Master's Thesis

Detecting Attacks using Program Alternatives

Marta Tasić

06. Januar 2015

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuende Mitarbeiter:	Dr. Björn Döbel
	Dipl.-Inf. Florian Pester

Technische Universität Dresden Fakultät Informatik	Dresden, 14.07.2014

Aufgabenstellung für die Masterarbeit

Name des Studenten:	Marta Tasić
Studiengang:	Distributed Systems Engineering
Thema:	Detecting Attacks using Program Alternatives

Attackers often use buffer overflows, heap overflows, or return-to-libC attacks to modify the control flow for their purpose. Orchestra [1], Address-Space Layout Randomization [2] and other strategies have demonstrated that many of these attacks can be mitigated by replicating an application and randomizing the replicas' address spaces.

ELKVM is a process isolation architecture developed at the TUD OS Chair. ELKVM allows full control over an application's resources and complete interception of system calls. ELKVM already supports replicated execution for the purpose of fault tolerance.

The goal of this thesis is to demonstrate ELKVM's suitability for mitigating security vulnerabilities by combining randomization and replication. This includes the development of necessary extensions to ELKVM's replication architecture as well as the implementation of one or more prototypes and the demonstration of their suitability to protect against certain attacks.

Possible ELKVM extensions include
* Randomization of heap regions (allocated through the mmap/sbrk system calls)
* Randomization of shared library regions similar to traditional ASLR
* Combination with compiler/linker extensions that reorder stack and data layout and the location of certain code segments

These extensions are just suggestions and concrete examples will be identified during the thesis in cooperation with the advisors.

verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dr. Björn Döbel
Institut:	Systemarchitektur
Beginn:	14.07.2014
einzureichen:	22.12.2014

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 6. Januar 2015

Marta Tasić

Abstract

Buffer overflow is known to be the most common form of vulnerability in software that allows attackers to hijack a system by feeding a specially crafted input to a vulnerable application running on it. Many techniques have been developed to prevent an intrusion, but none of them provide an ultimate solution. Multi-variant execution involves running several slightly different versions of a program in parallel. Discrepancies in execution of the variants indicate an attack. I develop a multi-variant execution environment with the help of ELKVM library. I implement a multi-variant execution monitor which produces variants for a given application using custom program diversification techniques and runs them while comparing their behavior. The monitor runs as a Linux user-space application. It provides security to the application against many buffer overflow based attacks with the geometric-mean performance degradation of 18.2%, commonly affordable to security sensitive applications.

Contents

List of Figures

List of Tables

1 Introduction

“The nation’s security and economy rely on infrastructures for communication, finance, energy distribution and transportation—all increasingly dependent on networked information systems. When these networked information systems perform badly or do not work at all, they put life, liberty and property at risk” [Sch+99].

According to the CERT [CER], buffer overflows have been the most common form of security vulnerability in computer and network systems for the last few decades. Being a requirement to launch a control-hijacking attack, buffer overflow vulnerability represents one of the most serious types of security threats. In a control-hijacking attack, the attacker gains control of a host by transferring the execution flow of a vulnerable program into the code he injected or chose. The attack code runs with the privileges of the victim application and allows the attacker to bootstrap any other functionality needed to control the host.

Wilander et al. divide prevention of buffer overflow based intrusions into static intrusion prevention and run-time intrusion prevention [WK02; WK03]. Static intrusion prevention tries to prevent attacks by finding and removing the security bugs from the application’s source code or porting the application to some of the safe languages. However, removing all security bugs from a program is infeasible. Porting the software to another language potentially introduces new security bugs. Additionally, costs of redesigning or replacing widely-used systems are often prohibitive.

The run-time intrusion prevention approach includes changing the run-time environment to make vulnerable programs less vulnerable. The more secure environment makes security bugs more difficult to exploit, without removing them from the program. Depending on which part of run-time environment the security technique addresses, compiler, operating system and hardware approaches can be differentiated.

The most common compiler approaches are to perform array bounds checks on all array accesses or integrity checks on code pointers before dereferencing them [JK95; Cow+98]. Operating system intrusion prevention approaches are commonly based on controlling the address space of a vulnerable process. Operating system support can be used to randomize the absolute locations of all process’s code and data or forbid execution in certain memory regions, such as the stack [BDS03; Des]. With hardware support, a mechanism that creates a process-specific randomized instruction set can be implemented [KKP03]. The array bounds checking method completely eliminates the buffer overflow vulnerability by making overflows impossible, but imposes substantial performance costs. The other approaches either ensure protection against an individual class of control-hijacking attack or provide only probabilistic reliability.

I implement a run-time monitoring mechanism that can detect buffer overflow based exploits by running two or more program variants and comparing their behavior against each other. It relies on the assumption of Salamat et al. that program variants have identical behavior under attack-free execution, but their behavior differs under any intrusion attempt [Sal+09]. The system builds different variants of a given program at run-time. It provides support for (1) stack protection mechanism and implements custom versions of (2) address space layout randomization and (3) instruction set randomization. By combining the three security techniques, my solution is able to thwart more types of control-hijacking attack than each of the methods is, when applied independently. With only two versions of a program running in parallel, the system improves reliability of introduced security techniques.

The system for detecting buffer overflow based exploits, which I present, is a software-based solution which runs as a user-space application and does not need kernel privileges to monitor the program variants. As process isolation tool it uses the ELKVM library, developed by Pester [Pes14]. ELKVM runs program variants inside virtual machines and intercepts all generated system calls, while imposing an average run-time overhead of 2.2%.

I introduce control-hijacking attack, the ELKVM library and replication monitor in Chapter 2. In Chapters 3, 4, and 5 I explain the three security techniques, which my solution supports, and describe related design and implementation choices I made. An evaluation of my solution is shown in Chapter 6. Chapter 7 concludes my work.

2 Technical Background

In this chapter I introduce a security problem — control-hijacking attack, as a common exploitation of a buffer overflow. Additionally, I explain a classification of control-hijacking attacks. For each of the introduced classes of control-hijacking attacks, I present general exploit schemes.

One of the strategies I employ to protect against control-hijacking attacks is process replication. For this purpose, I use the already existing ELKVM library. The second part of this chapter gives an introduction to ELKVM, the process isolation architecture developed at the TUD OS Chair.

At the end of the chapter I describe the attack model for hijacking the control of a program and the protection model I employ in my work.

2.1 Control-hijacking Attacks

In a control-hijacking attack an attacker intends to take over the execution flow of a running application by manipulating its control-sensitive data structures. The application's control-sensitive data structures include all data which can affect its control flow. Examples of data structures responsible for control flow of an application are return addresses, function pointers, C++ virtual functions table pointers, and the `jumpbuf`-s [Cow+00; CS02]. In a control-hijacking attack control-sensitive data is overwritten with an address of the code injected or chosen by an attacker. Once an attacker has the control of the victim application, he gains all the privileges the victim application's effective user is entitled to.

Control-hijacking is made possible due to security vulnerabilities found in software, the most common of which is buffer overflow. Buffer overflow attacks exploit a lack of bounds checking in the C and C++ programming languages [One96]. When storing input data to a buffer array, a buffer overflow is triggered in case the data is bigger in size than the memory allocated for the buffer. By writing data past the boundaries of an allocated array, an attacker overwrites the memory adjacent to the array. This way, the attacker makes arbitrary changes to program's control-sensitive data that can be reached by the buffer overrun.

The techniques to exploit a buffer overflow vulnerability depend on the memory region where buffer resides. Stack-based buffer overflow attacks use a buffer allocated on the program's stack to carry out a malicious act. By overflowing a stack buffer, an attacker aims to corrupt the return address of an active function call or some of other local variables allocated on the stack, such as function pointers. On the other hand, heap-based overflow attacks target dynamically allocated buffers. Writing beyond the bounds of a buffer in the heap data area has the goal to alter either data and function pointers allocated in the next memory block [Con99] or adjacent dynamic memory allocation meta data used by the memory manager, such as linked list pointers [ano01; Kae01]. By corrupting management information within the heap space itself, an attacker can further manipulate arbitrary existing function pointer.

Depending on the origin of the code, where an attacker transfers the execution flow of the victim application to, two types of the control-hijacking attacks can be differentiated: (1) code injection attack and (2) code reuse attack. In the remainder of this section I describe the two introduced types of the control-hijacking attacks.

2.1.1 Code injection attack

A code injection attack consists of injecting malicious code into a running application and causing the injected code to be executed [PAM09]. In the code injection attack, an attacker first places the malicious code, which he wishes to execute, in the address space of the application. Subsequently, he uses a buffer overflow to overwrite a piece of control-sensitive data with the address of injected code. The attacker commonly injects the malicious code to the victim application by providing an input string that is actually executable, binary code native to the machine being attacked.

Engineering this type of an attack is not trivial. Primarily, an attacker needs to find an appropriate piece of memory where to place his code. Program's stack and heap are usually the locations accessible to the attacker. Often, a memory policy that disallows execution is enforced on stack region of memory. In order to execute injected code the attacker can either find a way to disable the execution protection of the region, or to place the code in an unprotected region of memory.

After successfully placing the malicious code in the address space of the program, it is necessary for the attacker to determine the exact address of his code. To soften this requirement, the attacker prepends the injected code with a sequence of NOP instructions. This way, in order to execute the injected code, the attacker needs to redirect the control flow to any location in the address range of the NOP instruction block.
The attacker particularly uses this method in heap-based code injection attacks. Locating the injected code within the heap is more difficult than doing so within the stack region, since dynamic memory allocation makes addresses of heap objects less predictable. Allocating large NOP instruction blocks in the heap area, is known as heap spraying [Tea11]. All the attacker needs to do is to transfer the control flow to one of the NOP blocks. The NOP block will "slide" the execution to the malicious code.

In addition to locating the injected code, the attacker needs to know the exact position of the control-sensitive data he wishes to modify. To increase the likelihood of successfully overwriting the chosen piece of control-sensitive data, such as return address within the stack frame or a function pointer, the attacker can place multiple copies of the desired value in the approximate region of the target location [Cow+98].

2.1.2 Code reuse attack

A code reuse attack exercises an attack method by which non-executable memory protection scheme is entirely evaded. Instead of injecting the attack code into the victim application, an attacker in the code reuse attack performs a malicious act using the program code itself and code of linked shared libraries. The code used in this type of attack is already stored in memory and marked as executable.

A type of code reuse attack which uses functions provided by shared libraries linked to the victim application for malicious purposes is known as return-into-library attack. The attacker constructs the attack by executing complete library functions one after the other. Namely, when the program returns from the current function, control flow is redirected to the entry point of another function chosen by the attacker. By chaining the function calls in this manner, attacker executes a sequence of arbitrary functions. Using the stack-based buffer overflow, the attacker injects series of stack frames to the stack, which allow him to control the function calls and their parameters. Standard library functions are the most likely candidates for assembling an attack, since they offer subroutines for performing system calls essential to the attacker [Ner01].

Since it only allows attacker to execute complete functions one after the other, return-into-library attack is generally considered limited. Tran et al. [Tra+11] demonstrate that this type of attacks are capable of arbitrary computation, contrary to believed restrictions. However, return-into-library attacks rely on complete set of library functions. A possible security measure is to remove vulnerable functions from applications, such as `system()` and `execve()` which launch another program or `mprotect()` which can disable memory execution protection. Since these functions are of particular interest to the attacker, crafting the attack becomes more difficult.

Another type of code reuse attack, which preserves its expressive power in face of restricted set of library functions, is return-oriented programming. Comparable to return-into-library, it uses existing application's code to perform a malicious act. Instead of chaining calls to the functions provided by the shared library, in return-oriented programming the attacker assembles the attack code using existing application and library instructions. By carefully choosing short instruction sequences, the attacker can perform arbitrary operations even with reduced amount of library executable code. The attack computations are constructed by linking together short code snippets. The code snippets start at an arbitrary position in the existing instruction sequences and end with a `ret` instruction. The `ret` instructions are there to chain instruction sequences together. With a help of the stack-based buffer overflow, the attacker supplies the sequence of addresses on the stack, to be used by `ret` instructions [Sha07; PR12].

However, to perform a successful code reuse attack, the attacker needs accurate locations of all instructions and functions he wants to mount in the attack chain. Therefore, code

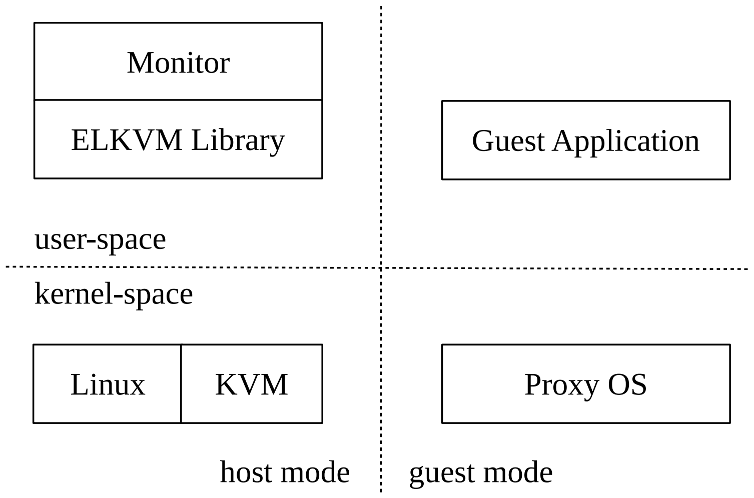


Figure 2.1: Units involved in ELKVM-based virtualization solution. Figure taken from [Pes14]

areas such as library base and main executable need to be discovered precisely. Predicting target addresses can be made additionally difficult with the help of address space layout randomization. I present address space layout randomization in Chapter 3 as technique which randomizes positions of different memory regions, including code regions, in the process's address space [Sha+04].

2.2 ELKVM & Monitor

"ELKVM is a library which allows execution of a guest application inside a virtual machine without a full-scale operating system. It uses KVM, the Linux Kernel Virtual Machine built into the Linux kernel, as a hypervisor. KVM executes virtual machines as normal Linux processes" [Pes14].

KVM exposes an API to control the virtual machines from the host's user space. KVM's API includes functions for creating a virtual machine and adding a virtual CPU to it, mapping a piece of memory to the virtual machine, manipulating the register values of the virtual CPU, and running a virtual CPU inside the virtual machine [Kiv+07].

Figure 2.1 shows the building blocks involved in ELKVM-based virtualization solution. ELKVM abstracts from the KVM interface and provides functions for virtual machine architecture initialization. The monitor is an application which uses the ELKVM library and runs in the host user-space. ELKVM sets up the address space of a guest application, which shall be executed inside a virtual machine, by loading the application's code and data in the guest user-space and an operating system kernel in the guest kernel-space. To provide the guest's operating system, ELKVM implements a minimal proxy OS. The proxy OS defines only necessary kernel structures, such as Interrupt Descriptor Table for keeping locations of interrupt handlers and Global Descriptor Table as list of memory

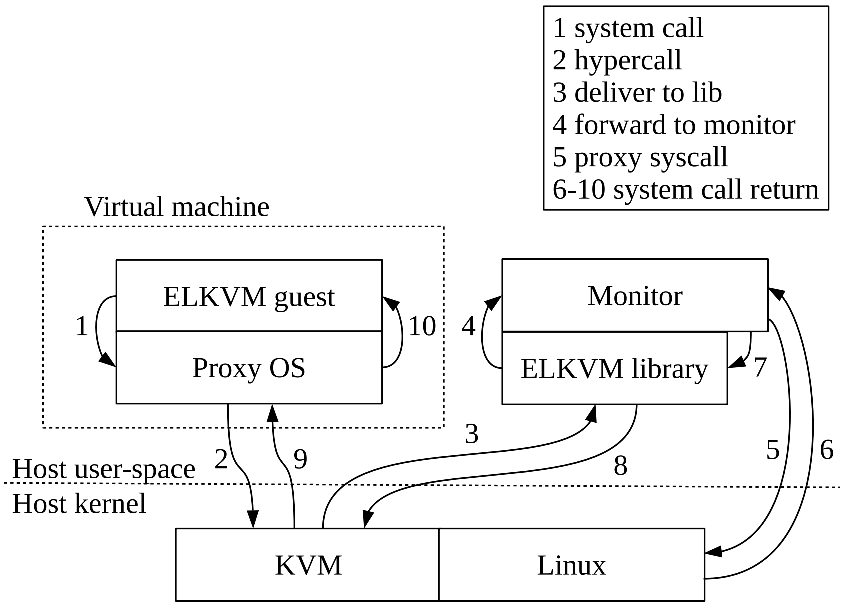


Figure 2.2: System call interception scheme. Figure taken from [Pes14]

segments' descriptors.

All implemented interrupt handlers in the proxy OS are set to forward the interrupt to the hypervisor, which in turn reports the interrupt to the ELKVM. This mechanism enables ELKVM to intercept all interrupts generated during execution of the guest application, and specify the way they shall be handled.

The way a system call propagates through the system until it is executed is depicted in Figure 2.2. When the guest binary issues the `syscall` instruction, control is transferred to the operating system inside the guest (1). The system call handler, implemented in the proxy OS, by executing the `vmcall` instruction gives control to the host hypervisor — the KVM module in the Linux kernel (2). The KVM's `handle_vmcall()` routine delivers the system call to the ELKVM library (3). ELKVM provides an API to delegate handling of intercepted system calls (4). By implementing this API, monitor takes responsibility for handling each system call itself or forwarding it to the Linux kernel (5, 6). Monitor returns the system call result to ELKVM (7). ELKVM, after writing the result to the appropriate guest's register, returns control to the KVM module (8). System call handling is completed once the KVM's `handle_vmcall()` routine and the system call routine of the proxy OS executed a return instruction (9, 10).

The Listing 2.1 presents a simple ELKVM monitor. It is an unprivileged user application that receives path to the executable to run. To implement ELKVM's API, the monitor defines a set of functions that represent system call handlers, a couple of which are shown in lines 5 - 11. Handler functions defined in this manner only forward the system calls to the C standard library linked to the monitor application. As it only acts as an intermediary for system call requests between guest application and kernel, it is called proxy.

As the first step, the proxy obtains a handle to the KVM subsystem, presented as a call to the `elkvm_init()` function in the line 21 of the listing. This handle is later used in issuing commands that control various aspects of the virtual machine. As the second step a new virtual machine is created using an `elkvm_vm_create()` call.

The `elkvm_vm_create()` function takes a number of virtual CPUs the virtual machine shall have, a structure of function pointers defining the system call handlers, and a name of a guest application that shall be executed as arguments. The function first adds the requested number of virtual CPUs to the created virtual machine. Afterwards, memory is mapped to the virtual machine. At this point, page tables necessary for the virtual memory management are initialized. When the virtual machine has acquired the memory, the virtual memory address space of the guest application can be set up. ELKVM loads the minimal proxy OS and the application's text and data sections into it, and allocates memory regions for the stack and heap. Before returning the created instance of KVM virtual machine, ELKVM sets the initial values of the virtual CPU's registers, so that the virtual CPU can be started.

In line 34 of the listing the virtual CPU is started. This is done by issuing a call to the `elkvm_vm_run()` function and passing the structure representing a KVM virtual machine to it.

```
#include <elkvm.h>
#include <kvm.h>
#include <cstdio>

long pass_read(int fd, void *buf, size_t count) {
  return read(fd, buf, count);
}

long pass_write(int fd, void *buf, size_t count) {
  return write(fd, buf, count);
}

struct elkvm_handlers syscall_handlers = {
  .read = pass_read,
  .write = pass_write,
  /* ... */
};

int main(int argc, char **argv) {

  int err = elkvm_init();
  if(err) {
    printf("ERROR initializing VM\n");
    return EXIT_FAILURE;
  }

  char *binary = argv[1];
  struct kvm_vm *vm = elkvm_vm_create(1, &syscall_handlers, binary);
  if(vm == NULL) {
    printf("ERROR creating VM");
    return EXIT_FAILURE;
  }

  err = elkvm_vm_run(vm);
  if(err) {
    printf("ERROR running VCPU\n");
    return EXIT_FAILURE;
  }
}
```

Listing 2.1: Using the ELKVM library

In my work, I am using ELKVM to build a process replication architecture. The main challenge in building such an architecture is making the replication transparent to the end user. The overall system shall have the same behavior compared to the conventional execution of the guest application on the host operating system. ELKVM-based process replication environment relies on described techniques of process isolation inside a virtual machine and system call interception.

To run a guest application in a replicated fashion using ELKVM, the monitor creates more than one virtual machines and starts the guest application inside each of them. The virtual machines run independently from each other, as replicas. To ensure deterministic behavior of the replicas, the replication monitor provides identical copies of the data to all replicas on each input operation. Additionally, the monitor makes sure that each output operation is executed only once, which hides the presence of replication. As application typically communicates to the outer world using system calls, system call interception mechanism enables monitor to control all input and output operations requested by replicas. The monitor uses system call handler functions to manage communication between replicas and the kernel. As a result, defined handler functions are more complex comparing to ones presented in Listing 2.1 and specify replica synchronization procedure.

When a replica issues a system call, the control is transferred to the system call handler function registered by the replication monitor. To synchronize the replicas, monitor arranges that each replica waits for the other replicas to arrive, until all replicas have reported the system call. When all the replicas reported the system call, the one that arrived last executes the system call on behalf of all other replicas. The replication monitor copies the result of the system call to all other replicas and allows them to continue with execution. This way, each system call is performed only once, but all replicas receive the copy of the result needed to resume the execution.

2.2.1 ELF

ELKVM supports execution of any guest application in the Executable Linking Format (ELF). ELF is a standard format for binary files used by compilers, linkers, loaders and other tools that manipulate object code on the Unix-based operating systems [Com+95]. To load the guest application into the virtual machine, ELKVM reads the segments present in an ELF file. Each such segment is a region of data inside the ELF object that is associated with a particular memory protection attribute and that should be placed at a specified virtual memory address. ELKVM uses the information regarding memory protection and virtual address when initializing page table entries associated to these memory regions. One or more ELF sections map to a segment. ELF sections keep the actual content of an ELF object. ELKVM copies the content of text and data sections into the memory allocated for the virtual machine.

Besides the executable's code and data, ELKVM acquires program's entry address from the ELF file as well as the ELF's type, necessary for the initial check of the file in hands. A mandatory header named the ELF executable header provides this information.

2.3 Attack & Protection Model

As explained in Section 2.1 a control-hijacking attack is made possible due to the buffer overflow security vulnerabilities found in software. A common attack technique is to find a buffer overflow in a program, and then exploit the buffer overflow to manipulate the program's control flow. When building the security system against the control-hijacking attacks, I suppose that the attacker knows about the existing application's vulnerabilities related to buffer overflows. As the attacker controls the program's input, he can supply an arbitrary string to the victim application, crafted to perform a malicious act.

My security system protects only the user application, as I assume it is the only vulnerable component in the software stack. The ELKVM monitor, introduced in Section 2.2, is a separate process with its own address space, isolated from the user application by the ELKVM library. Moreover, the monitor is a regular unprivileged process and if compromised, limits the attacker to user-level privileges, which are not sufficient to attack the operating system kernel.

My work does not depend on security mechanisms provided by operating systems, such as address space layout randomization or executable space protection which prevents execution in certain memory regions. These mechanisms can be either activated or deactivated.

3 Address Space Layout Randomization

Attacks which exploit memory vulnerabilities, such as buffer overflows, require an in-depth understanding of the memory layout of a victim process, including the locations of stack, heap and code. Address space layout randomization is a security technique which makes it unlikely that attackers possess such an information.

Each process runs in its own virtual address space, which is divided in two parts: kernel space and user space. Figure 3.1 illustrates the Linux process virtual address space layout on the x86 64-bit CPU. The hardware allows only the least significant 48 bits of a virtual address to be used, while requiring that bits 48 through 63 are copies of bit 47. Therefore, valid virtual addresses are from 0 to 00007FFFFFFFFFFF, and from FFFF800000000000 to FFFFFFFFFFFFFFFF, which makes 256TB of usable virtual address space in total. Operating system takes the higher-addressed part of the address space for its kernel, while the lower-addressed part is available for application code and data. Area between user space and kernel space is used to trap illegal accesses.

The memory regions of the user space are application code and data segments, heap, stack, and libraries. Standard Linux address space layout assigns predefined, constant values to starting addresses of these memory regions. This results in the same memory layout for each process.

The simplicity of address space layout of a process, where each memory region starts at the fixed address, makes the process vulnerable to buffer overflow based exploits. Consider a plain stack buffer overflow attack, shown in Figure 3.2. A program accepts data from the standard input and writes it to a buffer located on the stack. An untrusted user supplies data bigger in size than memory allocated for the buffer. The program writes outside of the buffer boundary and corrupts adjacent data on the stack: local function data, the saved frame pointer and the function return address. The attacker crafts the input so that it fills the buffer with the executable code and overwrites the function return address with a pointer to this buffer. Once the function returns, execution resumes at the location pointed by the return address, in this case, user-input filled buffer. The absolute buffer address, which is used to overwrite the return address, is equal to stack base address plus a constant offset. Discovering the buffer address is a simple task for the attacker, since the address space layout defines a constant stack address for every process.

Address space layout randomization (ASLR) is a technique used to randomly arrange user space memory regions in the process's address space. Namely, the position of each of the regions is set by adding a random offset to its predefined starting address.

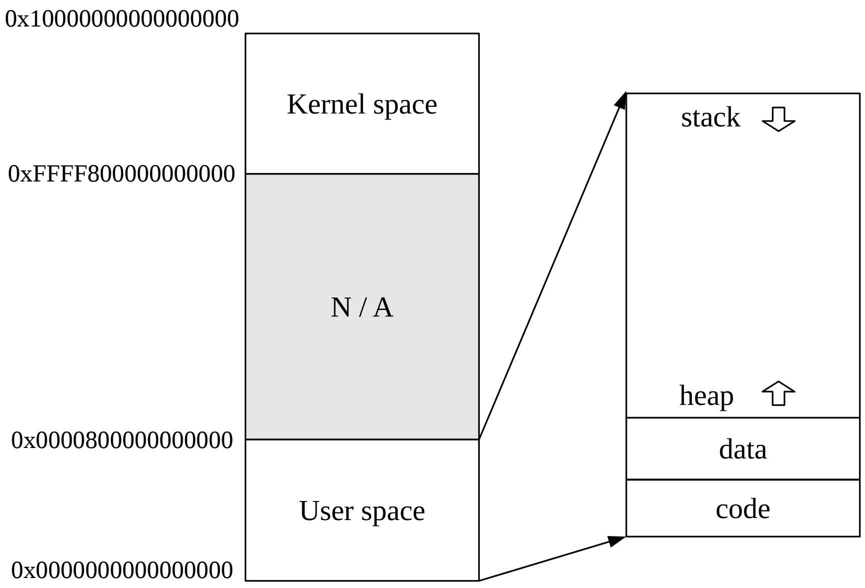
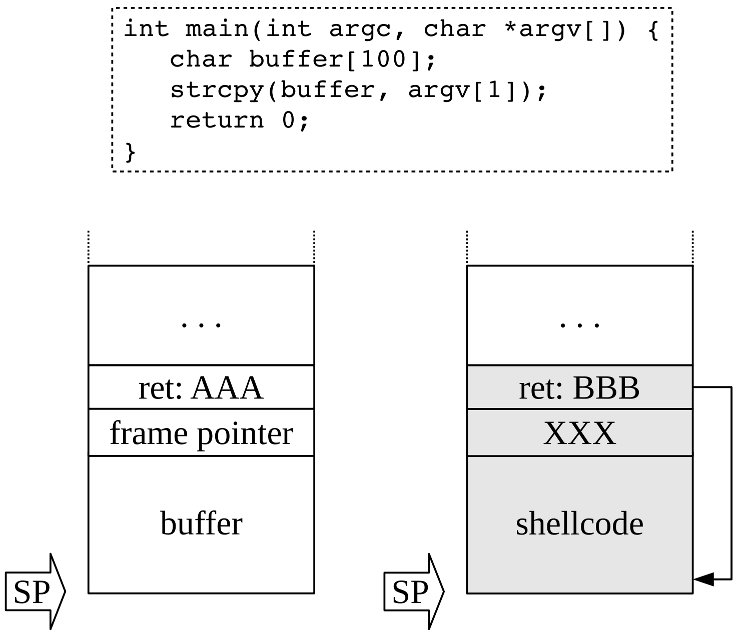


Figure 3.1: 64-bit Linux process virtual address space layout



```
int main(int argc, char *argv[]) {
   char buffer[100];
   strcpy(buffer, argv[1]);
   return 0;
}
```

Figure 3.2: Stack buffer overflow attack

3.1 Security goal and effectiveness

The goal of ASLR is to make the exploitation of buffer overflow based security vulnerabilities more difficult. As explained in Section 2.1, exploits usually include overwriting the return address in a stack frame or a function pointer. For this purpose, an attacker uses a pointer to the shell-code, previously allocated on the stack or heap, or a pointer to the library function. Additionally, within the attack code, the attacker possibly references data needed for the attack, which is living in the process's address space. With ASLR, crafting the pointers to desired memory locations becomes more difficult for the attacker.

Having randomized stack and heap positions, the absolute addresses of all variables allocated on the stack and heap change with every new run. Similarly, ASLR randomly shifts the memory mapping segment base, which will affect loading positions of all dynamic libraries and other file-backed and anonymous memory mappings created by a `mmap()` request. The precise location of a target library function, required by the attacker, will depend on the location where the appropriate library was loaded in the virtual address space.

To guess the address space layout, a brute-force attack is worth considering. The bigger the search space, the lower the probability of guessing the location of a randomly placed area. The size of the virtual address space is limiting the number of possible process's memory layouts. On a 32-bit architecture, each process runs in the virtual address space of 4 GB in size. Shacham et al. show in their work that ASLR implemented on a 32-bit architecture is ineffective and suggests using a 64-bit architecture as the most promising solution against brute force attack [Sha+04].

However, there exist methods to defeat randomization. As mentioned in Section 2.1.1, by crafting the code to inject so that it begins with a sequence of NOP instructions or by building a large "NOP slide" using heap spraying, the attacker has increased chance to successfully perform an attack. Instead of guessing the address of a single location, the attacker needs to trigger execution starting from an address in a certain address range, where NOP instruction block is allocated. Analogously, data needed for the attack can be allocated in multiple copies.

3.2 ASLR in Linux

Modern general-purpose operating systems implement ASLR. At task creation time, the randomized address space layout is established by generating random values to set each region's offset. For this purpose, PaX—Linux ASLR implementation—defines three variables: `delta_exec`, `delta_mmap` and `delta_stack` [Tea03]. The `delta_exec` variable affects base of the executable and heap region position, whereas `delta_mmap` and `delta_stack` are determining the offset of memory mapping and stack region, respectively. The example layout of the application's user space with applied ASLR is shown on Figure 3.3.

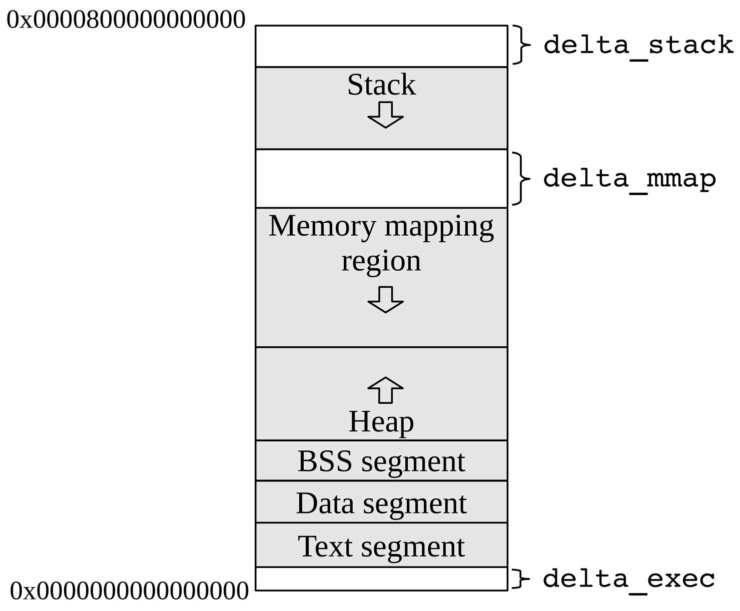


Figure 3.3: PaX applies ASLR to Linux processes

Randomization calculation leaves a portion of most significant bits as well as least significant bits of the address unchanged. This way, the ASLR implementation ensures that regions do not collide with each other and that resulting positions will be appropriately aligned.

Commonly, only position-independent executable can be loaded at the random address. Nevertheless, Linux PaX ASLR implementation is able to place non-position-independent code anywhere in the process's address space. It uses relocation information to adjust all accesses to constant addresses in the fixed position code, which causes an extra run-time overhead. However, missing relocation information can cause false alarms in which case PaX kills the process. To manage the aggressiveness of PaX, a user can activate or deactivate individual ASLR features on a system-wide or a process level, using the PaX interface.

3.3 ASLR in ELKVM

The ASLR technique prevents guessing the address space layout of a process with a quantifiable probability. Higher degree of randomness in the memory layout of the process means lower likelihood to predict an address of a certain location. In order to guarantee effectiveness of the ASLR, I implement a security mechanism which combines (1) randomization of the process's address space and (2) process replication. When more than one replicas of a process are executed in parallel, I ensure that each replica runs in differently organized address space compared to the other replicas.

I build the ASLR part of my security mechanism as an extension to ELKVM architecture. In Section 2.2 I explained the ELKVM monitor as an application that, using ELKVM, creates and runs a guest application inside a virtual machine. ELKVM has full control over virtual memory management of a virtual machine. To implement ASLR for a process running inside a virtual machine, ELKVM delegates setup of process's address space layout to the monitor.

As mentioned in the Section 2.2, ELKVM monitor is an unprivileged user-space application. This way, my security mechanism can provide ASLR to any guest application it executes, without kernel privileges. In addition, it can perform ASLR on a guest process independently of whether or not the host operating system itself supports ASLR and whether it has ASLR activated or deactivated.

The second part of my security mechanism related to process replication includes ensuring different address space layout for each replica of a process and comparing their memory accesses. To establish differently organized address spaces for different replicas, ELKVM monitor keeps one `aslr_struct` per replica. The starting positions of replica's memory regions are initialized using a newly generated `aslr_struct` assigned to the replica.

To monitor replicas' memory accesses, during their execution, I use the existing replica synchronization protocol. As I explained in Section 2.2, the replication monitor intercepts all system calls, which replicas generate, to control their input–output operations. I define memory access comparison mechanism to be executed by monitor, in addition to replica synchronization, on every received system call invocation. Namely, the mechanism ensures that each replica is operating on logically the same part of memory at a time. On every system call the monitor compares addresses of memory locations being accessed on that occasion. When replicas report a system call, monitor collects system call arguments whose value is the address of a memory location. At this point, monitor ensures that a system call argument of each replica is addressing the same memory segment and the same offset from the beginning of that segment in each replica's address space. Illustration of this procedure is shown on the Figure 3.4.

To compare replicas' system call arguments, replication monitor uses ASLR information from replicas' `aslr_struct`-s to learn about positions of appropriate memory segments. If it detects any inconsistency in replicas' arguments, the monitor reports the failure and exits. Discrepancies in replicas' system call arguments are considered as intrusion attempt, on which the system reacts with aborting the further execution, instead of letting the compromised application run.

In Section 3.1 I explained the way ASLR makes attacks which include predicting absolute addresses of certain memory locations more difficult. Having ASLR independently applied on multiple instances of a process and running them in parallel as replicas thwarts any address guessing attempt. Because of different memory layouts of different replicas, any absolute addressing results in detectable discrepancies in replicas' memory accesses. Even if attacker successfully guesses the address of a target location in one of the replicas, program's execution will be aborted as other replicas report nonconfirming system call arguments.

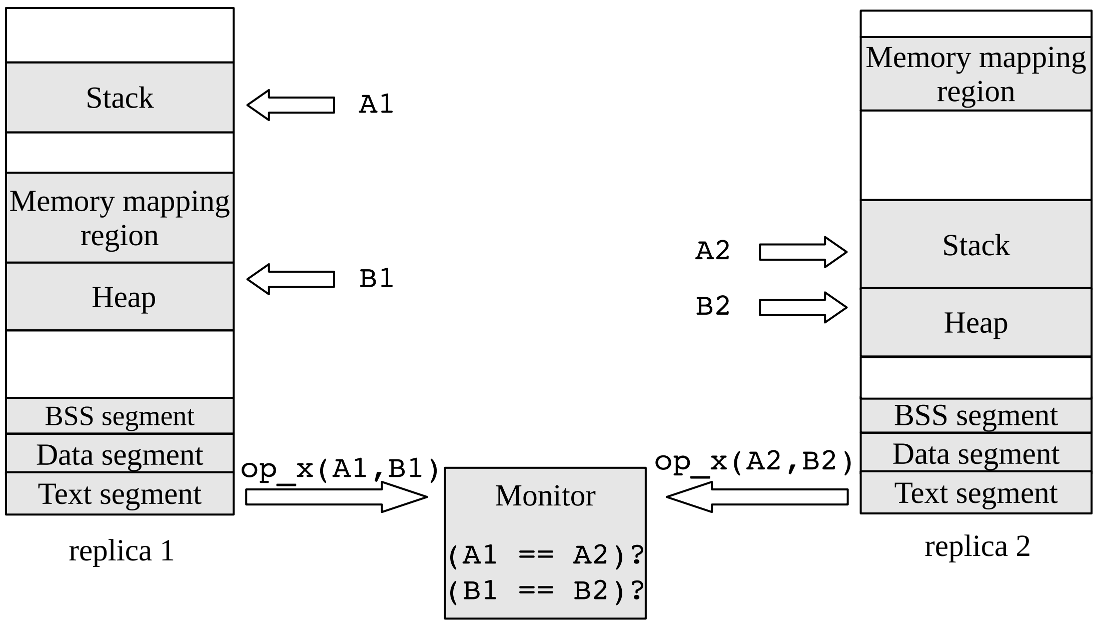


Figure 3.4: Monitor compares replicas' system call arguments

The system fails to detect the intrusion, if attacker manages to address the same location in all replicas. This can be achieved by performing a memory operation using relative addressing with fixed offsets that are valid for all replicas, such as a buffer size or current size of the stack. However, crafting an addressing scheme for every location needed in the attack in this way is much more difficult than using absolute addresses.

3.3.1 Implementation

My implementation of ASLR supports random positions of stack, heap and memory mappings in the virtual address space of the process. For guest application running in ELKVM, the monitor initializes structure `aslr_struct` responsible for randomized layout. The values of variables in the structure are determining random offsets of memory regions that support random positioning.

On startup, monitor creates one `aslr_struct` structure per ELKVM replica. The structure assigned to each replica contains three variables: `aslr_stack`, `aslr_heap` and `aslr_mmap`, initialized to random values. These values are used as offsets to randomize positions of stack, heap and memory mapping segment.

In the initialization phase, ELKVM allocates memory regions for stack and heap for each replica. To decide on the virtual addresses of these memory regions, ELKVM invokes callback functions `stack_init()` and `heap_init()` implemented by the monitor. The initialization function receives ELKVM's address proposal according to the standard memory layout of the Linux process. The function modifies the received address so as to shift the memory segment location by a certain offset. The offset to be added is a value read from `aslr_struct` assigned to each ELKVM replica. In particular, `aslr_stack` variable is deciding the position of the stack, `aslr_heap` the position of the heap.

For creating a memory mapping, ELKVM implements `elkvm_do_mmap()` function. This function is responsible for finding a memory region for the mapping and initializing it. On every memory map request, ELKVM calls a handler function registered by the monitor. The handler function receives information about the allocated mapping. The monitor edits the virtual address of the mapping by obtaining a random offset `aslr_mmap` from the current replica.

When randomizing virtual addresses in stack, heap and memory mapping regions, I choose 35 as the most significant randomized bit position. This already yields a high degree of randomness in the memory layout of a process, which I am now going to demonstrate.

`aslr_stack`, `aslr_heap` and `aslr_mmap` are 64-bit values, out of which I use 36. In case of `aslr_stack` and `aslr_heap` values I set 28 most significant as well as 4 least significant bits to zero, while randomizing the remaining bits. Using the values crafted in this way as offsets to stack and heap regions, resulting memory addresses have randomness on bit positions 4–35. This way, I place the stack and heap base somewhere in area 64 GB wide in increments of 16 bytes, which results in 2^{32} possible valid positions. By leaving 4 least significant bits of the address intact, memory region alignment is preserved.

In case of `aslr_mmap` value, in addition to 28 most significant bits, I fix 12 least significant bits to zero. The remaining bits are having the arbitrary value. By adding this value to the base of the memory mapping segment, I randomize the segment position with a period of one page in the area 64 GB wide. This yields 2^{24} possible valid positions.

3.4 Summary

My security system implements ASLR and uses ELKVM's support to run one or several instances of the guest application, isolated from each other and the kernel. The ASLR implementation protects the guest program against traditional control-hijacking attack where an attacker relies on knowing the memory layout of the program. Because of the introduced randomness, guessing the absolute addresses of memory locations in face of ASLR is much harder. When the system runs two or more instances of a guest application in parallel, it ensures their different memory layouts and compares their memory accesses on each system call. In order to spawn an undetected intrusion in such a system, the attacker is forced to employ more complex relative addressing. That is, the attacker can break the defense mechanism only if he manages to craft an attack which keeps program instances synchronized regarding memory accesses.

The security monitor provides ASLR to the guest application without relying on the ASLR facility of the host operating system. Therefore, deactivating ASLR in the host operating system does not affect monitor's functioning. Host's ASLR mechanism is there to protect monitor itself. However, the monitor application runs in its own address space, isolated from its guests, and it is difficult to compromise it by taking control of a guest program.

4 Instruction Set Randomization

The instruction set specifies the set of machine language instructions implemented by a particular processor. Each instruction has its defined operation code and possibly operand specifiers. The idea of instruction set randomization (ISR) is to change relations between instructions and operation codes. Preferably, a new random instruction set is created for each process run on the current system.

Instruction set randomization system can be developed as compiler feature or independent utility that operates on executable files before or during application loading. To correctly execute a randomized application, it should be derandomized before the CPU starts processing it. Code derandomization can be implemented as the instruction decoding process supplement, which requires modifying the conventional CPU or executing the randomized application on top of a custom CPU emulator. To make code transformation completely reversible, randomization and derandomization process are driven by a shared key.

4.1 Security goal

In the code-injection attack presented in the Section 2.1.1, the attacker is trying to redirect the control flow to the machine code he has previously allocated in some area within the address space of the victim process. Injected code needs to be compatible with the target execution environment for the attack to succeed. ISR makes attacker's code unlikely compatible, since randomization process has not been applied to it. When control flow is transferred to unrandomized code, instead of executing it, CPU will abort execution reporting illegal op-code [KKP03].

Kc et al. [KKP03] state that the effectiveness of this approach increases with the number of possible instruction sets. Increasing the size of a randomized instruction is a way to achieve better security.

4.2 ISR in ELKVM

Jiang et al. [Jia+07] argue that per-instruction derandomization would cause significant overhead and that key instruction to perform malicious action is a system call instruction. The proposed solution is to implement ISR only for system calls, named system service interface randomization.

In Chapter 3 I introduced ASLR, the implemented defense mechanism against control-hijacking attacks. I also stated its limitations regarding more complex intrusion attacks. With these limitations in mind and more effort, an attacker can still successfully launch a control-hijacking attack. System service interface randomization complements ASLR when faced more sophisticated code injection attacks.

The idea of system service interface randomization is to prevent injected code to execute a system call correctly, even though the code injection succeeds. The attack code usually directly interacts with the kernel by specifying the system call number and arguments. System service interface randomization is based on randomizing the mapping between system call numbers and their functionalities.

In my implementation, I follow the idea of system service interface randomization and combine it with process replication. I ensure that a new random system service interface is created for each instance of a process.

The C standard library provides the system service interface and implements associate functions which are making system calls on user's behalf. These functions invoke a system call using the `syscall` assembly language instruction. The standard library defines a set of system call numbers according to the operating system. Each system call's associate function sets the system call number using predefined constants. Along with the system call number, it is also responsible for storing the system call arguments in the appropriate registers according to the application binary interface.

I built system call number randomization mechanism into C standard library. The randomization mechanism maintains a table of random system call numbers per process. When a guest application is run in replicated fashion, every replica of a process will use its own distinct set of random system call numbers.

ELKVM intercepts all system calls generated by guest application, as described in Section 2.2. If several replicas of the guest application are executed, ELKVM monitor is designed to synchronize the replicas. I integrate derandomization mechanism into the existing system call interception scheme. This way, a system call number is translated as each system call instruction arrives. Additionally, I extend replica synchronization protocol to compare incoming system calls, in case of replicated execution. After translating the system call number, replication monitor performs the check if all replicas are issuing the same system call at a time.

The resulting system defines a custom set of numbers to be used when requesting a kernel service and reports illegal usage of the kernel interface in any other case. Initializing a set of accepted system call numbers using random values, makes it difficult for the attacker to perform desired operations from the code he injects. Any unsuccessful attempt to guess the valid system call number will be detected by the system call number translation mechanism. On the other hand, process replication prevents a potential successful attacker's guess. Since each replica of a process has an independent set of valid system call numbers, attacker's guess will be caught by either number translation mechanism in other replicas or by replica synchronization and comparing protocol.

4.3 Implementing the System Service Interface Randomization

```
 1 pid_t getpid(void)
 2 {
 3   /* ... */
 4   __asm__ __volatile__ ( "syscall"
 5         : "=a"(ret)
 6         : "a"(SYS_getpid)
 7         : "rcx", "r11"
 8         );
 9   return ret;
10 }
```

Listing 4.1: Original `getpid()` function implementation

```
 1 pid_t getpid(void)
 2 {
 3   /* ... */
 4   __asm__ __volatile__ ( "syscall"
 5         : "=a"(ret)
 6         : "a"(get_syscall_num(SYS_getpid))
 7         : "rcx", "r11"
 8         );
 9   return ret;
10 }
```

Listing 4.2: `getpid()` function implementation after modification

I modified the musl C standard library to perform system call number randomization mechanism. To enforce usage of the modified standard library, I recompile the user application and link it against the new standard library. The modification of the musl library includes defining a table of random system call numbers and a table entry getter function `get_syscall_num(i)`, which for a given `i`, returns the content of the `i`-th entry in the table. Every system call associate function, implemented in the standard library, is modified to obtain a system call number by invoking the getter function, instead of using common system call numbers. I show the standard library modification on the example of `getpid()` function. Listing 4.1 contains original implementation of a function, whereas Listing 4.2 shows the applied modification in line 6.

The described scheme does not cover all the requirements for implementing system call randomization technique. It leaves the following problems unsolved:

1. Building the replica-specific system call number randomization scheme.
2. Protecting part of memory containing table of randomized system call numbers.
3. Handling direct system call invocations in the guest application.

I discuss the possible solutions as well as my approaches to address these requirements, until the end of this chapter.

4.3.1 System call numbers randomization & derandomization

In Section 2.2 I explained the way system call propagates from guest application, over ELKVM to the monitor's system call handler. When ELKVM intercepts a randomized system call, it receives a random value representing a certain system call number. It is a requirement to authorize ELKVM to perform system call number derandomization. In order to handle the system call properly, ELKVM needs to translate this value into one of the common system call numbers.

The second requirement is to implement system call numbers randomization on a replica level. By doing so, each replica of a process uses a set of random system call numbers that is different comparing to sets of other replicas. This way, different versions of the same program are achieved, when it comes to system call numbers randomization.

To fulfill the two requirements, I implement both system call number randomization and derandomization in ELKVM. A virtual machine object, represented by the `VMInternals` class is assigned a table of system call numbers and responsible for its initialization. When a virtual machine object is created, it constructs a consistent and reversible mapping between system call numbers and random values and fills the table. `VMInternals` class implements functions `get_rand_scnum()` and `get_derand_scnum()` for converting from original system call number to randomized value and the other way around, respectively.

To initialize the table of random system call numbers in the process address space, ELKVM first needs to find table's location in the memory. The ELF format provides information about all generated symbols and their virtual addresses. I introduced the ELF format in Section 2.2.1. To make the search more efficient, I chose to place the table in a separate ELF section. For this purpose, I define an additional section `scnums_section` when declaring a table of random system call numbers in the standard library. By reading the appropriate section header, ELKVM gets the information about where the table of random system call numbers lives in memory after the program has been loaded. On ELF loading, ELKVM allocates memory regions for all loadable segments and copies their content from ELF file into the address space of each replica. When it comes to the `scnums_section` where the table of randomized system call numbers is, instead of copying the content from the binary, ELKVM uses data provided by virtual machine object to initialize this memory region. The in-memory table of system call numbers is put together by calling `get_rand_scnum()` function for each table entry. This way, the table will be initialized differently for different replicas.

When a process issues a system call, ELKVM relies on respective virtual machine object to translate the system call number. The `get_derand_scnum()` function performs number derandomization for ELKVM.

4.3.2 Protecting the table of system call numbers

Theoretically, an attacker can learn about system call number mappings by examining the binary file for the address of the mapping table or by guessing the mapping table's location. Once the attacker knows how to access the table of randomized system call numbers, he is able to modify it to fit his needs or simply use it when performing an attack. To prevent unauthorized access, only the standard library should have read access to the memory region containing the table of random system call numbers.

The first requirement for manipulating access permissions to the in-memory table of system call numbers, is to have it in a separate memory page. A way to achieve that is to use a compiler directive to specify an *align* attribute. By specifying a minimum alignment of a 4096 bytes, when declaring a table of system call numbers, causes the compiler to allocate the table on a page boundary. However, I chose to write a custom linker script, since it gives me the full control of the the memory layout of the output file. Once the table is in a separate memory page, the desired access permissions can be set by adjusting the protection bits of the page in the page table. ELKVM has a full control of page tables of all replicas it runs as guests.

However, memory page access permissions are enforced on the complete process. Setting certain access rights to only part of the process — that is, to the standard library code — is not supported by memory management mechanism. Therefore, it is required to implement a special access control mechanism for this memory page and its authorized user — the standard library. A possible approach is to build the access control mechanism into the C standard library. Initially, access permission to the memory page that contains system call numbers is set to "no access allowed". To make reading this memory page possible for the standard library instructions, each access to the table is surrounded with special access controlling functions. The first of the function pair is called to allow reading access and the other to set protection flags of the memory page in question back to "no access allowed". This however, with every access to the system call numbers table, leaves a certain time period during which the page is accessible to the complete process. Having the right timing and with the help of multithreading, the attacker is able to read the page content without causing access violation.

Another approach is to implement the access control mechanism in the page fault handler. Namely, instead of having its protection bits repeatedly adjusted, a page table entry about the page that keeps system call numbers is removed from the page table. Every access to this page would result in a page fault. I implemented this approach because it ensures that every access to the table of system call numbers first passes the access control check, which is not the case with the first approach.

When ELKVM's page fault handler is called, the cause of the page fault is examined. If the page fault address belongs to the memory page that contains the table of system call numbers, ELKVM calls the `scnums_pf_handler()` function that handles the interrupt. `scnums_pf_handler()` function first checks if it is the trusted instruction that tries to read this memory page. To decide if the instruction is permitted to access the table of system call numbers, ELKVM uses the instruction origin. Only if the instruction belongs to the `get_syscall_num()` function, which is a part of system call randomization mechanism explained in the Section 4.2, ELKVM assumes it is a trusted instruction and then emulates this memory access operation. Otherwise, `scnums_pf_handler()` function reports the unauthorized memory access.

To determine the origin of the instruction, ELKVM uses its address. When interrupt happens, processor places the return address on the stack, before it calls interrupt handler. That is the address of instruction which caused the page fault. The instruction is assumed to belong to the `get_syscall_num()` function, if its address is in the address range of the function. By reading the ELF symbol table, ELKVM learns about the address range of the `get_syscall_num()` function. To handle the instruction which access the table of system call numbers without mapping the memory page which contains it, ELKVM needs to emulate the behavior of this memory access operation. This includes reading the table entry addressed in the instruction and storing the result in the destination register. For this purpose, ELKVM uses the ELKVM VCPU interface for accessing the appropriate registers and ELKVM virtual machine interface for getting the system call number mapping. As the last step, which makes sure the execution proceeds from the next instruction, the return address from the interrupt handler is adjusted.

4.3.3 Direct system call invocations

The system call randomization technique built into the standard library relies on the assumption that all system calls the guest application generates are made using standard library API. System call invocations that do not go through the standard library API, but instead request a system service directly using the `syscall` assembly instruction, would circumvent the randomization procedure. These system call invocations would cause reporting an attack during attack-free execution and therefore, have to be managed.

The way to apply the randomization procedure on every assembly language system call invocation, found in guest code, is to transform it to a call to a standard library function. The `syscall()` function, implemented by the standard library, fits this purpose and retains the program semantics. It takes the system call number and arguments and makes a system call. Being one of the system call wrapper library functions, this function is modified to perform system call number randomization. To transform an in-line assembly system call invocation to in-line assembly function invocation different parameter passing conventions have to be handled. The task of the transformation, which needs to be applied to the guest application, is to replace the `syscall` instruction with a `call` instruction and convert from the system call convention to the function calling convention. Listing 4.3 shows the example of direct system call invocation. The desired result of the transformation is in the Listing 4.4. As a function call and a system call expect their arguments in different set of registers, the system call arguments are reallocated as shown in lines 3 and 4 of Listing 4.4.

```
__asm__ __volatile__ (  "movq $0x0, %rdi;"
        "movq $0xe7, %rax;"
        "syscall;"
            );
```

Listing 4.3: Direct system call invocation example

```
__asm__ __volatile__ (  "movq $0x0, %rdi;"
        "movq $0xe7, %rax;"
        "movq %rdi, %rsi;"
        "movq %rax, %rdi;"
        "call syscall;"
            );
```

Listing 4.4: System call transformed to a function call

Transformation of direct system call invocations can be performed on already compiled application or on the application's source code. Building a binary-to-binary compiler, is the approach to perform described transformation on the compiled application. Designing a tool which substitutes one instruction with another one, equal in length, is an easy task. However, nonsymmetric substitutions affect the overall code size and, consequently corrupt instructions which use relative as well as absolute addressing. As transformation of the direct system call invocation is not a trivial instruction exchange, it requires usage of a tool that dynamically or statically instruments the program to modify its execution while maintaining its correctness. Valgrind offers a framework for building such a tool [NS07].

Implementing additional compiler feature is a way to perform the source code transformation of direct system call invocations. All the operations that make up a Clang compiler are performed within LLVM passes. LLVM Pass Framework is a set of tools which provide optimizations, program transformations and static analyses over intermediate representation [LA04]. I choose to add a custom LLVM pass, because LLVM offers a powerful set of functions for manipulating low-level code representation. Figure 4.1 shows LLVM compiler framework architecture containing the custom pass for direct system call transformation.

The task of a module pass, I am implementing, is to examine all user code in-line assembly instructions for system call invocation. Before replacing the `syscall` instruction with a `call` instruction, the pass inserts assembly language instructions which reorder system call arguments according to function argument passing convention. Additionally, registers that are required to retain their values across the call have to be saved and later restored. Therefore, around the instruction block in question, the pass adds register saving–restoring instructions.

4.4 Summary

In this chapter I introduced a custom ISR built into my security system. It serves as a defense against code injection attacks which are able to overcome ASLR.

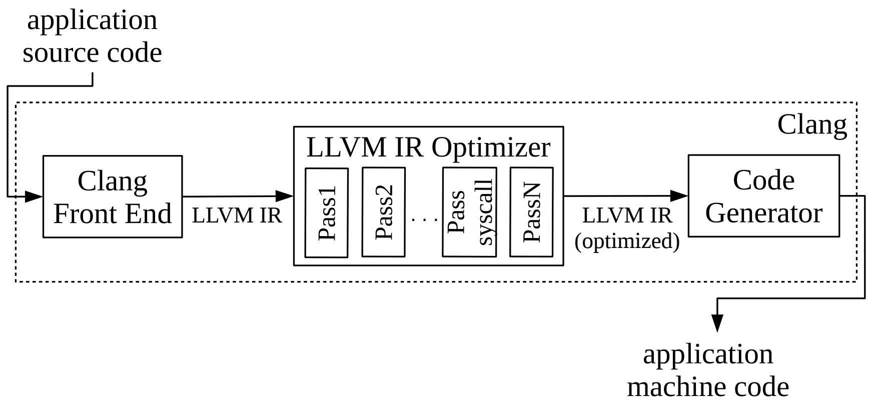


Figure 4.1: Custom pass added to the LLVM System architecture

The implemented system service interface randomization creates new mappings between system call numbers and their functionalities for every process. By randomizing the kernel interface and therefore, making it unknown to the attacker, it prevents the attacker to invoke a system call correctly. My security system implements the process specific system service interface and provides support for the parallel execution of two or more instances of a program.

The mechanism is effective against code injection attacks in which an attacker invokes the system service directly, using the kernel interface. If the attacker, in order to perform the malicious act, uses the standard library functions in the code he injected, the intrusion will not be detected. The standard library implements system service interface randomization, as its functions can be invoked by the application during the attack-free execution.

Similarly, this security mechanism does not provide protection against any type of code reuse attacks. The code reuse attacks are constructed using existing functions or code snippets, which by rule, implement system call randomization.

5 Stack Protection

As presented in Section 2.1.2, to craft a return-into-library as well as a return-oriented attack the attacker places a series of malicious stack frames or return addresses on the program's stack. When the program returns from the current function, each of the attacker's frames divert control flow into the another function chosen by the attacker or into the middle of existing instruction. Stack protection is a mechanism that shall detect any manipulation of function's activation frame on the stack, and consequently hinder return-into-library and return-oriented attacks.

Storage needed for a function's execution is organized in an activation frame (record). The activation frame keeps function's local variables, arguments passed to the function, function result and information necessary for context restoring on function return: nonvolatile registers, pointer to caller's activation frame and program counter. Before execution of a function begins, an activation frame is created and context data is saved. On function exit, state of the caller, frame pointer and program counter are restored using saved data. Restoring the program counter returns the control to the caller.

Activation frames in most of the general-purpose programming languages are allocated on the stack. As explained in Section 2.1, stack-based buffer overflow attacks can gain control of execution by overwriting the control information saved on the stack. Specifically, an attacker manipulates the saved program counter, which is part of the data used for restoring a function caller's context. To overwrite the control data saved on the stack, an attacker writes to a function's local buffer an amount of data bigger than the buffer's size. When function returns, program counter is restored using value set by the attacker, and thus execution continues starting from location of the attacker's choice.

Stack protection is a security scheme aiming to detect that a stack buffer overflow has occurred. When corruption of the saved control information inside an activation frame is identified, stack protection prevents redirection of program's control flow on function return.

5.1 Protection scheme

Stack protection is a compiler-based defense mechanism, described by Cowan et al. [Cow+98]. When stack protection is activated, the compiler allocates an additional slot called canary to the stack activation frame. On function entry, along with writing control information to the stack frame, the canary slot is initialized. For this purpose, a special guard value is used. With a help of the canary, the code generated by the

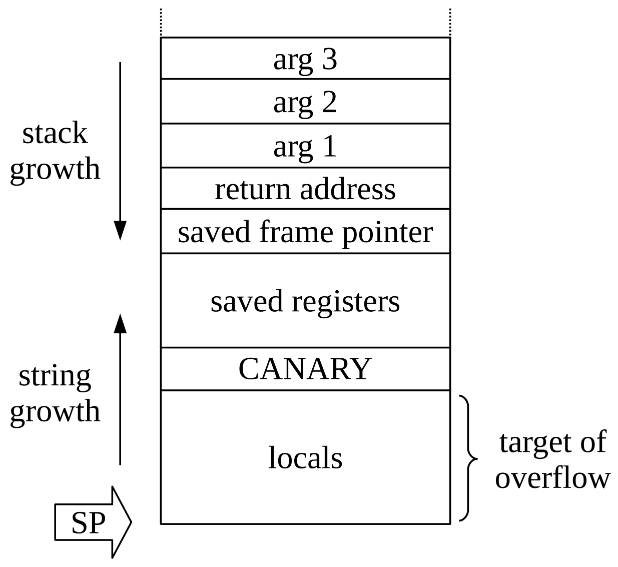


Figure 5.1: Stack layout with the canary slot in place

compiler performs integrity checking on an activation frame to detect buffer overflow.

The canary slot is placed between control information and function's local variables within the function's activation frame, as shown in Figure 5.1. Consequently, any out-of-bounds sequential write in the area of function's local variables, with the goal to overwrite function's control information, is forced to overwrite the canary. On function exit, before restoring the context of the caller using information saved in the stack frame, the integrity of the value in the canary slot is checked. If any inconsistency between expected value and the value found in the canary is detected, the program reports intrusion attempt and exits. This way, stack protection mechanism ensures that corrupted control information is never used as destination where control flow of the program will be transferred.

5.2 Stack protection in LLVM

LLVM provides mechanism to insert stack canaries during code generation. In a function's prolog, a canary value is placed on the stack. In the function's epilogue, the value stored on the stack is compared with the canary value of the process. If the values do not match, the execution is halted.

LLVM uses a random number of a pointer size for all canary values within one process. The random number is retrieved from the `__stack_chk_guard` global variable, defined in the C standard library. Since the variable is initialized at run-time, the attacker cannot learn the canary value by inspecting the executable file. Additionally, this yields a new random value each time the program is executed.

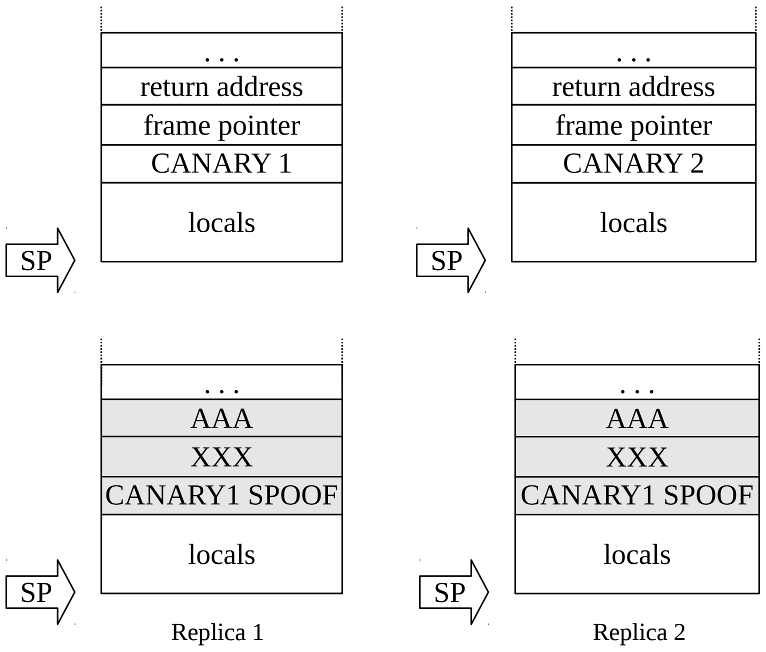


Figure 5.2: Canary guessing is successful for at most one replica

Random canaries provide protection against buffer overflow attacks which do not know its secret value. By simulating the canary word, an attacker can overcome the stack protection. That is, the attacker can embed the precise canary value at the canary slot place in the attack string [BK00]. Overwriting the canary slot with the correct value bypasses the protection. The main weakness of this canary type is that the exploit can obtain its value by either guessing or locating and reading the `__stack_chk_guard` variable.

5.3 Stack protection in ELKVM

I built a security extension to support stack protection mechanism. In particular, my solution addresses canary spoofing as the main weakness of the stack protection scheme implemented in the Clang compiler, explained in the Section 5.2.

A way to achieve random canary spoofing is to successfully guess or learn its value. Using multi-variant execution, I try to prevent any lucky canary guessing attempt. In Section 2.2 I explained ELKVM as a process replication architecture. In addition to replicated execution, provided by ELKVM, my system implements independent stack protection mechanism on the replica level. That is, when a program is run in replicated fashion, each replica uses its own random canary, different to the canary values of the other replicas. As depicted in Figure 5.2, the canary guessing attempt can be successful for at most one replica at a time, but results in detectable corruption of control information in the other replicas.

Another way to undetectably perform buffer overflow, in the presence of stack protection, is to gain control over the `__stack_chk_guard` variable. The canary value of a process is set using this variable, defined in the standard library. If an attacker is able to access the `__stack_chk_guard` variable, he can learn about, or even tune, the canary value of the process. My solution implements a memory protection technique which resists any unauthorized reading or writing access to the `__stack_chk_guard` variable.

The existing stack protection mechanism, implemented as a Clang compiler feature, guarantees a different canary value for every new program execution. To ensure a different canary value for each replica of a process, when the program is run in replicated fashion, canary initialization procedure has to be managed.

As explained in the Section 5.2, the canary value of a process is set using the `__stack_chk_guard` variable, defined in the standard library. The musl standard library, which I use for my work, initializes this variable in the `__init_ssp()`, one of the startup routines, which is executed before application's main function is invoked. The routine sets the `__stack_chk_guard` initial value by reading the auxiliary vector entry of type `AT_RANDOM`. The auxiliary vector is a mechanism to communicate information from the kernel to user space in the UNIX systems. The entry of type `AT_RANDOM` keeps a pointer to sixteen random bytes provided by the kernel. Each replica of a process, in the setup phase of its address space, receives an identical copy of the kernel space and the auxiliary vector. As a result, all replicas will be assigned the same canary value, contrary to the desired property.

5.3.1 Randomizing the canary

A special handling of the data used in canary initialization is one way to achieve distinct canary values for process's replicas. Namely, the memory location pointed by auxiliary vector entry of type `AT_RANDOM` needs to be initialized differently for all replicas. A possible solution is to have ELKVM generate a new random value to set the content of this memory area, on replica's address space initialization, instead of simply copying the data from one replica to the other.

Another way to ensure distinct canary values for process's replicas is to modify the way the canary is initialized. This approach includes locating the `__stack_chk_guard` variable in the address space of each replica and manipulating its content, before execution starts. Additionally, preventing the re-initialization of the variable in the `__init_ssp()` startup routine is needed, as it would overwrite the value set earlier. I chose to implement this approach because it can be combined with memory protection technique of the `__stack_chk_guard` variable, which will be explained later in the Section 5.3.2.

To initialize the `__stack_chk_guard` variable in the replica's address space, ELKVM first locates it in the memory by reading ELF's symbol table. On ELF loading, ELKVM sets the variable's content to the desired canary value of replica in question. ELKVM delegates canary generation to the virtual machine object. For this purpose, the `VMInternals` class, produces a random value on every instantiation, and provides an interface to acquire the generated value. This way, each replica of a process will use a random canary of different value compared to the other replicas.

I prevent the `__init_ssp()` startup routine to re-initialize the `__stack_chk_guard` variable, and thus invalidate the described randomization procedure, using memory protection mechanism described in the following section.

5.3.2 Protecting the global canary

In the Section 5.3 I motivated a memory protection technique which shields the `__-stack_chk_guard` variable from unauthorized reading or writing. I design this protection mechanism similarly to the mechanism protecting the table of system call numbers, described in the Section 4.3.2. More specifically, the memory protection scheme consists of designating one memory page for the `__stack_chk_guard` variable, and excluding this memory page mapping information from the process's page table. This enables me to perform a special access control procedure in the page fault handler, as every access to the page will trigger the page fault interrupt.

I implement access control mechanism in the `canary_pf_handler()` function. The function is invoked from ELKVM's page fault handler when the reason of page fault is the memory page which keeps the `__stack_chk_guard` variable. The `canary_pf_handler()` function first checks if it is the trusted instruction that tries to access this memory page. The `__stack_chk_guard` variable is read and written only in the `__init_ssp()` routine, and therefore, only instructions which belong to the routine are assumed to be trusted. I design the `scnums_pf_handler()` function to emulate memory accesses issued from the `__init_ssp()` routine, and report all other reading and writing attempts as unauthorized memory accesses.

The `__init_ssp()` routine first initializes the `__stack_chk_guard` variable using the random bytes in the auxiliary vector. Next, it copies the content of the variable to the place reserved for the canary in the thread local storage. Emulation of the first memory operation, which includes writing to the `__stack_chk_guard` variable, is intentionally omitted. The variable initialization is performed on ELF loading, described in the Section 5.3.1. To emulate the second memory operations of concern, which includes reading the `__stack_chk_guard` variable and storing the data in the destination register, ELKVM uses the virtual machine interface for getting the canary value and the ELKVM VCPU interface for accessing the appropriate register.

5.4 Summary

In this chapter I introduced stack protection, a compiler approach to perform return address integrity checking using stack canaries. My security system supports stack protection and, with a help of multi-variant execution and variant-specific canaries, fights against canary spoofing.

Stack protection addresses control-hijacking exploits based on corrupting a return address within a stack frame. Stack protection contributes to my security system by providing defense against code-reuse attacks, crafted in a way to resist ASLR. As described in Section 2.1.2, code-reuse attacks corrupt one or more function frames on the stack in order to link pieces of the attack together.

Stack protection does not detect corruption of a return address before it is used at the function return. Moreover, attacks which do not corrupt any active stack frame cannot be detected by this mechanism. If the attacker manages to divert the control flow of a program with the help of some other control-sensitive data and finds the way around system service interface randomization and ASLR, he might be able to break the security which my system provides.

6 Evaluation

The security system I developed combines process replication and diversification. For process replication, I use ELKVM, a general-purpose library designed to support process isolation by running a given executable inside a virtual machine. To produce different versions of a process, I implemented three security techniques, presented in Chapters 3, 4, and 5. This Chapter evaluates control-hijacking detection mechanism, built using ELKVM library, regarding implementation effort, memory and performance overhead.

6.1 Implementation Effort

The replication monitor, which is the core of my security system, uses ELKVM to run a process with replication and supports process diversification at run-time. It implements address space layout randomization, system service interface randomization and stack protector randomization. The replication monitor I developed adopts numerous concepts regarding replication from ELK Herder, another ELKVM monitor which focuses on fault tolerance [Pes14].

Component		**LOC**
Replication monitor		1175
	replication	934
	ASLR	41
	replica comparison	200

Table 6.1: Implementation effort of replication monitor

Component	**LOC**
ELKVM	214
musl	22
clang	104

Table 6.2: Extensions of existing tools

Table 6.1 presents the effort, in terms of lines of code, it took me to implement different parts of replication monitor. Table 6.2 lists all the tools I needed to extend and the number of lines of code I added to each of the tools.

The code responsible for process replication makes the largest part of the monitor. More than 900 lines of code are required for running the replicas and synchronizing them at system calls. The support for ASLR takes 41 line of code in the monitor. This code includes initializing replicas' ASLR information and implementing callback functions for random positioning of replicas' memory regions. The protocol for comparing replicas' system calls, takes 200 lines of code. The protocol defines comparison of system call numbers and system call arguments, and therefore is related to both ASLR and system call randomization techniques.

I extended the ELKVM library for 214 lines of code to support system call number randomization and stack protector randomization, out of which roughly 150 are related to system call number randomization. The ELKVM extensions include a function for generating table of system call number mappings to random values as well as a function for generating a random canary value. Furthermore, I add the code which loads this table and canary variable into the address space of a guest, and protect them against any unauthorized access. With performance optimization in mind, I also implement this memory protection scheme in proxy OS residing inside a virtual machine. Finally, as part of the system call number randomization support, ELKVM's system call interceptions mechanism is extended to perform number derandomization.

As I explained in Section 4.2, I built system call randomization mechanism into the musl standard library. To define a table of system call numbers and a getter function takes 22 lines of code.

To implement an LLVM pass, as an extension of Clang compiler, requires 104 lines of code. This pass makes sure that no legitimate system call invocation circumvents system call randomization procedure, which is explained in more detail in Section 4.3.3.

6.2 Memory Overhead

When running a process with system call number randomization, a separate memory page in its address space is designated to host a table of system call number mappings. This is a requirement for implementing the memory protection mechanism, described in Section 4.3.2. An additional memory page in the guest's address space is used for a canary variable, in case the guest application shall support stack protector randomization. Memory overhead imposed by ASLR can be neglected, as a few 64-bit variables represent all data structures necessary for ASLR implementation. In total, the memory overhead introduced by the three security techniques is 2 memory pages for each replica of a process.

Pester [Pes14] in his work measures 16 KB of run-time memory overhead of ELKVM and ELK Herder, for each replica. This is the memory used for data structures needed for process virtualization and replication and therefore, applies to my system as well. The memory ELKVM maps at start up, designated for running a guest application, is 16 MB in size per replica.

6.3 Run-time Overhead

All the measurements in these section were executed on a quad-core Intel Core i5-3550 running at 3.3 GHz. The cores share 6 MB of L3 cache, each of the cores has 256 KB of L2 cache. The system works with 4 GB of main memory and runs Ubuntu 14.04. TurboBoost of the CPU was turned off.

To evaluate my system in terms of the run-time overhead, I measure the execution times of test applications with each of implemented security techniques applied independently as well as with them all applied together. I also measure the run-time overhead caused by each of the two security levels offered by my system, which are achieved by running one version of the program or by running two different versions in parallel. For the purpose of comparison, I measure the execution times of test applications when they are run natively, without ELKVM.

For the need of attack detection, running two different versions of a program suffices. As my system does not support restoring the state of execution after it detects the attack attempt, there is no further benefit of running in parallel more than two instances of a program.

In my measurements, I also wish to show the run-time overhead introduced only by ELKVM as well as by ELKVM and replication monitor without providing any of the security features. In order to measure the overhead that comes from ELKVM, I use the proxy application presented in Section 2.2. Proxy is a simple ELKVM monitor that only intercepts and delivers the application's system calls to the kernel. The performance costs of replication and replica synchronization I measure by running the replication monitor with all security extensions disabled.

In Section 4.3.2 I explained the memory protection scheme which I use to prevent any unauthorized access to the table of system call numbers. I use the existing paging mechanism to trap every access to this memory area. That is, every access to the in-memory table of system call numbers is checked inside of the page fault handler function. This results in transferring the control to the host twice on every system call generated by the guest application: first to the page fault handler function, and next time to the appropriate system call handler function. To save one of the trips to ELKVM, I implement described memory protection scheme in the minimal proxy OS inside the guest. I expect that this approach saves some execution time for applications running with system call number randomization and possibly for applications running with stack protection. My stack protection mechanism uses comparable mechanism to protect in-memory canary variable, but the protected variable is accessed only a few times on the process startup, as explained in Section 5.3.2. The ELKVM monitor that runs with described potential performance enhancement I call Monitor++.

The test applications which I use to evaluate performance of my system are part of the SPEC CPU 2006 benchmark suite. I chose to run only benchmarks implemented in C programming language, as my system addresses the security vulnerabilities of this language. I compile all the benchmarks with the clang compiler and link against the musl standard library, even in case of native execution. This way I ensure comparable results.

Unfortunately, the 482.sphinx3 fails to run on top of the ELKVM architecture. The benchmark stops execution after reading unexpected data from the memory, which indicates a possible memory corruption bug in ELKVM. I could not make possible for this benchmark to run with ELKVM on time for this work. The 462.libquantum benchmark, when run with the ELKVM monitor, produces an incorrect result for the input specified by the benchmark suite. What this benchmark does is factorize the number given as an input parameter. I run the 462.libquantum benchmark with a modified input number, for which the benchmark gives a correct result.

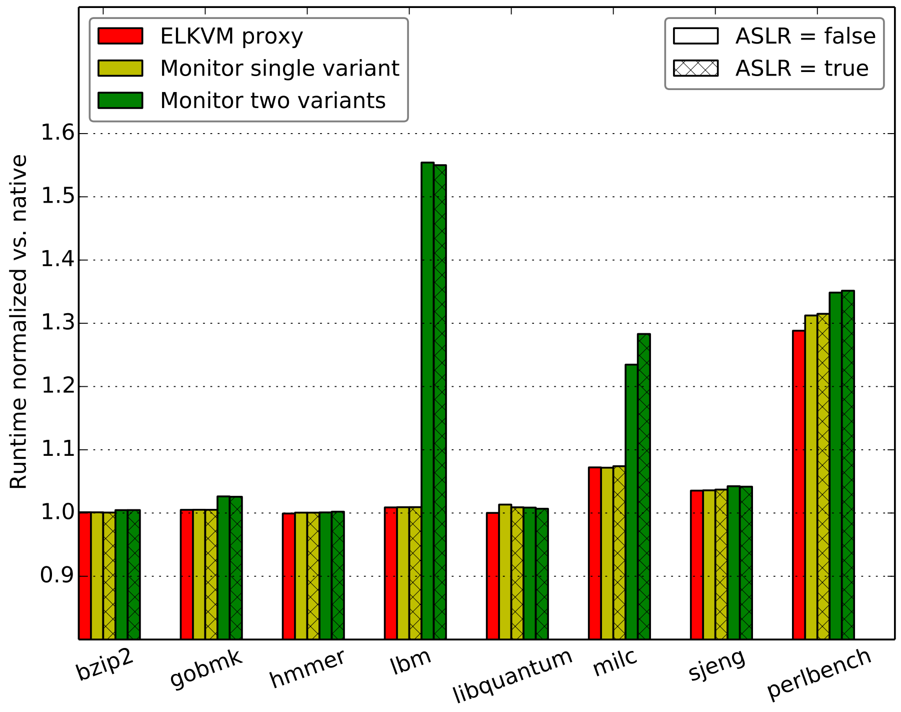
For all of the different configurations, I executed each C benchmark in three iterations and took the best execution time of the benchmark over these runs. The Figure 6.1 shows normalized execution time overheads of the benchmarks for different security techniques. When all the security techniques are activated and single version of the program is run, my system introduces 5.3% overhead in average comparing to native execution. However, the monitor, when running a single version of a program with all security extensions disabled, reports similar execution times comparing to the runs with security extensions enabled. This indicates that measured overhead comes from ELKVM and monitor itself.

In case when two versions of a program are run in parallel, my system shows more than 50% overhead for all security extensions in case of the 470.lbm benchmark and about 30% overhead for 400.perlbench and the 433.milc benchmark. The rest of the benchmarks have less then 5% overhead. By comparing, in the sense of overhead, the resulting runs to the runs of the system with two replicas without any of security techniques, I conclude that this run-time overhead is not introduced by the security extensions. However, very different run-time overheads seen in different benchmarks argue that replication is not the only reason of performance degradation either.

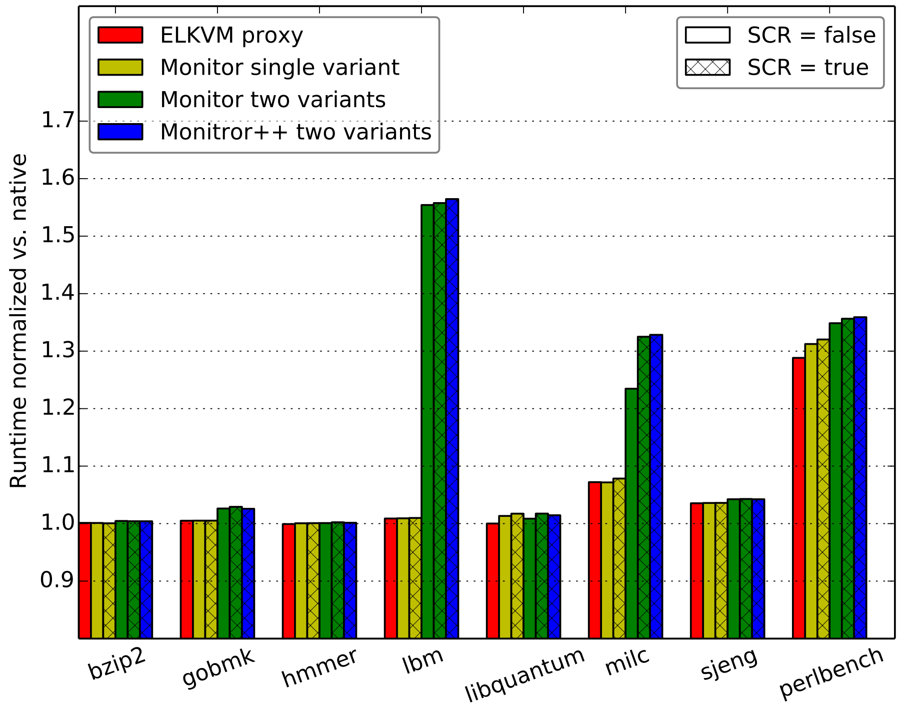
In his work, Döbel examines the encountered different replication overheads of benchmarks [Döb14]. He measures the time a benchmark spends executing user code as well as replication code. The 470.lbm and 433.milc benchmarks show increase of user execution time when run in replicated fashion. Therefore, the overall overhead of the replicated runs of the benchmarks cannot be ascribed only to the execution of replication code. The increased cache miss rates explains increased user execution time and argues that the respective benchmarks are cache-bound. Cache misses happen due to several replicas competing for the cache or due to cache flushing on every context switch to the replication monitor.

In the case of the 400.perlbench the performance degradation is already present when it is run with ELKVM proxy. With close to 1000 system calls in a second, this benchmark has the highest system call rate among all the benchmarks which I run [Döb14]. With a high system call rate the system call interception becomes more expensive. Replicating the 400.perlbench additionally decreases the performance for up to 10%. This overhead comes only from additional execution of synchronization and replication code, as user execution time does not change with replication, also measured by Döbel.

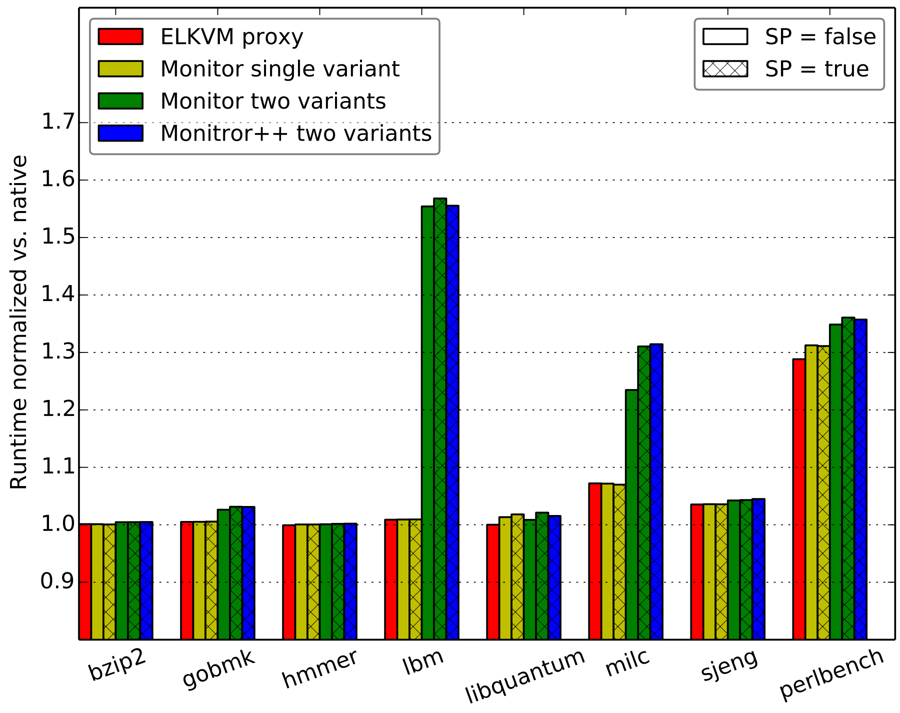
Running the system using Monitor++ does not bring any substantial enhancement regarding performance. Monitor++ addresses the performance of system call handling



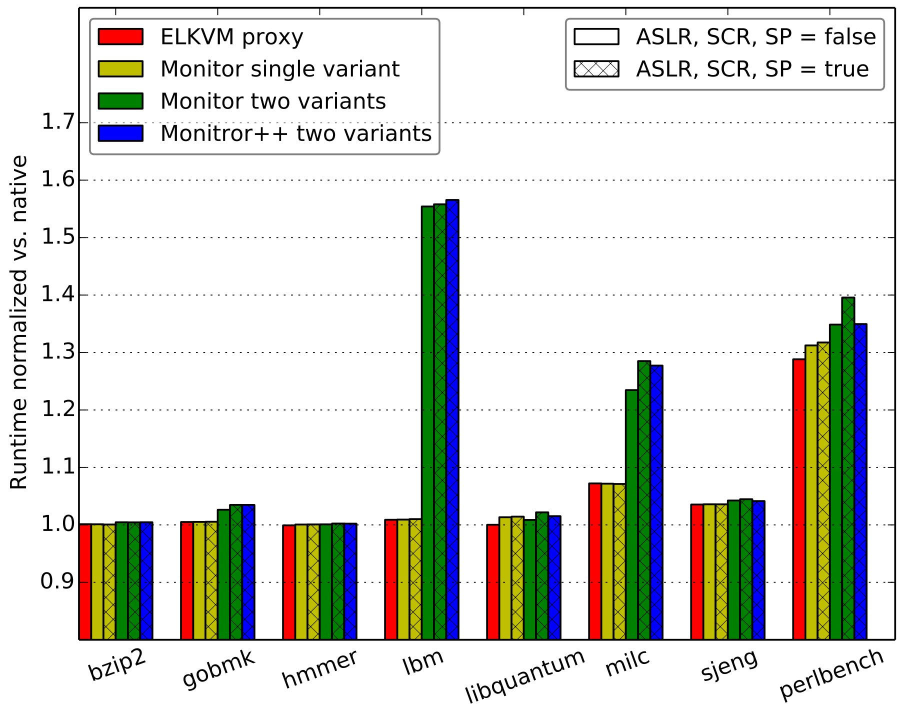
(a) address space layout randomization



(b) system service interface randomization



(c) stack protection randomization



(d) all security techniques combined

Figure 6.1: Normalized run-times of the C SPEC benchmarks

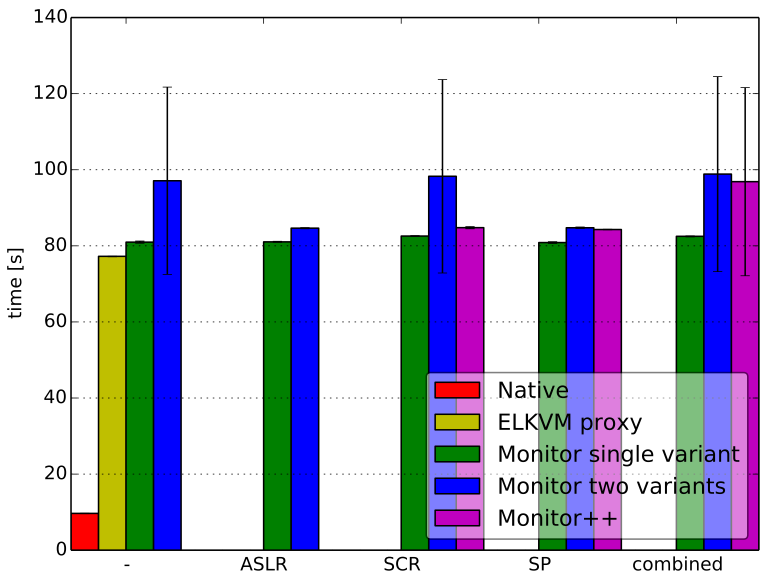


Figure 6.2: Run-time of the micro-benchmark

in the presence of system call number randomization. Performance cost that comes from the system call number randomization is hard to notice using computation-heavy benchmarks which do not intensively communicate with the operating system. To examine the potential advantage of using Monitor++ as a replication monitor, I run an additional benchmark. The micro-benchmark executes 40000 system calls per second when run natively without ELKVM. The system calls executed in the micro-benchmark do not require big replication effort as they are proxied to the standard library. Figure 6.2 shows that Monitor++ saves up to 15% run-time overhead when two variants of a program are run.

7 Conclusion And Future Work

I have presented a defense against control-hijacking attacks that combines process replication and diversification. My security solution produces different variants of a program at run-time and compares their execution while running them in parallel. As process diversification techniques, it implements address space layout randomization, system service interface randomization and stack protector randomization. My system uses the ELKVM library, a process isolation solution greatly suited for process replication.

By combining the three security techniques, my solution addresses different types of buffer overflow based control-hijacking attack. It supports stack protection to thwart well known intrusion strategy that uses function call return address corruption. Implemented system service interface randomization makes it difficult for the attacker to request any operation from the kernel, crucial to perform the malicious act. Address space layout randomization is there to hinder guessing the absolute address of a certain memory location, which is commonly involved in the control-hijacking attacks. These security techniques, when combined with multi-variant execution, become more reliable. The full benefit of the multi-variant execution is achieved with running only two variants of a guest application in parallel.

My security solution runs on Linux as unprivileged user-space processes. It can run on all hardware architectures to which ELKVM can be ported. The SPEC CPU 2006 benchmarks written in C showed that the system imposes 18.2% of overhead in average when providing the highest level of security. Usually, for security sensitive applications, the performance is not the main concern.

In order to perform transformations described in Section 4.3.3, my system requires application's source code. There, I already discuss alternative approach to perform the necessary transformations using a custom binary-to-binary compiler, which is left for the future.

To provide support for implemented security techniques I extend the C standard library. At the moment I only provide a patch for the musl standard library. Implementing the extensions for other, more broadly used C libraries, such as the GNU C library, may benefit to the overall applicability of my system.

For applications which are required to have very high uptime it is important to recover and continue running normally, after the intrusion attempt has been detected. This includes repairing the corrupted state of the compromised application. Process re-initialization and resuming mechanism can be considered for the future work. The implemented replication monitor already possesses control over processes it runs and insight into their state, which are requirements for such a work.

Bibliography

[ano01] anonymous. “Once upon a free()”. In: *Phrack Magazine, Vol. 11* (2001).

[BDS03] Sandeep Bhatkar, Daniel DuVarney, and Ron Sekar. “Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.” In: *USENIX Security*. Vol. 3. 2003, pp. 105–120.

[BK00] Bulba and Kil3r. “Bypassing stackguard and stackshield”. In: *Phrack Magazine, Vol. 10* (2000).

[CER] CERT. *CERT coordination center*. URL: `http://www.cert.org/`.

[Com+95] Tool Interface Standards Committee et al. *Executable and Linking Format (ELF) specification*. 1995.

[Con99] Matt Conover. *w00w00 on heap overflows*. 1999. URL: `http://www.cgsecurity.org/exploit/heaptut.txt`.

[Cow+00] Crispin Cowan et al. “Buffer overflows: Attacks and defenses for the vulnerability of the decade”. In: *DARPA Information Survivability Conference and Exposition, 2000. DISCEX’00. Proceedings*. Vol. 2. IEEE. 2000, pp. 119–129.

[Cow+98] Crispan Cowan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. In: *USENIX Security*. Vol. 98. 1998, pp. 63–78.

[CS02] Eric Chien and Péter Ször. “Blended attacks exploits, vulnerabilities and buffer-overflow techniques in computer viruses”. In: *Virus* 1 (2002).

[Des] Solar Designer. *Linux kernel patch from the Openwall Project*. URL: `http://www.openwall.com/linux/README.shtml`.

[Döb14] Björn Döbel. “Operating System Support for Redundant Multithreading”. Dissertation. Technische Universität Dresden, Nov. 2014.

[Jia+07] Xuxian Jiang et al. “RandSys:Thwarting Code Injection Attacks with System Service Interface Randomization”. In: *SRDS*. IEEE Computer Society, 2007, pp. 209–218.

[JK95] Richard Jones and Paul Kelly. *Bounds Checking for C*. July 1995. URL: `http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html`.

[Kae01] Michel Kaempf. “Vudo malloc tricks”. In: *Phrack Magazine, Vol. 11* (2001).

[Kiv+07] Avi Kivity et al. “kvm: the Linux virtual machine monitor”. In: *Proceedings of the Linux Symposium*. Vol. 1. 2007, pp. 225–230.

[KKP03] Gaurav Kc, Angelos Keromytis, and Vassilis Prevelakis. “Countering Code-Injection Attacks With Instruction-Set Randomization”. In: *Proceedings of 10th ACM Conference on Computer and Communications Security.* Oct. 2003, pp. 272–280.

[LA04] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Code Generation and Optimization, 2004. International Symposium.* IEEE. 2004, pp. 75–86.

[Ner01] Nergal. “Advanced return-into-lib(c) exploits (PaX case study)”. In: *Phrack Magazine, Vol. 11* (2001).

[NS07] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *ACM Sigplan Notices.* Vol. 42. 6. ACM. 2007, pp. 89–100.

[One96] Aleph One. “Smashing the stack for fun and profit”. In: *Phrack Magazine, Vol. 7* (1996).

[PAM09] Michalis Polychronakis, Kostas Anagnostakis, and Evangelos Markatos. “An empirical study of real-world polymorphic code injection attacks”. In: *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET).* 2009.

[Pes14] Florian Pester. “ELK Herder: Replicating Linux Processes with Virtual Machines”. Diploma thesis. Technische Universität Dresden, Feb. 2014.

[PR12] Marco Prandini and Marco Ramilli. “Return-oriented programming”. In: *Security & Privacy, IEEE* 10.6 (2012), pp. 84–87.

[Sal+09] Babak Salamat et al. “Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space”. In: *Proceedings of the 4th ACM European conference on Computer systems.* ACM. 2009, pp. 33–46.

[Sch+99] Fred B Schneider et al. *Trust in cyberspace.* National Academies Press, 1999.

[Sha+04] Hovav Shacham et al. “On the Effectiveness of Address-Space Randomization”. In: *Proceedings of 11th ACM Conference on Computer and Communications Security.* ACM. Oct. 2004.

[Sha07] Hovav Shacham. “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)”. In: *Proceedings of the 14th ACM conference on Computer and communications security.* ACM. 2007, pp. 552–561.

[Tea03] PaX Team. *PaX address space layout randomization.* 2003. URL: `http://pax.grsecurity.net/docs/aslr.txt`.

[Tea11] Corelan Team. *Exploit writing tutorial part 11: Heap Spraying Demystified.* Dec. 2011. URL: `https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/#0x0c0c0c0c`.

[Tra+11] Minh Tran et al. “On the expressiveness of return-into-libc attacks”. In: *Recent Advances in Intrusion Detection.* Springer. 2011, pp. 121–141.

[WK02] John Wilander and Mariam Kamkar. “A comparison of publicly available tools for static intrusion prevention”. In: *Citeseer* (2002).

[WK03] John Wilander and Mariam Kamkar. “A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention.” In: *NDSS*. Vol. 3. 2003, pp. 149–162.