УДК 004.453

THE UNIVERSAL VULNERABILITY EXPLOITATION PLATFORM FOR CTF

P. Y. Sviridov, G. Y. Zaytsev, A. S. Ivachev

Capture the Flag (CTF) is a command educational computer security competition. The aim of all CTF games is to capture flags from vulnerable services of other teams. There are a lot of routine tasks in CTF games according to the rules. In order to automate the tasks, a big software project named *Pechkin* and implemented in C++ is built. The aim of Pechkin is to automate the exploitation of enemy services vulnerabilities. It runs instances of exploits, manages the instances, calculates statistics, performs logging, etc. Pechkin has a modular architecture. Each module implements one of the pointed functions and is started by the main one which is called a platform. The platform connects all the modules by passing messages between them. In different games, many parameters (e.g. the jury system interface and rules) may vary setting some restrictions. Pechkin cares about them, and the team members are free of them. The only offensive concern left for the participants is the creative process of finding vulnerabilities and writing exploits. The architecture allows the implementation of a scalable system with a load-balancing which is very important to CTF, because the game is long, unpredictable, and resource-draining.

Keywords: *CTF, flag, vulnerability, exploit.*

CTF, or Capture the Flag, is a computer security competition that is usually designed to serve as an educational exercise to give the participants an experience in securing a computer and in designing, developing, and reacting to the real-world sort of attacks. The aim of all CTF games is to capture *flags* from vulnerable services of other teams. The flags are some secret data that jury gives to the teams and the vulnerability is a service weakness which is caused by incorrect service usage or implementation. The only way to capture a flag is to exploit a *vulnerability*. It may seem that a team should find the greatest amount of vulnerabilities to win the game, but it is not necessary the truth. The team with the greatest number of enemies flags is awarded the victory.

To capture flags the participants write *exploits* — small programs that utilize vulnerabilities in order to make the enemies services behave in the favour of the exploits authors, i. e. to give them back enemies flags. The instances of the exploits are run against each team, and the collected flags are sent to the jury. The jury decides on success of capturing each flag using the information about the original distribution of flags and answers whether a flag was truly given to one team and then received from another, i. e. captured, or not. In the case of successful capture, the team that provided the flag to the jury scores points that are usually called *offensive points*, and the team which flag was captured receives less *defensive points* than usual. The overall scores are calculated as an aggregate of both offensive points and defensive points.

In order to automate a lot of routine tasks in CTF games, a big software project named *Pechkin* and implemented in C++ is built. Particularly, it does the following: runs instances of exploits, manages the instances, sends flags to the jury, checks whether the flag has already been sent, calculates statistics, performs logging.

Pechkin has a modular structure. Each module implements one of the following features:

— receiving flags from exploits;
— storing flags to the team storage (that is database);
— reading flags from the team storage;

— sending flags to the jury;
— managing instances of exploits;
— calculating statistics;
— logging the game events.

Receiving module provides the interface to the team members for accumulation of the captured flags in the team storage. The storage is a database which stores flags in a certain format.

Sending module sends flags from the storage to the jury and deals with the jury answer. Normally, the answer simply states whether the flag is accepted as properly captured or not, and some explanation is provided. However, there are some CTF games that do not allow participants to submit the captured flags to the jury under certain circumstances. In this case, the team should wait until the conditions for submission hold true and then resubmit the flag. The sending module handles this situation automatically. Another obstacle to be dealt with is the possible restriction on the number of connections from one team to the jury server. Because all team members send their flags to Pechkin, and the sending module of it has the only connection to the jury, this possible limit is also never broken.

Statistics module collects statistics and presents it at the web page. This helps the team members to understand whether their exploits are useful or not. Statistics also include the information about the teams that has already patched their services and expelled the vulnerability.

Logging module registers events and records them to the journal. It helps to analyse the game process after the game ends.

Exploit manager module starts instances of exploits, gives addresses of vulnerable hosts and takes flags returned by the instances.

Every module is started by the main one which is called a platform. The platform connects all the modules by passing messages between them. The platform has message queue for each module. If one of the modules does not cope with the tasks from its queue, the platform starts another instance of this module. All instances of one module work separately and the message mechanism allows the platform to load-balancing between their queues.

The architecture allows the implementation of a scalable system with load-balancing which is very important to CTF, because the game is long, unpredictable, and resource-draining.

All modules are C++ classes inherited from the abstract class, for they should have the same interface. The interface comprises Init(), Run(), Pause(), and Stop() methods and the pointer to its message queue. A module also must have a pointer to the platform for sending messages to other modules. However, it should not access other methods of the platform class. Therefore, a module has the pointer to the wrapper of the platform class providing only one method of the platform.

In different games, many parameters (e. g. the jury system interface and rules) may vary. When a module is initialized, it gets the name of the configuration file as an argument. The file describes some of the module parameters, e. g. database authentication parameters or the IP range of the enemy vulnerable hosts.

A module instance can be in one of the five states: NEW, READY, RUNNING, PAUSED, and STOPPED. After initialization, an instance goes to the state NEW. When the platform runs a module instance, it changes its state to RUNNING and starts to handle messages from its queue. If the queue is empty, state is changed to READY. The instance also can be paused and stopped by the platform.

The platform class implements the design pattern Singleton, for there always should be only one instance of the class. All modules classes are loaded to the platform memory dynamically [1]. The platform only has the hash table of loaded modules. This mechanism allows to create arbitrary number of modules instances and create modules during the work of the platform. The platform managing is done via the administration console. The console has the commands allowing to load the modules, run, pause, and stop them.

Pechkin automates all the attack processes in the CTF games. Team members are freed from sending flags to the jury directly or checking whether other teams have already patched the vulnerabilities. The only offensive concern left for the participants is the creative process of finding vulnerabilities and writing exploits. After submission of an exploit to Pechkin, the author can check whether his script is useful and improve it if necessary. Of course, the written exploit should correspond to the internal format of Pechkin.

BIBLIOGRAPHY

1. *Norton J.* Dynamic class loading for C++ on Linux // Linux Journal. 2000. Iss. 73. `http://www.linuxjournal.com/article/3687`