean :
      rm -f *.o core
```

```
/*
 * Secure & Flexible Wrapper
 * Header file for sfw.c
 *
 * by Minsu Choi
 * Graduate Student
 * Computer Science Dept.
 * Oklahoma State Uiversity
 * Stillwater, Oklahoma
 */

/*
 * Secure and Flexible CGI Wrapper header file
 */

#include <stdio.h>
#include <ctype.h>
#include <sys/stat.h>          /* for file permission stating */
#include <sys/param.h>         /* for parameter manipulation */
#include <errno.h>             /* for error handling */
#include <stdlib.h>            /* for standard library functions */
#include <stdarg.h>            /* for standard arguments */
#include <pwd.h>               /* forpassword file handling */
#include <grp.h>               /* forpassword file handling */
#include <unistd.h>            /* for setting user id */
#include <sys/types.h>         /* types for uid */
#include <string.h>            /* for string manipulation */
#include <time.h>              /* time for logging */
#include <syslog.h>            /* for system log file generation */

/* Define who runs httpd. Usually nobody */
#define  HTTPD_USER   "nobody"

/* CGI path */
#define  CGI_PATH     "/public_html/cgi-bin/"
```

```
/* SFW log file location */
#define  LOG_SFW      "/var/log/httpd/sfw_log"

/* user document root */
#define  USER_DIR     "public_html"

/* system document root */
#define  DOC_ROOT     "/home/httpd/html"

/* safe path info */
#define  SAFE_PATH     "/usr/local/bin:/usr/bin:/bin"

/* minimum UID and GID (lower IDs are assigned system users) */
#define  MIN_UID      500
#define  MIN_GID      500

/* maximum length of path */
#define  MAX_PATH     8192

/* maximum number of environment variables */
#define  ENV_BUFF     256

extern char **environ;    /* environment variable list */
static FILE *log_file;    /* SFW log file */

/* environment variable templete from apache's suEXEC model */
char *safe_env_lst[] =
{
    "AUTH_TYPE",
    "CONTENT_LENGTH",
    "CONTENT_TYPE",
    "DATE_GMT",
    "DATE_LOCAL",
    "DOCUMENT_NAME",
    "DOCUMENT_PATH_INFO",
    "DOCUMENT_ROOT",
    "DOCUMENT_URI",
    "FILEPATH_INFO",
    "GATEWAY_INTERFACE",
    "LAST_MODIFIED",
    "PATH_INFO",
    "PATH_TRANSLATED",
    "QUERY_STRING",
    "QUERY_STRING_UNESCAPED",
    "REMOTE_ADDR",
    "REMOTE_HOST",
    "REMOTE_IDENT",
    "REMOTE_PORT",
    "REMOTE_USER",
    "REDIRECT_QUERY_STRING",
    "REDIRECT_STATUS",
    "REDIRECT_URL",
    "REQUEST_METHOD",
    "SCRIPT_FILENAME",
    "SCRIPT_NAME",
    "SCRIPT_URI",
    "SCRIPT_URL",
    "SERVER_ADMIN",
    "SERVER_NAME",
    "SERVER_PORT",
    "SERVER_PROTOCOL",
    "SERVER_SOFTWARE",
    "USER_NAME",
    "TZ",
    NULL
};
```

```
/*
 * Secure & Flexible Wrapper
 * Source code in C
 *
 * by Minsu Choi
 * Graduate Student
 * Computer Science Dept.
 * Oklahoma State Uiversity
 * Stillwater, Oklahoma
 */

/* header file of header files */
#include "sfw.h"

/* writes message into log file */
void log_msg(const char *format_str, ...)
{
    va_list     v_list;       /* varable argument list */
    time_t curr_time;         /* numerical format current time */
    struct tm *local_time;    /* structured current time */

    /* open log file if it is closed */
    if (!log_file)
      if ((log_file = fopen(LOG_SFW, "a")) == NULL) {
          fprintf(stderr, "failed to open log file\n");
          perror("fopen");
          exit(1);
      }

    /* prepare to get variable argument list */
    va_start(v_list, format_str);

    /* get current time in numerical format */
    time(&curr_time);

    /* convert it into human recognizable time */
    local_time = localtime(&curr_time);

    /* write time info into the log file */
    fprintf(log_file, "[%.2d:%.2d:%.2d %.2d-%.2d-%.2d]: ",
            local_time->tm_hour, local_time->tm_min,
          local_time->tm_sec, local_time->tm_mday,
            (local_time->tm_mon + 1), local_time->tm_year);

    /* write messages into the log file */
    vfprintf(log_file, format_str, v_list);

    /* flush log file pointer */
    fflush(log_file);

    /* specifies end of variable argument list */
    va_end(v_list);
    return;
}

void main(int argc, char *argv[])
{
   char *user_name;        /* user name */
   char *group_name;       /* group name */

   char *real_user_name;   /* real user name */
   char *real_group_name;  /* real group name */

   char *script_name;      /* script name */
                           /* ex : test.cgi */

   char *owner_home;       /* script owner's home dir */
                           /* ex : /home/choim */

   char *script_path;      /* path where scipt exists */
                           /* ex : /home/choim/public_html/cgi-bin */
```

```
char *script_full_path; /* script_path + script_name */
                        /* ex : /home/choim/public_html/
                                cgi-bin/test.cgi*/

char cwd[MAX_PATH];     /* current working directory */
char dwd[MAX_PATH];     /* document working directory */

int user_dir_flag=0;    /* checks ~ */

uid_t sfw_UID;          /* UID of secure & flexible wrapper */
uid_t UID;              /* UID of owner */
uid_t GID;              /* GID of owner */

struct passwd *owner;   /* password information structure
                           for owner */
struct passwd *execute_user; /* for executer of script */
struct group *group;     /* for executer of script */

struct stat path_stat;   /* for directory stating */
struct stat script_stat; /* for file stating */

char tmp_path[512];      /* temporary storage for path info */
char **safe_env;         /* safe environment variable pointer */
char **env_ptr;          /* temp env pointer */
int  env_index;          /* index for checking env var list */
int  tmp_index;          /* same as above */

/* apache server passes 4 arguments user name, group name
   CGI program name and arguments to the program */
if(argc!=4)
{
   log_msg("incorrect number of arguments.\n");
   exit(101);
};

/* retrive script name */
```

```
script_name = strdup(argv[3]);

/* figure out who owns script */
user_name = strdup(argv[1]);
group_name = strdup(argv[2]);

/* get UID of sfw process */
sfw_UID = getuid();

/* figure out who wants to execute CGI script */
if(!(execute_user=getpwuid(sfw_UID))) {
   /* executer not found in password file */
   log_msg("Executer of script is not found in password file\n");
   exit(102);
};

/* script name check
   contains leading /?
   contains leading ../?
   contains any ..? */
if((script_name[0]=='/')||
   (!strncmp(script_name,"../",3))||
   (strstr(script_name,"/../") != NULL))
{
   log_msg("script name check failure (%s)\n", script_name);
   exit(104);
};

/* HTTPD user and executor are same? */
if(strcmp(HTTPD_USER,execute_user->pw_name))
{
   log_msg("user mismatch (%s/%s)\n",HTTPD_USER,
                              execute_user->pw_name);
   exit(103);
};
```

```
/* user name check
   contains leading ~? */
if(user_name[0]=='~')
{
   user_name = strdup(++user_name);
   user_dir_flag = 1;
};

/* check password info of owner */
if(!(owner=getpwnam(user_name))) {
  /* owner not found */
   log_msg("Owner of script not found in passwd file (%s).\n",
           user_name);
   exit(105);
};

/* check group info of owner */
if(strlen(group_name)!=strspn(group_name,"0123456789"))
{
   if((group=getgrnam(group_name))==NULL)
   {
   /* group not found */
   log_msg("Group of script not found in group file (%s).\n",
           group_name);
   exit(106);
   };

   GID=group->gr_gid;
   real_group_name=strdup(group->gr_name);
}
else   /* numerical group name */
{
   GID=atoi(group_name);
   real_group_name=strdup(group_name);
};
```

```
/* get UID, real user name and user's home dir */
UID=owner->pw_uid;
real_user_name=strdup(owner->pw_name);
owner_home=strdup(owner->pw_dir);

/* log request */
log_msg("UID: (%s/%s) GID: (%s/%s) %s\n",user_name,real_user_name,
         group_name,real_group_name,script_name);

/* user ID minimum check */
if((UID==0)||(UID<MIN_UID))
{
   log_msg("minimum UID check failed (%d/%s)\n",UID,script_name);
   exit(107);
};

/* group ID minimum check */
if((GID==0)||(GID<MIN_GID))
{
   log_msg("minimum GID check failed (%d/%s)\n",GID,script_name);
   exit(108);
};

/* Now, change UID and GID with setuid and setgid */
if(setgid(GID)!=0)
{
   log_msg("setgid failed (%d/%s)\n",GID,script_name);
   exit(109);
};

if(setuid(UID)!=0)
{
   log_msg("setuid failed (%d/%s)\n",UID,script_name);
   exit(110);
};
```

```
/* get the current working directory */
if(getcwd(cwd,MAX_PATH)==NULL)
{
   log_msg("cannot get current working directory\n");
   exit(111);
};

/* path check */
if(user_dir_flag)    /* user dir */
{
   if(((chdir(owner_home))!=0)||          /* go to home dir first */
      ((chdir(USER_DIR))!=0)||            /* and then public_html */
      ((getcwd(dwd,MAX_PATH))==NULL)||    /* and then get
                                             document dir */
      ((chdir(cwd))!=0))                  /* then go to current dir */
   {
      log_msg("cannot get document root info (%s)\n",owner_home);
      exit(112);
   };
}
else
{
   if(((chdir(DOC_ROOT))!=0)||            /* go to system doc root */
      ((getcwd(dwd,MAX_PATH))==NULL)||    /* get the path */
      ((chdir(cwd))!=0))                  /* then go to current dir */
   {
      log_msg("cannot get document root info (%s)\n",DOC_ROOT);
      exit(113);
   };
};

/* comapre suffix of current working directory and
   document working directory */
if((strncmp(cwd,dwd,strlen(dwd)))!=0)
{
   log_msg("CGI executable not in docroot(%s/%s)\n",cwd,dwd);
```

```
    exit(114);
};

/* stat the current working directory */
if(((lstat(cwd,&path_stat))!=0)||
    /* make sure it is directory */
    !(S_ISDIR(path_stat.st_mode)))
{
    log_msg("dirctory stat failed: (%s)\n",cwd);
    exit(115);
};

/* check current directory is not writable by other */
if((path_stat.st_mode&S_IWOTH)||
    (path_stat.st_mode&S_IWGRP))
{
    log_msg("directory is writable by others: (%s)\n",cwd);
    exit(116);
};

/* check CGI program file */
if(((lstat(script_name,&script_stat))!=0)||
    /* make sure it is not symbolic link */
    (S_ISLNK(script_stat.st_mode)))
{
    log_msg("CGI program stat failed: (%s)\n",script_name);
    exit(117);
};

/* check CGI program is not writable by other */
if((script_stat.st_mode&S_IWOTH)||
    (script_stat.st_mode&S_IWGRP))
{
    log_msg("CGI program is writable by others: (%s/%s)\n",cwd,
              script_name);
    exit(118);
```

```
   };

   /* check CGI program is setuid or setgid */
   if((script_stat.st_mode&S_ISUID)||  /* setuid? */
      (script_stat.st_mode&S_ISGID))   /* setgid? */
   {
      log_msg("CGI program is setuid or setgid:
(%s/%s)\n",cwd,script_name);
      exit(119);
   };

   /* target UID and GID check */
   if((UID!=path_stat.st_uid)||
      (GID!=path_stat.st_gid)||
      (UID!=script_stat.st_uid)||
      (GID!=script_stat.st_gid))
   {
      log_msg("target UID/GID (%ld/%ld)",UID,GID);
      log_msg(" mismatch with directory (%ld/%ld)",path_stat.st_uid,
                 path_stat.st_gid);
      log_msg(" or rpogram (%ld/%ld)",script_stat.st_uid,
                 script_stat.st_gid);
      exit(200);
   };

   /* check CGI environment variables */
   env_index=0;

   /* dynamic allocation of environment variable pointer list */
   if((safe_env=(char **)calloc(ENV_BUFF,sizeof(char *)))==NULL)
   {
      log_msg("dynamic allocation of env list failed\n");
      exit(120);
   };

   for(env_ptr=environ;*env_ptr&&env_index<ENV_BUFF;env_ptr++)
```

```
{
   if(!strncmp(*env_ptr,"HTTP_",5))  /* variables with leading
                                        HTTP_ are just skipped */
   {
      safe_env[env_index]=*env_ptr;
      env_index++;
   }
   else
   {
      for(tmp_index=0;safe_env_lst[tmp_index];tmp_index++)
      {
         if(!strncmp(*env_ptr,safe_env_lst[tmp_index],
             strlen(safe_env_lst[tmp_index])))
         {
            /* check and move one by one */
            safe_env[env_index]=*env_ptr;
            env_index++;
            break;
         };
      };
   };
};

/* modify path info environment variable to block
   malicious execution */
sprintf(tmp_path,"PATH=%s",SAFE_PATH);
safe_env[env_index]=strdup(tmp_path);
safe_env[++env_index]=NULL;

/* copy safe list back to original */
environ=safe_env;

/* close log file */
fclose(log_file);
/* assign NULL to make sure that it is closed */
log_file=NULL;
```

```
    /* execute requested CGI program with exec system call */
    execv(script_name,&argv[3]);

    /* if exec fails, log it */
    log_msg("exec failed (%s)\n",script_name);
    exit(255);
}
```

```
/*
 * Resource-User Mapper for SFW
 * Header file for sfwmapper.c
 *
 * by Minsu Choi
 * Graduate Student
 * Computer Science Dept.
 * Oklahoma State Uiversity
 * Stillwater, Oklahoma
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define HTPASSWD_FILE "/etc/sfw_passwd"
#define HTGROUP_FILE "/etc/sfw_group"
#define TMP_FILE "/tmp/.sfw_group_tmp"
#define ACCESS_CONF "/etc/httpd/conf/access.conf"
#define HTPASSWD "/usr/sbin/htpasswd"

#define LF 10
#define CR 13

#define MAX_ARRAY_LEN 300
#define MAX_STRING_LEN 256
```

```
/*
 * Resource-User Mapper for SFW
 * Source code in C
 *
 * by Minsu Choi
 * Graduate Student
 * Computer Science Dept.
 * Oklahoma State Uiversity
 * Stillwater, Oklahoma
 */

#include "sfwmapper.h"

main(int argc, char *argv[])
{
   FILE *g_file;                /* SFW group file */
   FILE *tmp_file;              /* temporary flie */
   FILE *map_file;              /* SFW map file */
   FILE *conf_file;             /* SFW configuration file */

   char tmp_str[MAX_STRING_LEN];           /* temporary strings */
   char tmp1[MAX_STRING_LEN];
   char tmp2[MAX_STRING_LEN];
   char tmp3[MAX_STRING_LEN];
   char tmp4[MAX_STRING_LEN];

   char m_string[MAX_ARRAY_LEN][MAX_STRING_LEN];     /* map strings */
   char host_list[MAX_ARRAY_LEN][MAX_STRING_LEN];    /* list of hosts */
   char user_list[MAX_ARRAY_LEN][MAX_STRING_LEN];    /* list of users */
   char group_list[MAX_ARRAY_LEN][MAX_STRING_LEN];   /* list of groups */

   char *found_group;           /* matched group name */
   char tmp_char;               /* temporary character */

   int i,j,k;                   /* index */
   int num_host;                /* number of hosts */
int num_user;                /* number of users */
int num_group;               /* number of groups */

int m_flag;                  /* flag */

/* this program is to have 3 arguments
   name, option and third argumnt */
if(argc!=3)
{
   printf("Invalid number of arguments.\n");
   exit(1);
};

/* examine option */
switch(argv[1][1])           /* options */
{
   /* option -c makes a SFW passwd file with initial entry */
   case 'c':
      /* write arguments to temp string buffer */
      sprintf(tmp_str,"%s -c %s %s",HTPASSWD,HTPASSWD_FILE,argv[2]);
      /* runs apache password file generator */
      system(tmp_str);
      break;

   /* option -u adds user */
   case 'u':
      /* write arguments to temp string buffer */
      sprintf(tmp_str,"%s %s %s",HTPASSWD,HTPASSWD_FILE,argv[2]);
      /* runs apache password file generator */
      system(tmp_str);       /* add or modifiy user */
      break;

   /* option -C makes SFW group file with initial entry */
   case 'C':
      /* open SFW group file */
      if(!(g_file=fopen(HTGROUP_FILE,"w")))
```

```
        {
           printf("Could not open group file.\n");
           exit(1);
        };

        /* get a list of group member */
        i=0;
        while(1)
        {
           printf("Enter member (enter 0 to end) : ");
           scanf("%s",m_string[i]);
           /* if 0 is entered, stop */
           if(strcmp(m_string[i],"0")==0) break;
           i++;
        };

        printf("Adding group entry %s.\n",argv[2]);

        /* write entry to group file */
        fprintf(g_file,"%s: ",argv[2]);

        i--;

        /* write group members */
        for(j=0;j<i;j++)
           fprintf(g_file,"%s, ",m_string[j]);
           fprintf(g_file,"%s\n",m_string[j]);

        /* close group file */
        fclose(g_file);
        break;

     /* option -g adds or modify group info */
     case 'g':

        /* initialize flag */
```

```
m_flag = 0;

/* open group file for read */
if(!(g_file=fopen(HTGROUP_FILE,"r")))
{
   printf("Could not open group file.\n");
   exit(1);
};

/* decide add or modification */
while(!feof(g_file))
{
   i = 0;
   /* read a line from group file */
   while(1)
   {
      tmp_str[i] = fgetc(g_file);
      if(feof(g_file))
      {
         tmp_str[0] = '\0';
         break;
      };

      if(tmp_str[i] == 13)
         tmp_str[i] = fgetc(g_file);

      if((tmp_str[i]==0x4)||(tmp_str[i]==10)||(i==255))
      {
         tmp_str[i]='\0';
         break;
      };
      i++;
   }

   /* extract group name */
   i = 0;
   while(tmp_str[i]!=':')  /* skip to delimeter */
      i++;

   tmp_str[i]='\0';
   found_group = strdup(tmp_str);

   /* compare group names */
   if(strcmp(found_group,argv[2])==0) m_flag = 1;
};

/* no match means addition */
if(m_flag==0)
{
   fclose(g_file);

   /* open group file to append */
   if(!(g_file=fopen(HTGROUP_FILE,"a")))
   {
      printf("Could not open group file.\n");
      exit(1);
   };

   i=0;

   /* get list of members */
   while(1)
   {
      printf("Enter member (enter 0 to end) : ");
      scanf("%s",m_string[i]);
      if(strcmp(m_string[i],"0")==0) break;
     i++;
   };

   printf("Adding group entry %s.\n",argv[2]);

   /* add group entry */
```

```
    fprintf(g_file,"%s: ",argv[2]);
    i--;
    for(j=0;j<i;j++)
       fprintf(g_file,"%s, ",m_string[j]);
       fprintf(g_file,"%s\n",m_string[j]);

}
else            /* match means modification */
{
    /* close groufile and reopen */
    fclose(g_file);

    /* open group file */
    if(!(g_file=fopen(HTGROUP_FILE,"r+")))
    {
       printf("Could not open group file.\n");
       exit(1);
    };

    /* open temp file */
    if(!(tmp_file=fopen(TMP_FILE,"w+")))
    {
       printf("Could not open group file.\n");
       exit(1);
    };

    /* read a line from group file */
    while(!feof(g_file))
    {
       i = 0;
       while(1)
       {
          tmp_str[i] = fgetc(g_file);
          if(feof(g_file))
          {
             tmp_str[0] = '\0';
```

```
            break;
         };

         if(tmp_str[i] == 13)
            tmp_str[i] = fgetc(g_file);

         if((tmp_str[i]==0x4)||(tmp_str[i]==10)||(i==255))
         {
            tmp_str[i]='\0';
            break;
         };
         i++;
      }

      i = 0;

      /* skip to delimeter */
      while(tmp_str[i]!=':')
         i++;

      /* copy group name */
      tmp_str[i]='\0';
      found_group = strdup(tmp_str);

      /* write all group entries but matched one */
      if(strcmp(found_group,argv[2])!=0)
      {
         tmp_str[i]=':';
         if(strlen(tmp_str)!=0)
            fprintf(tmp_file,"%s\n",tmp_str);
      };
   };

   /* copy tmp file to group file */
   rewind(tmp_file);
   fclose(g_file);
```

```
/* open group file again */
if(!(g_file=fopen(HTGROUP_FILE,"w")))
{
   printf("Could not open group file.\n");
   exit(1);
};

/* copy char by char */
tmp_char = getc(tmp_file);

while(tmp_char!=EOF)
{
   putc(tmp_char,g_file);
   tmp_char = getc(tmp_file);
};

/* remove temporary file */
remove(TMP_FILE);

i=0;

/* get list of members */
while(1)
{
   printf("Enter member (enter 0 to end) : ");
   scanf("%s",m_string[i]);
   if(strcmp(m_string[i],"0")==0) break;
   i++;
};

printf("Modifing group entry %s.\n",argv[2]);

/* write modified group at the end of group file */
fprintf(g_file,"%s: ",argv[2]);
i--;
```

```
            for(j=0;j<i;j++)
                fprintf(g_file,"%s, ",m_string[j]);
                fprintf(g_file,"%s\n",m_string[j]);
        };

        fclose(g_file);
        break;

    /* option -f processes map file */
    case 'f':
        /* open map file */
        if(!(map_file=fopen(argv[2],"r")))
        {
            printf("Could not open map file.\n");
            exit(1);
        };

        /* open access configuration file */
        if(!(conf_file=fopen(ACCESS_CONF,"r")))
        {
            printf("Could not open access configuraion file.\n");
            exit(1);
        };

        /* open access configuration file */
        if(!(tmp_file=fopen(TMP_FILE,"w+")))
        {
            printf("Could not open temp file.\n");
            exit(1);
        };

        /* copy block above SFW section */
        /* read a line from group file */
        while(!feof(conf_file))
        {
            i = 0;
        while(1)
        {
            tmp_str[i] = fgetc(conf_file);
            if(feof(conf_file))
            {
                tmp_str[0] = '\0';
                break;
            };

            if(tmp_str[i] == 13)
                tmp_str[i] = fgetc(conf_file);

            if((tmp_str[i]==0x4)||(tmp_str[i]==10)||(i==255))
            {
                tmp_str[i]='\0';
                break;
            };
            i++;
        }

        i = 0;

        /* find a line with "# SFW section" */
        if(!strstr(tmp_str,"# SFW section"))
        {
            fprintf(tmp_file,"%s\n",tmp_str);
        }
        else
        {
            fprintf(tmp_file,"%s\n\n",tmp_str);
            break;
        };
    };

    fclose(conf_file);
    rewind(tmp_file);
```

```
/* open access configuration file */
if(!(conf_file=fopen(ACCESS_CONF,"w+")))
{
   printf("Could not open access configuraion file.\n");
   exit(1);
};

/* copy char by char */
tmp_char = getc(tmp_file);

while(tmp_char!=EOF)
{
   putc(tmp_char,conf_file);
   tmp_char = getc(tmp_file);
};

/* remove temporary file */
remove(TMP_FILE);

/* interprete map file line by line */
while(!feof(map_file))
{
   i=0;

   /* read a line from map file */
   while(1)
   {
      tmp_str[i] = fgetc(map_file);
      if(feof(map_file))
      {
         tmp_str[0] = '\0';
         break;
      };

      if(tmp_str[i] == 13)
```

```
        tmp_str[i] = fgetc(map_file);

    if((tmp_str[i]==0x4)||(tmp_str[i]==10)||(i==255))
    {
        tmp_str[i]='\0';
        break;
    };
    i++;
};

/* if line is not empty and not comment */
if((strlen(tmp_str)>0)&&(tmp_str[0]!='#'))
{
    /* read four entries */
    sscanf(tmp_str,"%s %s %s %s",tmp1,tmp2,tmp3,tmp4);

    /* if host list is not empty, get host names */
    if(strcmp(tmp2,"*")!=0) {
        i = 0;
        j = 0;
        k = 0;

        /* get denied host names */
        while(1)
        {
            if(tmp2[i]=='\0')
            {
                host_list[j][k] = tmp2[i];
                break;
            }
            else
            if(tmp2[i]==',')
            {
                host_list[j][k] = '\0';
                j++;
                k = 0;
        }
        else
        {
           host_list[j][k] = tmp2[i];
           k++;
        };
        i++;
     };

     num_host = j;          /* number of hosts */
  };

  /* if user list is not empty, read users */
  if(strcmp(tmp3,"*")!=0) {
     i = 0;
     j = 0;
     k = 0;

     /* get access controlled user list */
     while(1)
     {
        if(tmp3[i]=='\0')
        {
           user_list[j][k] = tmp3[i];
           break;
        }
        else
        if(tmp3[i]==',')
        {
           user_list[j][k] = '\0';
           j++;
           k = 0;
        }
        else
        {
           user_list[j][k] = tmp3[i];
```

```
            k++;
         };
         i++;
      };

      num_user = j;          /* number of users */
   };

   /* if group list is not empty, read them */
   if(strcmp(tmp4,"*")!=0) {
      i = 0;
      j = 0;
      k = 0;

      /* get access controlled group list */
      while(1)
      {
         if(tmp4[i]=='\0')
         {
            group_list[j][k] = tmp4[i];
            break;
         }
         else
         if(tmp4[i]==',')
         {
            group_list[j][k] = '\0';
            j++;
            k = 0;
         }
         else
         {
            group_list[j][k] = tmp4[i];
            k++;
         };
         i++;
      };
   num_group = j;          /* number of groups */
};

/* writes a directory access control directive
   to the access configuration file             */
fprintf(conf_file,"<DIRECTORY %s>\n",tmp1);

/* disable user based access control */
fprintf(conf_file,"AllowOverride None\n");

/* enable authentication module */
fprintf(conf_file,"AuthType Basic\n");
fprintf(conf_file,"AuthName SFW security check\n");

/* SFW user and group file locations */
fprintf(conf_file,"AuthUserFile %s\n",HTPASSWD_FILE);
fprintf(conf_file,"AuthGroupFile %s\n",HTGROUP_FILE);

/* access control for GET and POST methods */
fprintf(conf_file,"<LIMIT GET POST>\n");

/* denied hosts */
if(tmp2[0]!='-')
{
   fprintf(conf_file,"   deny from ");
   for(i=0;i<=num_host;i++)
      fprintf(conf_file,"%s ",host_list[i]);
   fprintf(conf_file,"\n");
};

/* access controlled users */
if(tmp3[0]!='-')
{
   fprintf(conf_file,"   require user ");
   for(i=0;i<=num_user;i++)
```

```
                    fprintf(conf_file,"%s ",user_list[i]);
                fprintf(conf_file,"\n");
            };

            /* access controlled groups */
            if(tmp4[0]!='-')
            {
                fprintf(conf_file,"    require group ");
                for(i=0;i<=num_group;i++)
                    fprintf(conf_file,"%s ",group_list[i]);
                fprintf(conf_file,"\n");
            };

            fprintf(conf_file,"</LIMIT>\n");
            fprintf(conf_file,"</DIRECTORY>\n\n");
        };
    };
    fclose(conf_file);            /* close configuration file */

  break;

  default:
    printf("Invalid switch.\n");  /* invalid switch */
    exit(1);
 };
}
```

APPENDIX B

EXAMPLE SFW CONFIGURATIONS

```
#
# SFW Password File
# /etc/sfw_passwd
#
# Format : <User_name>:<Encrypted_password>
#

kim:UQS/lE4pau4Gk
lee:IqvPLAeB.Ynks
park:dgUVxW0xwtZ6I
joe:ElWNnjno0r1M.
```

```
#
# SFW Group File
# /etc/sfw_group
#
# Format : <Group_name>:<User_name_list>
#

team: kim, lee
party: lee, park
project: kim, lee, joe
```

```
#
# SFW Resource-User Map File
#
# Format : <CGI_resource_name> <Host_name_list> <User_name_list>
#          <Group_name_list>
# Remark : - indicates null list
#

/home/choim/public_html/cgi-bin - kim,lee -
/home/minsu/public_html/cgi-bin *.bad.org - party
/home/john/public_html/cgi-bin - - project
```

APPENDIX C

EXAMPLE SFW LOG FILES

```
#
# /var/log/httpd/sfw_log
# Log file for CGI transactions based on SFW
#

[03:22:43 18-03-98]: UID: (choim/choim) GID: (choim/choim) test.cgi
[02:33:20 19-03-98]: UID: (minsu/minsu) GID: (minsu/minsu) test.cgi
[02:33:20 19-03-98]: directory is writable by others:
(/home/minsu/public_html/cgi-bin)
[02:33:35 19-03-98]: UID: (choim/choim) GID: (choim/choim) test.cgi
[12:37:38 20-03-98]: UID: (jane/jane) GID: (jane/jane) multiply.cgi
[05:50:45 25-03-98]: UID: (choim/choim) GID: (choim/choim) test.cgi
[05:56:51 25-03-98]: UID: (john/john) GID: (john/john) counter.cgi
[06:37:19 25-03-98]: UID: (minsu/minsu) GID: (minsu/minsu) env.cgi
```

```
#
# /var/log/httpd/access_log
# Global HTTPD log file
#

127.0.0.1 - - [19/Mar/1998:02:33:29 -0600] "GET /~choim/cgi-bin/test.cgi
HTTP/1.0" 401 350
127.0.0.1 - park [19/Mar/1998:02:33:29 -0600] "GET /~choim/cgi-
bin/test.cgi HTTP/1.0" 401 350
127.0.0.1 - lee [19/Mar/1998:02:33:35 -0600] "GET /~choim/cgi-
bin/test.cgi HTTP/1.0" 200 247
127.0.0.1 - - [20/Mar/1998:12:36:15 -0600] "GET /~choim/cgi-bin/test.cgi
HTTP/1.0" 401 350
127.0.0.1 - kim [20/Mar/1998:12:37:39 -0600] "GET /~choim/cgi-
bin/test.cgi HTTP/1.0" 200 247
127.0.0.1 - - [25/Mar/1998:05:50:35 -0600] "GET /~choim/cgi-bin/test.cgi
HTTP/1.0" 401 350
127.0.0.1 - kim [25/Mar/1998:05:50:45 -0600] "GET /~choim/cgi-
bin/test.cgi HTTP/1.0" 200 247
127.0.0.1 - - [25/Mar/1998:05:54:16 -0600] "GET /~choim/cgi-bin/test.cgi
HTTP/1.0" 401 350
127.0.0.1 - kim [25/Mar/1998:05:56:51 -0600] "GET /~choim/cgi-
bin/test.cgi HTTP/1.0" 200 247
127.0.0.1 - - [25/Mar/1998:05:57:46 -0600] "GET /~choim/cgi-bin/test.cgi
HTTP/1.0" 401 350
127.0.0.1 - joe [25/Mar/1998:05:58:21 -0600] "GET /~choim/cgi-
bin/test.cgi HTTP/1.0" 401 350
127.0.0.1 - - [25/Mar/1998:06:35:09 -0600] "GET /~choim/cgi-bin/test.cgi
HTTP/1.0" 401 350
127.0.0.1 - kim [25/Mar/1998:06:37:20 -0600] "GET /~choim/cgi-
bin/test.cgi HTTP/1.0" 200 247
127.0.0.1 - - [25/Mar/1998:06:37:55 -0600] "GET /~choim/cgi-bin/test.cgi
HTTP/1.0" 401 350
127.0.0.1 - joe [25/Mar/1998:06:38:31 -0600] "GET /~choim/cgi-
bin/test.cgi HTTP/1.0" 401 350
```

APPENDIX D

GLOSSARY OF TERMS AND ABBREVIATIONS

IN ALPHABETICAL ORDER

TERM	MEANING
Absolute Pathname	Absolute pathname points to file based on its absolute location on the file system. [15]
Access Control	Access control means that access to the resources within a webspace is somehow restricted. [13]
ACF	(Access Configuration File) HTTPD Global access control is defined by ACF. [13]
Apache HTTPD	It a freeware Web server from Apache organization. [13]
Authentication	It allows one to control access to resources so that remote users must enter a valid user name and password to be able to use them. [4, 5, 9, 13]
CGI	CGI stands for Common Gateway Interface, a method for running programs on the Web server based on input from a Web browser. [1]
CGI Environment Variables	They are a set of special variables that are in the environment when a CGI program is requested. [1]
Client-Server	The model of interaction on a distributed system in which a program at one site sends a request to a program at another site and awaits a response. The requesting program is called a client; the program satisfying the request is called the server. [2]
Current Working Directory	Every process has it. This directory is where the search for all relative pathnames start. [15]
FTP	(File Transfer Protocol) The standard, high-level protocol for transferring files over the Internet. [2]
Gateway	It refers to an application program that interconnects two services. [2]
GID	Group Identification Number. [15]
Gopher	A text-based information service used throughout the Internet. [2]
Host	Any end-user computer system that connects to a network. Hosts range in size from personal computers to supercomputers. [2]

Host Name	The host name is the system on the Internet where the information is stored, such as www.cs.okstate.edu. [2]
HTML	(Hypertext Markup Language) The language used to develop web documents. [10]
HTTP	(Hypertext Transfer Protocol) Web servers and browsers communicates using this protocol. [13]
IP	Internet Protocol is the standard communication protocol of the Internet. [2]
IP Address	A 32-bit address assigned to each host that participates in the Internet. [2]
NCSA	The National Center for Supercomputing Applications.
Nobody	This special user name can be used by network servers that allow remote users to log in to a system, but with a UID and GID that provide no privileges. [15]
Port	The abstraction that TCP/IP transport protocols use to distinguish among multiple destinations within a given host computer. [2]
Query String	The arguments to the CGI program or the form input (if submitted using GET). It contains everything after the question mark in the URL. [1]
Relative Pathnames	Relative Pathnames points to files based on their locations relative to the current working directory. [15]
RFC	(Request For Comments) The name of a series of notes that contain surveys, measurements, ideas, techniques, and observations, as well as proposed and accepted TCP/IP protocol standards.
Root (superuser)	The special user whose UID is 0 is called either root or superuser. If a user has superuser privileges, most file permission checks are bypassed. [15]
SFW	Secure & Flexible CGI Wrapper is a security CGI wrapper suite which ensures more secure and flexible CGI transaction.
Symbolic Link	A type of file that points to another file. [15]

TCP/IP	(Transmission Control Protocol/Internet Protocol) A standard connection-oriented communication protocol for the Internet. [2]
TELNET	The TCP/IP standard protocol for remote terminal service. TELNET allows a user at one site to interact with a remote system at another site. [2]
UID	User Identification Number. [15]
URL	A Universal Resource Locator specifies address of a web document. [11, 12]
WWW	World Wide Web is a global, interactive, dynamic, cross-platform, distributed, graphical hypertext information system that runs over the Internet. [2]

2

VITA

Minsu Choi

Candidate for the Degree of

Master of Science

Thesis : A SECURE AND FLEXIBLE COMMON GATEWAY (CGI) WRAPPER

Major Field : Computer Science

Biographical :

Education : Graduated from Dae-Sung High School, Seoul, Korea in 1988; received Bachelor of Science Degree in Computer Science from Oklahoma State University, Stillwater, Oklahoma in 1995. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University, Stillwater, Oklahoma in May 1998.

Experience : Employed by Oklahoma State University, Department of Computer Science as a graduate teaching assistant; Oklahoma State University, Department of Computer Science, 1996 to present.

Professional Membership : Golden Key National Honor Society.