Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

Applying Partial Virtualization on ELF Binaries Through Dynamic Loaders

Tesi di Laurea in Sistemi Virtuali

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Federico Pareschi

Sessione III
Anno Accademico 2011/2012

Applying Partial Virtualization on Elf Binaries Through Dynamic Loaders

Federico Pareschi

March 9, 2013

Abstract (Italian)

La tecnica della virtualizzazione parziale è un approccio rivoluzionario nel mondo della virtualizzazione. Viene a collocarsi nel mezzo tra la virtualizzazione completa di interi sistemi (come QEMU e XEN) e quella di singole applicazioni (come JVM e CLR). ViewOS può essere considerato l'avanguardia di tale tecnologia, sviluppato ad-hoc dal laboratorio Virtual Square con lo scopo di fornire ad ogni singolo processo una visuale personalizzata delle risorse del sistema su cui si trova, contrastando il principio della *Global View Assumption*.

Virtual Square fornisce diverse tecniche per ottenere una virtualizzazione parziale all'interno del sistema di ViewOS, sia a livello utente che a livello kernel. Ognuno di questi diversi approcci ha i suoi vantaggi e svantaggi. Questa tesi si occupa di fornire un'analisi delle diverse metodologie di virtualizzazione, sia tradizionale che parziale, e dei problemi ad esse correlati.

Questa tesi è il risultato di un'estesa ricerca per individuare una nuova tecnologia da impiegare nella creazione di virtualizzazione parziale, basandosi sullo standard di eseguibile ELF. Inizia con una breve analisi delle attuali alternative nel campo della virtualizzazione, per poi focalizzarsi sullo studio del sistema di ViewOS, evidenziando i problemi correnti presenti in esso. Il progetto *vloader* viene quindi proposto come una possibile soluzione ad alcuni di questi problemi, con tanto di proof of concept funzionante ed esempi che ne dimostrano le potenzialità.

Facendo injection di codice e librerie nel mezzo del processo di loading dinamico degli eseguibili ELF, il progetto vloader riesce a promuovere un approccio semplificato ed immediato per tracciare svariate system call. Con i vantaggi elencati in seguito, questo metodo presenta performance migliori e più portabilità fra i vari sistemi, rispetto alle attuali implementazioni di ViewOS. Per concludere, vengono presentati anche alcuni svantaggi e problemi, con tanto di possibili soluzioni.

Abstract (English)

The technology of partial virtualization is a revolutionary approach to the world of virtualization. It lies directly in-between full system virtual machines (like QEMU or XEN) and application-related virtual machines (like the JVM or the CLR). The ViewOS project is the flagship of such technique, developed by the Virtual Square laboratory, created to provide an abstract view of the underlying system resources on a per-process basis and work against the principle of the Global View Assumption.

Virtual Square provides several different methods to achieve partial virtualization within the ViewOS system, both at user and kernel levels. Each of these approaches have their own advantages and shortcomings. This paper provides an analysis of the different virtualization methods and problems related to both the generic and partial virtualization worlds.

This paper is the result of an in-depth study and research for a new technology to be employed to provide partial virtualization based on ELF dynamic binaries. It starts with a mild analysis of currently available virtualization alternatives and then goes on describing the ViewOS system, highlighting its current shortcomings. The vloader project is then proposed as a possible solution to some of these inconveniences with a working proof of concept and examples to outline the potential of such new virtualization technique.

By injecting specific code and libraries in the middle of the binary loading mechanism provided by the ELF standard, the vloader project can promote a streamlined and simplified approach to trace system calls. With the advantages outlined in the following paper, this method presents better performance and portability compared to the currently available ViewOS implementations. Furthermore, some of its disadvantages are also discussed, along with their possible solutions.

"You can't trust code that you did not totally create yourself. No amount of source-level verification or scrutiny will protect you from using untrusted code. As the level of program gets lower, [...] bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect."

- Ken Thompson, Reflections on Trusting Trust, 1984

Contents

List of Figures

1 Introduction

In the modern world of computing, virtual machines are employed in the majority of development environments. If you are a developer, a system administrator or even just a normal user, chances are you have been interacting with virtualization every day. As a system administrator, managing multiple virtual machines on a single host is much easier and more scalable than running everything on individual and standalone machines. It provides an abstraction layer from hardware defects and software incompatibilities.

On a smaller scale, targeting individual developers, virtualization can be provided through an integrated development environment. With quick prototyping assisted by an interpreter and an appropriate language, like Java or C#, applications are often run on their own specialized virtual machine. It is much simpler to program with a higher level language that can take advantage of runtime optimizations and automatic memory management, compared to some more traditional languages like C. It takes a great burden off the shoulders of individual programmers.

A lot of research effort is still currently invested in optimizing and improving different virtualization techniques and mechanisms. Most of this effort, however, is focused on researching new ways to achieve better results and gain more and more performance out of the already existing virtualization methods. Very few studies are actually aiming to propose new virtualization classes and approaches to make the lives of the end users even easier than they currently are.

The concept of partial virtualization is exactly that: a newly proposed class of virtualization that, unfortunately, has yet to reap the benefits of extensive research and development. The idea is to provide a virtual machine monitor to abstract individual applications and generate a separate view of the host's resources to each process. This approach has been formalized by the Virtual Square laboratory through the ViewOS project.

Furthermore, there are multiple techniques that can be employed to achieve partial virtualization. The current state of the ViewOS project supports different approaches, each with their own pros and cons. This paper tries to improve on that by proposing a new technology and proof of concept, the vloader, with also its own pros and cons, improving on performance and simplicity but losing on security compared to the alternatives. The vloader project applies system call tracing and hijacking in order to create partial virtualization to individual ELF binaries on Linux systems.

2 The ELF Binary Format

The Executable and Linkable Format (ELF) is the de-facto standard for executables in any modern Unix Operating System. It was first created and implemented on System V machines[1] and then properly standardized by the Tool Interface Standard[2]. It was later adopted by multiple Unix machines and Unix-like systems, notably by Linux.

There are three main types of ELF object files: relocatable, executable and shared objects.

Relocatable Files They hold all the data and code required to be linked together into a single executable or shared object by the static linker.

Executable Files These are what normal users refer to when talking about "programs", the file type contains the specifications and details required to be executed by the operating system.

Shared Object Files They hold the code and data required to be linked in two specific and different contexts. First, the static linker can process these together with other shared or relocatable objects to create a new object file. Second, the dynamic linker can process and combine them into a single executable file to create a running process image in the operating system memory.

The versatility of this format is that it can be both interfaced at static and dynamic link time: the file layout presents a dualistic interface depending on how and when the object file is being read.

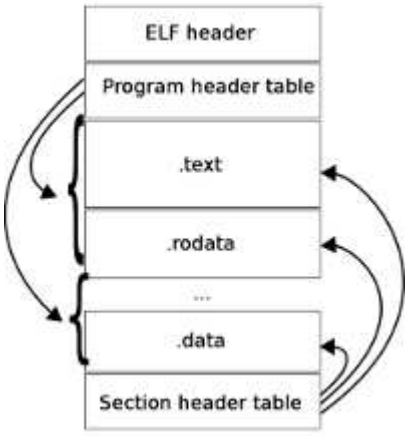

Figure 1: The layout of an ELF file.

To achieve such dualism and have each object suitable for multiple tasks the ELF binary layout, as portrayed in Figure 1, is composed of a file header, a

program header table and a section header table which instruct the linkers and loaders on where to find and how to read the required data:

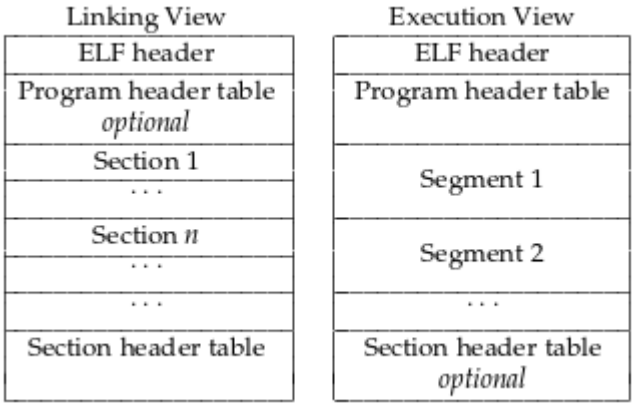

Figure 2: The structure of an ELF file being read.

- The ELF header resides at the beginning of the file and holds data describing its structure, which ELF version is being used, what is the target machine architecture and where to find the section and program headers.
- The section header table contains information describing each file section. Sections are used by the static linker to assemble relocatable files together as executable or shared objects.
- The program header table tells the system how to create a process image, it is not needed on relocatable files but is required on executable ones.

This paper will be focusing mostly on program header segments as they play a focal role in the virtualization process.

2.1 Shared Libraries

Shared ELF objects are often used as a means to achieve proper shared libraries on a system. A shared library is a piece of data and collection of sub-routines that can be shared and changed in a modular fashion[3]. There can be both static and dynamic libraries usually with either .a or .so extension and both of them play a different role in the process of building and running binary executables.

A static library is bound to an executable program at compile time, the static linker takes the code and data from the library and copies them into the final executable. A dynamic library, in contrast, is not bound at compile time and can be explicitly linked and loaded inside a process at runtime provided it is

present in memory when the process is executed.

There can be many advantages of using dynamic libraries compared to static libraries, the most obvious ones being having an executable smaller in size, having an easier time when upgrading/bugfixing commonly shared routines and being able to load code at runtime which makes development easier for plugin developers.[4] The majority of modern operating systems favor dynamic shared libraries over static shared ones for these reasons.

2.2 The Dynamic Loader

The dynamic loader (also known as runtime linker) is a program that is called by the operating system's kernel when it detects a dynamically linked executable being launched or some specific system calls like *dlopen(2)* being invoked. The task of the dynamic loader is to read and properly resolve all the symbols present in the libraries and program being executed. This is because dynamic libraries are usually compiled with PIC (Position Independent Code) and dynamically compiled executables often have unresolved symbol references that need to be tied together at runtime.[5]
To facilitate this process the ELF standard provides sections of data called GOT(Global Offset Table)[6] and PLT(Procedure Linkage Table)[7] which are tables of addresses and symbols to be filled at runtime by the linker when the required symbols are requested by the process (lazy binding).

To link together a dynamic executable and its runtime linker the ELF standard specifies a special segment called PT_INTERP which contains the pathname for the required interpreter on the system, this location is mostly standard and consistent on Unix-like systems.

The peculiar aspect of the ELF dynamic loader, though, is that it is itself an ELF binary, just more complex. It is not loaded in memory and it is handled differently compared to other dynamic binaries in the system: it does not specify any PT_INTERP segment, it expects the kernel to give it full control over its own memory mapping as it bootstraps itself.

3 The Virtual Square Project

Virtual Square is a set of different projects sharing the idea of exploiting virtuality by unifying concepts and creating tools for interoperability[8].

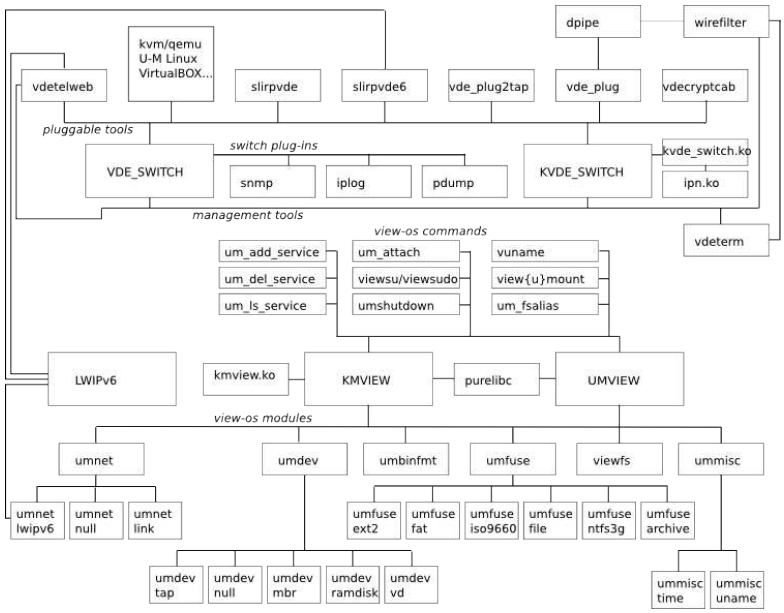

Figure 3: Tools and libraries in Virtual Square

The research field of Virtual Square involves several various aspects of virtualization providing a range of different tools for specific tasks and purposes as shown in Figure 3.

3.1 View-OS

View OS is a part of the Virtual Square project that focuses on virtual machines to give to each individual process its own view of the underlying system. View OS allows the users to change the perspective of their processes and work against the global view assumption standardized by all other operating systems[9]. Each process running inside the View OS virtual machine refers to a monitor that traps and captures all the system calls executed and, on discretion of the user, changes the context and resource requested by them. This makes it possible to have a process running in a partially virtualized environment where all accesses to a specific resource can be transparently rerouted to a different one.
Implementing virtual filesystem accesses (ViewFS) and virtual networking stacks (VDE/Umnet) is rather trivial thanks to the rest of the tools provided by the Virtual Square laboratory and users can individually manage different contexts

and sandboxed environments on a single machine without having to load an entirely new guest virtual machine.

4 Virtual Machines

Virtual machines were originally defined as an *efficient, isolated duplicate of a real machine*[10], their purpose is to provide a virtualized environment for processes and systems while keeping as much transparency as possible with the underlying host system. Virtual machines can be employed for a wide range of different tasks.
By deploying a series of virtual machines inside a single hardware machine it is possible to cut back a lot of budget and maintainability costs that usually come with the need to keep track of multiple hardware devices. All of this at the expense of some computational resources lost in the virtualization process itself.[11] There are two main categories of virtual machines: those involving full hardware/system virtualization and those concerning individual application and process virtualization. The focus of this paper, however, lies exactly in-between those two categories, through the definition of *partial virtual machines*.

4.1 System Virtualization

As technology evolves, new techniques are developed to improve the quality and stability for all types of virtual machines. The most effort is usually spent on full system virtualization, trying to cut down as much performance loss as possible while still maintaining a fully virtualized environment. The most common full virtual machine environments on Linux systems are Qemu/KVM, Virtualbox, VMWare Workstation, Xen and Usermode Linux.

4.1.1 Qemu and KVM

Qemu (short for Quick Emulator) is a free software virtual machine that performs full hardware virtualization. It can run in two different operating modes, full system emulation and user mode emulation. The former provides a full platform virtualization and emulates all the components and peripherals of a computer including the processor. The latter provides an application level virtualization which can be used to compile and execute foreign architectures on the host machine.[12] One of its great advantages is to provide a development platform and testbed for multiple architectures which would be too expensive to purchase and use as real hardware.

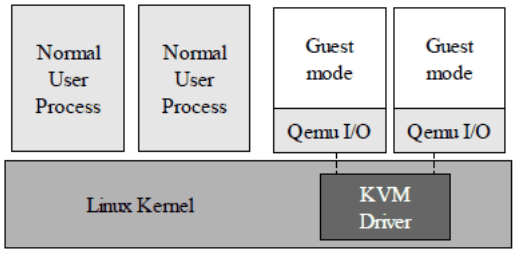

Figure 4: Qemu interfacing with kvm device.

In contrast, KVM (Kernel-based Virtual Machine) is a free software virtualization infrastructure which usually goes hand in hand with Qemu. It is not by itself a real virtual machine, it just provides a kernel-based virtualization interface which can be used by user-space processes (usually Qemu) to set up guest virtual machines, as shown in Figure 4. The advantage of using KVM virtualization instead of native Qemu, aside from the obvious performance improvements at the kernel layer, is that it simplifies the virtualization process using hardware features of the processor itself like Intel VT or AMD-V.[13][14][15]

4.1.2 Virtualbox

Virtualbox is an open source virtualization software package developed by Oracle. Like KVM it can take advantage of hardware virtualization technologies but does not require them except for some specific cases. The concept behind its virtualization design is very similar to Qemu, in some cases it also takes advantage of its recompiler for software based virtualization.[16] One of its main disadvantages, though, is that it supports only x86 (32 and 64 bit) virtualization and no other foreign architecture.

4.1.3 VMWare and VMWare Workstation

VMWare is a company that develops proprietary virtualization products, their flagship is the VMWare Workstation, a hypervisor running on x64 architectures that lets the user set up multiple guest virtual machines as full system virtualization. Very similarly to its open source competitors VMWare workstation employs several virtualization techniques at binary level, translating machine code at runtime and taking advantage of hardware virtualization features of the CPU.[17]

4.1.4 Xen and Hardware Hypervisors

Unlike the other examples, Xen should probably be listed in a category of its own. It is a free software native hypervisor, it runs on the bare metal and acts as host operating system in order to run in a more privileged CPU state. It can

host several full system virtual machines as guests called domains. The main one is labeled dom0 and it operates on a higher privilege level, it is the only domain with direct access to the underlying hardware of the machine and it is used to setup and launch all the other guest domains(domU).

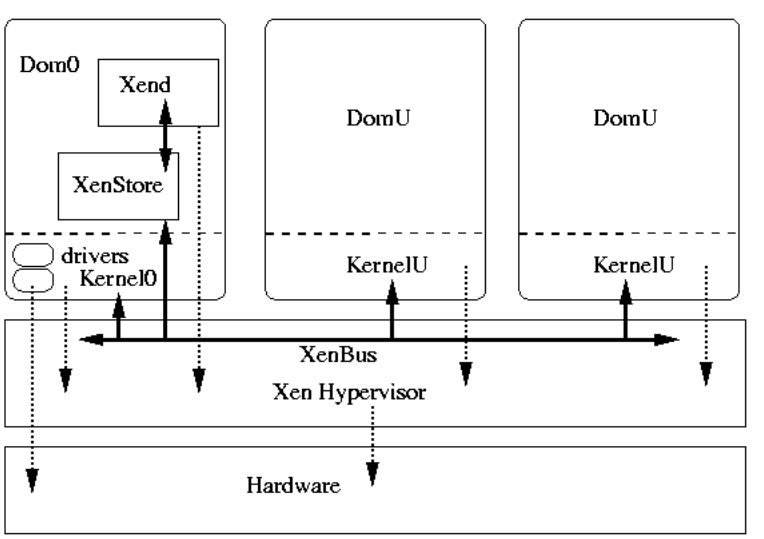

Figure 5: Xen system architecture.

What is revolutionary about Xen, which excels over other virtual machine designs, is that it runs its own operating system kernel on the bare metal to provide as little overhead as possible. It has its own scheduler and memory manager and it offloads all the hardware support and drivers to the guest OS running as dom0. It contains only the code required to detect and start secondary processes, set up interrupt routing and perform PCI bus enumeration.[18] This makes it very small with very little memory footprint as it tries to be as unintrusive as possible.

4.1.5 User Mode Linux

User mode Linux(UML) should also deserve its own category, it is a virtual Linux machine that runs on Linux itself. As its creator said *"UML is a port of Linux to Linux [...] it is a port to the software interface defined by Linux rather than the hardware interface defined by the processor"*.[19]

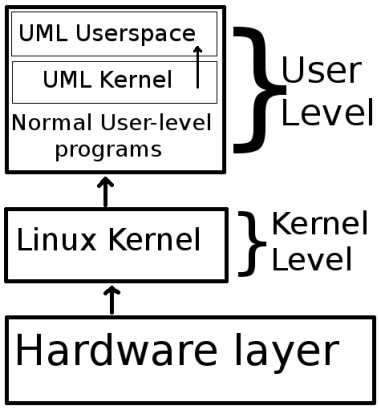

Figure 6: Structure of UML running inside Linux.

The main trait that differentiates UML to other virtualization technologies is that it is not really a virtual machine by traditional meaning, it is an actual process running a custom virtual OS as user mode, as seen in Figure 6. This makes it more independent and isolated from both the host and the underlying hardware architecture, although it makes it lack in performance.

4.2 Process Virtualization

On the other side of the coin, opposed to full system virtualization, lies process virtualization. This type of virtualization focuses on providing a contained environment for individual processes without having to create a full system under them. Examples of notable process virtual machines are the JVM (Java Virtual Machine), Microsoft's CLR (Common Language Runtime) and Dalvik. This type of virtualization usually comes with specific platforms and programming languages being compiled to a target intermediary bytecode which is then interpreted by the virtual machine.[20]

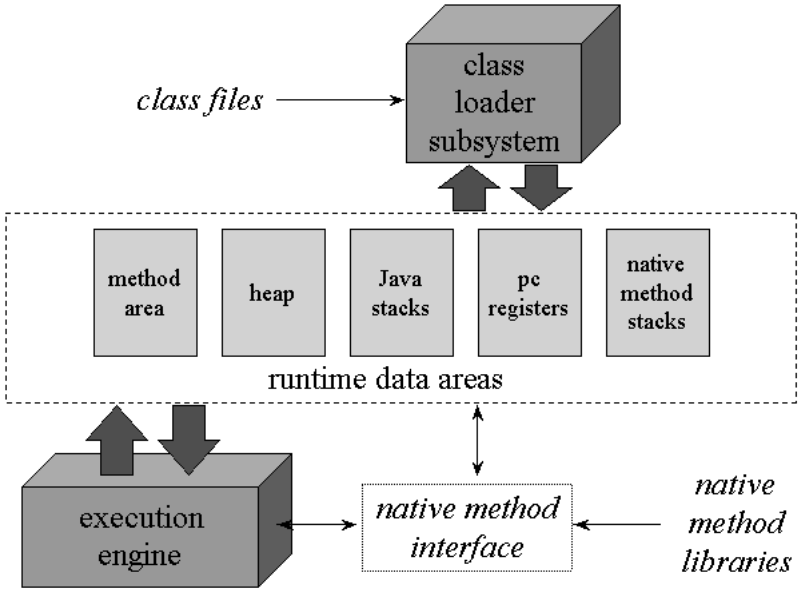

Figure 7: Architecture of the JVM

There are several advantages with this type of approach, by providing an intermediate step between compilation and actual execution it is possible to defer specific types of optimization at runtime while offloading most of the compilation process to the actual virtual machine. This type of technique is known as Just-in-time Compilation or Selective Optimization[21] and has become very popular in modern high level programming languages. Another important advantage is the portability and universality of a single application which can be executed on any host and architecture as long as an appropriate interpreter is provided.[22]

4.3 In-between: Partial Virtual Machines

In the middle, between these virtual machines, lies the idea of a partial virtual machine. Its purpose is to virtualize only portions of a specific application (or multiple applications) on demand by the user. It cannot be considered a full virtual machine because it does not apply full virtualization to a whole environment and yet it is not an application-specific virtual machine because its virtualization can be applied to generic processes and not only to a specific subset of programs (unlike the JVM, for example, which requires binaries compiled in Java bytecode). It is thanks to this type of technology that it is possible to

develop tools like View OS for the Virtual Square laboratory.

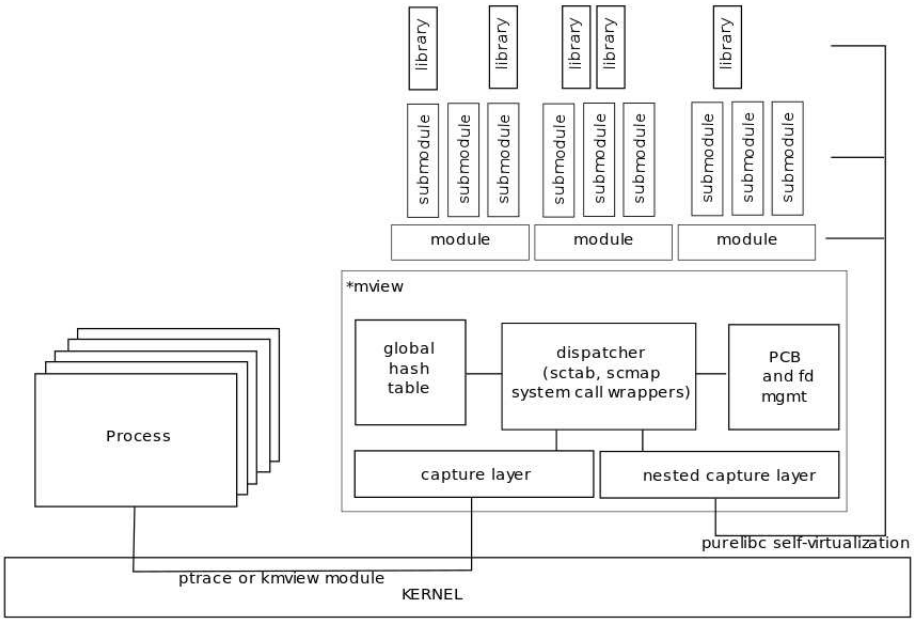

Figure 8: Virtualization in ViewOS

What a partial virtual machine needs to do, in order to be considered such, is to have a way to detect and re-route all possible system calls from a running process. System calls provide a way from user space to communicate with kernel space and request/operate system-wide resources at the control of the underlying kernel. By placing a monitor in the middle of each system call it is possible to redirect them (either some specific syscall or all of them at once) somewhere else and create a fake kernel that virtualizes the machine.
Currently there are three ways to inject this type of virtualization inside executable processes in ViewOS: Ptrace, Utrace and LD_PRELOAD[23], this paper will introduce a fourth method in the following sections: the virtual loader (or *vloader* in short).

4.3.1 Ptrace

Ptrace (which stands for process trace) is a debugging and tracing facility found on most Unix-like systems, its primary use is to trace programs and it is widely used by tools like gdb and strace. ViewOS uses this type of technology as well to listen to all the system calls and capture the ones requiring virtualization, routing them to the umview monitor. Ptrace, however, was not designed for virtualization since its original purpose was debugging, its performance is not

optimal and has some limitations and downsides.

First of all, the standard ptrace interface is only able to transfer just one word of memory to the kernel for each syscall trapped which in turn forces many context switches between user and kernel if more memory is required. This causes a lot of performance loss since context switches are very slow and taxing operations. Furthermore it is not possible to nest ptrace calls, this makes it impossible to use programs and tools that require the ptrace interface to work (like gdb and strace) inside the virtual machine itself.
Another big problem of ptrace, again performance related, is that it is not possible to avoid a context switch between kernel and user space when the traced process invokes a system call. This gives a lot of performance overhead, especially on those syscalls that do not require any kernel interaction because they should just be rerouted to the virtualization monitor instead.

4.3.2 Utrace and Kmview

Utrace is the natural evolution of ptrace on the kernel side. It is a tracing utility for kernel modules developed by Roland McGrath. It serves as a general purpose substitute of ptrace at kernel level for developers to implement in their own modules. Kmview takes advantage of that and enables ViewOS to apply the required virtualization at kernel-level thanks to its kernel module.

Performance-wise, compared to ptrace, utrace is an obvious winner. The speed up received by running the virtualization monitor at kernel level is similar to kvm compared to normal qemu, it is simply faster. There is no forced context switching on system calls and there are no memory limitations when communicating with the monitor.

However, there are two big problems to utrace: first of all, it is not officially recognized by the Linux Kernel and requires custom patches and custom builds, this makes it harder to distribute and run for common users and less likely to be implemented on most Linux distributions. As a consequence, the second issue, its development has mostly been dropped and the maintainers have all moved on to other projects. Every time a new kernel is released the utrace patches have to be applied and with no developers willing to work on it, it simply means that the utrace/kmview approach is not the optimal solution.
Another problem that utrace does not solve, compared to ptrace, is that it does not allow for properly nested virtual machines.

4.3.3 LD_PRELOAD and Purelibc

Purelibc is a C library developed at the Virtual Square laboratory to provide "pure" access to the system call interfacing functions of the C language. It runs on eglibc and glibc but tries to abstract the actual features of the language from their implementation, keeping them "pure". Through the LD_PRELOAD fea-

ture it is possible to preload the purelibc functions and interpose them between the process and the monitor. This way it is possible to capture the syscalls and either let them through the kernel or reroute them to ViewOS.

With this approach it is possible to bypass the forced context switch for virtualized system calls and provide a performance improvement to the virtualization process. Although it does not perform as well as kmview it is easier to implement because it does not require any additional modification to the kernel and is widely available to all Linux users.

At the same time, though, it does have its own problems. It does not behave properly on some specific syscalls like *dlopen(2)* and *fopen(2)* and is overall inconsistent. Some specific setuid programs cannot be injected with LD_PRELOAD because of security reasons and it is still required to fall back to the ptrace approach for most cases.

5 Virtualization on ELF Binaries

There are different ways to possibly overcome some of the the partial virtualization limitations outlined in the previous section and develop a new way to capture system calls. The most straightforward approach would be to analyze and modify the binary format used to interface with the kernel. When a system call is executed, a very specific instruction is passed to the processor and is then trapped by the kernel. Originally, on the old i386 architecture, the int 0x80 interrupt vector was used to transfer control to the kernel.[24] This would have been the perfect point to attack on programs in order to capture every possible system call being executed.

5.1 ASM Injection

This approach requires proper analysis of the program to be able to provide modifications before it gets executed. Eventually, all instances of int 0x80 in the compiled executable must be replaced with trampolines pointing to a virtual system call dispatcher embedded into the target itself.

Implementing this solution manually is trivial, using Unix provided tools like objdump and hexedit it is possible to create copies of system binaries and save them as their own independent executable. Doing the same automatically through a script or a program written in C is a bit more troublesome: detecting all proper int 0x80 instructions is not immediate and requires more advanced analytical techniques. Sometimes it is possible to encounter a byte sequence that translates to an int 0x80 while not being actually so, it could be a payload of a function, for example, some variable in the code or even just a misaligned memory address in the binary. A big limitation to this is also imposed by the system itself and its security features that prohibit the user from patching and excuting binaries at runtime with arbitrarily injected code. Re-creating a single binary in a specially allocated slot of memory and then running it (through a jmp instruction for example) is far from simple and often denied by several kernel hardening features on specific systems.

Regardless of the difficulties outlined above, in reality this strategy would still be irrelevant on most modern systems for the following reasons:

Permissions
: Due to the impossibility of running patched binaries from memory at runtime, it is necessary to save the patched program someplace else on the filesystem. A good location would be /tmp for example (ignoring some systems which deny executing applications from /tmp, easy to circumvent). As a consequence to this, however, all permissions and setuid flags from the original binary are lost and most operations will most likely fail to execute properly.

Sysenter/Sysexit
: Due to the processor's architecture, starting from the Pentium II, x86 platforms have been very slow at handling int 0x80 requests, hence Intel provided new instructions called sysenter and sysexit for faster context switching between user and kernel mode. Since the 2.5 kernel, Linux systems have implemented support for these instructions pushing forward this new interface. Through a complex and ingenious trick the Linux kernel now provides an abstraction layer called linux-vdso.so (previously linux-gate.so) under the shape of a virtual dynamic library. With this library all system calls are routed through this gateway and then processed by the kernel without relying on interrupts[25], effectively rendering manual interrupt patching impossible to achieve.

Dynamic Binaries
: Except for some peculiar systems and specific corner cases, nowadays all Linux systems make heavy use of shared libraries and dynamic binaries when compiling software. This means that one would have to recursively analyze, re-write and re-assemble most libraries and that implies a magnitude of complexity too bigger to handle. Especially when considering advanced dynamic binding techniques like *dlopen(2)* which allow a process to add new libraries at runtime.

5.2 Libc Manipulation

The next logical step is to look for a commonly shared interface between all processes that wraps around every system call at a higher level of the call chain. This abstraction layer is the actual libc library: the implementation of the standard C functions provided by the system which also include every syscall available to the kernel.

By monitoring the libc it is possible to capture every single system call at a native library level regardlessly of the kernel's implementation or architecture. This idea resembles the LD_PRELOAD hack used by ViewOS and in reality the base concept is not far from that. As outlined in the previous section, lowering the abstraction level to machine code is not feasible and raising it higher than the libc lets through too much unmonitored data. Directly in the middle between these two levels lies a program whose task is to assemble code from multiple libraries into a single process: the dynamic loader.

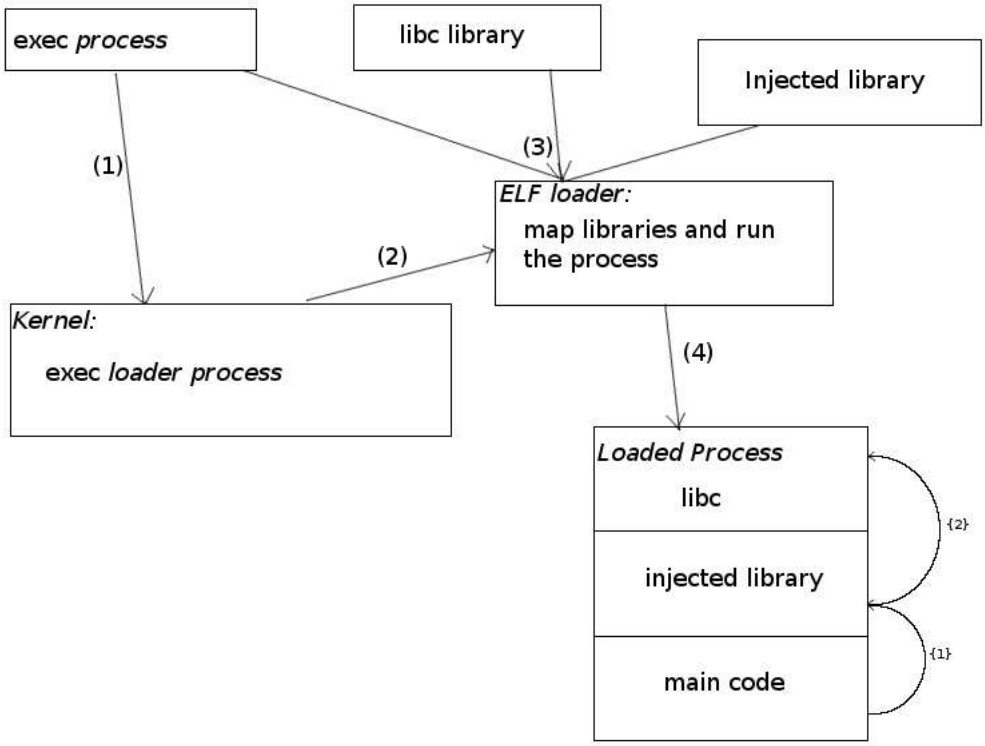

Figure 9: Injection example with a dynamic loader.

Through specific techniques and hacks used to capture the loading process it is possible to let an attacker successfully interpose his own library implementation in the middle between the target application and the native libc, as shown in Figure 9. Doing so efficiently re-routes all the system calls to custom functions without the added overhead that comes with ptrace or the inconveniences of an utrace-patched kernel.

6 The Vloader Project

The purpose of the virtual loader project is to provide a simple and easy interface to partial virtualization through modifications of the Linux ELF dynamic loader in a similar fashion to the one explained in the previous section.

There are several problems to take in consideration when working on ELF binaries and especially on the dynamic loading process. Although the ELF standard is unified and universally accepted by every Unix-like system, there are still plenty of differences between architectures and implementations. The various processor architectures provide multiple ways to achieve memory relocation and dynamic loading and a loader has to keep that in mind when operating on each ELF segment to map in memory. The SPARC architecture, for example, is much less developer-friendly compared to x86 architectures when relocating addresses[26] and a lot of corner cases and specific hacks come into play to achieve a unified algorithm on all platforms. With the advent of multithreaded and multicore technologies, ELF binaries had to be adapted to support proper threading and data storage through TLS (thread-local storage) capabilities[27] which have not been fully supported on all systems (especially older ones) yet. This, in some ways, breaks compatibility with some specific binaries and libraries compiled for TLS on non-TLS operating systems and in turn makes some loaders effectively incompatible on unsupported platforms.

6.1 The Technology

The task of a dynamic loader is a very complex one, it has to allocate memory addresses and map the executable to be run. It has to resolve all various relocation issues, look for all the dynamic libraries in the specified paths, fix memory alignment problems and even bootstrap itself. Most of the time such procedure is ran recursively into each library, keeping track of all visited symbols, their version number, their precedence and whether or not they are compatible with the format requested by their dependencies.

The most important feature of a dynamic loader is the necessity to be hooked into the running process in order to be able to resolve lazy symbol bindings and relocations at runtime. Whenever a process calls for an unresolved symbol, the loader provides this hooking mechanism by pointing all the PLT entries on the ELF binary to the same GOT address, the first, which is an entry point for a shared lookup function.

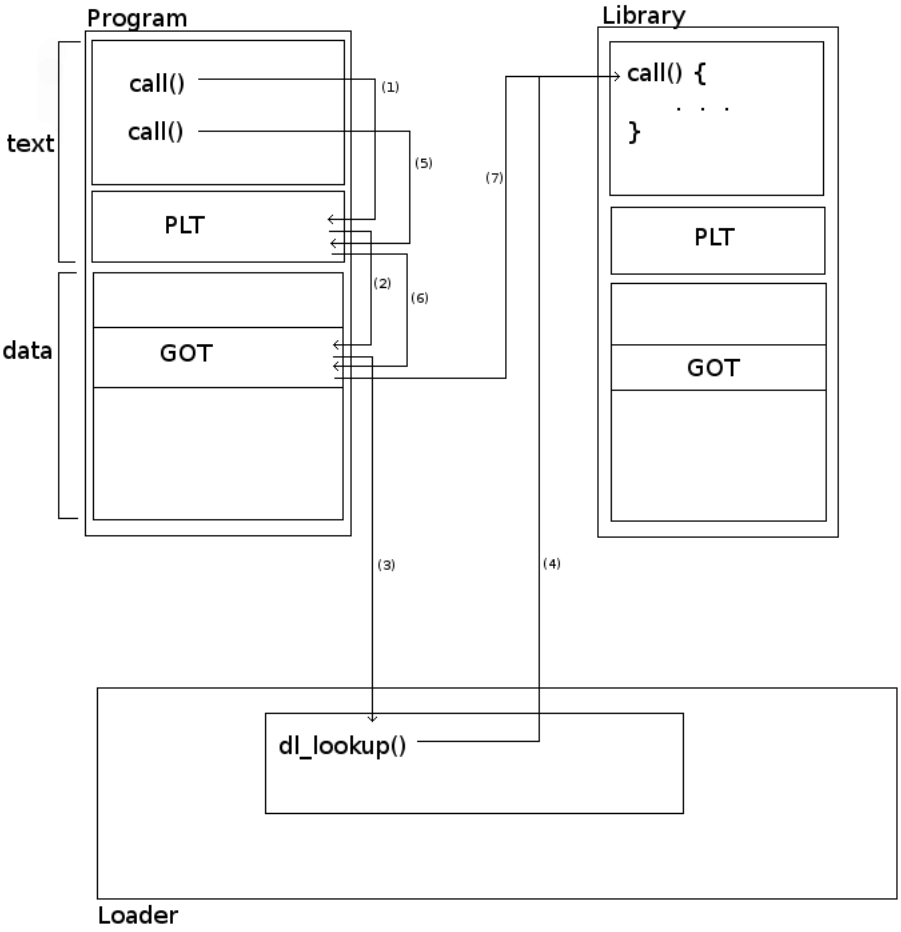

Figure 10: Resolving lazy binding with PLT and GOT lookup.

This function compares the requested symbol with a list of available hashes and, if the correct library was previously loaded, the selected PLT entry is modified to point to the proper GOT address.[28] This makes it possible for subsequient calls to the same function to directly jump to the resolved address without passing through the loader, causing only a slight overhead on the first call but not on the ones following, as shown in Figure 10. This is the method exploited by the vloader project to capture every single libc call before it reaches the kernel.

For all of the reasons outlined above and the huge scale of the project, the current version of the vloader is implemented on top of the existing glibc dy-

namic loader (specifically, the eglibc loader, version 2.13). The main purpose is to provide a working proof of concept to be improved and eventually specialized in the future.

The first step taken to develop the vloader was to find the appropriate lookup function in the glibc code. Adding a few lines of code to said function allows the loader to call specialized functions from an external library (called *libviewload.so*) and overrule some requested system calls. This lookup function is the *_dl_lookup_symbol_x()* found in the *elf/dl-lookup.c* source file.

```
lookup_t
internal_function
_dl_lookup_symbol_x (const char *undef_name,
                struct link_map *undef_map,
                const ElfW(Sym) **ref,
                struct r_scope_elem *symbol_scope[],
                const struct r_found_version *version,
                int type_class, int flags,
                struct link_map *skip_map)
{

  ...

  bool modified;
  undef_name = obtain_virt_name(undef_name, &modified);

  ...

}

static const char*
obtain_virt_name(const char *real, bool *mod)
{
        *mod = false;
        virt_library *list = virt_list;
        while(list != NULL){
                if(strcmp(real, list->check) == 0){
                        *mod = true;
                        break;
                }
                list = list->next;
        }
        if(*mod)
                return list->virt_addr;
        return real;
```

}

Libviewload.so is a library that can be provided ad-hoc by the user and has to be paired with a configuration file called vloader.conf. Through the usage of this external configuration file, parsed by the vloader, it is possible to add and remove function symbols to be virtualized without recompiling the whole project. If the vloader executable cannot find the libviewload.so library in its library paths it will print an error and virtualization will fail to start.

The following syntax is used to tell the vloader to map a certain function on top of another one. Through a double indirection and a dummy function it is then possible to call the originally redirected function, providing full transparency to the target process.

```
real_function -> virtual_function
dummy_function -> real_function
```

This snippet of code, in the configuration file, will transform all calls to *real_function()* into *virtual_function()* and all *dummy_function()* into *real_function()*. Note that the number and type of parameters are not specified. Implementing the correct signature among different functions is a task of the developer and not of the loader itself. With an incorrect number and/or type of parameters the behavior is undefined.

With this virtualization mechanism one could implement the Virtual Square purelibc library on top of the libviewload.so library and then drop the ViewOS dependencies from ptrace, or similar hooking techniques, in favor of this more streamlined and simplified approach.

6.2 Examples

Providing standard system call tracing with vloader is simple. Following are two examples of vloader.conf and libviewload.so files to generate a tracing application with a behavior similar to strace or even manipulate the results of libc system calls to create different outcomes in standard applications.

For instance, through a bash script, it is possible to read the system's configuration and header files to generate a list of all the accepted system calls. It is then possible to parse such results and generate the following setup (the actual files are much too bigger to include in this paper, these are small excerpts taken from the originals):

vloader.conf

```
. . .
#Redirect for read
read -> virt_read
real_read -> read
```

```
#Redirect for open
open -> virt_open
real_open -> open
. . .
```

viewload.c

```
. . .
//Dummy function for read
int real_read(void* params)
{
        return -1;
}

int virt_read(void *params)
{
        fputs("read was called\n",stderr);
        return real_read(params);
}
int real_open(void* params)
{
        return -1;
}

int virt_open(void *params)
{
        fputs("open was called\n",stderr);
        return real_open(params);
}
. . .
```

This code can be compiled with the following command and the result has to be placed in the appropriate LD_LIBRARY_PATH:

```
gcc -o libviewload.so -Wl,-soname,viewload \
-fPIC -shared viewload.c
```

Vloader paired with this type of library is now able to trace most system calls on the host system.

```
Linux $ ./vloader /bin/ls
getrlimit was called
uname was called
statfs64 was called
statfs64 was called
ioctl was called
lib  src  vloader  vloader.conf
```

```
exit was called
Linux $ /bin/more ../hello
Hello world!
Linux $ ./vloader /bin/more ../hello
access was called
access was called
access was called
signal was called
signal was called
signal was called
signal was called
signal was called
fcntl was called
Hello world!
exit was called
```

Another interesting setup for the vloader is to actually modify the behavior of some individual system calls to simulate partial virtualization. The following configuration makes vloader change every instance of *open("/etc/passwd")* into *open("/etc/hosts")*, in a naive attempt to prevent a user from reading the passwd file:

vloader.conf

```
open -> virt_open
real_open -> open
stat -> virt_stat
real_stat -> stat
```

viewload.c

```
int virt_open(const char *pathname, int flags)
{
        if(strcmp(pathname,"/etc/passwd") == 0)
                return real_open("/etc/hosts",flags);
        else
                return real_open(pathname,flags);
}

int real_open(const char *pathname, int flags)
{
        return -1;
}

int virt_stat(const char *path, struct stat *buf)
{
        if(strcmp(path,"/etc/passwd") == 0)
                return real_stat("/etc/hosts",buf);
```

```
        else
                return real_stat(path,buf);
}

int real_stat(const char *path, struct stat *buf)
{
        return -1;
}
```

readpasswd.c

This is the example program used to read the actual passwd contents.

```
int main(void)
{
        int fd = open("/etc/passwd", O_RDONLY);
        char buf[4096];
        ssize_t sz;
        while((sz = read(fd, buf, 4096)) != 0){
                printf("%s",buf);
        }

        close(fd);
        return 0;
}
```

If the readpasswd program is run normally, without vloader, it just opens the contents of */etc/passwd* and prints them on screen. However, since the *open(2)* and *stat(2)* system calls are being virtualized inside vloader, this is the result of readpasswd run inside the virtual environment:

```
Linux $ ./vloader ./readpasswd
127.0.0.1       localhost
127.0.1.1       linux

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
Linux $
```

These are the contents of the /etc/hosts file, not /etc/passwd. As expected, the virtualization mechanism is working properly.

There are several other ways to play around with this technology, injecting new features and experimenting with different system calls is very straightforward

and simple, there is no additional compilation time required. A user just has to modify the vloader.conf file and write his own version of the libviewload.so library.

6.3 Advantages

There are several advantages with this method compared to all the other currently used techniques in ViewOS:

Speed
: Injecting libraries through a native loader is fast. It is much faster than forcing a context switch with ptrace and there is virtually no overhead. Every injected function behaves exactly as if it existed in the original code, no slowdowns except for the symbol lookup. By disabling lazy binding (through the LD_LAZY environment variable) the loader will be slower to initialize but the lookup overhead will disappear completely as well. The injection process is not affected by it.

Architecture independence
: Using an already existing codebase (the glibc) gives the advantage of platform independence. Ptrace depends heavily on the specific architecture, different registers and cpu models behave differently and hooking system calls is harder to achieve through a universal algorithm. The loading and linking code in the glibc is already ported on all CPU architectures, relieving the burden of such task from the ViewOS developers.

No kernel modules required
: Contrary to the utrace approach, all Linux kernels support dynamic loading of binaries and the vloader project does not require specific kernel features that would need to be compiled by the end user.

Nested virtual machines
: The dynamic load hook process is completely transparent in regards of the running applications. Unlike ptrace and kmview it allows for a simple nesting mechanism to run virtualization inside an already existing instance of ViewOS. It just has to be handled by the partial virtual machine monitor, independently of the syscall trapping procedure.

Ptrace inside ViewOS
: It is theoretically possible to run a ptrace process inside the virtualized environment. This gives the advantage of some tools like strace and gdb inside the virtual machine which are not currently supported by the ViewOS project.

6.4 Inconveniences

Although the advantages may be tempting, vloader still has its own fair share of inconveniences too:

Glibc size

The glibc (and its eglibc variant) is massive. It has a very big amount of code and dependencies that are very cumbersome to work with, especially since most of its code is related to the actual C library and not to the dynamic loader itself. Although the platform independence advantages have already been explained, the disadvantages may as well outweigh them. In order to compile and run a simple instance of vloader, it is necessary to compile (with multiple passes) the whole C library and related GNU utilities. This process can take hours upon hours on weaker machines and most of its outcome is unnecessary for the purpose of the project itself. This also makes the glibc depend heavily on its version and platform: different Linux distributions support different versions and variations of the same C library and may or may not be entirely supported between each other. Custom patches have to be provided with the vloader project to run on each specific Linux system.

Security issues

Security-related problems are twofold for the vloader project. Firstly, running vloader with root privileges is not suggested and accidentally installing it on top of the original Linux loader will create a lot of security troubles. Vloader is designed to be run with individual user privileges in order not to cause any actual harm to the system.

Secondly, the security issue is rooted within the nature of virtual machines. Traditionally, one should not be able to escape outside of the virtualization environment. Due to the technology used by the vloader project, it is still possible to write native asm code and manually call the underlying kernel, escaping out of the virtualization's walls. Depending on the requirements of the end-user, a different approach (like ptrace) would be more secure, yet slower.

Statically compiled binaries

Statically compiled binaries do not require a dynamic loader to be executed. They are directly mapped in memory by the kernel and then executed without having to load external libraries or dependencies. Although more and more systems are moving to dynamic binaries for everything, there are still cases where running statically compiled executables is suggested (i.e. embedded systems and critical recoveries). In this case, the vloader program will not be able to run the virtualization environment, hence static binaries are not yet supported with this mechanism.

7 Current State of the Project

The current version of the vloader project is still in heavy development, it relies on the glibc and it is very sensitive to different versions and systems. Due to the peculiar versioning of the glibc and all its different forks and distro-specific backports, it can be very unstable on most systems. The current release can be successfully run on the Debian GNU/Linux distribution on x86, both 32 and 64 bits. It is technically possible to run the project on every glibc version, effectively targeting any Linux distribution, it is only required to select the proper libc version. The series of patches necessary to implement the vloader on top of the standard loader are not affected by the individual versioning number.

At the moment the virtualization process has come out successful with all types of function calls and system calls from the C library. The hooking mechanism is not affected by the parameters or by different symbol versions for the requested libraries, it presents no problems nor inconsistencies. It is not possible to escape the virtualized environment unless communicating with actual assembly code, as outlined in the previous section. It is also possible to load further libraries through standard means like LD_PRELOAD or simply running ptrace on top of it. The procedure is entirely transparent.

The current vloader is only presented as a working proof of concept and has not yet been implemented inside ViewOS or any other Virtual Square project.

8 Future Developments

There are still a lot of features to be implemented in the project, all the flaws and imperfections will have to be eventually fixed. The most troublesome of all is the huge net of dependencies taken from the glibc. A solution would be to effectively branch off from the original glibc code and separate the actual loader from the rest of the C library. This means modifying and removing all the makefile and scripts currently deeply rooted in the build process of the whole library. It is not an easy and simple task. A more sensible approach would be to re-write the actual loader code from scratch as a standalone project, taking architecture dependent code from the original GNU code. Separating the actual vloader from the glibc means keeping an independent codebase and not being influenced by every individual glibc versioning change which could potentially break compatibility with the various Linux distributions.

Another very important problem to tackle would be the current impossibility to virtualize statically compiled binaries. A possible theoretical solution would be to analyze the ELF binary structure of static executables and look for the GOT addresses encoded in the data. A simple re-route of such addresses to the injected library would then emulate the same behaviour presented in dynamically loaded binaries and provide an opening for the actual virtualization hook.

The matter of security is a tougher issue than the others. As explained earlier, when working with virtualization one should always be aware of what type of security and what type of performance are required. Undoubtly, ptrace is better security-wise: it is simply impossible to escape the virtual jail created by ptrace. The vloader approach, though, leaves space for asm code and old fashioned int 0x80 instructions to simply escape its virtual net. Employing extensive binary analysis techniques it could be possible to detect dangerous executables and libraries trying to escape the jail and have the vloader refuse to start them. This approach, unfortunately, is very taxing and difficult to develop successfully.

Another interesting improvement would be checking the linux-vdso.so library, the virtual library used by the kernel to provide a user space kernel mapped interface for system calls. If it were possible to hook the vloader library at that level, instead of the native libc layer, the whole mechanism would be faster, more secure and overall more independent from the Linux system and its supported system calls.

9 Conclusion

There are multiple ways to achieve proper virtualization on modern computer systems. Some of them focus on the task of "traditional" virtualization of complete system environments, like cloning a specific machine or platform to be run inside a host. Some others, instead, target single applications and provide a smaller virtual environment for such programs to live in. Both of these approaches live at the two antipodes of a fictional virtualization scale. Right in-between lies a different mechanism: the Virtual Square laboratory's idea for partial virtualization through the ViewOS project.

Partial virtualization, as employed by ViewOS and the Virtual Square project, takes care of abstracting the view each single process has of its own system, working against the global view assumption. It traps each system call, a mechanism used to communicate with the kernel, and uses a monitor to decide whether or not some specific resources should be virtualized. There are different ways to build a monitoring program to trap system calls, it is possible to hook to a process at user level via the ptrace interface, it is possible to develop specific kernel modules to avoid some context switching overhead and it is also possible to interpose custom developed libraries like the purelibc to intercept some function calls.

The vloader project is a new approach to partial virtualization, it is a working proof of concept designed to hook into the code in charge of loading processes and their related dynamic shared libraries. Modifying the loading behavior it is possible to intercept the C library interface of each individual system call, redirecting it from the application's level to either an external monitor or to the kernel itself. It is an alternative that shows no noticeable overhead but provides some possible security holes in the virtualization jail which have yet to be resolved.

The world of virtualization is still in heavy development, there are plenty of different approaches and techniques employed in the design of virtual machines. These mechanisms span from data analysis and runtime optimization to someting closer to the metal, like processor-based hardware virtualization and literal machine code translation. The vloader project itself, with its original design, is still a working experiment with various openings and challenges to be solved and implemented to allow the project itself to grow and mature enough to be eventually implemented inside the ViewOS system and the Virtual Square international laboratory.

References

[1] *System V Application Binary Interface*, 4.1 edition.

[2] *Executable and Linking Format (ELF) Specification*, 2.1 edition, May 1995.

[3] Reji Thomas and Bhasker Reddy. Dynamic Linking in Linux and Windows. `http://www.symantec.com/connect/articles/dynamic-linking-linux-and-windows-part-one`.

[4] IECC. Dynamic Linking and Loading. `http://www.iecc.com/linker/linker10.html`.

[5] Pierre-Alain Darlet. Runtime Loader-Linker Technologies. *ESC*, April 2001.

[6] Ian Wienand. Global Offset Tables. `http://bottomupcs.sourceforge.net/csbu/x3824.htm`.

[7] Ian Wienand. The Procedure Lookup Table. `http://www.iecc.com/linker/linker10.html`.

[8] Michael Goldweber Renzo Davoli. *Virtual Square: Users, Programmers & Developers Guide*. Lulu Book, 2011.

[9] Renzo Davoli Ludovico Gardenghi, Michael Goldweber. View-OS: A New Unifying Approach Against the Global View Assumption. ICCS, 2008.

[10] Robert P. Goldberg Gerald J. Popek. Formal requirements for virtualizable third generation architectures. Communications of the ACM, July 1974.

[11] Wei Huang. A Case for High Performance Computing with Virtual Machines. ICS, 2006.

[12] QEMU Internals. `http://qemu.weilnetz.de/qemu-tech.html`.

[13] Jeremy Andrews. Interview: Avi Kivity. `http://kerneltrap.org/node/8088`.

[14] Intel Virtualization Technology. `http://ark.intel.com/Products/VirtualizationTechnology`.

[15] AMD Virtualization. `http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx`.

[16] Oracle Corporation. Oracle VM Virtualbox User Manual. `https://www.virtualbox.org/manual/`, 2004-2012.

[17] Ole Agesen Keith Adams. A Comparison of Software and Hardware Techniques for x86 Virtualization. VMWare, August 2006.

[18] *Xen Users' Manual*, 3rd edition.

[19] Jeff Dike. *User Mode Linux*. Prentice Hall, 2006.

[20] Bytecode for the Dalvik VM. `http://source.android.com/tech/dalvik/dalvik-bytecode.html`, 2007.

[21] Stephen Fink Matthew Arnold. A Survey of Adaptive Optimization in Virtual Machines. Proceedings of the IEEE, February 2005.

[22] Jim Hugunin. Python and Java: The Best of Both Worlds. Corporation for National Rearch Initiatives.

[23] Michael Goldweber Renzo Davoli. View-OS: Change your View on Virtualization. `http://www.cs.unibo.it/~renzo/view-os-lk2009.pdf`, 2009.

[24] How System Calls Work on Linux/i86. `http://www.tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html`.

[25] Johan Petersson. What is linux-gate.so.1? `http://www.trilithium.com/johan/2005/08/linux-gate/`.

[26] Relocation . `http://www.iecc.com/linker/linker07.html`, June 1999.

[27] Ulrich Drepper. ELF Handling For Thread-Local Storage . `http://people.redhat.com/drepper/tls.pdf`, 2005.

[28] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, October 1999.