Developing a dynamic library loader for SODIUM, an educational operating system

Nicanor Casas[1], Graciela De Luca[1], Sergio Martín[1],
Gabriel Bonanno[1], Mario Bondar[1], Esteban Carnuccio [1], Elizabeth Guardia[1],
Alfredo Portero[1]

[1] Universidad Nacional de la Matanza
Departamento de Ingeniería e Investigaciones Tecnológicas
Dirección: *Florencio Varela 1703* - Código Postal: 1754
`{ncasas, gdeluca, smartin}@unlam.edu.ar`
`gab.peekaboo@gmail.com, mariobondar@yahoo.com.ar,`
`esteban_c23@yahoo.com.ar, eguardia@gmail.com,`
`acportero@yahoo.com.ar`

Abstract. The development of a dynamic program loader allows the processes to execute shared library functions across different memory segments. This research incorporates the use of shared memory, and the ability to link and invoke dynamic libraries into SODIUM, and educational operating system. Through the analysis of the Executable and Linking Format (ELF) generated by the GCC compiler, we were able to perform a runtime exchange of memory administrators, while showing the way that the shared memory is assigned, even with data segments, such as re-entering code. To this end, we made amendments to the implementation of a dynamic library for segmentation mode, also defining the changes for paging mode. These changes were made through adaptations to the memory loader of SODIUM, to allow it to recognize the ELF format in order to use for memory address assignment.

Keywords: ELF, Dynamic Library, Reconfigurable Memory Administrator, Dynamic Loader, STUB , Shared Memory.

1 Introduction

Originally, SODIUM executed on IA-32 architectures, and the user programs were only linked statically, i. e., that all its components were placed along in a single file. One of the main limitations of this approach is that it doesn't allow the usage of dynamic link libraries, thus rending code sharing impossible. In order to implement these libraries, we have studied the existing executable metadata standards, and decided to use ELF [1] due to the availability of documentation and the GCC compatibility. For this, we analyzed the diverse sections of the ELF files as a way to guarantee the correct implementation of the dynamic link libraries [2].

2 Dynamic Loader

2.1 Load-time reference resolution

All the references to dynamic libraries are resolved by a module linker that allows deferring the linking of their routines with the user programs after the compilation. The linking module of a program contains non-yet-resolved references to other programs or libraries. These references can be resolved in two different moments: while resolving links on the program load-time, or even while the program is being executed [3].

As the operating system created the image of a process, it copies the logical segments of the file to real memory segments. Illustrated in the Figure 1.a., there is an example of a metadata file in ELF format, while the contents of its heading members are depicted in Figure 1.b.

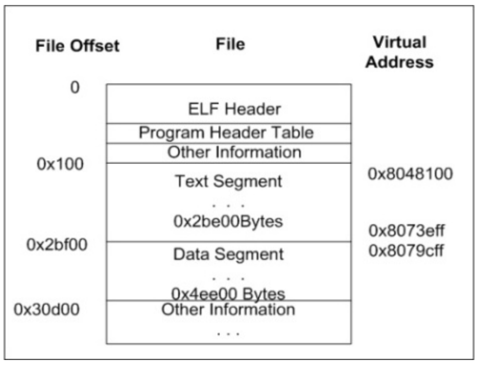
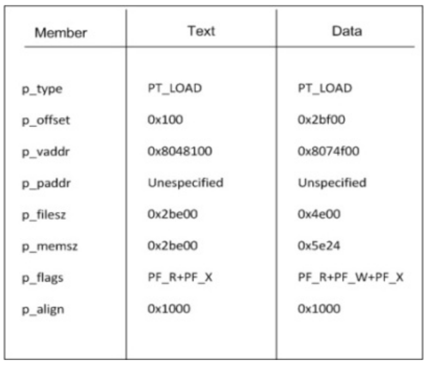


Figure 1.a. ELF metadata file



Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	Unespecified	Unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x1000	0x1000

Figure 1.b ELF file header contents

The end of a data segment requires a special handling for non-initialized data, which are defined initially with zero-values by the operating system.

The loading of an executable file differs from the loading of a shared object in the way that the first ones contain generally only absolute references. The segments must reside in the virtual address that is used to build the executable, whereby the operating system must use the *dep_vaddr* value as the virtual address [4], and, as opposite, the segments of a shared object contain *position-independent code* (PIC) [5]. This allows the virtual address of a segment to vary from a process to another without modifying its behavior, keeping its relative positions unaltered.

By utilizing this approach, we are able to perform *implicit linking* [8] that differs from the runtime reference resolution in that no stub [6] is created to aid the operating system to resolve the references in a later moment. When a call to a dynamic link library function is found, the linker adds metadata to the executable file to indicate the operating system where the corresponding code can be found.

When the program is loaded into memory, the operating system looks for all the calls to dynamic link libraries that the program will use [5], within the ELF metadata file. If the library is not yet present in memory, it is then loaded and a code segment descriptor (CS) is assigned to it in the global descriptor table (GDT) [7]. In order to perform this, the operating system must maintain a table with all the libraries' addresses loaded at that moment.

When a call or branch referencing a dynamic link library function is found within the loaded code, the ELF table must be checked in order to verify that there is an entry for it. If that is the case, its CS descriptor is retrieved and a lookup search is performed for the invoked routine's relative address offset. Then, the unresolved reference is replaced by the code segment descriptor plus the obtained offset. When all the unresolved references are replaced, the program is ready to be executed.

2.2 Runtime reference resolution

When compiling a program with the ELF format, the compiler replaces all the external routines calls within the code by non-resolved symbolic references. Afterwards, when the binary image of the process is created, a little fragment of code called *stub* is injected. All the added stubs are organized within the file by a structure called *procedure linkage table* (PLT) that contains a jump to a *global offset table* (GOT) [9]. Through the GOT, the linker allows the dynamic link libraries to be shared among several processes. This is possible because the libraries are built with process-independent code (PIC).

The GOT table contains the absolute addresses for all the static data being referenced within the program; but also possesses the real direction of the extern routines, although this one is not known until the application actually executes. The role of the runtime linker is to fill the entries of the GOT table with the actual process addresses, allowing the GOT to keep direct references to all of a library's symbols [4]. The GOT address is usually kept in the EBX register that relatively references it. Each shared library and executable object that use shared libraries have their corresponding PLT and GOT tables (see Figure 2).

Each entry on a PLT corresponds to an external routine, and consists of an indirect branch to the GOT table. Then, if a dynamic link library routine must be acceded by a process in runtime, its real memory address can be obtained simply by an indirect branch to the corresponding GOT entry through the PLT. [6]

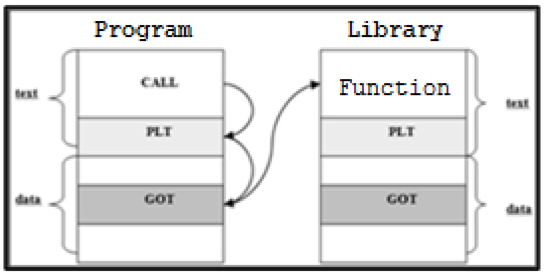


Figure 2. Executable file and shared memory scheme.

This approach implies the usage *lazy procedure linkage*, that is, the explicit invocation of external libraries through the usage of the instrumentation of the program's code in order to let the linker decide whether and when they should be loaded [7].

3 Design

3.1 Memory management

We decided to work with the simple segmentation memory scheme due to its relative ease of analysis and coding, and to the fact that, currently, SODIUM shows more stability while executing with segmentation mode. The paging mode needs just little adjustments [10] that should be easy to carry on over the current SODIUM's implementation of dynamic link libraries.

When using segmentation, each process –whether dynamic or statically linked– is loaded into a single memory segment, and the library is loaded into another, making sure that both are present within the same user memory space.

3.2 Dynamic library loading

The different approaches of implementing dynamic library loading change according to the time that they are carried on –whether in runtime, or while loading the process into memory–.

Due to the SODIUM's particular architecture, we opted for a runtime reference resolution implementation. These are resolved on demand, determining the undefined functions' virtual addresses just-in-time, that is, exactly before they are executed. In order to achieve this implementation, special instructions are created to load the dynamic link libraries in memory, obtain their functions' virtual addresses, execute them, and then free the utilized memory segment. These special instructions must be used within the code of the semi-dynamic processes. The names used for these directives are similar to those generally used in Linux [12].

3.3 Libc library dependencies

The compilation of all the SODIUM core files is done within a LINUX environment, where the compiler implicitly adds the the *libc.so* library (C language standard library for LINUX) [11] to all of the dynamic ones. To be able to generate an executable over SOIDUM, the *libc.so* library was changed in all its makefiles, generating the necessity of linking the *_start.o* and *libsodium.o* libraries statically, in order to compensate for the original *libc.so* absence. These libraries main responsibilities include, but are not limited to:

- Initialize the process' heap and stack
- Call the program's main procedure
- Receive the main's return code

Therefore, we had to modify the way that those files were compiled in the way that no association exists, using a similar library for the programs and libraries handling, generating the implementation of the *_start* routine, that is the very first one to be invoked when executing a semi-dynamic process. This routine is in charge of executing tasks such as initialize the process' heap and stack, and also to point to the program's start procedure. This last task is carried on by invoking the *main* symbol of the dynamic program.

3.4 Compilation and loading from SODIUM virtual drive

In order to carry the corresponding tests, a new dynamic link library named *libiblio.bin* had to be created and placed within the */usr/lib* folder. This library currently counts with two simple functions: *cycle* and *sleep*. Also, two semi-dynamic processes invoking those functions were generated using two code files, named *program1.bin* and *program2.bin.* These files were placed within the */usr* folder.

Manually created PLTs had to be created instead of the GCC's automatically generated PLTs [11], in order to be able to correctly implement the runtime reference resolution. All programs and libraries files are statically linked to the manually created PLTs.

Multiple SODIUM *makefiles* also had to be modified. The necessary changes included:

- **/usr/lib:** the *libiblio.bin* library compilation was added and linked statically to the *pltInput.o* file.
- **/usr:** The compilation of the semi-dynamic executable files *program1.bin,*and *program2.bin* was added. Those are dynamically linked to *libiblio.bin* and statically linked to *_start.o, libsodium.o,* and, *pltOutput.o*
- **/solo** and **/build:** The makefiles for both folders were modified to allow the semi-dynamic processes and the dynamic link library within the root folder of SODIUM's virtual drive in memory.

The execution of the *ls* command from SODIUM's command line allows the display of a list with all the stored filed within the root folder of SODIUM's virtual drive. All of the three files created are included there:

- *Program1.bin***:** semi-dynamic object file.
- *Program2.bin***:** semi-dynamic object file.
- *Libiblio.bin*: Dynamic link library.

3.5 Dynamic Loader

In order to avoid breaching the SODIUM security levels, the dynamic loader is implemented as part of the operating system's kernel. By doing this, we allow the direct invocation of functions that are placed within the *gdt.c* file, as they are placed within the same file, forming part of SODIUM's kernel.

3.5.1 Dynamic program object files loading into memory

Both *program1.bin* and *program2.bin* semi-dynamic object files are loaded into memory is performed when any of them is executed from the command prompt.

If *program1.bin* is executed, the *iFnExecuteBinary* function of *sodshell.c* is invoked. This function executes a *fork* syscall (to duplicate the parent process, thus creating a child process), and then the *execve* syscall is invoked to instantiate the child process with the code image from the *program2.bin*. Whenever SODIUM receives the *fork* and *execve* syscalls, it executes the *iFnDuplicateProcess* and *iFnReplaceProcess* functions present in the kernel's *gdt.c* code file, respectively.

We had to make a modification to the normal loading of each executable to skip reading the address representing the actual object file start, in order to avoid including the ELF metadata header as executable code. This modification worked correctly with just statically compiled programs, and it took a simple adaptation for dynamically linked programs with a generalized load procedure for both types of executables.

Finally, we opted for implementing the loading retrieving the *LOAD* segments from the *program header section* from the ELF headers. The statically compiled programs count with just one *LOAD* segment, while the dynamically linked programs count with at least two. We determined that the best way to perform the loading was unifying both *LOAD* segments into one, and loading it into a single segment. To achieve this, we had to create a file named *linkerscript.lds* that is used during the *program1.bin*, *program2.bin*, and *libiblio.bin* for determining the virtual address of the start of the *LOAD* segment and where it is going to be loaded in memory, as well as which sections will conform them. The last step was to modify the size of the ELF header in the function *iFnReadExecutableHeader* from the *gdt.c* code file, just to avoid any load address miscalculation.

3.5.2 Dynamic library files loading into memory

The inclusion of the dynamic link library is done by loading the *libiblio.bin* object file in memory. We implemented the *dlopen* syscall to aid the resolution of dynamic references during the execution of the semi-dynamic processes. This syscall must be invoked from the *main* procedure of *program1.bin* or *program2.bin* using the following syntaxis:

```
selector = dlopen("library_name");
```

Dlopen invokes *iFnLoadLibrary* –present in the *gdt.c* kernel code file– automatically, while the ES selector register is associated to the code descriptor of the dynamic library in memory, allowing the process to access it. In this manner, while, initially after one semi-dynamic process performs a *fork* syscall, the child's process ES descriptor is set to point to the GDT entry of the parent process, when it executes the *dlopen* syscall, that value is replaced by the base segment address where the dynamic library is loaded (See figure 3).

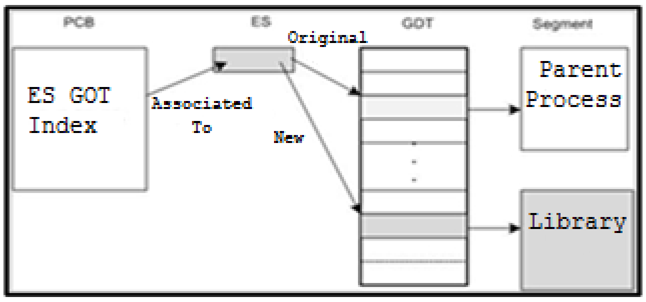


Figure 3. The ES register selector is replaced by the dynamic library GDT position

In a temporal context, the dynamic library is loaded into memory just after the first time a process invokes it. When another –or the same– process needs to invoke it too, it is already present in memory and does not need to be reloaded.

In a spatial context, the dynamic library is loaded always in higher memory position than the calling process. This was implemented this way simply because in order to load it before we should have evaluated the program's library requests in load-time, which might incur in unnecessary overhead and memory usage if the process ends up not using them. Also, one cannot preventively fix up free space upfront, because the size of the library is unknown until it is invoked. However, one exception for inverse library positioning is that a first process invokes a library before another process comes into memory and uses it (see Figure 4).

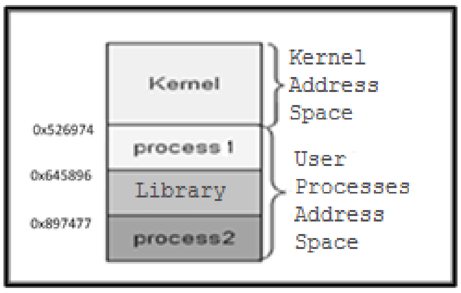


Figure 4. Possible memory setup for memory distribution using dynamic link libraries

4 Results and achievements

After the research findings and further development, SODIUM counts with a dynamic link library that allows:

- The use of the *dlsym* syscall that allows obtaining the relative offset of the address of any function within a dynamic link library. This offset is obtained from the *value* field of the library's symbols table. The *dlsym* syscall must be invoked within the *main* procedure, and must be executed only after having used *dlopen*, using the following syntax:

  ```
  offset = dlsym("library_name", "function_name");
  ```

  Upon execution, *dlsym* automatically invokes the kernel function *iFnObtainDirFunction,* placed in the *gdt.c* code file, which looks up for the function's offset within the library's symbols table. Once found, retrieves it to the semi-dynamic process through the EAX register.

- The actual execution of a function within a dynamic link library. Right now it is limited to be invoked from the main of a user process, but it can be straightforwardly upgraded to a more general usage. It must be executed after *dlopen* and *dlsym* with the following syntaxis:

  ```
  vFnFuncDin(unsigned int selector, unsigned int offset);
  ```

  This function is a very simple assembler programmed stub –which we placed in a code file named *pltOutput.asm*– which is compiled and statically attached to every semi-dynamic compiled program. It receives two parameters: the first one, *selector*, is the GDT selector position where the dynamic link library is loaded obtained from the previous execution of *dlopen*; and the second, *offset*, is the relative offset of the address where the function to be executed is placed obtained from the previous execution of *dlsym*.
  Upon receiving the offset value, it is stored into the EAX register to be used as input for the *vFnPLTlibrary* –which we placed in a code file name *pltInput.asm*

that is statically linked into the called library–, to use it for the actual call to the external function.
The scheme of both –input and output– PLTs placement and function is depicted in Figure 5. The narrow arrows represent the calls to the input PLTs, and the thick ones represents the ones that execute the *retf* return instruction.

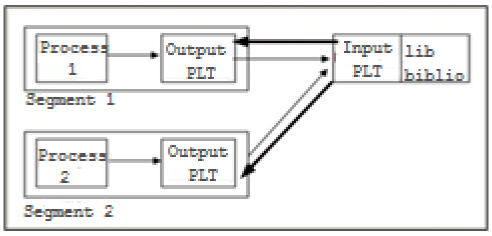


Figure 5. Call and return to an input PLT and from an Output PLT, respectively.

One of the main reasons why we decided to implement two different PLT files (for input and output) and not just one (only for output) is that all the functions from the dynamic link library are compiled with normal *ret* instruction returns that is limited only to the current memory segment. However, since all the libraries will be placed in their own segments, we rather needed to execute a *retf* instruction that allows a inter-segment return. The *pltInput.asm* file serves then as a wrapper for each function call that takes the EAX register as the offset and re-calls internally with a simple *call* instruction. When the function returns, it does so with a *ret* instruction back to the PLT wrapper that takes the return value and does execute the needed *retf* return instruction. Otherwise, when we first implemented direct calls, we would be executing the functions, but not being able to return (usually a CPU exception occurred).

- The ability to clean up the memory segment assigned to the library after its usage. For this purpose, we developed the *dlclose* syscall that must be invoked from the *main* procedure after having executed the *vFnFuncDin* function. This syscall automatically calls the *iFnCloseLibrary* function placed in the *gdt.c* code file, that frees the segment occupied by the library, modifies the segment descriptor of all the processes pointing to it back to their parent processes' code segments.
  One inconvenience carried by this approach is that if one of the semi-dynamic processes executes the *dlclose* syscall, it won't be able to further share the library. This happens because, when a process executes *dlclose*, the segment is freed and the other process will have to reload the library. In order to fully seize the benefits of the sharing library, none of the processes must execute *dlclose*. By doing this, the loaded libraries will still be available after the first process that needed it finally exited; if a further process needs to use it, it will be already available in memory, and there won't be any need for reloading. However, it is the responsibility of the operating system to keep track of those libraries that are still open and most likely won't be used in the future.

5 Conclusions

Due to the static nature of all the previous SODIUM's user processes, many modifications in the way of compiling the kernel source code files had to be done, in order to allow the copy of a program to a virtual drive and then loaded into memory. Until now, the performed adaptations aimed to achieve a more stable operating system that could run both static and dynamically compiled and linked programs, and that also could handle dynamic calls through different memory segments.

SODIUM allows now the usage of dynamic link libraries with limited functionality, although clearing the path for further needed extensions in the future. Its use will be extended to a n-quantity of processes, and the SODIUM standard library (*libsodium)* will be changed from statically to dynamically linked. These changes will greatly improve stability and the horizon for further developments.

This research was carried out by both professors and alumni of the Universidad Nacional de La Matanza Advanced Operating Systems class, and had a double educational value: in part, for those alumni involved in the research that could learn about both intricacies of program linking and memory segments administrations; and for the forthcoming alumni that will be able to experiment and learn about dynamic library loading by consciously experimenting with the explicit syscalls hereby presented.

6 References

[1] Youngdale, E.: The ELF Object File Format by dissection. Linux Journal, 1995.
[2] Darlet, P.: Runtime Loader-Linker Technologies. Embedded System Conference, 2001.
[3] Stallings, W.: Operating Systems Internals and Design Principals, Prentice Hall, 2008.
[4] Executable and Linking Format (ELF) Specification Version 1.2, Tool Interface Standard (TIS). 1995.
[5] Levine, J. R.: Linkers & Loaders. Morgan Kaufmann Publishers, San Francisco, 1999.
[6] Silverschatz, A., Galvin, P., Gagne, G.: Operating System Concepts. 7th edition. John Wiley & Sons, 2004.
[7] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M. Intel, 2008.
[8] Tanenbaum, A.: Structured Computer Organization. 4th edition. Prentice Hall,1998.
[9] Muchknick, S. S.: Advanced compilers: Design and Implementation. Morgan Kaufmann Publishers. 1997.
[10] Casas, N., De Luca, G., Cortina, M., Puyo, G., Valiente, W.: Implementación de diferentes tipos de memoria en un sistema operativo didáctico. In proceedings of XIV Congreso Argentino de Ciencia de la Computación (CACIC). Argentina, 2010.
[11] Stallman, R. M.: Using and porting the GNU compiler collection. Iuniverse Inc, 2000.
[12] Gorman, M.: Understanding the Linux, Virtual Memory Manager. Prentice Hall, 2004.