Security Aspects of Web–based Unix Programming Environment

Richard Perry
Villanova University
Department of Electrical and Computer Engineering
Villanova, PA 19085
richard.perry@villanova.edu

Abstract

The security aspects of a web–based Unix environment for C, Java, and shell programming are presented. In contrast to a sandbox with limited functionality, the environment allows full access to the underlying system, and the user can execute arbitrary Unix commands. System security is provided in multiple layers involving the process environment, file system, network configuration, resource limits, and cryptographic tickets. Although implementation details will be presented specifically for a Solaris 9 operating system, the techniques are applicable to any Unix system. The environment is implemented using an "old–school" combination of C code, Perl, and shell scripts, and is freely available (open source).

1 Introduction

The educational aspects of a web–based Unix environment for C, Java, and shell programming were originally described in 2004 [8], with only a brief discussion of the security aspects. Since then many new features have been added, with an associated increase in the complexity of the security mechanisms, including use of cryptographic tickets. After a short functional overview of the system from the user perspective, we will describe all of the security mechanisms used to provide this safe web–based Unix programming environment. Section 3 will cover the underlying operating system security, then Section 4 will focus on security mechanisms inside the environment itself. The implementation and security aspects of web–based interactive I/O will be presented in Section 5.

2 Functional Overview

The environment is oriented towards students taking a C programming course, although Java and shell script programming are also supported. After "logging in" to a course–specific URL (e.g. using LDAP authentication) users are presented with their project homepage for that course, as seen in Figure 1 for user *alice* and course *fc*, a freshman C programming course. There are ten project assignment subdirectories, *a1* ... *a10*, links to the environment documentation and source code, and a link to download a zip archive of all of the users files. *iPlot* is a special subdirectory preinitialized with code to provide an interactive plotting environment, and in the *HTML* subdirectory student programs can generate HTML forms to create their own interactive environments.

If *alice* selects the link for *a2* she is presented with the page shown in Figure 2, listing specific files that can be edited or created, and all existing files which can be viewed. Each project subdirectory is initialized with only a *Makefile* to start with.

Now if *alice* decides to edit *p1.c* she proceeds to the main work area of the environment where files can be edited, saved, compiled, etc. as displayed in Figure 3. Here, files can be uploaded from the users local

Figure 1: Project Homepage

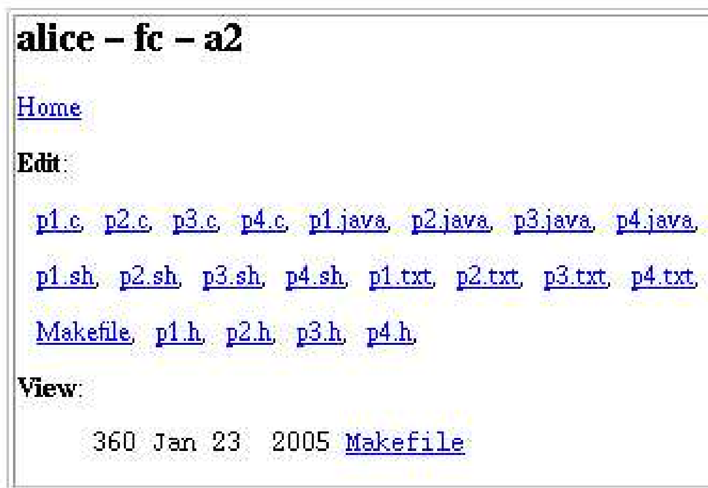

Figure 2: Project a2 Subdirectory

computer or copied from an archive of example programs, and there are various options related to compiling, debugging, plotting, audio, and interactive I/O. Note the *args* input area: this is used to pass command–line arguments when running a program, or to execute an arbitary Unix command using the *Command* button. This area of the environment will be discussed further in Section 4 in the context of its security.

3 System Security

For most types of Unix systems, after installing the operating system many changes must be made to make the system more secure. Here we describe the particular changes and other configuration settings made on a Solaris 9 installation in order to safely allow web–based access. This includes changes to the file system, process and network configuration, and webserver configuration.

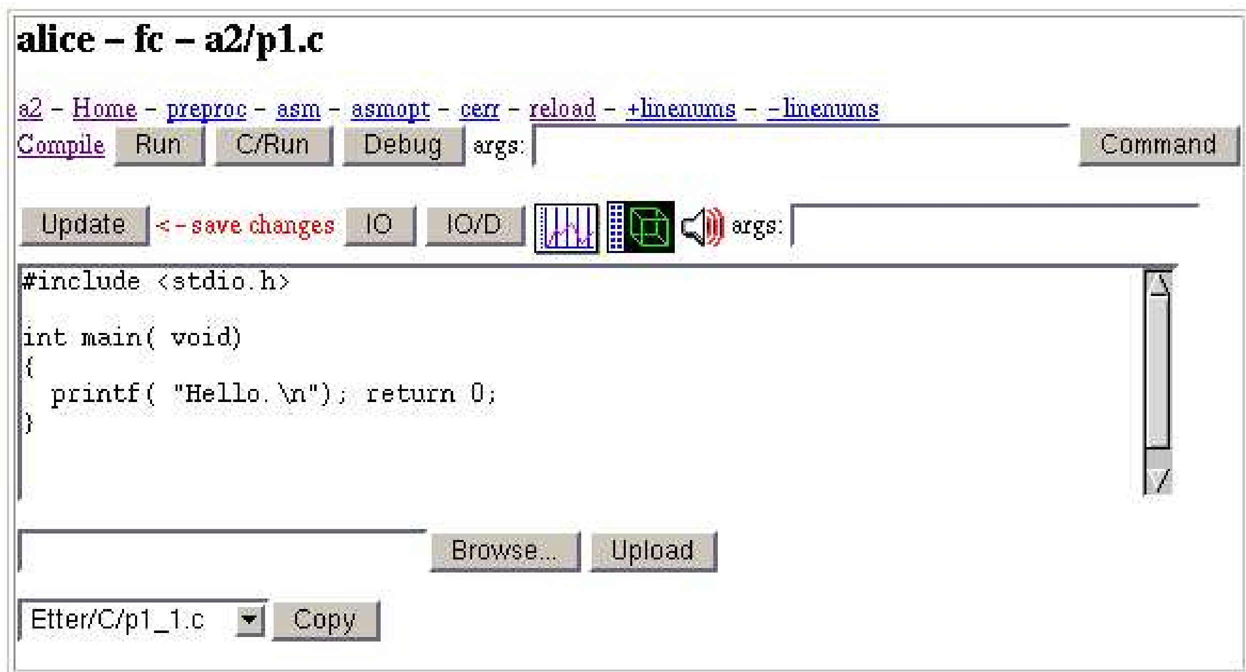

Figure 3: Editing a2/p1.c

3.1 File System Security

A complete install of Solaris 9 contains over 100 setuid and/or setgid programs in /usr. Many of these programs, like *admintool*, are not intended to be used by regular users, but are setuid root and executable by anyone. Some of these programs have had ongoing security problems, such as buffer–overflows related to command–line arguments or environment variables, which could be exploited by a user to obtain elevated privileges.

To avoid the risk of setuid/setgid program exploits, all such programs have been moved to a separate */usr/root* directory tree, accessible only by users in group root. The only users in group root are system administrators who have root access anyway. The original programs are replaced by symbolic links in /usr.

Example permissions, owner, group, and file:

```
drwxr-x---   root   root   /usr/root
-r-sr-xr-x   root   bin    /usr/root/bin/sparcv9/uptime
lrwxrwxrwx   root   root   /usr/bin/sparcv9/uptime ->
                             /usr/root/bin/sparcv9/uptime
```

For web–based users, with no controlling terminal, no display, and no real login (they have unique uid numbers, but no /etc/passwd entry, see Section 4 for more information) very little functionality is lost by not having access to any of the setuid/setgid programs. In particular for them, *ps* does not work, and pseudo–terminals in general do not work due to /usr/lib/pt_chmod being unaccessible.

3.2 Process and Network Security

Two particular aspects of process security are addressed by the following settings in the */etc/system* specification file:

```
set coredefault=0
set maxuprc=40
```

The first setting prevents core dumps, which can reveal privileged information if a system daemon crashes. Also, it eliminates the core files which would otherwise be created in every project directory of every student using the system (assuming that all student programs crash, which is pessimistic but mostly true). Rather than examining core files, students can reproduce crashes using the *Debug* or *IO/D* options, which run their program under the gdb debugger.

The second setting limits the number of simultaneous processes for users other than root. This prevents the "fork–bomb" attack as well as other unintentional runaway spawning of processes. The setting should not be too low, since it limits all non–root processes system–wide, including webserver processes (e.g. running as userid *httpd*).

Basic network security of the system starts with disabling *inetd* or equivalently commenting out all lines in */etc/inetd.conf*. Network access for system administration is then provided using *sshd* with *tcp–wrappers*.

3.3 Webserver Security

The Apache *httpd–2.0* webserver is used, with *mod_ssl* enabled, running as user *httpd* and group *httpd*, and started with a *chroot* command to make the environment root directory */home/httpd/*. The web programming system, */vecr/*, is contained in the chroot area, but except for the front–end CGI interface it is outside of the webserver DocumentRoot (*/htdocs/*). However, to allow browsing of the live environment code [7], parts other than users files are made accessible using symlinks:

```
lrwxrwxrwx  root  root  htdocs/vecr/IO -> ../../vecr/IO
lrwxrwxrwx  root  root  htdocs/vecr/audio -> ../../vecr/audio
lrwxrwxrwx  root  root  htdocs/vecr/dist -> ../../vecr/dist
lrwxrwxrwx  root  root  htdocs/vecr/doc -> ../../vecr/doc
lrwxrwxrwx  root  root  htdocs/vecr/images -> ../../vecr/images
lrwxrwxrwx  root  root  htdocs/vecr/local-bin -> ../../vecr/local-bin
lrwxrwxrwx  root  root  htdocs/vecr/proto -> ../../vecr/fc/proto
lrwxrwxrwx  root  root  htdocs/vecr/src -> ../../vecr/src
```

with the IO and audio links actually being necessary for the applets described in Sections 5 and 4.4 respectively to work.

The front–end CGI interface is invoked whenever the string */prog/* is present in a URL. This is configured by the following Apache directives:

```
ScriptAlias /prog/ "/vecr/cgi-bin/"
<Directory /vecr/cgi-bin>
  Options None
  AllowOverride AuthConfig Limit
</Directory>
```

which enable use of a *.htaccess* file to require user authentication, as discussed further in Section 4.

The chroot setup prevents web users from accessing anything outside of the /home/httpd/ directory. To allow user programs and the webserver itself to run, /usr and /opt are loopback–mounted read–only on /home/httpd/, and the dev directory there contains the following device files:

```
crw-rw-rw-  root  root  dev/null
crw-r--r--  root  root  dev/random
crw-rw-rw-  root  root  dev/tcp
crw-rw-rw-  root  root  dev/ticotsord
crw-rw-rw-  root  root  dev/udp
crw-r--r--  root  root  dev/urandom
crw-rw-rw-  root  root  dev/zero
```

4 Environment Security

Access to the environment starts with user authentication to run one of the course–specific wrapper CGI programs. The CGI exec's a Perl script *view.pl* which parses the request and passes control to a shell script *view.sh*. Then, depending on what actions are to be performed, additional subsidiary programs and scripts are invoked.

The complete source code is available at [7]. The implementation is an "old–school" [4] combination of C code, Perl, and shell scripts, and should be easily portable to any Unix system.

4.1 Authentication and Wrapper

The *cgi-bin* file permissions, link counts, owner, and group appear as follows, configured for three courses:

```
-rw-r-----  1 root  httpd  vecr/cgi-bin/.htaccess
---s--x---  3 root  httpd  vecr/cgi-bin/ccs
---s--x---  3 root  httpd  vecr/cgi-bin/fc
---s--x---  3 root  httpd  vecr/cgi-bin/osp
```

The *.htaccess* file specifies a combination of LDAP and local password file authentication:

```
AuthName "Villanova LDAP"
AuthType Basic
AuthLDAPEnabled on
AuthLDAPURL ldap://ldap.villanova.edu/o=villanova.edu?uid?sub
AuthAuthoritative off
AuthUserFile /vecr/private/.htpasswd
AuthGroupFile /dev/null
require valid-user
```

All regular users authenticate using LDAP; the *.htpasswd* file only contains dummy users for testing.

The wrapper has separate hard links for three courses, *ccs*, *fc*, and *osp*. It is setuid–root, runnable only by group httpd, and is a C program executable which basically does the following (error checking code removed for brevity):

```
char *val, course[1024];  gid_t grouplist[1] = { 0 };

setgroups( 1, grouplist);  setgid( 0);  setuid( 0);

val = strrchr( argv[0], '/');  val = val ? val+1 : argv[0];

strcpy( course, "COURSE=");  strncat( course, val, 1024-10);

putenv( course);  execv( "/vecr/local-bin/view.pl", argv);
```

that is, it sets the real, effective, and saved user and group IDs to root (necessary on Solaris, beyond the file permission effective setuid, to really run as root), creates an environment variable *COURSE* containing the file name, then exec's the *view.pl* Perl script.

Given the aversion to setuid programs evidenced in Section 3.1, one may question why a setuid–root program is used here. One reason is that the original environment ran under a webserver which was not chroot, so the wrapper had to run as root to perform its own chroot call. Since the current system already runs chroot under the webserver, the remaining two reasons for the setuid–root wrapper are, first, so that the environment can perform setuid to a unique userid for each user. As discussed further below, a directory structure is created for each user such that they can not see other users files and can not change permissions such that other users could see their files. This would not be possible if the environment ran under a single generic userid. The second reason is to allow root–only access to private data, such as the database of web userids, the usage log file, and the secret used in the cryptographic token for interactive I/O (Section 5).

The wrapper is derived from a small C program, *chroot+setuidgid*, which can perform chroot and setuid/gid, as well as redirecting stdout after the setuid, based on either command–line arguments or hard–coded values set using conditional compilation. The wrapper uses hard–coded values, i.e. from the Makefile:

```
UID = 0
GID = 0
PROG = /vecr/local-bin/view.pl

wrapper: chroot+setuidgid.c Makefile
        $(CC) -o wrapper -s \
          -DUID=$(UID) -DGID=$(GID) -DPROG=\"$(PROG)\" \
          -DPUTENV_COURSE=1 chroot+setuidgid.c
```

and is installed setuid–root in the cgi–bin/ directory using course–specific file names, one link for each course.

Another version of the same program is compiled without hard–coded values, so it will obtain the values at run–time from the command line:

```
chroot+setuidgid: chroot+setuidgid.c
        $(CC) -o chroot+setuidgid -s -DSTDOUT chroot+setuidgid.c
```

and is installed as a helper program in the local–bin directory:

```
---x--x--x  root  root  vecr/local-bin/chroot+setuidgid
```

This version of the program does not have setuid file permissions since it will be run as root from a descendent process of the setuid wrapper. It is used as described below to create processes running under the user uid and gid.

4.2 Running as root

The *view.pl* Perl script invoked as root by the wrapper checks whether the request was "GET" or "POST", and safely passes that information as well as any arguments and post data to the *view.sh* script as seen in this excerpt:

```
$prog="/vecr/local-bin/view.sh";

if ($ENV{REQUEST_METHOD} eq "GET")
{
```

```
    @a = ("GET", @ARGV);  exec("$prog", @a);  exit;
  }
  # else POST
  ... parsing code omitted, $mode, $args, and $data set here ...
  open( EDIT,"| $prog POST");
  print EDIT "$mode @ARGV $args\n$data\n";
  close( EDIT);
```

The *view.sh* script first logs the request, then carefully extracts the username from the *REMOTE_USER* environment variable provided by the webserver. For LDAP authentication in particular, if AuthAuthoritative is turned on, *REMOTE_USER* will contain the full distinguished name (dn), for example:

```
uid=alice,ou=a,ou=Students,o=villanova.edu
```

The username *alice* is easily extracted from this, but additionally, the username is "sanitized" by removing any '/' and '.' characters. This is a necessary precaution because it will be used as a directory name, and it is conceivable that an attacker with control over the LDAP server would craft a special filename–like username in an attempt to disrupt the web environment, e.g. "uid=../../../conf/".

In the web environment, users are assigned unique userid/groupid values starting with 3000, and these stored in a user list file. The users do not have a real Unix login, and are not listed in any password or group file, not even in the chroot area. *view.sh* obtains the userid/grouid value corresponding to the username from the user list file, or automatically creates a new entry for a first–time user. The users web home directory and project subdirectories are created if they do not already exist.

The following example showing permissions, owner, group, and file name illustrates that *alice* has userid/groupid 3025, she has read–only access in her top–level home directory, and write access in the project subdirectories:

```
drwxr-x---  root  3025  vecr/fc/home/alice
drwx------  3025  3025  vecr/fc/home/alice/a1
```

Since *alice* is not the owner of *home/alice/*, she can not change permissions to allow other users to see her files. Similarly, she can not see other users files.

All further actions in the environment, such as compiling or executing Unix commands, are run under the users userid/groupid, with resource limitations as described next.

4.3 Running as the user

To execute a command as the user, *view.sh*, running as root, first uses the shell *ulimit* command to restrict resources, then invokes the *chroot+setuidgid* to set the userid/groupid and run a special *RUN* program which will be described below:

```
RUN="/vecr/local-bin/run"
SETUIDGID="/vecr/local-bin/chroot+setuidgid"
RUNSH="/vecr/local-bin/run.sh"
#
ulimit -c 0     # maximum core file size (in 512-byte blocks)
ulimit -d 8192  # maximum size of data segment or heap (in kbytes)
ulimit -f 1024  # maximum file size (in 512-byte blocks)
ulimit -n 64    # maximum file descriptor plus 1
ulimit -s 8192  # maximum size of stack segment (in kbytes)
```

```
ulimit -t 10    # maximum CPU time (in seconds)
ulimit -v 65536 # maximum size of virtual memory (in kbytes)
#
exec "$SETUIDGID" --setuidgid "$uid" "$uid" \
  "$RUN" "$RUNSH" "$mode" "$top/$fname" "$sub" "$file" "$args"
```

Shell variables set up prior to running the command contain the userid, mode (what to do, e.g. run, debug, plot, audio, command, etc.), home directory, subdirectory, file, and arguments.

run.sh, running as the user, checks the mode argument to determine what to do, and for executing a Unix command (Command button in Figure 3) simply does:

```
eval $args
```

which enables use of shell wildcards, pipelines, I/O redirection, etc. in the user command.

The *RUN* program, executed as the user by *chroot+setuidgid*, is a C program executable which: forks *run.sh* as a child process; sets an alarm for 30 seconds real–time; then waits for the child process to finish. If the time–limit is reached, it kills all processes running as the user (including itself); otherwise it simply reports the exit status of the child process. The combination of cpu–time and real–time limits prevents potential problems with runaway or stuck processes overloading the system.

4.4 Audio Security

For audio projects, students are generally given an input data file containing raw audio samples at 8000 Hz. corrupted by noise or interference. They create a program to filter the data using signal processing techniques to reduce the corruption, and then play their program output using the audio option (speaker icon in Figure 3). The audio option first converts the program output to a temporary file in 8-bit ulaw format, then plays that file in a loop using a Java applet. An example is shown in Figure 4.

The ulaw conversion has to store all of the data in memory in order to find the maximum value for use in scaling its output, so there is a potential problem with memory usage growing indefinitely if the user program has unlimited output. Although memory usage is limited by the ulimit shown in Section 4.3, a smaller limit seems more appropriate here, so a limit of 1 million samples (about 2 minutes of audio) is imposed by the conversion program.

For compatibility with older browsers and old versions of Java, the applet uses the simple *getAudioClip()* method to download the audio file. This does not support user authentication with the webserver, but even if it did we would avoid that because it would force the user to reenter their userid/password every time the applet is run. In Section 5 a similar applet authentication problem is encountered, and it is solved using a cryptographic ticket. To do the same for audio would be overkill since here the user is only accessing a temporary data file, not their whole web environment.

Therefore, the audio file is created in a */tmp/* directory which is publically accessible through the webserver, and is writable to all web users. To avoid file name collisions, the temporary audio file names are created by concatenating the username, course name, project subdirectory and program name, and current process id, for example */tmp/alice-fc-a2-p1.sh-19141.audio*. This presents some risk to the users, from other users, since one could manually create such files to try to cause a collision and subsequent denial–of–service.

A more significant risk caused by the writable public directory is that if the webserver configuration for */tmp/* allows following symlinks, then users could create links to anything in the entire chroot area. Any files readable by user/group *httpd* could then be made publically available, for example the *cgi-bin/.htaccess* file shown in Section 4.1, and other files outside the webserver DocumentRoot not intended for public access. However, this risk is eliminated by the webserver configuration settings *Options None* and *AllowOverride None* for the */tmp/* directory.

Figure 4: Audio Applet

Another problem with temporary directories and files in general is accumulation of old files and resulting loss of disk space. This is currently handled in the web environment using a cron job to remove old files:

```
2 * * * * /opt/bin/gfind /home/httpd/htdocs/tmp -type f -amin +59 -delete
```

5 Interactive I/O Security

Interactive programs are run using the *IO* or *IO/D* (run under gdb debugger) buttons shown in Figure 3. An example is shown in Figure 5 using a program which prompts to read numbers and displays the number and its reciprocal. The top area shows diagnostic messages regarding the applet and network connection. The middle area shows program input and output; the contents of this area can be saved to a file using the *Save* button. The user enters input in the bottom area above the buttons.

It should be noted that although interactive I/O is limited to running user programs and scripts, the user can create a one–line shell script like this:

```
exec ksh
```

and run it interactively to obtain an interactive shell. This is allowed intentionally, and does not present any additional risk to the system since the user is already capable of executing arbitrary Unix commands using the *Command* button.

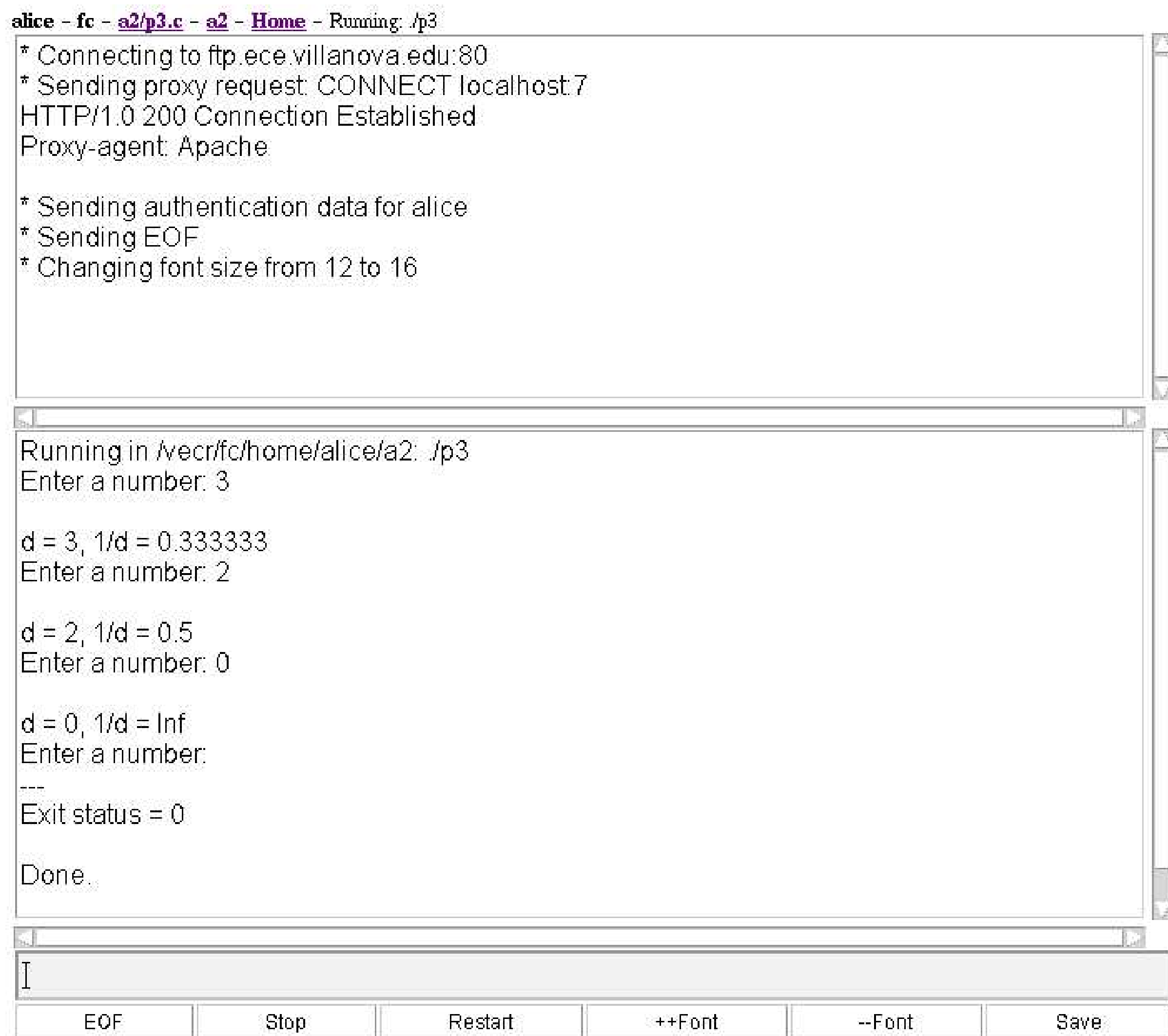

Figure 5: I/O Applet

The execution of interactive I/O starts with an applet, which connects to a web proxy. The proxy forwards the connection to a Java server, which spawns a client thread for each connection.

5.1 I/O Applet and Ticket

In the web page generated to display the I/O window, the applet is passed parameters containing the userid, current time, a cryptographic ticket, the project subdirectory path, the program name to run, and the program

command–line arguments, if any.

The cryptographic ticket is a message authentication code based on a hash (HMAC, RFC 2104 [5]). It is generated using javax.crypto.Mac() with algorithm *HmacSHA256*, which uses SHA256 [6] as the hash function. HMAC operates by combining data with a secret as input to a hash or "one–way" function. Given the hash output, one can not determine the inputs. However given the original data and secret, one can regenerate the hash output and compare it with the value sent with the data.

For the I/O applet ticket, the secret is 64 bytes from */dev/random* stored in a file readable only by root:

```
-rw-------  root  root  vecr/IO/bin/secret.dat
```

This file can be changed to a new secret at any time, and although that would invalidate any I/O tickets currently in use, the user can just click the *IO* button again to obtain a new valid ticket, as must be done when tickets expire anyway.

The ticket itself is a 256–bit HMAC created using the secret, the current time, the username, and userid. It is passed to the applet as a 64–byte hex string parameter. Since the user was already known to be authenticated at the time the IO option was initiated, the ticket allows that authentication to propagate from the applet to the server without having the user resend their userid and password.

5.2 I/O Proxy

Rather than open another port on the perimeter firewall, the webserver is used as a proxy to forward requests to the Java I/O server as diagramed in Figure 6. The Apache configuration directives for the proxy are:

```
ProxyRequests On
ProxyTimeout 300
AllowCONNECT 7
<Proxy *>
  Order Deny,Allow
  Deny from all
</Proxy>
<Proxy localhost:7>
  Order Allow,Deny
  Allow from all
</Proxy>
```

Once the proxy connection is established, the web server is transparent and simply forwards data, including authentication data, back and forth between the remote user and the server on localhost port 7.

5.3 I/O Server and Authentication

The I/O server is a Java process started at system boot–time using a *chroot* command to make its root directory the same as the webserver. It runs as root, and listens for connections on localhost port 7. The applet uses the webserver proxy to connect to it by sending the string "CONNECT localhost:7".

When it receives a connection, the I/O server creates a client thread to communicate with the applet. The applet then sends its authentication data parameters (username, userid, and ticket creation time), and ticket. It also sends its other parameters (project subdirectory path, program name to run, program arguments), but these are not used in the authentication. The client thread creates an HMAC using the authentication data and the same secret described above in Section 5.1. If this HMAC does not match the ticket, then the authentication fails and the client throws a SecurityException which is logged before it drops the connection and exits. If the HMAC does match, then the client thread is assured that the username and userid presented

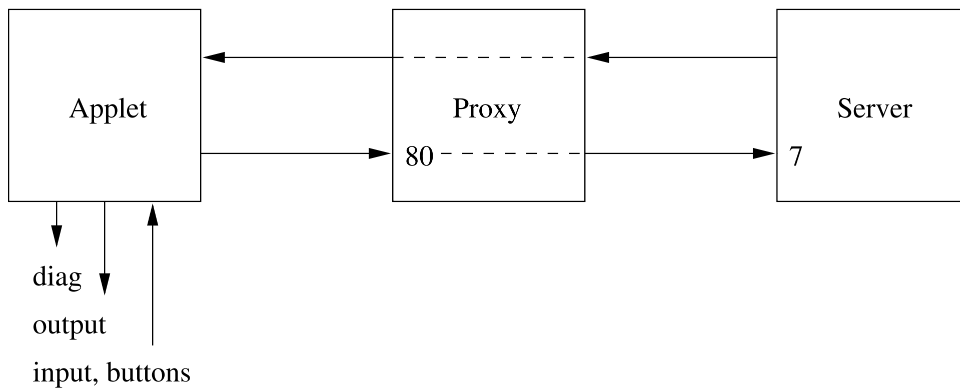

Figure 6: I/O Web Proxy

by the applet are valid, so it proceeds to compare the ticket timestamp with the current time. If the ticket is older than 10 minutes, the client throws a SecurityException and exits. Otherwise it proceeds to create a subprocess to execute the user program.

One may question why the authentication data and ticket do not include the subdirectory, program, and arguments. This is because there is no need to check or limit the commands or programs that users execute; that is already restricted by the file system security and chroot environment. A clever user could change the applet parameters, as long as none of the authentication values are changed, which would cause authentication failure. The server only needs assurance of the validity of the username and userid – the username simply for logging activity, and the userid to perform setuid/setgid as described next.

5.4 I/O Run Subprocess

After authentication succeeds, the I/O client creates a subprocess to execute *Run*, a C program executable which combines aspects of *RUN*, *chroot+setuidgid*, and *ulimit* from Section 4.3. It also spawns a *Copy* thread to copy output from the user program to the applet, while itself handling copying input from the applet to the user program as seen in Figure 7.

Before invoking the user program, *Run* sets its userid and groupid to that of the user, then it changes directory to the subdirectory specified in the applet parameter, and then it forks. In the child process of the fork, it uses *setrlimit()* calls to restrict resources similar to the ulimit shell command described in Section 4.3. An excerpt, with error checking code removed for brevity:

```
struct rlimit rl;

rl.rlim_cur = rl.rlim_max = 0;  setrlimit( RLIMIT_CORE, &rl);

rl.rlim_cur = rl.rlim_max = 10; setrlimit( RLIMIT_CPU, &rl);
```

The child process then exec's the user program.

The parent process sets an alarm for 120 seconds, then waits for the child to exit. If the child exits within the time limit, the exit status is reported. Otherwise the parent kills all processes running as the user (including itself).

The applet communicates with the I/O client using a simple protocol to indicate whether data is being sent or the user selected one of the *EOF* (send end–of–file), *Stop*, or *Save* buttons (see Figure 5). *Save* creates a subprocess which runs a C program executable similar to *Run* above, which sets the userid, groupid, and

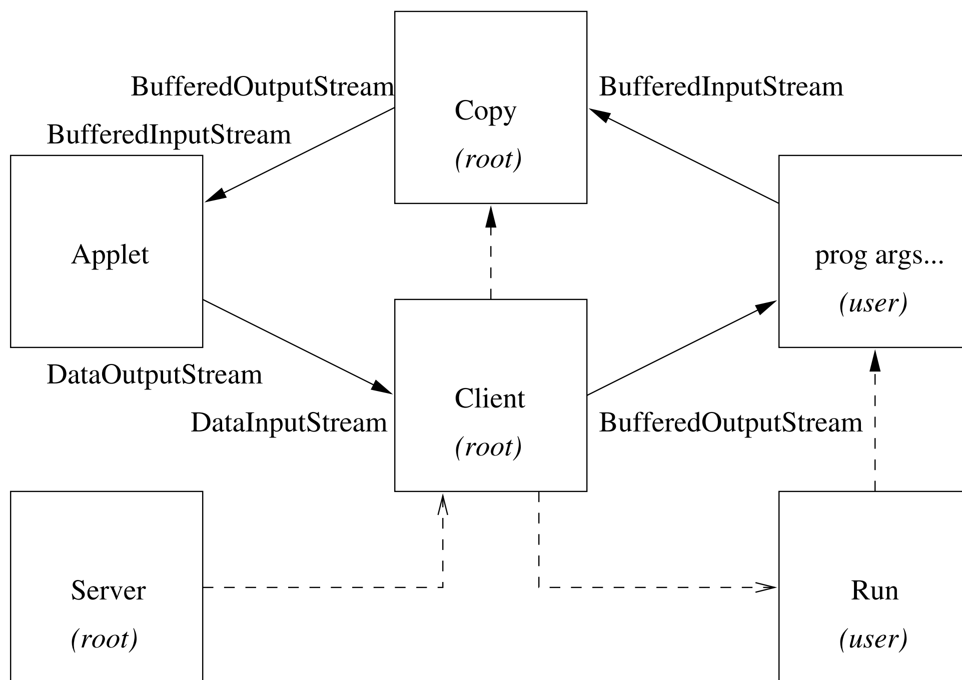

Figure 7: I/O Server, Client, and Run Processes

default directory, then copies the contents of the applet output window (sent from the applet with the *Save* command) to a file.

6 Conclusion

The programming environment described here had its origin in a much simpler form created several years ago to support a single course assignment. It enabled editing and running a single shell script generating HTML to implement an interactive directory/file browser. Security of the system was of utmost priority from the beginning, since users had access to execute arbitrary Unix commands. By using a chroot environment, eliminating access to potentially insecure commands (Section 3.1), using a setuid–root wrapper (Section 4.1), and limiting resources (Section 4.3), users were prevented from accidently – or maliciously – disrupting the system. Evolving from that simple beginning to its current complexity, with audio, interactive I/O, applets and a Java server, security of the system has continued to be the core requirement in implementing any new features.

The oldest and most widely used web–based application today may be web–mail. Word processing and spreadsheet applications have recently been incarnated in web–based form as “Google Docs & Spreadsheets” [3]. To the author’s knowledge, web–based programming environments similar to the one presented here do not exist, although the commercial product CodeLab [2], formerly WebToTeach [1], deserves mention even though it is instructor–oriented and designed mainly for testing and homework–checking. Although the programming environment described here may be unique, the security aspects are applicable to any web–based application environment.

References

[1] David Arnow. The WebToTeach Project. http://www.sci.brooklyn.cuny.edu/~arnow/WebToTeach/.

[2] Turing's Craft. CodeLab. http://www.turingscraft.com/.

[3] Goggle. Google Docs & Spreadsheets. http://www.google.com/.

[4] Stephen B. Jenkins. Musings of an "old–school" programmer. *Communications of the ACM*, 49(5):124–126, May 2006.

[5] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. ftp://ftp.isi.edu/in-notes/rfc2104.txt.

[6] NIST. FIPS 180-2: Secure Hash Standard (SHS). http://csrc.nist.gov/publications/fips/.

[7] R. Perry. VECR documentation and source code. http://ftp.ece.villanova.edu/vecr/doc/.

[8] R. Perry. View/edit/compile/run web-based programming environment. In *Frontiers in Education Conference*, pages T3H 1–6, October 20–23, 2004.