Software Security project- Tracer

Vinko Mlačić, Université de Rennes 1

Introduction

The Tracer program attaches to a running process, injects payload and instruments the execution in the attached process depending on the input user provided.

There are two scenarios:

- **Cleanup:** Tracer executes the payload, cleans up every trace of injecting anything and releases the attached process so the process runs in the same manner as it ran before the Tracer attached to it
- **Inject virus:** Tracer replaces the entry function with the payload so that every time the said function is called, the payload gets executed. After injecting, Tracer releases the attached process and exits

The attached process will be referred to as the Tracee from now on and the payload as the virus function.

The Tracer has a defined virus function which gets injected into the Tracee. An interesting feature would be if the Tracer could take the payload from an external source and then inject it. However, this was not a requirement for the project.

It should be noted that currently the virus function has a limitation that it can't be linked with any library. It is possible to write a custom linkage for every payload but that felt cumbersome and to achieve an actual functionality like this which is more general, it is necessary to implement a linker.

Target

Tracer is developed for Linux operating system and x86-64 architecture and is not portable. Uses in other settings would result in unexpected behaviors and most probably malfunction of the Tracer program and the Tracee.

Requirements

There are several standard Linux CLI tools that are dependencies of Tracer. The dependencies are listed below.

- nm – used for inspecting binaries and getting function offsets
- cat – used for reading process memory and process memory mappings
- grep – used as parser so that the wanted output is easier to extract
- ldd – used for scanning for dynamically linked objects in the Tracee

- pgrep – used for getting process ID

If one of the dependencies is missing the program will not work.

Installation

For information on how to install the program or the Tracee given as example, please refer to **README.md** in the project folder.

Usage

The program has a command line interface. There are several options (both required and optional) that could be passed to the program. You can also get the help for the program by using the "-h" flag.

Example of using the program:

```
./tracer -p tracee -b /bin/tracee -e f1 -c
```

Required options:

- "-p": specifies the name of the process which will be used in getting the PID
- "-b": specifies the path to the process binary executable
- "-e": specifies the function which is the entry point where Tracer will assume control

Optional options:

- "-c": specifies what will the Tracer do after it gets control of the running process (for reference on possible scenarios see Introduction)
- "-h": displays the help (if this option is specified the program will exit without error even though required options are not provided)

More on entry function

The entry function is where Tracer will start to change process state. In the beginning, Tracee attaches and sets a **breakpoint** in the beginning of the entry function. When the program calls this function, breakpoint is hit, and Tracer will change process registers and code in order to execute arbitrary code.

The code

The code is divided into several modules. These modules handle everything from logging to code injections. They are explained in detail in the following sections. In case more detailed explanation is needed, there are Javadoc style comments detailing every exposed function in the header files.

While writing, I tried to be consistent with some coding conventions and a readable style as much as possible given the language's expressiveness. Furthermore, sometimes programming at a low level requires the programmer to do some hacks which may not be immediately clear to the reader.

Option parsing

For option parsing, Tracer leverages the getopt library. In the file "option_t.h" all options that could be passed through the command line are specified as members of a struct.

The file where all behavior is defined is "options.h". The only exposed function is parse_options which returns the fore mentioned struct.

Binary inspection

Several functions in "process_info.h" could be used for inspecting the Tracee's binary as well as shared libraries (e.g. libc).

The binary inspections usually use the "nm" utility and parse its output.

Also, in order to find out exact libc which is used by the Tracee, "ldd" is used.

Process inspection

Other function that are declared in "process_info.h" are used to inspect the actual running process.

- get_process_base_address: reads /proc/${pid}/maps
- locate_libc_in: reads /proc/${pid}/maps
- get_function_code: reads /proc/${pid}/mem
- get_pid: uses "pgrep" to find out the PID by process name

Low-level APIs

For this kind of program to work there are a few low-level modules that are needed to provide resource management and reading from important files and handling operations on a byte scale.

- pread.h: handles creating and reading from a pipe (mostly used for executing external tools such as "nm" or "cat")
- procmem.h: handles reading and writing to a process memory and contains functions that allow doing for bytes, words or arbitrary sized memory blocks

Code injection

For injecting code into a running process there are functions declared in "inject_code.h" header. They allow injecting breakpoints, indirect calls to functions or trampolines.

With exception of trampolines, all code injections after the actual instruction vector inject an interrupt. This was done so that the Tracer can regain control after it lets the injected code to run.

Ptrace

For the program to work there was no other alternative than using the ptrace library. The ptrace library proved to be very cumbersome to work with being that more functionalities are combined inside one function.

Therefore, a wrapper over the API was made in “ptrace_wrapper.h” header.

Several benefits are gained by this. Firstly, the error handling is placed inside the wrapper functions so that it doesn’t clutter the calling code. Secondly, in case a better alternative becomes an option in the future, migrating from ptrace to it would take simply changing the functions in the wrapper, not everywhere the ptrace is needed.

Process state

Managing process’ register values and code changes proved to be a very daunting task since these things are very low level and sometimes it couldn’t be clear what is being done in some pieces of code (even for the author).

In headers “pstate_t.h” and “pstate.h”, there is a definition of a data structure and its behavior respectively. This abstraction is used to simplify the problem with state management of the Tracee.

Data, which is stored in the struct, includes code changes and original registers from when the Tracer gained control so the control could be returned to the Tracee.

The code changes are generated by code injections. Every code injection means overwriting the original code. The original code is stored in the pstate so it can be reverted when the injected code finishes executing.

This abstraction is very important since the bugs that occur from mistakes made in Tracee’s state management are uninformative (usually segmentation faults) and make debugging a hard task.

Logging

There are five logging levels implemented in the program in this order of importance: trace, debug, info, warn and error.

The logging is defined in modules “console.h” and “log.h”. In “log.h” there are macro definitions which refer to functions in “console.h”. The macros are used so that specific logging output could be suppressed if needed. The only exception is error output which is can’t be suppressed and always outputs to stderr.

Info is the most important log level which outputs information about a normal program flow.

Warning log level is used to report strange occurrences in the program input which could lead to an error later in the program flow but could also result in a normal program execution.

The makefile specifies which log level are on by default (info and warning) but you can suppress this by setting MACROS variable when invoking make.

Macros that enable/disable logs are:

- TRACE_ENABLE
- DEBUG_ENABLE
- INFO_ENABLE
- WARN_ENABLE
- LOG_ALL (enables all log levels)

Error reporting

Error reporting is defined in modules “t_errno_t.h”, “t_error.h”, “t_error.h”.

Modules:

- “t_errno_t.h”: defines t_errno_t enumeration which holds all possible error codes which could occur in a program flow
- “t_error_t.h”: defines a more complex struct which holds the t_errno_t error code and more details about how the error occurred
- “t_error.h”: provides functions which allow error reporting (RAISE macro) and error catching

The error reporting is done in a similar fashion as it is done with the standard errno global variable.

There is a global variable of type t_error_t which is initialized to success state. When a programmer wants to report an exceptional behavior, she can do so by calling RAISE macro. This will set the global t_error to a corresponding error code, it will provide the error message and more information that are extracted from a place in the code where the RAISE macro is called (like line number and source file).

It is important to check the global variable after functions that could have set it return in order to validate their postconditions.

Validation

In order to remove the undefined behaviors and bugs the code was compiled with almost all warnings in compilers gcc and clang.

The code was also checked by a static analyzer.

After this the code was checked by sanitizers.

The reported bugs were corrected and false positives by setting correct flags in the makefile.