# OPERATING SYSTEMS PROJECT ON SANDBOXING

## TAJ Defender

## A brief report on its objectives

<u>**Prepared by:**</u>

**Adarsh Mohata – 12IT03**

**Ajith P S – 12IT04**

**Ashish Kedia – 12IT14**

Department of Information Technology,

National Institute of Technology, Karnataka Surathkal

# Sandboxing

**Abstract:** The objective of this project to develop a sandboxed environment for running code whose origin or content is untrusted.

In computer security, a sandbox is a security mechanism for separating running programs. It is often used to execute untested code, or untrusted programs from unverified third-parties, suppliers, untrusted users and untrusted websites. The sandbox typically provides a tightly controlled set of resources for guest programs to run in, such as scratch space on disk and memory. Sandboxing technology is frequently used to test unverified programs which may contain a virus or other malignant code, without allowing the software to harm the host device.

## Technical Objectives:

1. Capture system calls and arguments invoked by sandboxed binary programs in runtime, and block malicious actions through user-defined policy modules
2. Specify quota limit of resources allocated to the sandboxed program, including CPU cycles, wall clock time, memory, and disk output
3. Minimize privileges of sandboxed programs, and isolate their execution from critical parts of the operating system. This will be achieved through already available commands like “chroot”
4. Prevent the program from accessing network
5. Allow blind-substitution based custom-error message for reserved "keywords"
6. Prevent the function from creating a new process/thread
7. Impose a limit on time taken to execute a code i.e., output should be deterministic in time
8. Impose a memory limit on the code i.e., the process should not allocate more than the specification of the user-defined policy
9. Impose other file read-write restriction
10. Granting access to only limited system calls
11. Input and Output should be performed only through the standard input output streams
12. Prevent any kind on hack into the system

We will try to implement this as a separate kernel module if possible, so that it can be port and scaled easily. Minix already provides several tools to play around with kernel and build an easy-sandbox. The main challenge is to have an architecture-independent configuration so that the same application can be used for different system (including virtual machines) and servers.

Our support will cover C and C++ languages. Java already has an in-built sand-boxing mechanism and including this will be easy. Optionally we even plan to add support for python and shell scripts. We plan to release the end-product as open-source software for people across the world to review and re-use our code. Currently there are no-such open-source software available. The code for the project will be written mostly in C and shell script.

## Optional Objectives:

We also plan to extend our project so that it can be used to build automated profiling tools and watchdogs that capture and block the runtime behaviors of binary executable programs according to configurable / programmable policies. This will help a lot in tuning of several other programs. Implementing a sandboxed remote execution is also one of our optional objectives. Extending support to windows platform is also a great prospect for our project.

## End Delivery:

We are actually planning to implement sandboxing in our online judge systems to test programs in programming contests and/or lab assignments. The system will be able to compile and execute codes, and test them with pre-constructed data. Submitted code may be run with restrictions, including time limit, memory limit, security restriction and so on. The output of the code will be captured by the system, and compared with the standard output. The system will then return the result. When mistakes were found on the standard output, a re-judgment using the same method must be made.

## Application:

One important problem while running programs in the Linux environment is that under normal circumstances any program is allowed access to any system call. This leniency can be misused by malicious programs to do some harmful operations on the running environment. To avoid this, a solid secure-computing facility called SECCOMP for the Linux Kernel was devised by Andrea Arcangeli in January 2005. It is enabled via the prctl(2) system call. Our project TAJ Defender uses SECCOMP at the core of its operation.

## Introduction to SECCOMP:

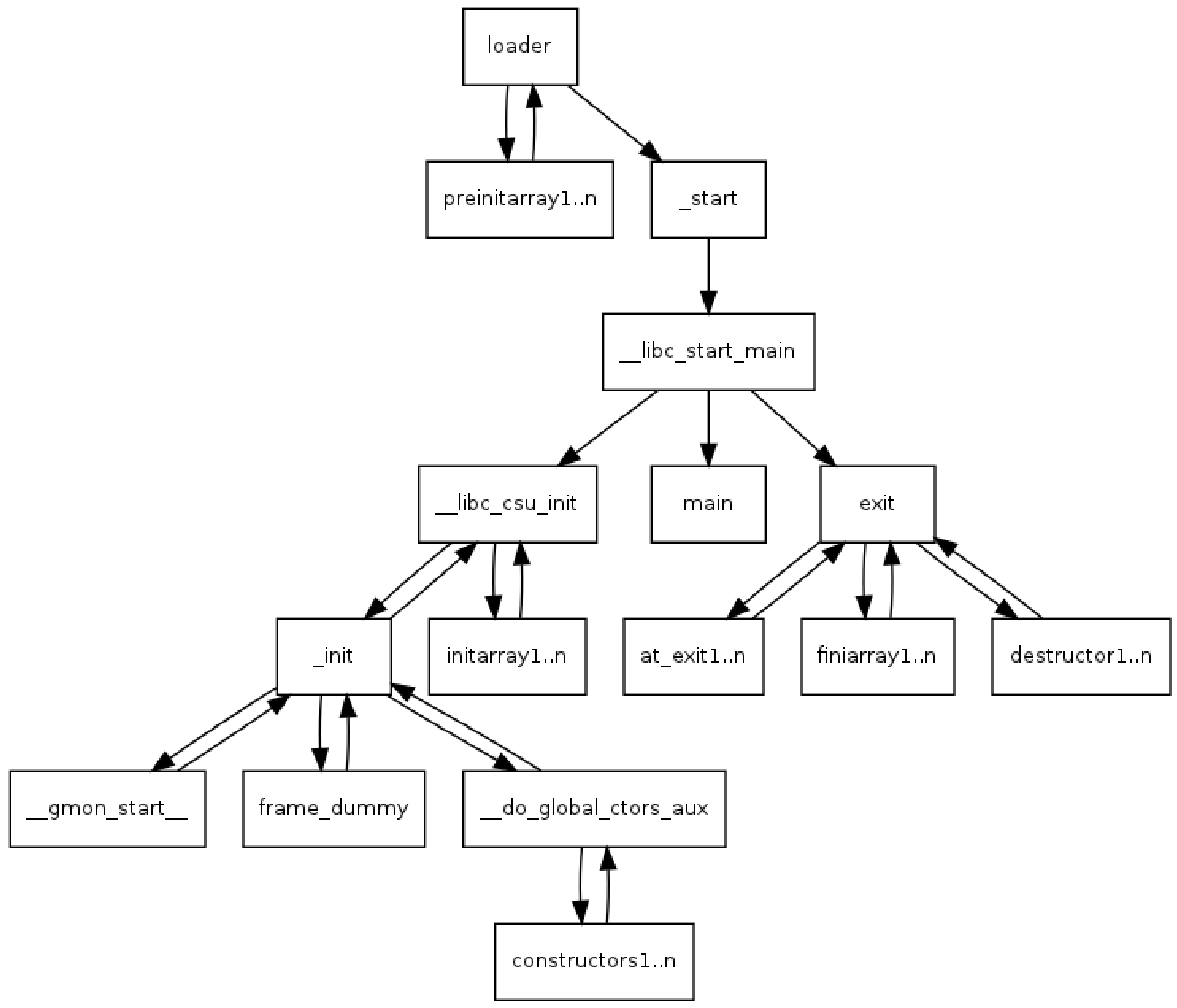
SECCOMP (short for Secure Computing Mode) is a simple but solid secure-computing facility (sandboxing mechanism) for the Linux kernel. It was added to version 2.6.12 of the Linux kernel mainline on March 8, 2005. It allows a process to make a one-way transition into a "secure" state where it cannot make any system calls except exit(), sigreturn(), read() and write() to already-open file descriptors. Should it attempt any other system calls, the kernel will terminate the process with SIGKILL. In this sense, it does not virtualize the system's resources but isolates the process from them entirely.

SECCOMP mode is enabled via the prctl(2) system call using the PR_SET_SECCOMP argument.

## Working of TAJ Defender:

In a C program, before the main function is called, an initialization function calls __libc_start_main. The __libc_start_main() function performs any necessary initialization of the execution environment, calls the main function with appropriate arguments, and handle the return from main(). If the main() function returns, the return value will be passed to the exit() function.

The following diagram explains in detail what happens before and after the main function is called.



In general __libc_start_main does the following things.

- Takes care of some security problems with setuid setgid programs
- Starts up threading
- Registers the fini (our program), and rtld_fini (run-time loader) arguments to get run by at_exit to run the program's and the loader's cleanup routines

- Calls the init argument
- Calls the main with the argc and argv arguments passed to it and with the global __environ argument.
- Calls exit with the return value of main.

Our program implements its own version of __libc_start_main and using the concept of LD_PRELOAD makes the user program start off by calling our implementation rather than the normal __libc_start_main() function.

Our implementation allocates specified amount of memory using the mmap system call and then calls the system implementation of __libc_start_main(). Our program has also implemented versions of runtime loader , sbrk, malloc, calloc, free, init and atexit functions.

- malloc() was implemented because most programs will be using malloc to allocate memory and since malloc uses sbrk system call, SECCOMP would kill it.
- init() was implemented because the SECCOMP mode must be activated before the program calls the main function of the user program. Note that we are not editing the source code rather changing the way program starts to activate SECCOMP.
- exit() was implemented because the normal exit function would involve system calls making SECCOMP kill the program.
- atexit() and __cxa_at_exit() was also implemented because that is handled by the exit function and since we implemented exit function we had to invariably implement both atexit() and _cxa_at_exit().

TAJ Defender has two layers of protection.

- Shield Layer (#define layer)
- SECCOMP Layer

1. The first layer is setup by adding the contents of the file called define.h which will contain some #define MACROs of disallowed function names. If the program contains calls to disallowed function then it will be replaced by some garbage names resulting in compilation error. This layer can take care of various hazardous code-snippets trivially. Administrator can modify the contents of this define file and can thus control the restrictions. This layer however cannot account for kernel-access without any system call for example, messing with device files using simple IO.

2. The second layer i.e., SECCOMP layer is activated in our implementation of init() using process control call prctl. Before it is activated we open all 3 file descriptors (stdin, stdout, stderr) by doing some operations on it. This is done because SECOMP will allow only those file descriptors which were opened before the SECCOMP mode is activated. This additional layer is necessary since the administrator may not have blocked all the barred system calls using #define MACROs and SECCOMP takes care of that. Moreover there are a numerous examples of malicious codes that do not contain any of the well-known system calls.

## <u>**Project Contents (Files):**</u>

- define.h :- Contains the #define macros which would limit calls to certain functions. User can edit this file to block the use of certain keywords/functions in the code being run.
- TAJdefender.so :- Contains the set of objects codes obtained after compiling the 2 main files of TAJ Defender i.e., malloc.c (Custom Memory allocation function) and TAJdefender.c
- test.sh :- A shell script to which you have to pass the filename of the source code to be executed. It should be in the same folder as the sandbox. It will first append the source code to the define.h file and then compile the source code using TAJdefender.so (LD_PRELOAD)

## Usage:

The TAJdefender is very easy to run. The source code of the user program should be placed in the sandbox folder and the shell script test.sh should be run with arguments being the filename of source code and the filename of input file. The shell script will say whether the compilation was unsuccessful, if not it will run the program and save the output file in the same folder. It will terminate with runtime error in-case there is a malicious code snippet.

## Future plans and uses:

Some features in sandbox which will be implemented in the future are:

- Time limit constraints.
- Memory Limit constraints.
- Profiling Codes

Both these constrains are a necessary part of any sand-boxed environment. The TAJ defender is part of our ongoing project. This project is an online judge and we call it TAJ – Teaching Assisting Judging. In near future this will be implemented inside TAJ customizing a little to suite the process of automatic judging. It will be able to judge a C or C++ source code submitted by the students. A sand-boxed environment for such code is necessary because the code will be executed on a server and a malicious system call can create havoc. Adding memory and time constraints to the code-execution will also enable the user to correctly predict the time and space complexity of the source-code which in turn will enforce the implementation of the intended algorithm. We hope that our TAJDefender will be able to serve the purpose. It will have 4 possible outcomes:

1. Correct Answer – Source Code ran successfully in the sand-boxed environment and the output in stdout was as expected
2. Wrong Answer – Source Code ran successfully but the output was not as expected

3. Run Time Errors – Source Code terminated during execution due to a variety of reason (Segmentation Fault, Unauthorized system call, allocating more memory etc)
4. Time Limit Exceeded – Source Code could not be executed in the given time frame.

## Group:

Adarsh Mohata – 12IT03 (amohta163@gmail.com)

Ajith P S – 12IT04 (ajithpandel@gmail.com)

Ashish Kedia – 12IT14 (ashish1294@gmail.com)

## References :

1. http://en.wikipedia.org/wiki/Seccomp
2. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
3. http://lxr.free-electrons.com/source/kernel/seccomp.c
4. http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html