Apriorit Code Protection for Linux: Scheme, Main Components

The white paper describes the technology of code protection for Linux applications, which is based on the so-called "Nanomite" approach applied previously for Windows systems.

It is one of the modern antidebugging methods that can be also effectively applied for the process antidumping.

Apriorit Code Protection for Linux is provided as commercial SDK with various types of licensing.

Project Description

The project was written for Linux OS 32-bit applications. But the principles can easily be implemented for other operating systems, so further development is planned.

First, we will take a look at creating a custom debugger for Linux. After that, we will move on to the implementation of nanomites. Binutils and Perl are used for the compilation of the project.

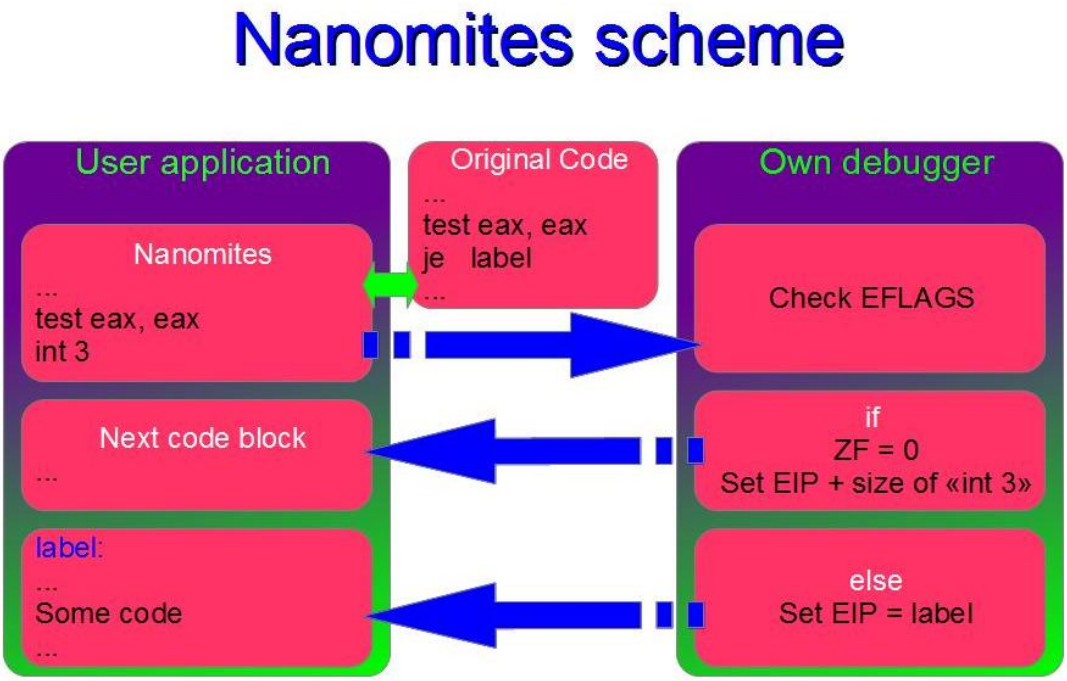
We apply the combination of two techniques: Nanomites and Debug Blocker.

Nanomites are code segments, containing some key application logic, marked with specific markers in source files. Protector cuts such segments out from the protected program for packing. When unpacking, they are obfuscated, written to the allocated memory, and jumps replace them in the original code. The table of conditional and unconditional jumps is built, and it contains not only nanomite jumps abut also some non-existent "trash" ones. Such "completness" is a serious obstruction to recover this table.

Debug Blocker implements **parent process protection**. Protected program is started as a child process, and protector - parent process - attaches to it for debug. Thus, for a third party, it is possible to debug only parent process. Combined with nanomite technology, Debug Blocker creates reliable protection for an application, making its debugging and reversing very complicated and time-consuming.

Both techniques were successfully used in commercial Windows protectors. Apriorit Code Protection is the first product to implement them for Linux application protection.

General Idea



Apriorit Code Protection includes 2 main components:

1. Nanomites: a static library that contains the debugger process logic.
2. Nanomites Debugger: a debugger executable file, it is compiled with the Nanomites library.

Also we provide Nanomites Demo: a demo application protected by nanomites.

There's also a script collection for adding the nanomites to an application and for creating nanomites tables.

Protected Application Creation Sequence

An application with an –S key for creating an assembler listing is created;

The assembler listing is analyzed with Perl script. All jump and call instructions (e.g., `jmp, jz, jne, call`, etc.) are processed and replaced with `instructionOffsetLabel(N): int 3;`

After that, the user application, which consists of modified assembler listings, is compiled.

With the help of a Perl script, a compiled application is parsed and the table of nanomites is built.

Debugger Library Description

Our debugger is based on the **ptrace** (process trace) system call, which exists in some Unix-like systems (including Linux, FreeBSD, Mac OS X). It allows tracing or debugging the selected process. We can say that ptrace provides the full control over a process: we may change the application execution flow, display and change values in memory or registry states. It should be mentioned that it provides us no additional permissions: possible actions are limited by the permissions of a started process. Moreover, when a program with setuid bit is traced, this bit doesn't work as the privileges are not escalated.

After the demo application is processed with scripts, it is not independent anymore, and if it is started without a debugger, the «segmentation fault» appears at once. The debugger starts the demo application from now on. For this purpose, a child process is created in the debugger, and then parent process attaches to it. All debugging events from the child process are processed in a cycle. It includes all jump events; parent process analyzes nanomite table and flag table to perform correct action.

The Advantages of Apriorit Solution Compared to Armadillo

Armadillo (also known as SoftwarePassport) is a commercial protector developed for Windows application protection. It introduced nanomite approach, and also uses Debug Blocker technology (protection by parent process).

In Armadillo, the binary code is modified. That's why when a 2-5 bytes long jump instruction is replaced with a shorter 1 byte long `int 3 (0xcc)` instruction, some free space remains. Correspondingly, we need to write the original jump instruction over `int 3` to restore a nanomite.

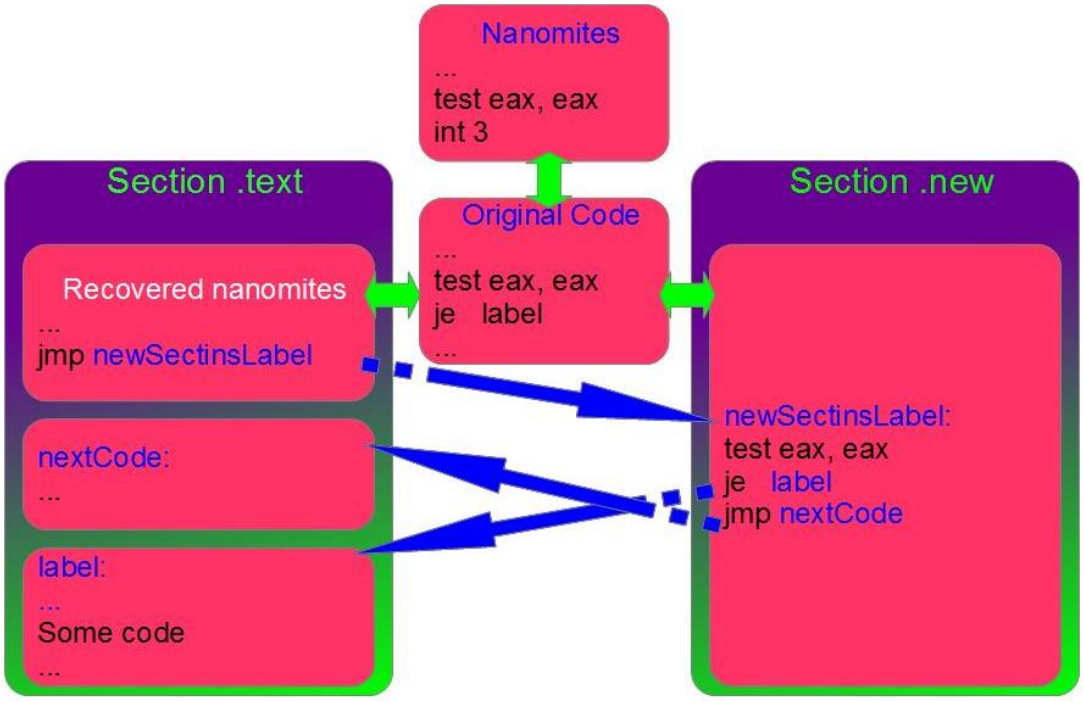
We change the code on the sources level in our approach. That's why the nanomite will be 1 byte long. Correspondingly, we won't be able to restore the nanomite by writing the original instruction over it. And we cannot extend the code in the place of the nanomite as all relative jumps would be broken. But there is a way to restore our nanomites, for example the following.

A Way to Recover Apriorit Nanomites

A hacker can create an additional section in the executable file, then find the nanomite and obtain its jump instruction and jump address.

Then the restoration goes as follows:

Recover nanomites



Such solution is complex in implementation. Firstly, a disassembler engine is required for automation, secondly, the moved instructions may contain jump instructions with relative jumps, which will require corrections.