

View/Edit/Compile/Run Web-based Programming Environment

Richard Perry¹

Abstract—A web-based environment has been developed for students to perform C, Java, and shell programming. The environment runs on a Unix server, uses password authentication, and provides each student with separate project subdirectories that can not be seen by other students. Options are available to view files, edit source code, compile, run, run in debug mode, run with output plotted and displayed as a GIF image, display C preprocessor output, display generated assembly code, display optimized assembly code, and insert compiler error messages as comments into the source code. The environment is implemented using a combination of C code, perl, and shell scripts, and is freely available (open source). The source code of the environment itself can be used as examples in an advanced Unix/C programming or security course. The environment has been used successfully in both sophomore and senior-level C programming courses, a graduate Unix/C programming course (C and shell programming), and a senior/graduate computer communications security course (Java programming).

Index Terms—Unix, open source, programming, web-based.

I. INTRODUCTION

For ECE and CSC computer programming assignments, students at Villanova have access to a variety of systems including Unix and Microsoft workstations in departmental and college laboratories, as well as their own personal computers. But the programming environments available on these systems vary widely – some may not have Java installed, or may have an unsuitable old version; some only have professional C++ development tools with complex interfaces that overwhelm beginners and do not enforce strict ANSI C compliance; and the Microsoft systems generally do not have the Unix shells and other tools (e.g. Cygwin [1]) installed.

To alleviate this problem, the View/Edit/Compile/Run (VECR) environment [2] was created to provide an easy-to-use interface to a set of standard programming tools, with consistent options. For example, for C programming, the GNU GCC compiler [3] is used with options *-ansi -pedantic -Wall* to enforce strict ANSI C compliance and enable all warnings. This helps students to learn standard C and avoid use of system-specific functions or inappropriate constructs from C++.

Students are encouraged to use whatever programming environment is most convenient for them for initial development of programs, then upload to VECR for testing and turning in. However most students have reported that they do all development directly in VECR, after having used it and found

it to be both convenient and consistent, providing the same interface for C, Java, and Bourne shell programming.

In the next section, this paper will provide a tour of the VECR environment, with screen shots and descriptions of the available options. Section III will then describe the courses and projects which have successfully used the environment. Details of the implementation are provided in Section IV, followed by conclusions and ideas for future work.

II. VECR TOUR

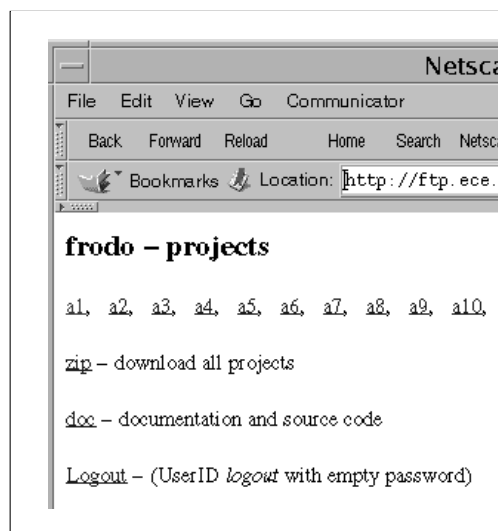


Fig. 1
VECR HOME PAGE FOR STUDENT FRODO

Figure 1 shows the VECR home page for student *frodo*. There is a choice of 10 available projects, *a1*, ..., *a10*, as well as a link to download all of the project files in zip format. There is also a link to the VECR documentation and source code; all of the source code and files within the running VECR system (except for student project files) are readable by anyone, and students are encouraged to explore the system. The Logout link simply allows reauthentication using a dummy *logout* UserID which clears the browser-cached authentication information.

The VECR home page for course instructor *rperry* shown in Figure 2 is similar, but also allows the instructor to set their effective UserID to any student from a drop-down list. After setting the UserID, an instructor can access the associated student project files without having to know the student

¹Richard Perry, Villanova University, Department of Electrical and Computer Engineering, richard.perry@villanova.edu



Fig. 2
VECR HOME PAGE FOR INSTRUCTOR RPERRY

password. This is useful for helping students in debugging their programs and for grading finished projects.

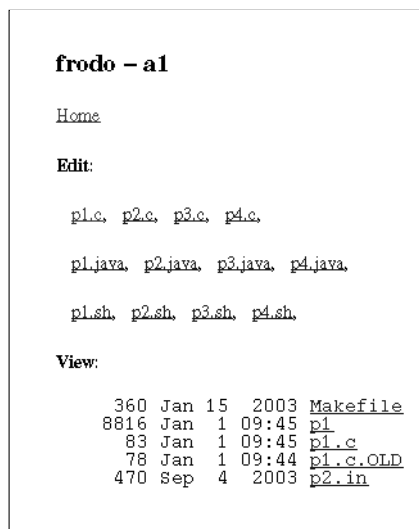


Fig. 3
VECR A1 PROJECT PAGE

Figure 3 shows an *a1* project page; there is a choice of four C, four Java, and four sh programs to edit. At the bottom of the page is a directory listing with links to view any files in the project directory. In this example, the Makefile and data file p2.in were provided by the instructor; pl.c and pl.c.OLD are the current and previous versions of the student work on that program, and pl is the associated executable.

Figure 4 shows the result of selecting *pl.c* from the *a1* project page. Here the student program source code is in an editable textarea window at the bottom of the page. At the top are links for the various actions which can be performed:

- *preproc* - display C preprocessor output
- *asm* - display generated assembly code
- *asmopt* - display optimized assembly code
- *cerr* - insert compiler error messages as comments into the source code
- *reload* - reload the current page
- *Compile* - compile C or Java program
- *Run* - run C, Java, or sh executable

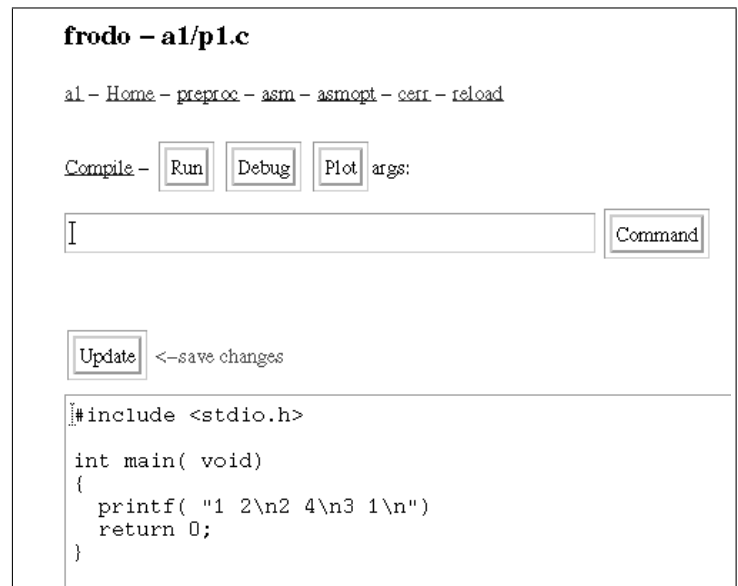


Fig. 4
VECR A1/P1.C EDIT PAGE

- *Debug* - run C program using gdb, or sh script using -x
- *Plot* - pipe program output into plotting program
- *args* - specify program command-line arguments
- *Command* - execute args input directly as command
- *Update* - save edit changes
- *Upload* - not shown in the figure, at the very bottom of the page, upload local source file

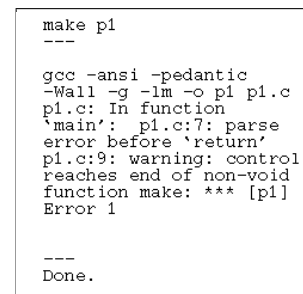
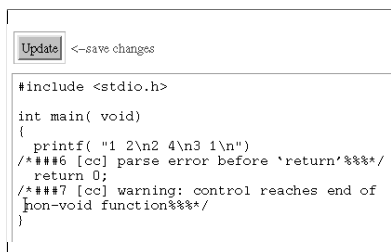


Fig. 5
VECR A1/P1.C COMPILE

The program shown in Figure 4 contains a syntax error; it is missing a semicolon after the printf() statement. Attempting to compile produces the error messages shown in Figure 5. By selecting the *cerr* option, the error messages are inserted as comments into the source code as shown in Figure 6. This can be very helpful for debugging larger programs, and is implemented simply by redirecting standard output and standard error from make into the Unix *error* utility: *make ... 2 > & 1 | error*

After fixing the syntax error, saving the changes, and recompiling, selecting *Run* from Figure 4 produces the output shown in Figure 7. It shows the command-line used to run the program, as well as the program exit status (i.e. return value



```

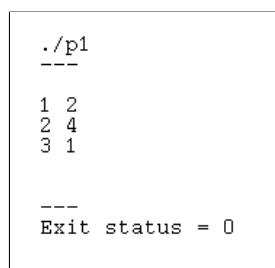
Update <-save changes

#include <stdio.h>

int main( void)
{
    printf( "1 2\n2 4\n3 1\n")
    /****6 [cc] parse error before 'return'****/
    return 0;
    /****7 [cc] warning: control reaches end of
    non-void function****/
}

```

Fig. 6
VECR A1/P1.C CERR



```

./p1
---
1 2
2 4
3 1

---
Exit status = 0

```

Fig. 7
VECR A1/P1.C OUTPUT

from main() in C).

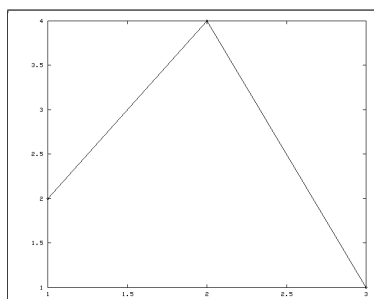


Fig. 8
VECR A1/P1.C PLOT

For programs producing one or two columns of output data suitable for plotting, *Plot* may be selected instead of *Run*. As seen in Figure 8 *Plot* will run the program with output piped into a plotting program and the result will be displayed as a GIF image. The plot program is just a simple shell script which uses gnuplot [4] and netpbm [5]:

```

(echo "set term pbm\nset output\n\
plot '-' notitle with linespoints"; cat) |
/usr/local/bin/gnuplot |
/usr/local/netpbm/bin/ppmtogif

```

III. COURSES AND PROJECTS

Starting over a year ago, the VECR environment grew out of an initial web-based setup which allowed students to create one Bourne shell script for a project in a graduate-level Unix/C programming course. The project was to dynamically generate HTML web pages listing system directories and files, with

links back to the same script to traverse the file system and view files. Providing an environment where the student shell script was run by a web server made the project much more interesting that other course projects where programs or shell scripts were developed for a simple interactive Unix command-line environment.

VECR was then expanded to include ten course projects, with four C, four Java, and four Bourne shell scripts in each project. The default *Content-Type* for program output is now *text/plain*, but that can be changed to *text/html* for specific course projects to enable dynamic HTML programming as in the initial environment.

The VECR environment has since been used successfully in both sophomore and senior-level C programming courses, a graduate Unix/C programming course (C and shell programming), and a senior/graduate computer communications security course (Java programming).

A. Introductory C Programming

For the C programming courses, the first and several subsequent course projects took advantage of the VECR plotting capability to enable very simple C programs to produce graphics output, thus making the projects more interesting for students. For example, one initial course project involved a noisy sine-wave data file given to the students and automatically available in the VECR environment. The first part of the assignment was to just read the data as floating-point numbers using `scanf()` and output using `printf()`; subsequent parts involved computing 3-point or N-point moving averages of the data, and estimating the period of the sine wave. By using the VECR *Plot* option to run their program, students could visually see the results of smoothing the noisy input data.

Another motivating project (in February) was to write a C program using various functions from *math.h* to produce a plot which looks like a Valentine's day heart. And for a Monte-Carlo simulation project, one aspect involved using an array of counters, which represented a probability density function when plotted.

B. Advanced C Programming

In more advanced C programming projects, VECR provided students with header files and code for functions such as dynamic matrix allocation, matrix singular value decomposition, and network I/O. The student projects then involved writing code to use the provided functions to develop their application. Checking code which may have worked on the students own system or other systems at Villanova, but which failed to compile or run properly in the VECR environment, invariably revealed errors in their code, such as OS portability and data endianness issues.

C. Java Programming

VECR was also used for Java programming projects in senior elective and graduate computer communications security courses. These courses covered the theory and practice of computer privacy, authentication, and encryption, with hands-on

projects at every level. VECR provided the students with the current Java development environment, together with the Java security extension packages and other packages developed by the instructor. Of particular interest in these courses was the security of the VECR environment itself, and students were encouraged to explore the system and examine the source code for potential security flaws. The instructor, as developer of VECR, worked hard to stay one step ahead of the students in that regard.

IV. VECR IMPLEMENTATION

The VECR environment is implemented using a combination of C code, perl, and shell scripts, and is freely available [2] (open source). The source code of the environment itself can be used as examples in an advanced Unix/C programming or security course.

The implementation is designed to be secure. It runs in a chroot environment, with most actions performed under the student userid with run-time limits on resources such as CPU and real-time usage (the Unix *chroot* environment sets the root of the file system to a subdirectory from which one can not escape, thus preventing access to the rest of the file system).

A. System Environment

The VECR system at Villanova is a Sun Ultra-5 workstation (*ftp.ece.villanova.edu*) running Solaris 7, and has handled courses with up to 50 students. The Apache web server [6] is used, running under userid *nobody*, and is not itself running chroot. The following Apache configuration directive:

```
ScriptAlias /prog/ /home/chroot/cgi-bin/
```

redirects requests for *http://ftp.ece.villanova.edu/prog/view* to the executable file */home/chroot/cgi-bin/view* which is the main interface to VECR. The *view* executable is setuid-root, with permissions:

```
---s---x--- root nobody ... view
```

and is responsible for performing chroot to */home/chroot* and then running the next phase of VECR processing (*view.pl*, discussed below in Section IV-D).

The following abbreviated *df -k* output shows the file systems which are loopback mounted (readonly for */usr* and */opt*) on */home/chroot*:

Fs	bytes	capacity	Mounted on
/usr	1015542	71%	/home/chroot/usr
/opt	3410878	85%	/home/chroot/opt
/proc	0	0%	/home/chroot/proc

This allows access from within VECR to the standard Unix utility programs in */usr/bin* and optional software such as gcc, gdb, java, etc. in */opt/bin*. Access to */proc* is not strictly required, but without that some commands like *ps* and *who* will not work.

B. Authentication

Users access the VECR web interface using password authentication, as specified in the */home/chroot/cgi-bin/.htaccess* file:

```
AuthName "Villanova LDAP"
AuthType Basic
LDAPAuth On
LDAPSSLDisable
LDAPServer "ldap://ldap.villanova.edu:389/"
AuthAuthoritative off
AuthUserFile /home/chroot/private/.htpasswd
AuthGroupFile /dev/null
# LDAP or .htpasswd user
require valid-user
```

Villanova uses University-wide LDAP authentication, so users do not need to have separate accounts set up for web access. However, there is sometimes a need for access by users not associated with Villanova who are not listed in the LDAP server, and by dummy accounts for testing. Thus, the *.htaccess* file specifies a fallback to a local *.htpasswd* file where non-LDAP users can be listed (e.g. the dummy *frodo* account used in the examples of Section II).

Each user accessing VECR is dynamically assigned a userid/groupid starting with 3000. These userid's do not exist in the system */etc/passwd* file, that is they are not real login accounts. The authenticated *REMOTE_USER* environment variable is looked up in the chroot */local-lib/id.list* flat database file, and if not found, the next sequential userid/groupid is assigned.

C. Project Directories

For each user, a directory is created in the chroot *home/* directory, owned by root with readonly access for the users' group. Subdirectories for each project are created owned by the user, for example:

```
drwxr-x--- root 3004 home/frodo
drwx----- 3004 3004 home/frodo/a1
drwx----- 3004 3004 home/frodo/a2
drwx----- 3004 3004 home/frodo/a3
...
```

The file permissions ensure that each user only has access to their own project directories. Furthermore, a user can not change the permissions of their top-level home directory to allow access by other users, since that directory is owned by root.

Each project subdirectory can be automatically initialized with files provided by the instructor, e.g. a Makefile, generic input data files, specific per-student data files, auxiliary source files, etc. Common initial files reside in subdirectories of a *proto* directory, one for each project, for example:

```
-rw-r--r-- root root proto/a1/Makefile
-rw-r--r-- root root proto/a1/p2.in
```

Student project directories are initialized with symbolic links to the *proto* files, for example *home/frodo2/a1/Makefile* is a symlink to *../../proto/a1/Makefile*

Specific per-student data files can be created by adding code to the *view.sh* script (discussed below in Section IV-D). For example, in one course *a2/p3.in* was initialized with an encrypted random fortune for each student:

```
# create a2/p3.in if necessary
f="$dir/a2/p3.in"
if [ ! -r "$f" ]; then
  cd /src/sdes-cbc || exit 1
  /usr/local/bin/fortune |
  /usr/local/jdk/bin/java CBCEncrypt \
  "$user" > "$f"
  chown "${uid}:${uid}" "$f"
  chmod 400 "$f"
fi
```

The code above contains a security flaw; there is a race condition between the time the file is determined to not exist and the time it is created from the encrypted fortune output. Between those two times a student process could have created the file as a symbolic link to any existing file in the chroot environment, which would then be overwritten by the encrypted fortune output. Code such as the above is more safely implemented running as the student *userid* instead of as root.

D. Running as root

As noted in Section IV-A, entry to VECR starts with the *setuid-root* executable *view*, which performs the *chroot()* system call. *view* also sets the real and saved *userid* and *groupid* to root, which is necessary for Solaris to run subsequently invoked scripts as root. Running as root is dangerous since root has access to all aspects of the system and can even break out of the chroot environment. The parts of VECR which run as root should not contain flaws which may allow creation or overwriting of arbitrary files or execution of arbitrary user input.

Running as root is necessary in order to create the student project directories with proper owner and group, and to change the running *userid* and *groupid* to that of the student for actions such as compiling and running programs. Overall this is more secure than running everything under a common *userid* such as *nobody* which would lead to less separation of student file ownership and less process control.

view runs the next step of VECR, the *view.pl* Perl script, which decodes data sent by POST requests, such as updating a project source file. *view.pl* then invokes the *view.sh* Bourne shell script, which is the main part of VECR. *view.sh* handles creating new accounts and project directories and files. Based on whether it was invoked using GET or POST, and depending on supplied arguments, it generates the appropriate HTML web pages on the fly as needed. There are no static HTML pages in VECR, other than the documentation. All HTML content is generated dynamically.

E. Running as student *userid*

For actions to be run as the user, such as compiling or running programs, *view.sh* not only sets the *userid* and *groupid* to that of the student, but also runs the request using an executable named *run* which limits the run-time to 30 seconds real-time, to prevent runaway processes. If the 30 seconds run-time is exceeded, all processes owned by the student are killed. *run* also checks the exit status of its child process and will display a descriptive message if the process was terminated by a signal, e.g. "SIGSEGV - segmentation violation".

The shell *ulimit* command is also used to restrict the resources used by the student process, placing limits on maximum core file size (0), maximum size of the data segment or heap (8192 KB), maximum file size (512 KB), maximum number of file descriptors (64), maximum size of the stack segment (8192 KB), and maximum CPU time (10 seconds). Furthermore, */etc/system* sets a system-wide limit on number of processes per *userid*, which prevents *fork bombs* from disrupting the system (a *fork bomb* is code such as *while(1) fork();*).

The student process spawned by *view.sh* uses a separate script, *run.sh*, to set appropriate compile options and/or perform the requested action. Note that by selecting the Command button shown in Figure 4 the student can execute an arbitrary Unix command. Since the student can create and execute arbitrary C, Java, or sh programs, there would be little security gained by attempting to restrict what commands can be executed. In fact, *run.sh* uses the sh *eval* command on purpose to execute commands as well as run the student programs, which enables use of pipelining and I/O redirection. For example, to run the compiled *p1.c* program with input from file *p1.in* and output piped to the input of compiled program *p2.c*, the student would use the following args in the *p1.c* edit page:

```
< p1.in | p2
```

V. CONCLUSION AND FUTURE WORK

The VECR environment currently satisfies the needs of several courses for programming in C, Java, and sh, and it continues to evolve every semester that it is used. New features will be added as the need arises, and security of the system will always be a priority in design.

One current limitation when creating new accounts is that no locking mechanism is used to prevent conflicting simultaneous updating of the *userid* database files. This has not been a problem so far, but would be if used in an environment where many new users access the system simultaneously.

Another limitation is that interactive programs are not supported; program input must come from data files or the output of other programs, the user can not interactively type input. This limitation will be removed in the future by having an *Interactive* selection in addition to the *Run* and *Plot* options for running programs. For interactive programs I/O will be managed by a Java process running on the server with a network connection to a Java applet running on the client system.

REFERENCES

- [1] "Cygwin – Unix tools for Windows." <http://cygwin.com/>.
- [2] "VECR documentation and source code." <http://ftp.ece.villanova.edu/perry/VECR/>.
- [3] "GNU compiler collection." <http://gcc.gnu.org/>.
- [4] "Gnuplot – graphics plotting." <http://www.gnuplot.info/>.
- [5] "Netpbm – graphics library." <http://netpbm.sourceforge.net/>.
- [6] "Apache HTTP server." <http://httpd.apache.org/>.