Vrije Universiteit Amsterdam

Masters Thesis

Polymorphic Operating Systems

Candidate:	*Supervisor:*	*Second Reader:*
Anton Kuijsten	Prof. Andrew S. Tanenbaum	Cristiano Giuffrida

August 2012

Abstract

In recent years, kernel-level exploitation has gained popularity. Existing countermeasures do not offer a comprehensive defense against kernel memory error exploits.

In this paper, we introduce a novel design of system-level address space layout randomization (ASLR), that provides a comprehensive countermeasure against memory error exploits. While many ASLR implementations offer protection for the application layer, our design protects the operating system itself. We present a fine-grained randomization that covers all regions in the address space. Central to our design is the periodic live rerandomization of a running operating system, which counters brute force attacks. This rerandomization mechanism is fault-resilient, so it does not decrease system reliability. Results from SPEC and syscall-intensive benchmarks show a performance overhead less than 5% and a memory footprint overhead around 15%. This indicates that our design does not compromise performance, while providing effective and reliable protection for the operating system.

Contents

Chapter 1

Introduction

1.1 Kernel-level Exploitation

Kernel-level exploitation is increasing. Attackers use weaknesses in security measures to tamper with kernel memory, for example to disable security or execute injected code.

There are many proposed countermeasures, but they each protect against a very specific attack. These are examples of specific protections offered by individual countermeasures:

- Preserving kernel code integrity [1]
- Kernel hook protection [2]
- Control-flow integrity [3]

In order to protect memory, most countermeasures need virtualization support. Unfortunately, this introduces a significant performance overhead. Also, virtualization is not always desired for other reasons, such as complexity and licensing issues.

1.2 Address Space Layout Randomization

In this thesis, we propose to use address space layout randomization (ASLR) as a comprehensive countermeasure against kernel-level memory exploits. ASLR is a well known technique used to counter a wide range of attacks. These attacks have in common that the attacker uses information on the address space layout of an application or system that is attacked. ASLR is used to prevent the attacker from having this information. This is done by randomizing the addresses of data and instructions in the address space.

As an example, we will discuss a case in which an attacker wants to modify a variable in a process. If the attacker knows that the variable is preceded in the address space by an input buffer, he might be able to overflow that input buffer, and overwrite the variable (of course, for this example to work, input must not be checked on size). This is called a buffer overflow attack. It is only one example of using knowledge of the address space to execute an attack. ASLR can be used to prevent this attack. In this example, this can be done by randomizing the order of the variables, the input buffer and other data in the address space.

1.3 Goals

We want to use ASLR to protect kernel memory. In this section, we will state our goals for our solution.

1.3.1 System-level ASLR

Currently, most ASLR implementations focus on application-level ASLR. All mainstream operating systems offer support for securing applications this way. However, they do not protect themselves with this mechanism. Except for Microsoft Windows, which only offers a weak support for system-level ASLR, described in the next section. Our solution will offer a full featured system-level ASLR implementation.

1.3.2 Fine-Grained ASLR

Microsoft Windows only provides for a weak level of randomization; only the base address of the text segment is randomized. Our solution will be able to randomize the addresses of individual variables and functions independent of each other.

1.3.3 Live Rerandomization

Central to this thesis is a design and implementation of a novel ASLR feature that counters a new kind of attacks. Attackers are using brute force to overcome the lack of information on the address space layout caused by ASLR. The idea is that when the position of a target in the address space is not known, you find it by trial and error, with brute force. To counter this, we present the first ASLR implementation that can rerandomize the address space layout of a running system. Periodic and frequent live rerandomization prevents the randomness from being exhausted by brute force.

1.3.4 Reliability

Fine-grained live rerandomization of an address space is a complex operation. We want to avoid using error-prone techniques, such as binary rewriting. Also, if a rerandomization procedure fails, then this should not affect system reliability. Therefore, the failure must be isolated and recoverable.

1.3.5 Performance

Techniques such as binary rewriting, pointer indirection and code instrumentation can introduce a significant performance overhead. At the system level, this is not acceptable. Only a minimal overhead is acceptable when introducing changes to the system.

1.4 Design Overview

In this section, we will discuss how our framework will work on the MINIX 3 operating system, using the LLVM Compiler framework.

1.4.1 Microkernel

We implemented our system-level ASLR framework on MINIX 3, a microkernel operating system. Instead of running as one large kernel, the microkernel system is divided into multiple task-specific services that run as isolated processes. This design presents opportunities to solve the challenges introduced by live rerandomization. First of all, each individual system service can be stopped in a known state while the rest of the system continues to operate. This makes it much easier to implement rerandomization, and it's impossible to do on a monolithic kernel.

Secondly, for a system service, a new process can be started, to which the old state can be simultaneously transferred and randomized. It is easier to rerandomize into a new address space than to rerandomize in-place. Also, this design makes it possible to abort a rerandomization event when an error occurs; in that case, the old system service process can continue operation, because its state won't be changed. This fault tolerance greatly enhances the reliability of the ASLR framework. We will discuss process management in chapter 7.

1.4.2 LLVM Compiler Framework

Most of the address space layout, except for dynamic memory allocations, is normally generated at link time. Our design presents the opportunity to randomize the layout of the new process with the compiler at link time. This way, a new layout can be generated by just re-linking the executable.

This prevents the difficult and error prone task of changing the address space layout by binary rewriting in a running process. Prior work includes binary rewriting and pointer indirection, which introduces performance overhead. We will discuss prior work in chapter 2. This method of rerandomization creates no performance overhead between rerandomizations for data and instructions that are allocated at link time, and only introduces a very small overhead for dynamic memory allocations. This will be discussed in chapter 5.

We use the LLVM compiler framework to implement link time address space layout randomization. The LLVM framework is built with extendability in mind, so it's easy to extend the LLVM framework with a link time state randomization module. The compiler framework ensures that the transformations to the address space layout are correct. It also makes it possible to randomize the address space layout in a fine-grained manner. Instead of changing the base address of memory segments, we can permute and pad individual functions and variables. Padding is generated by placing dummy functions or variables in between each pair of consecutive functions or variables. These dummy functions and variables will not be references anywhere in the application, and only serve to take up space.

In order to rerandomize, we need metadata on the address space layout of the old and new service process. This metadata is necessary to transfer all data and recalculate all C pointers based on the new address space layout. Therefore, another LLVM pass, called the Magic Pass, injects metadata on functions and variables into the executables. This metadata, made available by LLVM, is used at runtime to find addresses of all variables and functions in the old and new system service process. The metadata is accessed at runtime with the help of an additional user library, called the Magic Library. Together, the Magic Pass and the Magic Library form the Magic Framework. The initial version of this framework was implemented outside of the scope of this project, and will be discussed in chapter 6.

1.5 Outline

In chapter 2, we first discuss the literature on ASLR and prior solutions. Then, we discuss the design of MINIX 3 in chapter 3. Our design is discussed in chapter 4. In chapter 5 and chapter 6, we will discuss the ASLR and metadata instrumentation. In chapter 7, we discuss the rerandomization event, and we go into more details of state transfer in chapter 8. Finally, in chapter 9, we will evaluate our implementation.

Chapter 2

Background

In this chapter, we will discuss literature on other ASLR implementations. We will discuss features and issues with operating systems, legacy applications and distribution models. Also, the effectiveness of rerandomization will be analyzed.

2.1 Protections Against Specific Attacks

Many of the proposed countermeasures protect against very specific attacks. StackGuard [4] is an example of a specific protection. StackGuard was developed specifically to protect against stack smashing. With this attack, the return address on the stack is overwritten using a buffer overflow [5]. StackGuard puts canary values around the return address. Stack smashing is detected by StackGuard when these canary values are modified by a buffer overflow.

While StackGuard protects these specific pointers, it does nothing to prevent other kinds of memory error exploits. For example, non-pointer arguments to *execve()* are not protected. Pointers in the heap area are also not protected. Therefore, various ASLR implementations are presented as a generic protection against memory error exploits [5–8].

2.2 Operating Systems and Legacy Applications

Some ASLR solutions require changes to the operating system or applications. This may be a problem when legacy software or existing distribution models have to be supported.

2.2.1 Kernel and User Level Solutions

PaX [8] is a patch for the Linux kernel that provides some ASLR features to applications. It randomizes the base address of the stack, heap, code, and memory-mapped segments. It requires no changes to applications, except that the binaries must contain position-independent code (PIC).

On the other hand, Bhatkar et al. [6] provides an ASLR implementation that does not require changes to the operating system. Instead, it requires a source-to-source transformation of the application source code.

2.2.2 Compiler Modifications and Software Distribution

This source-to-source transformation does not require manual programmer intervention or changes to the compiler. However, it does put a requirement on the availability of the application source code.

Other implementations feature binary rewriting, such as Bhatkar et al. [5]. This is a technique that is more error prone than source-to-source transformation, and is limited in the complexity of transformation. This approach allows applications to be distributed as a binary, so no changes to the distribution model are necessary.

2.3 Relative Address-Dependent Attacks

As discussed in the previous section, PaX limits its ASLR transformations to randomizing the various memory segments. This will prevent the attacker from overwriting a pointer with the address of a specific memory object. For this kind of attacks, the attacker has to know the absolute address of the memory object that must be pointed to, in order to successfully change the pointer.

This protection does not guard against attacks on non-pointer data. For a buffer overflow attack, the attacker only needs to know the relative distance between the buffer and the target non-pointer data. This distance will not be changed when only the base addresses of the memory segments are randomized.

In order to change the relative distance between memory objects, we need solutions such as Bhatkar et al. [6], that introduce fine-grained ASLR. These solutions change the relative distances between individual code and data objects by introducing object permutation with random gaps between individual objects.

2.4 Fine-grained ASLR

Before we discuss the various ASLR techniques for different memory segments, we will now define a general approach to fine-grained ASLR.

In order to randomize the absolute addresses of memory objects, each segment gets a random offset, similar to PaX. Because there is only one offset per segment, we can choose a large random value without the risk of running out of virtual memory space.

In order to randomize the distance between objects, each segment is permuted. Therefore, we need to define the individual objects. Bhatkar et al. [6] splits the code segments into individual functions, the stack and static segments into individual variables, and the heap and map segments into individual allocations.

In case the order of some memory objects cannot be changed, we also introduce gaps between individual objects, which we call padding.

In the rest of this section, we will discuss different ASLR techniques used for other implementations. Because of their individual characteristics, there are different techniques for the various memory segment. Therefore, we will discuss each segment separately.

2.4.1 Stack Randomization

The global offset of the stack is created by subtracting a random value from the stack pointer when a process is started. The stack pointer is modified by Bhatkar et al. [5], using code that is added to the start of the main function. The kernel is modified by PaX [8] and Kil et al. [7] to change the stack pointer.

The stack offset is protected with *mprotect()* by PaX [8] and Bhatkar et al. [5]. An attack that writes into the stack offset will cause the target process to crash. This prevents illegal access to process memory, and is likely to be noticed by an administrator.

In Bhatkar et al. [6], random padding is generated between stack frames at compile time. This is done by inserting *alloca()* calls at the start of each function. Instead of padding and permuting individual stack variables, buffer variables and variables whose addresses are taken are allocated on a shadow stack. This prevents buffer overflow attacks on the variables whose addresses are not taken.

2.4.2 Heap Randomization

In Bhatkar et al. [5, 6], A global offset is added to the heap by adding a call to *brk()* to the start of the main function. A call to *malloc()* would not work, because for large allocations, *malloc()* will call *mmap()*.

In order to generate padding between *malloc()* invocations, *malloc()* is instrumented to allocate a random amount of extra space. Permutation is not implemented, as this would make the *malloc()* implementation too complex with too much overhead.

The heap is in Kil et al. [7] randomized by the kernel, and for this only a randomized offset is determined, just like the stack is randomized.

2.4.3 Map Randomization

In Kil et al. [7], the kernel randomizes the allocation of pages for *mmap()* calls.

2.4.4 Static Data Randomization

Global variables in the data segment are randomized by Bhatkar et al. [6] with a source-to-source transformation. First, each variable is replaced by a pointer of a similar type. For example, each "int i;" variable is replaced by "int *i_ptr;". Next, all instructions are modified to work with these pointers. For example, "i++;" is changed to "*i_ptr++;". Finally, a constructor function is added that dynamically allocates and initializes all global variables, and updates the pointers. Allocation is implemented by calling *mmap()*. A region is allocated at a randomized address that is able to hold all variables permuted and interleaved with padding.

In Kil et al. [7], the global variables are permuted using binary rewriting before execution. For this, it requires the symbol table and the following sections: relocation, global offset table, procedure linkage table, relocation data, and relocation text. These sections are provided by the linker when providing a few extra command line arguments. Using these command line arguments is the only requirement for Kil et al. [7]. Any binary can be randomized without needing source or object files.

These sections provide information on the name and start address of variables and functions. Their end address can be inferred by looking up where the next object starts. Furthermore, it provides information on where in the code segment these objects are used. This way, all variables and functions can be relocated, and the usage of their pointers in the code and data segments can be updated.

2.4.5 Code Randomization

The code segment is not permuted or padded by Bhatkar et al. [5]. This is introduced in Bhatkar et al. [6]. Instead, only the global offset of the code segment can be randomized. Bhatkar et al. [5] can use two different methods to offset the segment. The first option is to build the entire binary as a dynamic library, and dynamically load it with a wrapper *main()* function at a randomized address. The second option is to add a random offset at link time. Only the first option introduces an overhead.

There are two distinct steps in code randomization in Bhatkar et al. [6]. First a source-to-source transformation is performed that adds an array of function pointers. For each function, a pointer is initialized to hold its address. Next, all function calls are made indirect: functions are invoked by dereferencing the function pointer. The array of function pointers is write-protected. After this transformation, the binary is created.

The next step actually randomizes the functions. Using the LEEL binary-editing tool [9], the function pointer array can be modified, and the functions can be moved around. When sorting the array, we can determine the start and end address of each function. The end of a function is determined by assuming that it precedes the start of the next function. Using this information, we can permute and pad the functions, and update the function pointer array. Application code is randomized by periodically modifying the binary before execution.

In Kil et al. [7], the functions are relocated in the same way global variables are relocated, as is discussed in the previous subsection.

2.4.6 Dynamic Library Randomization

In order to randomize the offset of dynamic libraries, the dynamic loader is modified by Bhatkar et al. [6]. The offsets of dynamic library addresses are only randomized statically at link time by Bhatkar et al. [5]. The latter technique has the effect that each execution of the same binary results in the same dynamic library offset. However, it does not need a modification to the dynamic linker.

2.5 Performance

The pointer indirection of global variables, as used by Bhatkar et al. [6], introduces an overhead of 9% during a test with the gzip binary. However, a lot of the variable accesses in the gzip test were done on global variables (40%). Most other tested applications

accessed global variables for only 0-10% of all variable accesses, and did not have a high performance overhead.

A random heap or stack offset does not introduce a performance overhead. Also, link time random stack frame padding in Bhatkar et al. [6] does not introduce any overhead.

For Bhatkar et al. [6], shadow stack allocations and code indirection (function pointers) both resulted in a 0-10% overhead with most tested applications. The code segment offset that is created at load time by Bhatkar et al. [5] introduces a 0-20% overhead for different applications.

In Kil et al. [7], *fork()* and *exec()* calls are slowed down by 6.86% and 12.53% respectively. For PaX [8], this is 13.83-21.96%.

2.6 Effectiveness

A test published in Shacham et al. [10] shows that it takes 216 seconds on average to compromise Apache running on a PaX-enabled Linux system. This is achieved by using a brute force attack that guesses the address-space layout.

2.6.1 Effectiveness of Different Randomization Methods

Compile and link time randomization techniques have more control over the address space, and are able to provide a more fine-grained randomization with a higher entropy.

Also, kernel level approaches, such as PaX do not provide a lot of entropy. That is because only global offsets for different memory segments are applied. Without object permutation and padding between individual objects, there is no guard against relative address-dependent attacks.

2.6.2 Run Time Rerandomization

According to an analysis in Shacham et al. [10], run time rerandomization will give only one bit more randomness. This means that it will only take twice as long to compromise the target, compared to link or load time randomization.

The best rerandomization frequency would be between each attempted attack. It is an intuitive idea that when the attacker attempts the exact same attack with each attempt, it is expected to succeed at some point, because the rerandomization procedure will then

choose the layout that the attacker has in mind. As stated above, this is calculated to take only twice as long.

However, that paper does not take into account that for most attacks, multiple memory object have to be targeted. After successfully finding the first target object, a failed attempt to hit the second target can trigger a crash, followed by a rerandomization. This way, an attacker will find it difficult to find more than one target.

Chapter 3

MINIX 3

We have implemented our ASLR design on the MINIX 3, a POSIX-compliant UNIX clone. In this chapter, we will discuss properties of MINIX 3 that are relevant to the implementation of ASLR. This includes the architecture, interprocess communication and the System Event Framework (SEF).

3.1 Microkernel Architecture

MINIX 3 has a microkernel architecture. As stated in the introduction chapter, and to be discussed in chapter 4, our implementation of ASLR depends on this. In contrast to a monolithic kernel architecture (Linux, Windows), in a microkernel architecture, most of the system functionality is moved from the kernel to a collection of processes. This has two advantages: modularity and reliability.

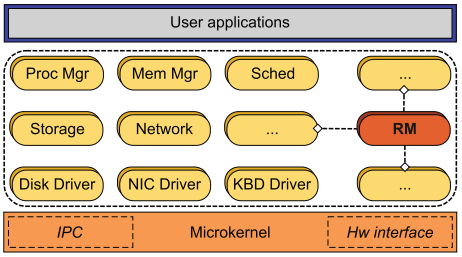

FIGURE 3.1: MINIX 3 Architecture.

3.1.1 Modularity

Modularity is achieved by separating the system functions into different processes, according to functionality. For example, in MINIX 3, the virtual file server (VFS), virtual memory server (VM), process manager (PM) and network server (INET) are all processes outside of the kernel. Their names are self-explanatory. Hardware drivers are also isolated in separate processes, such as the SATA driver and Realtek RTL8139 driver. Some servers depend on a driver. For example, the network servers needs the RTL8139 driver to use the RTL8139 network interface. The modular design offers the well known advantages: Separation of concerns, maintainability, extendability, and interchangeable modules. Individual system processes are shown in Figure 3.1 in the middle.

3.1.2 Reliability

Reliability is another property of the microkernel architecture. The system processes (servers and drivers) do not run at the highest privilege level. Instead, they run as regular processes, restricted by the MMU. This prevents faults in a system process affecting other parts of the system. For example, a crash of the network server will not affect the operation of the file server. In addition to fault isolation, this design also makes it possible to restart crashed system processes by starting a new instance.

3.1.3 Kernel

To perform I/O, system processes make calls to the kernel, which checks if they're authorized to do so. In addition to I/O, the kernel only manages process scheduling, interprocess communication (IPC) and hardware interrupts. Because of this, the MINIX 3 kernel contains only 9000 lines of code, compared to 6 million lines of code in the Linux kernel. This greatly reduces the predictable number of bugs in the kernel, the part of the operating system that is critical to its reliability. The kernel is shown at the bottom of Figure 3.1.

3.1.4 Overhead

The microkernel architecture introduces IPC calls and extra context switches. This results in a performance overhead of 5-10%, measured by comparing MINIX 3 user-mode drivers to MINIX 2 kernel-mode drivers [11]. The focus on reliability and security instead of performance, and the differences between MINIX 3 and other operating systems (e.g. lack of DMA in the disk driver), make it difficult to compare the performance of MINIX

3 to other operation systems. For L4, a microkernel that was developed with a primary focus on performance, a performance overhead of only 5% is reported for a system call-intensive benchmark [12].

3.2 Interprocess Communication

In this section, we will discuss how system calls are implemented, and how processes can communicate with each other on MINIX 3.

3.2.1 Message passing primitives

In MINIX 3, Interprocess Communication (IPC) is implemented with fixed-length messages. The following functions offer the available message passing primitives.

send	Send a synchronous message	blocking
sendnb	Send a synchronous message if other process is receiving	nonblocking
senda	Send an asynchronous message	nonblocking
receive	Receive a message	blocking
sendrec	Send a message and wait for reply	blocking
notify	Send an empty asynchronous message	nonblocking

3.2.2 Kernel

When a process wants to send or receive a message, it traps to the kernel, which handles message passing. For each message passing request, the kernel checks if the process is authorized to execute that particular primitive between itself and the other process involved. The kernel also translates hardware interrupts into messages to the drivers.

3.2.3 System Calls

Each system call is handled by a designated system process. For example, the virtual files system server (VFS) handles all file system calls. A few system calls are handled by the kernel itself. A user process makes a system call by sending a request message to the designated system process, which handles the call and responds with a reply message. Each message contains a system call number, system call arguments, and a sender endpoint number set by the kernel, which identifies the sender.

3.2.4 System Processes

Each system process contains a handler function for each system call that it supports. The handler functions are accessed by using the system call number as an index into a global array of function pointers.

After initialization, each system process enters a never ending service loop. At the top of the loop, the process receives a system call from a user process by means of a message. Then, the appropriate system call handler is called. After the system call handler returns to the loop, the process sends a reply message to the user process, and jumps back to the starts of the loop. Because a loop iteration handles only a single request, it is short-lived.

A fair number of messages are sent between system processes, because they need to support each other. For example, other system processes need VM for memory allocation, just like user processes. Also, they need to synchronize or update each other. For example, when PM handles the fork system call, it needs to update VM and VFS on the newly created process.

3.2.5 User Processes

To make a system call, user processes can only execute the *sendrec()* primitive: after sending, the process awaits a reply. This way, after receiving a request message from a user process, a system process can always send a reply message that is received immediately. This guarantees that a system process won't be blocked by a busy user process. User processes are only authorized to exchange messages with system processes. For communication between user processes, system processes handle system calls for standard UNIX IPC, such as signals, sockets and pipes. User processes are situated in the application layer at the top of Figure 3.1.

3.2.6 Interprocess Copying

If a system call argument is too large to fit in a message, a pointer and a size value is put in the message instead. The system process can then copy the data by using a set of special kernel calls. A small subset of the system processes are allowed to use the *sys_vircopy()* and *sys_physcopy()* kernel calls, using a source and destination pointer and a size value. These functions can also be used to copy the result of a system call back to the user process.

3.2.7 Safecopy

Because these calls circumvent the protection offered by the MMU, this reduces system security and reliability. Therefore, the other system processes have to make use of a safe system of grants that processes can create for each other. A grant provides a specific process read or write access to a specific memory region. Some system processes are authorized to create grants to memory regions from other processes. It is also possible to create indirect grants to memory regions that are accessible through another grant. After a grant is registered with the kernel, data can be copied with the sys_safecopyfrom and sys_safecopyto kernel calls.

3.3 System Event Framework (SEF)

In this section, we will discuss how the System Event Framework (SEF) takes care of system process management in MINIX 3.

3.3.1 Reincarnation Server

In order to have a reliable system, MINIX 3 has a crash recovery mechanism built into its system processes. Crash recovery is achieved by replacing a crashed system process with a new instance, which is handled by the reincarnation server (RS). Besides crash recovery, RS also handles other system process management tasks, such as starting, stopping and updating system processes.

3.3.2 SEF Library

In order to manage the various system process management events, RS has to communicate with the managed system processes using special messages. In order to handle these messages, the system processes rely on a set of message handler functions inside a special linked-in library. Together, this library and RS are called the System Event Framework (SEF).

In order to separate SEF messages from regular system call messages, system processes rely on a function in the SEF library. When called from the service loop of a system process, *sef_receive_status()* returns only regular system call messages back to the service loop. When it receives a message with a SEF system call number, it is dispatched to the specific SEF handler function.

The SEF library makes it possible for system processes to replace some of the default SEF handler functions with custom callback functions.

3.3.3 Data Store

In order to survive crashes and undergo updates, stateful system processes have to save their internal state in a place that's accessible to the process instances that replace them. The data store server (DS) offers a key value store to which data from the internal state can be stored and retrieved.

Currently only the drivers are implemented with the storage and retrieval of their complete internal state. The internal state of most servers is too complex to be accurately covered by a store and retrieve routine. Instead, there is experimental support to copy the entire heap, stack and global variables from the old to the new process. This requires that the exact same binary is used as replacement.

3.3.4 Crash Recovery

RS is the parent process of all other system processes. Therefore, when PM detects the crash of a system process, it sends a SIGCHLD signal to RS. Instead of crashing, a system process can also be caught in an infinite loop, or is blocked waiting for a message from another process that does not reply. In order to detect such an unresponsive system process, RS periodically sends a watchdog message to all system processes. The system processes have to send a response within a certain time frame, in order to be considered alive and healthy. These request and reply messages are both sent nonblocking, to prevent the mechanism from slowing down the system, and to prevent deadlocks.

When a system process is crashed or unresponsive, RS will start a new instance of the system process. After starting, processes always block to receive a message from RS. This message indicates if it concerns a crash, an update, or a fresh start. In case of a crash or update, the message contains information on the old process that is being replaced, such as the old endpoint number. The new process will then retrieve the internal state saved by the old process from the data store.

Most errors occurring in system processes are transient, and therefore recoverable with this recovery mechanism. For example, a crash can be caused by a segmentation fault that is triggered by dereferencing a dangling pointer. Another example is deadlock between communicating system processes, caused by a rare ordering of parallel actions. After recovery from transient errors, the system process is often able to continue operation without any noticeable glitch.

3.3.5 Starting, Stopping and Updating System Processes

Other system process management includes starting and stopping of system processes. For example, when a file system is mounted or unmounted, a file system server is started or stopped.

It would be possible to implement a live update of a system service if the current version saves it complete internal state to the data store. RS would have to send a special message to the old process, which would then save its state to the data store, and report back to RS. RS would then start a process instance of the new version, and instruct it to retrieve and convert the old state from the data store. Afterwards, the old process is killed, and the new process resumes control.

Recently, a more complex mechanism for live updating system processes has been implemented. In order to transfer and update its internal state, it relies on the same Magic Framework as our ASLR implementation. While our ASLR framework transfers the process state between two processes with a different address space layout, this live updating system does the same with two processes of a different software version. The framework can be extended with custom code that can translate the part of the state that is different between the two versions.

When an update event fails, RS is able to perform a rollback, and let the old process continue its service. This mechanism is part of a detailed discussion in chapter 4. Therefore, we will not discuss its details here.

3.3.6 Service Command-Line Tool

Events such as system crashes are handled by RS without any user intervention. For events such as live updates, or manually loading device drivers, the user has to be able to communicate with RS. In order to send commands to RS, and get back results, there is a command-line tool called service.

3.3.7 Signal Management

RS is registered as the signal manager of all system processes. It converts signals into SEF messages that are sent to the processes. For each SEF signal number, the default signal handler function is implemented in the SEF library. These functions can be replaced by custom callbacks.

3.3.8 Dependence On Other System Services

RS depends on PM to create new processes, and on VFS to load binaries from disk. VFS and the kernel are informed of the privileges of the newly created processes by RS. Also, as stated above, PM will inform RS when it learns about a system process crash.

Chapter 4

Design

This chapter gives a overview of the design of our ASLR framework. Here, we will discuss the necessary functionality in detail, including the detailed requirements for internal state management. However, implementation issues that do not affect our design will be left for other chapters.

4.1 Recapitulation of the Design

It is necessary for the reader to understand the section from the introduction chapter that introduces the design of our framework. We will quickly recapitulate the most important design features here.

- The MINIX 3 operating system is divided into a microkernel and a set of system processes. In order to rerandomize the system, we will rerandomize individual system processes.
- We will randomize the address space layout of a process at link time, using a new LLVM pass.
- In order to rerandomize a running process, we first load a new randomized instance from a different binary. Then, we transfer all internal data from the old instance to the new instance, and let the new instance resume execution.
- In order to locate all the data during transfer, we use address space layout metadata that is injected by another link-time LLVM pass.

4.2 Chapter Structure

The discussion in this chapter will be centered around our goal of live rerandomization. We will divide this discussion into different sections, each detailing a separate aspect of live rerandomization:

- ASLR Instrumentation
- Metadata Instrumentation
- Live Rerandomization Event
- State Transfer Routine

4.3 ASLR Instrumentation

In this section and the next section, we will discuss the two LLVM Passes that are part of our framework. At this point, we will only discuss the general design features of these two passes.

4.3.1 ASLR Overview

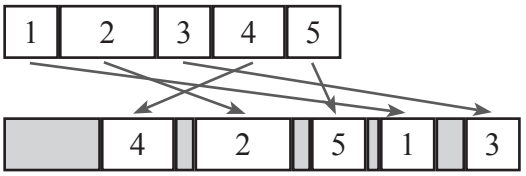

FIGURE 4.1: General memory region randomization approach.

The ASLR pass has to randomize both the text and data of a process. Also, there are different regions of data: global variables, stack, heap and map. Each of these regions can be viewed as a row of memory objects that has to be randomized. These memory objects can be individual function, global variables, and so on. The layouts of these different rows will be randomized according to the same general idea, illustrated in Figure 4.1:

- The order of objects is randomly permuted
- A padding of a random size is inserted between each pair of consecutive objects

- A large global offset of a random size is inserted at the start of the row. A large size offers more randomness. While it is not possible to generate a large padding between each pair of objects, because of the required memory, it is possible to create one large offset for each region.

In the rest of this section, we will discuss the specific instrumentation needed for each region, without going into implementation details.

4.3.2 Global Variables

Global variables have the same life span as the process itself. Therefore, their addresses are decided at link time. Therefore, the LLVM pass permutes the list of global variables, and inserts the paddings and the global offset. The size of the paddings and offset is randomly decided by the pass. It is not important that the pass decides on the actual address of each variable, as long as LLVM will respect the order, paddings and global offset.

4.3.3 Functions

Functions are randomized with the exact same approach as global variables.

4.3.4 Stack Instrumentation

Before each function call, a random offset is allocated on the stack. This creates an unpredictable gap between the stack frames of each function call on the stack. This way, exploits cannot execute illegal stack variable initializations between different calls to the same function, because they cannot predict the next location.

In a return-to-libc attack, the attacker overwrites the return address of the current stack frame. This address is used to jump back to the call site of the current function. This way, an attacker is able to put custom arguments on the stack, and jump to functions in the libc library that are dangerous when used improperly, such as *execve()*. The stack frame padding creates a randomness that prevents the return address from being overwritten.

In addition to stack frame offsets, a large random offset is allocated at the bottom of the start, at the beginning of the *main()* function.

All individual stack variables are permuted. Also, we separate two different kinds of stack variables. The variables whose addresses are used in pointer arithmetics are separated from those who are not. The former group is pushed to the bottom of the stack frame, with a random padding before and between them. This way, pointer arithmetics can't be exploited to change other variables.

The way in which stack variables, paddings and offsets are implemented, is discussed in chapter 5. There, we will also discuss optimizations that will prevent dynamic randomization of offsets and padding to have a negative impact on performance.

4.3.5 Allocator Function Instrumentation

It would be too much effort to permute dynamic allocations from *malloc()* and *mmap()* during normal execution. We would need to make changes to the allocator implementation, or create a pool of allocations to instantly permute allocations.

Instead, we rely on the rerandomization event to permute the order of dynamic allocations. When the state is transferred to a new process, the dynamic allocations have to be allocated again in the new process. The transfer routine can simple permute the order in which it reallocates the buffer. This randomizes the location of the dynamic allocations between the two processes.

We do need to create a random global offset and padding between allocated buffers. We do this by instrumenting the allocator functions. For each allocation, we add a random padding to the total allocation size. *munmap()* will need a size when deallocating a buffer.

4.4 Metadata Instrumentation

In this section, we will discuss how the binaries are injected with metadata and also instructions that dynamically generate metadata.

4.4.1 Magic Pass

The Magic Pass is the LLVM pass that is responsible for metadata instrumentation. It has to extract all necessary metadata from the LLVM API. It has to structure this data in a way that it is accessible to the system process at runtime.

Most of the metadata is generated at link time, and injected into the binaries as variables. There is also a need for metadata that is generated dynamically. This needs arises out of the need for metadata for dynamic memory allocations.

For the rest of this section, we will discuss the metadata that is generated and instrumented by the Magic Pass.

4.4.2 Metadata

In this section, we will discuss the metadata that is necessary for ASLR, without going into implementation details. We will find out what metadata we need by going over example code snippet below, line by line. On the right side of the snippet, we added the metadata in red. Actually, it is an informal, incomplete representation of the metadata, suitable at this point in the discussion. This is, of course, not part of the source code.

```
1  int *p;                                <"p", PTR, address = &p, size = 4>

2  int i;                                 <"i", INT, address = &i, size = 4>

3  struct {                               <"S1", STR, address = &S1, size = 12>
4    int x;                                   <"x", INT, offset = 0, size = 4>
5    struct {                                 <"S2", STR, offset = 4, size = 8>
6      int a;                                    <"a", INT, offset = 0, size = 4>
7      int *b;                                   <"b", PTR, offset = 4, size = 4>
8    } S2;
9  } S1;

10 int calc(int a){                       <"calc", FUNC, &calc>
11   int z = ...;

12   p = malloc(i * sizeof(int));         <"calc.p", ARR, address = p, size= 4*i>
13                                            <INT, size=4>
14   free(p);
15 }
```

LISTING 4.1: Example Code Snippet of memory objects in C. Pseudo metadata is shown on the right.

4.4.2.1 Pointers

```
1  int *p;                                <"p", PTR, address = &p, size = 4>
```

The first metadata value holds the name of the variable. We need this to match the variable in the old and new process together.

The second field holds the type of the variable. In this case, it is a pointer. We need the type, in order to decide which transfer strategy to follow. In this case, the value of

the pointer has to be changed. This is because it points to an integer variable that has moved.

Next, we find the address of the pointer. When we have the metadata of both the old and new process, we can transfer the pointer between their address spaces.

Lastly, the size of the variable is also needed for the transfer of the pointer.

4.4.2.2 Primitive Variables

```
2 int i;                                <"i", INT, address = &i, size = 4>
```

On the next line, we see the metadata information for an integer. It has the same kind of metadata as the pointer. In this case, the type is INT, which means that we have to transfer the value as is. Other primitive variable types get a similar kind of metadata.

4.4.2.3 Complex Types

```
3 struct {                              <"S1", STR, address = &S1, size = 12>
4   int x;                                  <"x", INT, offset = 0, size = 4>
5   struct {                                <"S2", STR, offset = 4, size = 8>
6     int a;                                   <"a", INT, offset = 0, size = 4>
7     int *b;                                  <"b", PTR, offset = 4, size = 4>
8   } S2;
9 } S1;
```

On line three, we find struct variable S1. On the first line of the metadata, we find information on the struct itself. From the type field, we can decide that we are dealing with a struct. The size field gives the total size of the struct.

The struct has different members. Each member may need a different transfer strategy, as we have in the previous sections about pointers and integers. Therefore, we need separate metadata for each separate member. The struct in this example even contains another struct. This way, the metadata becomes nested.

Complex types, such as structs, unions and arrays, are not randomized internally. There is no padding injected between struct members, and their order does not change. This will be discussed in chapter 5.

The members don not have an address field. Instead, they have an offset within their parent. Their address can be found by adding the offset to the address of their parent. This way, when metadata is generated dynamically, only the address of the outer struct has to be set dynamically. The member metadata can be generated at link time, because it will not change. This way, the performance overhead is minimized.

An example of an array will be discussed in a moment. Unions will be discussed in chapter 6.

4.4.2.4 Functions

```
10 int calc(int a){ ... }              <"calc", FUNC, &calc>
```

On line 10, we find a function. Functions do not have to be transferred, but we need metadata anyway. This is because we need the old and new addresses for function pointer transfer.

We also need the name, because the old and new metadata has to be matched. We need the type in order to find out that we are dealing with a function.

4.4.2.5 Stack Variables

```
11    int z = ...;
```

Until now, we have only discussed metadata that can be generated at link time. This is because the addresses of the global variables and the function that we discussed are all decided by the linker.

On line 11, we see a local function variable. This variable will be allocated dynamically on the stack. Due to the design of the MINIX 3 system services and the rerandomization event, we do not need to generate metadata for stack variables. This will be explained in subsection 4.5.3. This is good news, because dynamically generating metadata causes performance overhead. The huge number of stack allocations would probably cause a considerable performance drop.

Keep in mind that the stack layout is still randomized by the ASLR pass. We just do not need any instrumentation, because there will be no stack data transfer.

4.4.2.6 Heap and Map Allocation

```
12    p = malloc(i * sizeof(int));      <"calc.p", ARR, address = p, size= 4*i>
13                                          <INT, size=4>
```

On line 12, we see a call to *malloc()*. Unlike stack allocation, allocation on the heap and map has to generate metadata dynamically. Fortunately, this kind of dynamic allocation happens a lot less than stack allocation. Furthermore, MINIX 3 system processes are designed to do a minimal amount of dynamic allocation, so this reduces the overhead even more.

This function call illustrates that we do not only need the LLVM pass to inject static metadata, but also instrument the code to dynamically generate metadata.

Note that the metadata does not have to be linked to the pointer, p, which is assigned the allocated region pointer. The pointer and the allocated region are two separate objects in the address space, and get assigned different metadata.

The allocated region in this example is identified by a name, "calc.p", which is a combination of the function name and the pointer to which it is assigned.

This allocation example is also an example of the metadata structure of an array. We see that the array metadata holds the total size of the array. There is only one entry for member metadata, even if there are more than one array positions. That is because all members of an array are of the same type. This way, larger arrays do not require more space for metadata, which makes this approach scalable.

```
14   free(p);
```

On line 14, we find a call to free that deallocates the same region. This call is instrumented to remove the same metadata that we discussed above.

Calls to *mmap()* and *munmap()* are instrumented in a similar way as *malloc()* and *free()*.

4.4.2.7 Metadata Summary

These properties summarize the characteristics of metadata:

- We need name, address, type and size metadata for variables and functions.
- We need offset, type and size metadata for complex variable members.
- Metadata for global variables and functions is generated at link time.
- Allocated regions on the heap and map get assigned metadata dynamically by instrumenting *malloc()*, *free()*, *mmap()* and *munmap()*.
- Stack variables do not need metadata

The Magic Pass, used to generate metadata with LLVM, will be discussed in chapter 6.

4.5 Rerandomization Event

In this section, we discuss the process management of the rerandomization event. We will learn how the reincarnation server coordinates the system process instances. In this section, we will not discuss state transfer in detail. That part of the rerandomization event will be discussed in the last section of this chapter.

4.5.1 Reusing the Update Mechanism

In subsection 3.3.5, we discussed the live update event mechanism that is built into the System Event Framework (SEF). We found that this mechanism can be reused to implement the live rerandomization event mechanism. The way in which the old and new process must be put in the right state for the event, and monitored for errors, are the same for live update and live rerandomization. Therefore, the live update event is generalized to implement both events, using an extra flag in the associated SEF messages.

4.5.2 Rerandomization Event Flow

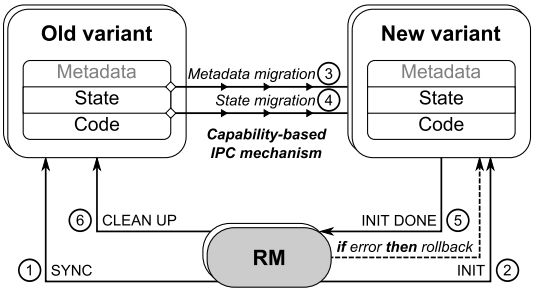

FIGURE 4.2: Rerandomization Event.

During rerandomization, the reincarnation server coordinates the old and new system processes between which the state is transferred. Figure 4.2 shows the order of messages that are sent and received by the reincarnation server and the two processes. These message dictate the ordering of the event. These are the different states of the event:

1. RS has already started a new instance of the process that has to be randomized. The message from the reincarnation server to the old process tells the process to prepare for rerandomization. The old process responds with a message indicating readiness.
2. RS sends a SEF message to the waiting new process, which indicates a live rerandomization event.
3. The new process copies the metadata from the old process to its own address space.
4. The new process copies the regular data from the old process to its own address space.
5. The new process sends a reply to the reincarnation server to indicate a successful state transfer.
6. RS kills the old process, and the new process resumes execution.

4.5.3 Stopping the Old Process With a Minimal State Size

In step one, after the old process has responded that it is ready for state transfer, it blocks waiting for a message from the reincarnation server.

This way, the state is fixed and the process is not processing any regular system call. During a regular system call, a lot of temporary variables would be generated on the stack, but we do not have to worry about this now. We only have to deal with the part of the state that is persistent outside the scope of a system call.

Any stack variables that are related to the rerandomization event can be ignored, as they will not be relevant to the new process after state transfer. This reduces the state to the situation at the start of the service loop iteration.

This means that the only important stack variables are the few that were allocated before entering the service loop. However, these variables are already correctly initialized in the new process. This means that no stack variables have to be transferred, so they do not need to be instrumented, as mentioned in subsubsection 4.4.2.5.

The fact that no process is waiting on a system call result from the old process also simplifies the state of IPC.

4.5.4 Error Handling

In the previous subsection, we stated that the old process responds indicating readiness. When the process is not in a state that can be rerandomized, it responds with a message that will cancel and roll back the randomization event. An example of a state that can not be randomized can be found in VFS. When VFS wants to read data from disk, it will send a request to MFS, which contains a pointer to a buffer that has to be filled with the requested data. If there is any outstanding request, the buffer may not be moved by a rerandomization event. This is because the pointer in the message to MFS will not be updated.

When the old process is blocked after indicating readiness, it is able to receive a message from the reincarnation server. This way, the reincarnation server is able to instruct the old process to resume execution, whenever an error occurs during the rerandomization event. This is possible, because the reincarnation server will notice when the new process sends a response indicating state transfer failure, or when it becomes unresponsive or crashes.

In that case, the new process is killed, and the old process resumes. This is the opposite from normal event execution.

State transfer will be discussed in section section 4.6.

4.5.5 Command-Line tool

The service command-line tool will have to be extended, so that rerandomization can be triggered by the user or by a scheduled job. Implementation details of the new additions to the reincarnation server and the SEF library will be discussed in chapter 7.

4.6 State Transfer Routine

In this section, we will discuss the actual state transfer routine in more detail.

4.6.1 SEF message handler

The state transfer routine will be implemented inside a SEF message handler in the new process. The SEF framework will block every initialized system process waiting for a startup message. This message will indicate whether or not this is a rerandomization event.

4.6.2 Interprocess Data Transfer

In order to give access to all of its internal state, the old process will create a large memory grant for the new process before it blocks. This way, big chunks of data can be copied, while the old process does not need to handle any data request while in a blocked state.

4.6.3 Transfer of Metadata

As mentioned above, in order to copy data between address spaces, the new process needs metadata from both processes. Therefore, before data transfer, metadata has to be transferred.

Because part of the metadata is generated dynamically, the metadata is not located in one contiguous memory region. Still, we must be able to locate all metadata in the remote address space of the old process.

Figure 4.3 gives a simplified view of the global metadata structure. We see a global struct that points to various arrays and linked lists of variable and function metadata.

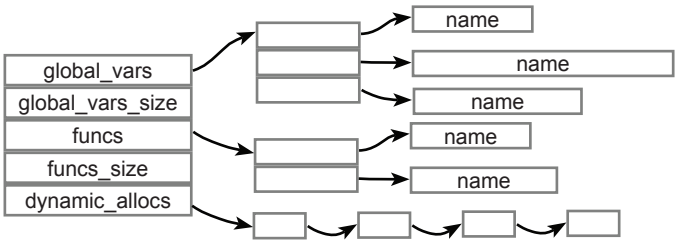

FIGURE 4.3: Metadata.

By following all pointer recursively, we can transfer the entire global metadata structure. Besides pointers, we need to know the size of the chunks of metadata. For that, we use the known size of a struct, the known number of elements in an array, or the maximum length of a string.

This way, the new process can obtain all old metadata when provided with just one pointer to the single global struct.

4.6.4 Metadata Pairing

In order to transfer a variable, we need to match the two sets of metadata. This way, when we transfer a variable from its address in the old address space, we can obtain its address in the new address space. Metadata pairing is done after metadata transfer.

4.6.5 Data Transfer

Before data is transferred, we loop through all the metadata of the dynamically allocated regions. While looping, we have to allocate each region that was allocated in the old process.

This has to be done before any data is transferred, because at that point, the addresses of all data must already be determined in the new process. These addresses are necessary to determine the new values of all transferred pointers.

All data is transferred by looping through the metadata of each global variable and dynamically allocated region. The state transfer routine delegates the actual transfer of each variable to the callback functions that are discussed in the next subsections.

4.6.6 State Transfer Callbacks

As discussed earlier, different data types require different transfer strategies (e.g., pointers vs. integers). The different strategies are each implemented in a separate state transfer callback.

```
1 int *p;                          <"p", PTR, address = &p, size = 4>

2 int i;                           <"i", INT, address = &i, size = 4>

3 struct {                         <"S1", STR, address = &S1, size = 12>
4   int x;                             <"x", INT, offset = 0, size = 4>
5   struct {                           <"S2", STR, offset = 4, size = 8>
6     int a;                              <"a", INT, offset = 0, size = 4>
7     int *b;                             <"b", PTR, offset = 4, size = 4>
8   } S2;
9 } S1;
```

Listing 4.2: Example Source Code Snippet

The code snippet above is a piece of the metadata instrumentation example in section subsection 4.4.2. On line one of the code snippet, we see an example of a pointer variable that has to be transferred. In the code snippet below, we see the pseudo code of a state transfer callback that is able to transfer pointers.

```
1 if (category == POINTER) {
2     adjust pointer(...);
3     return PROCESSED;
4 } else {
5     return NOT_PROCESSED;
6 }
```

Listing 4.3: Pointer Transfer Callback Pseudo Code

This pseudo code illustrates how a callback:

- determines if the variable type is supported
- transfers or skips the variable
- returns a value that indicates whether or not the variable is transferred

All callbacks are registered in a stack. This stack is represented in Table 4.1. The top row represents the pointer callback that we just discussed. The table describes the type support condition, the transfer strategy and the return value that is returned when the type and callback are compatible.

Description	**Condition**	**Action**	**Return Value**
pointer	type==pointer	adjust ptr	PROCESSED
primitive value	type==primitive	transfer value	PROCESSED
complex type	type==complex_value	do nothing	PROCESS_MEMBERS

TABLE 4.1: Default state transfer callbacks

The state transfer routine will start by calling the callback at the top, and continues calling the next callback until one callback succeeds. In our example, the first callback succeeds in transferring the pointer, and the state transfer routine proceeds to the next variable that has to be transferred.

The next variable in the example code is an integer. The first callback on the stack will not succeed, because it only supports pointers. However, the second callback succeeds, because it supports all primitive non-pointer types. We now see in the third column of the stack that the transfer strategies of pointers and other primitive types differ, as expected.

The third variable in our example is a struct. The first two callbacks will not succeed. The third one does, because it handles all complex types. This callback does not actually transfer anything. Instead, it returns a new kind of return value. This return value indicates that every individual member of the variable has to be transferred individually. This is exactly the behavior that we expect: This way, members of different types can be transferred using different strategies.

The state transfer routine will now process each variable member using the same callback stack. First it will successfully transfer the integer member with the primitive callback. Then, it encounters the inner struct. The complex type callback will again tell the state transfer routine to transfer each member individually.

4.6.7 Extendability with Callbacks

We will extend the previous example a bit, to illustrate the need for customization of the state transfer mechanism. Let's assume that the inner struct S2 has to be transferred as a whole with a custom callback.

Description	**Condition**	**Action**	**Return Value**
S2	name == "S2"	custom strategy	PROCESSED

TABLE 4.2: Custom state transfer callback for struct S2

In Table 4.2, this new callback is represented as a new row that has to be placed on top of the stack. Because all custom callbacks are placed on top of the stack, they are able

to override the default behavior from the built-in callbacks. In this example, the custom callback will reject to transfer all variables or variable members, except for struct S2. This is achieved by choosing the right condition for transfer.

Also, its return value is different compared to the return value from the default complex type callback. This callback will not tell the state transfer routine to transfer all members of S2 individually. Instead, it will transfer the whole struct itself.

Chapter 5

ASLR Instrumentation

In this chapter and the other chapters that follow, we will discuss the implementation details of the various components of our framework that were mentioned in this chapter. We will not re-iterate all the design details that are in this chapter. Instead, they are considered understood by the reader.

5.1 Internal Struct Randomization

In order to randomize the internal layout of a struct, its type definition should be changed by the ASLR pass. However, LLVM does not let a link time pass modify type definitions.

Creating copies of type definitions with a different internal order is a complex task that involves hacking the LLVM API. If this is done, all uses of the type should be updated to the new definition, too. Because types are used all over the place in LLVM, this will lead to a very complex pass implementation.

Arrays cannot be randomized internally in an efficient way. All pointer arithmetic that involves array indexes would have to be instrumented in the code.

There is another reason not to randomize complex types internally: When complex variables are sent to other system processes using IPC, the different systems are not allowed to use different internal layouts. Therefore, many complex types have to be excluded by hand from internal rerandomization.

5.2 Global Variables and Functions

As described in section section 4.3, global variables and functions are permuted and padded using a very similar approach. The only thing that was not discussed is how padding is implemented.

In order to pad global variables, we generate a dummy variable of a random size. We generate arrays of different sizes, in order to get randomly sized dummy variables.

Randomly sized function padding is obtained by generating dummy functions with a variable number of instructions inside.

5.3 Stack Randomization

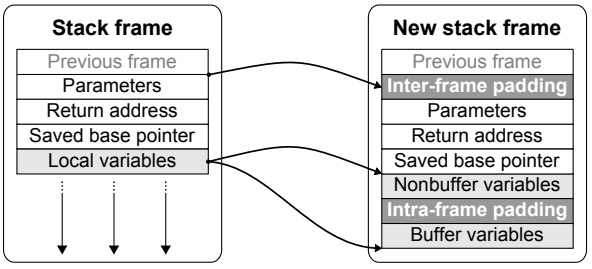

FIGURE 5.1: Stack Randomization.

On the stack, all paddings and offsets are generated with the same LLVM stack allocation instructions that are used to allocate stack variables. In other words, we use dummy stack variables as spacing. We will now describe how we apply randomization to the stack, as illustrated by Figure 5.1

Before each function call, a random offset is allocated on the stack. This creates an unpredictable gap between each pair of consecutive stack frames. It would be too expensive to generate a random offset dynamically. Even if we obtain a random offset by inlining a pseudo-random number generator before each function call, the overhead would be noticable. Therefore, each of these allocations is generated randomly at link time. This way, calls to the same function from different call sites generate a different offset.

For calls inside loops, we cycle through a random number of generated offset sizes that are generated at link time. This way, even consecutive calls from a loop have an unknown offset difference. This way, exploits cannot execute illegal stack variable initializations between different calls to the same function, because they cannot predict the next location.

For each function, stack variables that have their address used for pointer arithmetic, such as arrays, are separated from the ones that don't. That way, the address from those who have their address taken cannot be used by an exploit to access the other group. The former group is pushed to the bottom of the stack frame, and padding is inserted between each of those variables. A random offset is put between the two groups. Both groups are randomly permuted.

In order to avoid a big performance overhead, there is only one offset that is generated randomly at run time. In the main function, a global offset is allocated for the stack.

5.4 Heap and Map Allocation

We have already discussed in section 4.3 that padding is added to heap and map allocations by instrumenting the allocator functions. We did not mention yet how this works together with the metadata instrumentation of the same allocator functions.

In order to add padding, we actually do not instrument the allocators directly. Instead we add a new function to the binary that randomly chooses a padding size.

The metadata pass, discussed in the next chapter, does actually instrument the allocator functions. This instrumentation obtains a random padding size by calling the padding size function.

If the ASLR pass is not used, the metadata instrumentation just obtains a dummy zero padding value.

Chapter 6

Metadata Instrumentation

In this chapter we, will discuss the implementation details of the metadata instrumentation. We focus on both the instrumentation technique and the structure of the metadata. We will also introduce a naming scheme for the metadata, which we will also use in chapter 8

6.1 Units of Address Randomization

In chapter 4 and chapter 5, we discussed the fact that complex types are not randomized internally. This means that complex and primitive global variables are the atomic unit of address randomization. Members of complex variables only need to be assigned an offset, in order to calculate their addresses. Address information is only stored at the variable level. Variable members only contain type information and their offset.

As we saw in chapter 4, variable members of complex variables are most often transferred individually. This means that we need access to the individual type information of a variable member, so that it can be passed to state transfer callbacks.

6.2 Separation Between Sentries and Types

This leads to a structure of the metadata in which each variable is assigned a name and address. For type information, this metadata links to a tree data structure of type information. Each variable member is represented by a subtree.

The variable metadata structure is called a sentry. The type metadata structure is simply called a type.

6.2.1 Similarity With the LLVM Variable Structure

LLVM most often stores type information globally, and references this in variable declarations. Therefore, it is easy to use a similar kind of variable - type metadata separation.

6.2.2 Space Overhead Optimization

Separating type and variable information leads to a reduction in space usage, because type information does not have to be repeated for each variable. Instead, a pointer to the type metadata is sufficient.

6.3 Functions

Functions are also instrumented with metadata, because their randomized addresses can be stored in function pointers. They have a metadata structure similar to sentries: they contain a name, an address, and a link to their type metadata.

6.4 Dsentries

Variables on the heap and map cannot be assigned metadata at link time, because they are allocated dynamically. However, when their type metadata can be determined at link time, the instrumentation only has to generate variable metadata, and point to a global type metadata struct. This reduces performance overhead, because the type does not need to be generated.

However, when the size is not known at link time, type metadata has to be generated dynamically. For example, array allocations that use a variable as the allocation size, cannot be assigned a type at link time.

Dynamic variables are assigned a metadata structure called a dsentry. The dsentry simply contains space to hold a sentry and type. In the sentry, the usual variable metadata is stored. The type metadata space is only used for allocations of a dynamic size. Otherwise, the sentry will point to a global type

6.5 Instrumentation Techniques

In this section, we will discuss how different kinds of metadata are injected into the binaries.

6.6 Strings and Arrays

Names of types and variables are allocated in string buffers. The type and sentry simply contains a pointer to the string. When arrays of variable sizes are part of the metadata, they are also allocated separately, and pointed at.

6.6.1 Static Metadata

Static metadata is injected into the binary as global variables. Multiple metadata structures, such as all sentries, are grouped into arrays.

6.6.2 Dynamic Metadata

The allocator functions [*malloc()*, *free()*, *mmap()*, *munmap()*] are instrumented to generate and destroy metadata dynamically. The allocated buffer size is simply increased to also hold the metadata.

6.6.3 Traversability of Metadata

All metadata is made available through one global structure, as discussed in chapter 4. This struct is called the magic_vars struct, and holds pointers to the arrays of sentries, functions and types. It also points to the first allocated dsentry. Dsentries are ordered in a linked list, so that the collection of dsentries can change dynamically.

Chapter 7

Rerandomization Event

In chapter 1 and chapter 4, a lot of details were revealed on the System Event Framework. Therefore, in this chapter, we only need to discuss a few details of the rerandomization event. The design of the System Event Framework and the rerandomization event are described sufficiently in previous chapters.

7.1 Checking Internal State for Correct State Transfer Conditions

The old process has to respond to the SEF rerandomization message from the reincarnation server. In this response, it indicates whether it is ready for state transfer.

A good example for this state checking is the virtual file system server (VFS). This server often has outstanding file system requests that are processed by mounted file system servers. These requests can contain pointers into the local VFS address space, to which requested data has to be copied from disk.

VFS should not agree to rerandomization if one of the mounted file system servers is about to copy data into the VFS address space. If the address of the destination buffer get randomized, the data copy will corrupt VFS memory.

7.2 Handling Special System Processes

The process manager and the virtual memory manager are involved as much in system process management as in user process management. The process manager has to start

and kill system processes. The virtual memory manager has to allocate and deallocate memory for system processes.

The system process management functions that these two system processes implement are also needed for the rerandomization event. In order to execute a rerandomization event, the reincarnation server has to cooperate with these two processes.

When one of these two processes is scheduled for rerandomization, its services should not be relied on during the time that the old process is blocked, and the new process is not done transferring state.

Therefore, in the next two subsections we discuss features of the rerandomization event design that make sure we can still rerandomize these processes.

7.3 New Process is Started Early

When a rerandomization event is triggered, first, a new process is started. It seems strange that this is done so early; if the old process indicates that it is not ready to rerandomize, the new process has to be killed without being used. However, this way, the process manager can start a new instance of itself beforehand, in case the process manager is the process that is to be rerandomized.

7.4 Pre-allocated Memory

If the virtual memory manager is rerandomized, the new process cannot rely on the old process or itself to allocate memory when necessary. The old process is blocked, and the new process is not fully functional until state transfer is completed. Therefore, a buffer of memory has to be pre-allocated. All the memory allocator functions used by the state transfer routine are wrapped. The wrappers first check if memory is pre-allocated or not. Based on that information, they either use the pre-allocated buffer, or call the actual memory allocator functions.

7.5 Reincarnation Server

The reincarnation service itself is also a special case when it is rerandomized. In order to rerandomize reliably, the kernel will wake the blocking old process when the new process becomes unresponsive, that is, times out.

7.6 Batch Processing

It is also possible to batch multiple system processes into a single rerandomization event. The event will only succeed if all processes are rerandomized successfully. If not, every batched process is rolled back. If the reincarnation server, the virtual memory server or the process manager is part of a rerandomization batch, memory is pre-allocated for each process in that batch.

7.7 Linking Newly Randomized Binaries

At this moment, the LLVM framework and the LLVM passes run on a Linux system. This prevents the desired setup of a MINIX 3 system that can produce its own randomized binaries. Therefore, this framework is considered work in progress.

When the LLVM framework eventually runs on the MINIX 3 system, two separate jobs will have to be scheduled periodically. The first job will add new randomized binaries to the pool of available binaries. The second job will choose a new binary from the pool at random, and trigger an rerandomization event with this binary as the source for the new process.

Chapter 8

State Transfer Routine

In the design chapter, we discussed the general design of the state transfer routine. In this chapter, we will go into the implementation details. We will rely on the metadata structures that are introduced in the chapter on metadata instrumentation.

8.1 SEF message handler

The state transfer routine implements the state transfer phase of the rerandomization event. As we saw in the design chapter, the state transfer routine is located in the newly started process that will have replaced the old process at the end of the rerandomization event.

State transfer is triggered by a SEF message from the reincarnation server. The new process is blocked until it receive this message. At that moment, the old process is already blocked, waiting for a message from the reincarnation server. It's state is ready for state transfer.

The SEF framework automatically redirects the SEF message to the state transfer routine. The message contains the endpoint number of the old process, used for interprocess data copying.

8.2 State Transfer Phases

From the design chapter, we learn that there are three phases of state transfer. First, we need to transfer the remote metadata from the old process. Secondly, the remote metadata has to be paired with the local metadata in the new process. Finally, we use

the metadata to retrieve the regular data from the old process. These three phases are discussed in the next three sections.

8.3 Metadata Transfer

The only pointer that we need into the remote address space is the pointer to the magic_vars struct. From that struct, we can (indirectly) obtain the addresses of all other pieces of metadata. That pointer is obtained with a system call to the kernel. When starting, each system process sends its magic_vars struct pointer to the kernel to be stored, so that a rerandomization event can be executed later on.

We will not go into the details of transferring individual data structures, because it is sufficient to know the general approach of transferring metadata, explained in the design chapter.

It is important to keep a specific order in the metadata transfer. Type metadata is pointed at by sentries, dsentries and functions. It is best to have the types already transferred when those other metadata structures are transferred. That way, we can update their type pointers immediately.

8.4 Metadata Pairing

Remote and local metadata has to be paired. That way, we can find the local counterparts for remote sentries, dsentries and functions. This is needed to transfer data between its remote and local address, and to update pointer values.

Sentries, dsentries and functions are simply matched by their local and remote name.

Types have to be paired, because we need to find the local type when reallocating the remote dsentries, which is discussed in section 4.6.

Types are matched by their name, size, and other characteristics that make them unique. For complex types, we recursively compare the two entire type trees, in order to find out if two types can be paired.

8.5 Data Transfer

In the previous sections, we discussed how the metadata from the old process is transferred and paired with the metadata from the new process. Now, we will discuss how the application data is transferred using this metadata.

8.5.1 Transfer Buffer

In order to transfer all data, the state transfer routine loops through all sentries and dsentries. Before processing a variable by going through the stack of state transfer callbacks, the variable is first transferred to a local buffer.

8.5.2 Pointer Analysis

Now that the data is in a local buffer, the data and the (d)sentry is first analyzed by a special analysis function. When the variable is a pointer, the analysis function automatically finds the new pointer value.

The analysis function does this by searching for a remote sentry, dsentry or function whose data address matches the pointer value that is now in the transfer buffer. When found, we can find the local sentry, dsentry of function that it is paired with. The data address that it stored by this sentry, dsentry or function becomes the new pointer value.

In case a pointer points to a complex variable member, instead of a variable, we also need to take the offsets into the local and remote variable into consideration.

8.5.3 Complex Variables

As discussed in section 4.6, Complex variables are processed top down: First, the state transfer routine checks if one of the callbacks wants to process the whole variable. If not, each of the variable members is recursively processed individually.

The variable is always transferred as a whole into the transfer buffer, no matter if individual type members are processed separately or not. This is a huge optimization when we are dealing with large arrays of small elements.

8.5.3.1 Unions

All unions must be processed with a custom callback. This is the result of the way unions are implemented in C. Unlike some other languages, unions in C do not contain any information about which union member is currently stored by the union. It would be to complex to instrument code and unions with that information. Also, this would introduce an overhead.

Instead, the custom callbacks have to make a decision on how to process the union, based on the internal state.

There are not many unions used in MINIX 3, with the exception of the many occurrences of messages, which are implemented as a union. However, because rerandomization takes place between regular system calls, there are not many messages stored in the internal state of a process that is rerandomized. Also, SEF messages can be ignored, because they never have to be transferred to the new process. In our experience, any remaining messages can be transferred by value, without any modification.

There is a good example of a union used in the data store server. We had to build a custom callback for that union. The custom callback is able to infer which union member is stored purely by inspecting the internals of the union.

8.6 Out-Of-Band Dsentries

There are some dsentries that need special treatment. They are assigned as metadata to special memory regions. For example, DMA buffers get a special dsentry assigned to them.

These dsentries are flagged with a flag that indicates that these regions do not have to be allocated, and that they don't have to be moved for randomization. Basically, the dsentry metadata is transferred to the new process, and then only added to the local dsentries. This way, these special dsentries, called out-of-band dsentries are passed on between rerandomizations.

The memory pages in the virtual memory manager are another example of out-of-band dsentries.

8.7 Type Name Indexing

Many custom state transfer callbacks check if they can process a type by comparing its name to a constant string. This means that for possibly millions of variables and variable members (think of large arrays), a call to *strcmp()* is made by each of those callbacks. This has a big performance impact.

In order to avoid that many *strcmp()* calls, the programmer can register the type name of all types that he wants to have processed by his custom callbacks. This is done before state transfer. He also has to flag the callbacks that compare by type name.

During metadata transfer all types with matching names are flagged. Note that the number of types is much smaller than the number of variable (members). This means that *strcmp()* is called only a few times.

The state transfer routine now only has to call the flagged callbacks when the currently processed type is also flagged. This way, the number of *strcmp()* calls is greatly reduced.

Chapter 9

Evaluation

9.1 Performance

In order to test our implementation, we instrumented twenty system processes (7 drivers and 13 servers). All performance tests were executed on a PC system with a 12-core 1.9Ghz AMD Opteron Magny-Cours processor and 4GB of RAM. All tests were executed three times and averaged.

For the performance evaluation, we used the C programs in the SPEC CPU 2006 benchmark suite as a CPU-intensive workload. We also wanted to have a system call-intensive workload, for which we composed a devtools benchmark script that contained the following tasks: compiling (make world), find, grep, and copying and deleting files.

A user who chooses to protect his system with ASLR will probably choose to give the applications the same protection, in order to avoid weak spots in the system. Therefore, we made an effort to also instrument the benchmarks. In each test, the SPEC benchmarks are instrumented with the same compile and link time options. The devtools benchmark was not instrumented, as it would require a complete overhaul of the entire build system.

9.1.1 Instrumentation Performance Overhead

First, we want to measure the performance overhead that is generated by the ASLR and metadata instrumentation that is used for our framework. This test does not include the rerandomization event in its execution.

We measured the execution times of the benchmarks for three scenarios: no instrumentation, metadata instrumentation, and metadata+ASLR instrumentation. We normalized

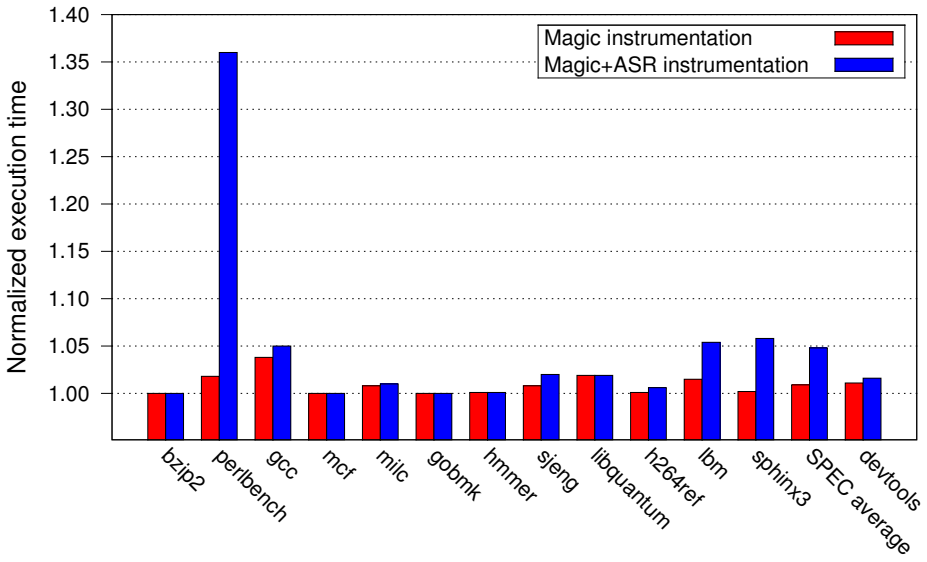

FIGURE 9.1: Execution time of the SPEC CPU 2006 benchmarks and our devtools benchmark normalized against the baseline (no OS/benchmark instrumentation).

benchmark	**metadata instr.**	**metadata+ASLR instr.**
SPEC	0.9%	4.8%
SPEC w/o Perl	0.9%	1.9%
SPEC	1.1%	1.6%

TABLE 9.1: Average Instrumentation Overhead

results from the tests with instrumentation against the results from the test without instrumentation. The results are shown in figure Figure 9.1.

Table 9.1 shows the average overheads derived from these results. The high ASLR overhead for SPEC was caused by Perl. Therefore, we also provide statistics that exclude the Perl benchmark. Excluding Perl causes the magic+ASLR performance overhead to drop from 4.8% to 1.9%/. With a little debugging of the Perl benchmark, we found out that it performs a large amount of small heap allocations during initialization. This is caused by an allocation pool that is implemented in Perl.

When ignoring the outlier results from Perl, we can conclude that our instrumentation is barely noticeable. An alternative, fine-grained implementation, described in [6] gets an average run time overhead of 11%, when instrumenting only the application layer.

9.1.2 Rerandomization Time

Table 9.2 reports statistics on the duration of a rerandomization event for a single system process. Because there is a difference in performance, we also show the differences

	all	servers	drivers
Average	272 ms	327 ms	162 ms
Median	194 ms	262 ms	163 ms
Max	633 ms	633 ms	197 ms

TABLE 9.2: Rerandomization Times

between servers and drivers. MINIX 3 servers tend to have a more complex internal state than drivers, which explains higher rerandomization times for servers.

Unfortunately, there is no other existing live rerandomization implementation that we can compare to.

9.1.3 Rerandomization Performance Overhead

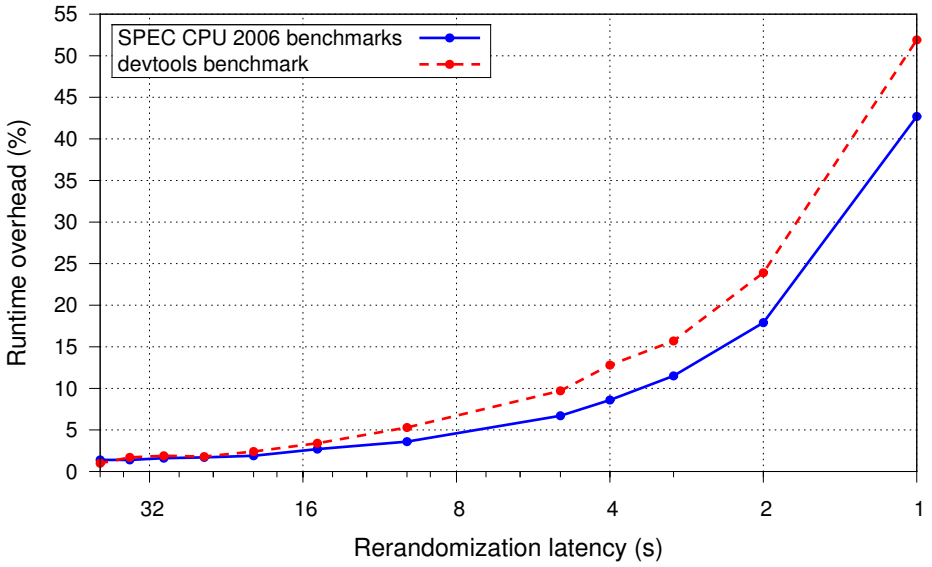

FIGURE 9.2: Run-time overhead against periodic rerandomization latency.

The rerandomization time statistics are useful as an indication for the impact of rerandomization on system process availability. However, this does not yet illustrate the performance impact of rerandomization. In order to measure this impact, we execute the benchmarks again. Only this time, we trigger a rerandomization event at the end of every predetermined time interval. Each time interval, one system process is rerandomized. For fairness, the system processes are chosen round robin. This gives us the runtime overhead for that specific time interval. We repeat this for a set of different time intervals, so that we can plot the runtime overhead against different rerandomization intervals. The results are in figure Figure 9.2. The devtools benchmarks results are plotted separately from the SPEC benchmark results. This shows that there is a higher

performance impact for devtools. This is understandable, because that benchmark is system call intensive; the benchmark will probably be blocked often, because a system call is sent to a system process that is currently randomizing.

The runtime overheads are 42.7% for SPEC and 51.9% for devtools at a one second rerandomization interval. This is clearly not acceptable in general. However, with an interval of 20 seconds or more, randomization has an acceptable runtime overhead. It seems possible to choose a randomization interval that does not give a window for brute force attacks, and yet has an acceptable performance overhead.

9.2 Memory usage

For testing, we chose a maximum random padding size that increased memory usage of each state object by 0-30%, similar to the default values suggested in [6].

On average, rerandomization instrumentation introduced 16.1% state overhead. This is the extra memory that is used to store metadata, with respect to the memory that is necessary to store all regular data.

Our implementation results in a 14.6% overall memory overhead. This is the state overhead that we mentioned above, together with the extra space needed to hold migration code, with respect to the original memory footprint.

Chapter 10

Conclusion

ASLR is a comprehensive approach to counter memory error exploits. Until now, no full-fledged solution targeted operating system security. Our Address Space Layout Randomization (ASLR) design is the first to target operating systems with fine-grained randomization.

Correctness is safeguarded by using compiler passes to instrument binaries, instead of binary rewriting techniques. The performance overhead of our implementation compares favorably to similar fine-grained application-level implementations.

Periodic live rerandomization, used to counter brute force attacks, is central to our design. Again, the compiler safeguards correct state management. Also, the design of the framework makes it possible to isolate and rollback any occurring errors during a rerandomization event. Individual system processes are rerandomized without noticeable interruption.

Our design depends on a microkernel design, and cannot be applied to monolithic kernels. However, the application level is in some ways similar to the system process level in MINIX 3. Our design could be adapted to the application level, and individual ASLR features can be integrated into other ASLR implementations.

Bibliography

[1] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of the 21st ACM Symp. on Oper. Systems Prin.*, pages 335–350, 2007.

[2] Heng Yin, Pongsin Poosankam, Steve Hanna, and Dawn Song. HookScout: proactive binary-centric hook detection. In *Proc. of the 7th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20, 2010.

[3] Nick L. Petroni,Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proc. of the 14th ACM Conf. on Computer and Commun. Security*, pages 103–115, 2007. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315260. URL http://doi.acm.org/10.1145/1315245.1315260.

[4] Crispin Cowan, Coltan Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattle, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the Seventh USENIX Security Symp.*, page 5, 1998.

[5] Sandeep Bhatkar, Daniel C DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proc. of the 12th USENIX Security Symp.*, page 8, 2003. URL http://portal.acm.org/citation.cfm?id=1251353.1251361.

[6] Sandeep Bhatkar, R. Sekar, and Daniel C DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of the 14th USENIX Security Symp.*, page 17, 2005. URL http://portal.acm.org/citation.cfm?id=1251398.1251415.

[7] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): towards Fine-Grained randomization of commodity software. In *Proc. of the 22nd Annual Computer Security Appl. Conf.*, pages 339–348, 2006.

[8] PaX Team. Overall description of the PaX project. http://pax.grsecurity.net/docs/pax.txt, 2008. URL http://pax.grsecurity.net/docs/pax.txt.

[9] L. Xun. A linux executable editing library, 1999. URL http://www.geocities.com/fasterlu/leel.htm.

[10] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM Conf. on Computer and Commun. Security*, pages 298–307, 2004.

[11] Minix 3 faq, 2012. URL http://http://wiki.minix3.org/en/FAQ.

[12] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5. doi: 10.1145/268998.266660. URL http://doi.acm.org/10.1145/268998.266660.