# C PORTAL
## *Online Educational Framework for C and C++ Languages*

Ádám Gábor, Dénes Németh and Imre Szeberényi
*Budapest University of Technology and Economics, Műegyetem rkp 3-5, Budapest, Hungary*

Keywords: C programming language, C++ programming language, Educational portal, Moodle framework, GRE security, Chroot, Resource limitation.

Abstract: This paper introduces the C portal educational framework, which helps teachers to assign C and C++ programming language problems to students and automatically compile and test the received solutions in a secured environment. The paper evaluates how the compiled untrusted binary can be executed with minimal security risk with focus on privilege escalation and uncontrolled resource usage. The paper also proposes how the system can be integrated in the existing systems intended for programming education like Moodle or dokeos.

## 1 INTRODUCTION

For more than a thousand year papers, letters or books were the main medium for delivering thoughts, ideas and theories between people. This has changed a lot with the spread of interconnected and telecommunication systems. These technologies make it possible to receive, send and share huge amount of information within a fraction of a second. The effects of these developments are estimated to be as big as the first industrial revolution.

These changes affect all aspects of the everyday life, which includes the educational system too. Our goal was to create an on-line interactive e-learning system for the C and C++ programming languages. There are several e-learning systems, but most of them can only handle tests or essay-like exams and hardly any supports the teaching of programming languages. The tools of this type mainly deal with interpreted languages like Prolog, ASP, PHP, Lisp, etc. or languages which can be easily tested in a secured container like Java or ran on the client side.

The C language is something completely different. It is a compiled language usually used for system programming, which means that it is hard to safely execute a C based program either by previous parsing of the source code or limit the capabilities of the binary executable by the operating system.

The goal of our system is to provide an environment in which solutions to C based e-problems can be safely tested. This environment consists of two parts: 1) a plugabble user interface described in the 2.1 section, 2) a batch system described in sections 2.3 and 2.4. The batch system is responsible for handling, executing and testing the submitted solutions. The last section of the paper covers the security risks and implications of the presented system.

## 2 THE INFRASTRUCTURE

The structure of the system consists of four logical components, which are illustrated on figure 1. The first is the *user interface* which handles the interaction with students and teachers. The *interoperability interface* stores all input and output data in a relational databases and shares it between the different components of the system. The third component is the *workload manager*, which is responsible for global job scheduling and separating the database from the unsafe running jobs, the last component is the *execution system*, which executes, tests and separates jobs from eachother. This last component can be composed of multiple physical computers. The last two components are called the batch system.

### 2.1 User Interfaces

The platform where certain users can interact with the system are the user interfaces. These are dynamic web pages, accessed via SSL using certificates,

Gábor Á., Németh D. and Szeberényi I. (2009).
C PORTAL - Online Educational Framework for C and C++ Languages.
In *Proceedings of the First International Conference on Computer Supported Education*, pages 247-252
DOI: 10.5220/0001975502470252

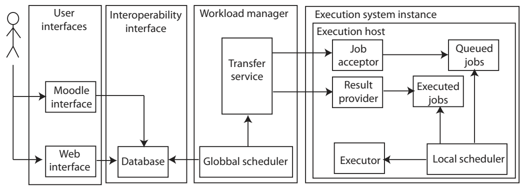

Figure 1: The structure of the system.

where the user has to authenticate with a username and password. The web pages are implemented using PHP, a widespread script language for developing dynamic web pages and applications (McArthur, 2008; Adam Trachtenberg, 2006). The user interface on development level is separated to two distinct layers: the data processing layer and the presentation layer.

The data processing layer collects all the input generated by the user (via submitting a form, or just clicking on a hyperlink), and executes the script for the actual target object. After the processing is finished, including verifications, validations and database transactions, the presentation layer is called by the appropriate parameters describing the structure of the page to be displayed and the generated content.

The presentation layer is a collection of self written and Smarty template engine (Sweat, 2005) templates. The Smarty template engine is also implemented in PHP, and gives a very simplified but enhanced way of generating web pages from predefined templates by providing an object oriented api for passing content.

All input generated by the user is being verified and validated to prevent malicious code to be inserted and executed, to prevent cross-side scripting and sql injections. The input types are classified in order to how the content is to be verified and validated. There are input types for what the user can only choose a value, but there are free input fields which have input scope boundaries. These inputs are supervised by regular expressions and built in methods.

Apart from php and database security the most threat comes from the user written C and C++ codes which are too difficult (impossible) to filter. Therefore we rely on the strength of the *execution system* defense mechanism.

There are two completed user interfaces ready and in use: the administrative and the student interfaces. The integration with the Moodle e-learning content management system is already started.

## 2.2 Interoperability Interfaces

The interoperability interface has two functionalities: First to provide asynchronous communication between the global scheduler and the user interfaces.

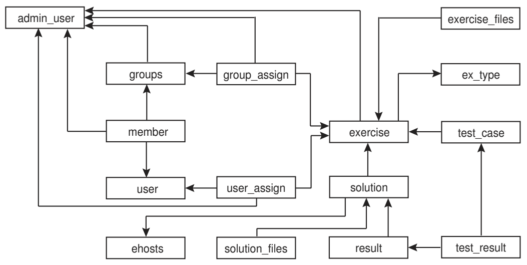

Figure 2: The schema of the database structure.

Secondly to act as a data storage for the neighboring components. The best choice for both tasks was an SQL database (Kofler, 2005; Dubois, 2007). The complex structure of the database used by these modules is shown on Fig. 2. This structure was chosen to provide maximal modularity and scalability for the system.

As this interface is used by more than one component, it has to provide granular data to each subsystem without the ability to identify the component which is doing the query. Tables have been divided to ensure flexibility, and to provide a simple way of accessing data independent from the type or content, while keeping the existing connections among the objects. As of security concerns, each component were granted different access privileges to the database - select, insert, update and delete rights -, keeping the number of these rights at the minimum, but still sufficient to ensure the expected functionality to be presented by each component. This is done by defining roles - collection of privileges -, and associating them to the appropriate subsystem.

Further concern was to enable the integration and cooperation of this system with existing, e-learning projects, such as Moodle (Cole and Foster, 2007; Korte, 2007). As being a complex e-learning content management system, Moodle already has a well documented, distributed database infrastructure with more than two hundred tables. It includes the storage of a wide variety of educational data templates, such as tests, quizzes, discussion boards, etc.

As the redesign of the database schema of Moodle is impossible we had to add new fields and tables to our database to ensure the functionality of the new Moodle module, while keeping the connections and data structure provided by Moodle untouched.

## 2.3 The Workload Manager

The workload manager runs the global scheduler, which implements two functionalities: the information collector for the execution system and the global resource broker. The information collector gathers all metrics (queued jobs, running jobs, available slots) of the execution system instances, these are needed for global load balancing. The global resource broker uses this information to track and schedule the jobs.

The job scheduling covers three tasks: 1) query of the interoperability interface for new jobs, 2) selection of an executor instance for job locations and 3) transfer of the results of the solutions from the executor instance to the database through the interoperability interface.

The workload manager also contains the transfer service, which moves the data between the execution system instances and the workload manager. This is implemented as an SSH client, which holds the private keys of the job acceptor and the result provider users of each execution system instance. This operates in a push/pull mode, which means that it pushes a job to the execution system instance, and periodically queries the result provider for available results. The pull mode on one side has a slight overhead, but on the other side it allows a much clearer separation between the two systems.

## 2.4 The Execution System

The execution system consists of several execution hosts. Each host runs four services. These are job acceptor, result provider, execution and local scheduler services. These services are explained in the following sections.

Execution system hosts are physically separated from the Internet by sitting in a private network which is connected to the workload manager over a switch but not to Internet. This separation prevents all computers except the workload manager to communicate with the executor hosts.

### 2.4.1 The Job Acceptor

The job acceptor accepts jobs *pushed* from the workload manager. Currently it is implemented as a simple SSH server, which only accepts connections from the workload manager and a unique user which can write to one single directory, which is the execution queue.

### 2.4.2 The Result Provider

The result provider is a *pullable service*, which allows the transfer service to move the produced result of the pushed solution back to the workload manager. In our system it is implemented as a simple SSH server, which allows a single user of the transfer service to handle the results queue:

- count the number of entries in a single directory, which represents the results queue

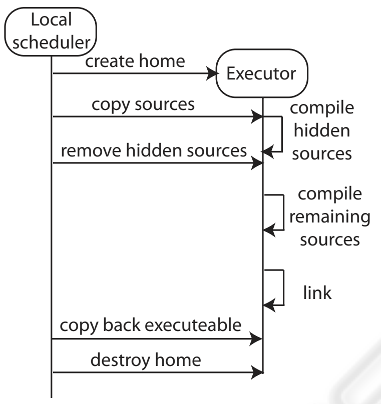

Figure 3: The comilation process.

- retrieve and remove elements from this queue

### 2.4.3 The Local Scheduler

The local scheduler is responsible for two tasks. 1) measure the metrics related to the execution hosts. These metrics are the load and the number of running and queried jobs. 2) drive the execution service, namely to decide which task should be executed from the received jobs. In the current implementation this is a shell script running in the background.

### 2.4.4 The Execution Service

The execution service driven by the local scheduler provides the safe container for running and testing solutions. The execution environment is a read only image, which contains a minimal operating system with pool users used for compiling and executing solutions. If the scheduler notices that a solution needs to be tested from the local queue it selects a pool user(PU) and compiles the user supplied and hidden source files as illustrated on figure 3.

The home of the user is a separate memory based temporary image created on demand and mounted inside the read only environment. After the home is mounted the sources are copied. First the hidden source files are compiled, than they are deleted and only the compiled object files are available in the further steps. After this the user supplied and non-hidden files are compiled. If all compilation was successful the objects are linked, and the final executable is copied outside of the read-only environment. Before the complete home is destroyed the log file is transported and parsed outside of the sandbox environment by the local scheduler.

If everything regarding the compilation process succeeded according to the settings defined by the in-

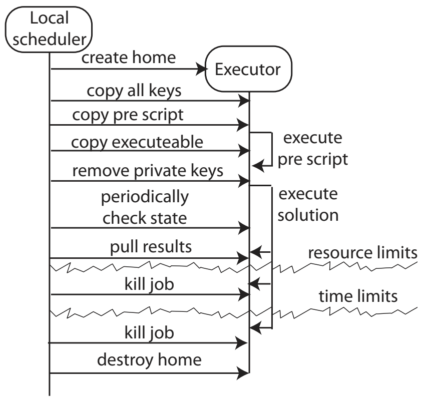

Figure 4: The execution process.

teroperability interface the testing phase begins. In this phase each test case is executed separately. The process of a test case execution is illustrated on figure 4. This process also starts with a temporary home creation and all keys, the executable and the *pre scripts* are copied there. After this step the system executes the pre script, which can create user specific input or data for the executable. After the pre script finished the compiled executable is started.

The local scheduler chroots(Joy, ) to the scratchbox and executes the wrapper of the job, which is binary that closes the standard input and outputs (stdin, stdout, stderr) of the wrapper and reopens them: The file called INPUT is used as standard input and the STD.OUT and STD.ERR files are opened for standard output and standard error respectively. If the streams are opened successfully the wrapper calls the exec system call for the compiled binary.

The following limitations are set for the solution execution:

1. The shell of the PU is the compiled executable
2. The PU can write only in its home
3. The PU can have only one process (it can not fork)
4. The PU can use maximum 32 MB memory
5. The PU can not use the network
6. The PU can not use more then 60 sec CPU time
7. The PU can not use more then 300 sec real time
8. The PU can not access device files
9. The PU can not access the sys or proc file system

The pool user is prevented from breaching 5, 8 and 9 limitations on kernel level, while the UNIX standard *limits* utility is used to enforce limitations 3, 4 and 6. The rule 2 is enforced by using a file system for the operating system which lacks write operations and ismounted as read-only. The home of the user is a memory file system. The 1 limit is hard coded into

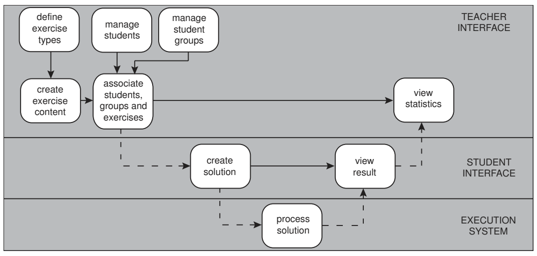

Figure 5: The usecase worrkflow of the user interfaces.

the read only image. The limitation 7 is enforced by the local scheduler, which terminates the running process if the consumed amount of real time is more than allowed.

If none of these limitations are breached and the test case process exits normally, then the *post* script is executed. which processes the output and decides whether the test case is accepted or not. This post script is a simple shell script, which has access to the all keys of the test case. If the post script exists normally it creates a *done* flag, which is checked by the local scheduler which moves all data to the results queue.

## 2.5 The Workflow

The workflow is centralized to the lifecycle of an exercise, shown on Figure 5. The account of the main administrator is automatically generated, and by authenticating with this account, the administrator has full management and administrative access to every object in the system. The main administrator has to initialize the database by inserting the available exercise types, and generating the student database. The admin has full access to the user management module, so it is possible to add users when the system operates.

As the user upload module accesses the interoperability interface, five pairs of public and private keys are generated automatically and associated to each student user, thus enabling the modules of the execution system to ensure privacy by accommodating encryption algorithms without querying the database for personal data.

The main admin may also generate sub-admin accounts in the same way as students (for the assistant lecturers for example). Sub-admins can create student groups conformly with the labor or seminar groups.

The next step is the creation of an exercise. This is a complex task for the teacher user, because all the required information have to be inputted. These parameters describe the method of giving the exercises to the students, and specify the parameters of how the solutions are to be verified, validated and evaluated.

Header file contents can be added to provide distinctive api to the students, as well as hidden files needed only for the compiler while building and testing the program from the source code handed in. When setting the specification of the exercise, the teacher may add student specific parameters automatically generated by the system at runtime. This can be done by explicitly the teacher: the body of the parameter generating script can be entered manually for each exercise, so as the students receive the exercise specification, the parameters are generated by the input function body as a static part of the specification, so the student acquire these parameters transparently. Any number of test cases can be assigned to the exercise.

At the point when the execution system successfully compiles the solution source code, it runs a series of tests defined by the creator by input and output. Prescripts and postscript can be also defined to each test case, enabling the configuration of the execution system to react and function in a unique way, while processing the solutions and evaluating each test case for each exercise. The creator has full access to manage and modify the settings of the exercise while it is not associated and distributed to the student users. Other teacher users have read privilege to exercises of other creators, and can easily create new ones with using similar settings or header files.

When an exercise is successfully entered into the system, every teacher user may assign it independently from the identity of the creator of the exercise to single student users. Exercises can also be assigned to groups of students (even if it is already assigned to individual student users), but with the limitation that the teacher user may only assign it to groups created by their own. The procedure of removing associations is similar.

As a student user authenticates on the student web interface, the assigned and completed exercises are prompted. In this page a student may hand in solutions to the associated exercises - the number of possible tries, the content length and the maximum number of uploadable file content are all defined by teacher users -, and is also able to view the results of each previously handed in solution. When a solution is submitted, it is passed to the interoperability interface, from where the execution service takes all the parameters of the actual exercise, and the contents of the solution. After the evaluation is completed, and result data is written back into the database, the status of the solution changes to evaluated, thus enabling the web page for the student to view the results of the provided solution and maybe create a newer one based on the evaluated solution.

## 3 SECURITY CONSIDERATIONS

During the development of the framework the two most important principles used were to keep the system simple, stupid and use only well proven, out of the box technologies. Since no system can provide 100 percent protection against all local exploits, one of the best choices is to keep the used technologies as updated as possible and use the strongest enforced privilege and service separation and prevent the spreading of the compromisation.(Bishop, 2003)

In an order to secure the infrastructure we run each functionality from 1-4 on separate nodes, and deploy firewalls between them.(Rash, 2007) These firewalls should allow only the absolutely necessary network communication between the services. The physically separated part of the infrastructure are the execution instances, the most easily breachable part of the system. They are only connected to the workload manager, which drops every incoming network packets and only allows the SSH client of the transportation service to reach the executor instances. This makes it impossible for the executor instance to reach the rest of the infrastructure. Even if the execution system is compromised, the rest of the system stays protected from internal attacks.

The workload manager runs the global scheduler as a non root user, which on one hand writes to the execution system through SSH and to the interoperability interface through PHP and MySQL. It accepts incoming connections only from the user interface while anything else is dropped. It is difficult to bypass this node as the rest of the system is connected only to this and there is no direct transfer of any network traffic.

The hardest part is the protection of the executor system. This not only includes the privilege protection, but the ensuring that the running test do not jeopardize the system, by creating dead locks, starvation or high load, which renders the system unusable. These problems are further explained in the (Daniel J. Barrett, 2003) and (Bishop, 2003) books.

The privilege protection is achieved by the technologies explained in the limitation enumeration, but the rest has to be achieved by the created architecture. The too extensive I/O usage limiting is achieved by using a read only scratch box and a temporary 2 MB size tmpfs created in the memory acting as the home of the pool user. This creates a maximal disk quota and converts disk usage to memory access, which is indirectly the already limited CPU usage.

The pool user is prevented from network usage by GRE security. This is required to protect the network from flooding. (Simson Garfinkel, 2003; Vacca, 2007)

The UNIX based systems normally can not handle more then 65536 processes, so the pool users must be prevented from flooding the system with huge amount of processes, so the maximal number of processes for a single pool user is limited to one.

The pool users are separated from each other by the standard UNIX UID/GID structure to prevent unwanted interference.

Above these limitations the solutions of the students are fairly queued, which means that the system tries to allocate the same amount of resources to each student. This prevents a single user to flood the system with jobs, which wait in the queue and delay the solutions of other users for a longer amount of time.

On the interoperability interface the backups of the databases must be periodically stored on a separate node. There are several tools to backup a database, so secured backups are out of the scope of this paper.

We can draw the following conclusion on the security side: This system is an educational supporting framework and not a mission critical banking system or airplane control unit and should not be treated as such. If an incident happens, the main goal is to protect the database, notice the event and identify the student who submitted the dangerous code.

If only the executor instance is compromised, it can be easily reinstalled, since it holds no states. The lost jobs, which are not transported back to the workload manager can be rerun.

## 4 CONCLUSIONS

As the implementation of the necessary core functionality of the system has been finished, our goal was to construct an environment consisting of distributed elements capable of compiling and testing C based problem solutions. Private and public testing proved that the modules of the system are well constructed and stable, and its usage did not produce any unexpected event.

Due to its distributed structure, the vulnerability of singular components does not affect the stability of the whole system, thus it provides a stable solution for a distributed system to be applied in an educational institute. As the system is already being used by more than half a year by more than two hundred students without severe errors or malfunctions, we can state that our security assumptions were correct. As usage increased many software ergonomical problems were resolved and interfaces were made more user-friendly.

There are lots of e-learning platform and portal implementations used by major institutions, and as our framework was designed to be compact but modularly structured, it is a bearable and sustainable development to integrate our system as a module to these bigger platforms. The integration into the Moodle e-learning content management system is almost finished.

## ACKNOWLEDGMENTS

Part of this work was funded by the Pázmány Ṕter programm (RET-06/2005) of the National Office for Research and Technology, the authors would like to thank the EGEE-III (INFSO-RI-222667) program.

## REFERENCES

Adam Trachtenberg, D. S. (2006). *PHP Cookbook*. O'Reilly.

Bishop, M. (2003). *Computer security: Art and science*. Addison-Wesley Publishing Co.

Cole, J. and Foster, H. (2007). *Using Moodle: Teaching with the Popular Open Source Course Management System*. O'Reilly Media, Inc., 2nd edition.

Daniel J. Barrett, Richard E. Silverman, R. G. B. (2003). *Linux Security Cookbook*. O'Reilly Media, Inc.

Dubois, P. (2007). *MySQL Cookbook*. O'Reilly, 2nd edition.

Joy, B. Chroot on unix operating systems. http://en.wikipedia.org/wiki/Chroot.

Kofler, M. (2005). *The Definitive Guide to MySQL 5*. Apress, 3rd edition.

Korte, L. (2007). *Moodle Magic: Make It Happen*. FTC.

McArthur, K. (2008). *Pro PHP: Patterns, Framework, Testing and More*. Apress.

Rash, M. (2007). *Linux Firewalls - Attack Detection and Response*. O'Reilly Media, Inc.

Simson Garfinkel, Gene Spafford, A. S. (2003). *Practical UNIX and Internet Security*. O'Reilly Media, Inc.

Sweat, J. E. (2005). *PHP Architect's Guide to PHP Design Patterns*. Marco Tabini & Associates, Inc.

Vacca, J. R. (2007). *Practical Internet security*. No Starch Press.