A Tale of Two Kernels: Towards Ending Kernel Hardening Wars with Split Kernel

Anil Kurmus
IBM Research – Zurich, Switzerland
kur@zurich.ibm.com

Robby Zippel
IBM Research – Zurich, Switzerland
zip@zurich.ibm.com

ABSTRACT

Software security practitioners are often torn between choosing performance or security. In particular, OS kernels are sensitive to the smallest performance regressions. This makes it difficult to develop innovative kernel hardening mechanisms: they may inevitably incur some run-time performance overhead. Here, we propose building each kernel function with and without hardening, within a single *split kernel*. In particular, this allows trusted processes to be run under unmodified kernel code, while system calls of untrusted processes are directed to the hardened kernel code. We show such trusted processes run with no overhead when compared to an unmodified kernel. This allows deferring the decision of making use of hardening to the run-time. This means kernel distributors, system administrators and users can selectively enable hardening according to their needs: we give examples of such cases. Although this approach cannot be directly applied to arbitrary kernel hardening mechanisms, we show cases where it can. Finally, our implementation in the Linux kernel requires few changes to the kernel sources and no application source changes. Thus, it is both maintainable and easy to use.

1. INTRODUCTION

It is no longer necessary to motivate the need for improved OS kernel self-protection. Kernel memory corruption vulnerabilities are routinely used for privilege escalation: To escape sandboxes, as in the recent Chrome sandbox escape on Linux [15]; To bypass code signing and secure boot, as in various "jailbreaks" on iOS; For remote kernel exploitation, as in the ROSE and SCTP exploits on Linux (CVE-2011-1493, CVE-2009-0065), and the kernel TrueType Font vulnerability on Windows (CVE-2011-3402).

Following the success of user-space exploit mitigations, kernel-specific *hardening* techniques were proposed [10, 16, 18, 19, 22, 25, 35]. Typically, these techniques render vulnerabilities either impossible to exploit, or make exploitation significantly more difficult.

Kernel hardening can inherently incur noticeable overhead, despite having clear security benefits. For instance, a simple approach such as clearing the kernel stack at each stack frame allocation prevents uninitialized kernel stack variable-related vulnerabilities, but its overhead is high.

To the security-conscious, performance overhead is an acceptable trade-off for improved security. And, in some use cases, security may prevail in practice (e.g., "free shell" providers, cloud providers). However, kernel developers are often performance-conscious and optimize the kernel as much as possible. Indeed, it is considered that even a small overhead in kernel code tends to add up and degrade system performance or power consumption noticeably, which in turn impacts the vast majority of its users.

However, in a large number of use cases and threat models, kernel hardening is only needed when kernel code is running on behalf of specific processes or hardware. In the case of a sandbox escape through a kernel privilege escalation exploit, the attacker will be exploiting kernel code while running in the context of a sandboxed process. Hence, there is an opportunity to take additional preventive measures in the form of kernel hardening (such as clearing each kernel stack frame) when system calls originate from a sandboxed process. In contrast, it is not beneficial to perform these extra operations when system calls originate from a trusted process (e.g., running as *root*).

In addition, the overhead of kernel hardening can vary greatly depending on workloads. Although this overhead would be acceptable on workloads involving low kernel processing, because existing mechanisms cannot be disabled at run-time, kernel distributors have to race to the bottom and disable hardening due to other workloads which would be greatly impacted by kernel hardening.

In this paper, we modify the Linux kernel build process to produce a SPLIT KERNEL that can be run either in *hardened* or *base* mode. Run-time policies *bind* a particular process, system-call, or hardware device to either mode, i.e., they specify whether kernel code running on behalf of these entities will be hardened.

We achieve this by creating a copy of each kernel function, and leverage the flexibility provided by Executable and Linkable Format (ELF) symbols and relocations to ensure all function symbols references are renamed. Only functions are copied while keeping the same references to data locations. We statically instrument all indirect function calls performed in hardened mode to ensure that the kernel remains in hardened mode during the duration of a kernel execution thread, even in the presence of function pointers pointing to base kernel functions.

Indeed, SPLIT KERNEL is designed to have no significant overhead when running in base mode, when compared to an unmodified kernel. A naive implementation would instrument (statically) each unmodified kernel function and redirect execution to a hardened version by branching on a flag of the current process: this would cause significant overhead when operating in base mode. Instead, we build the hardened set of functions such that, if execution is

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2957-6/14/11 ...$15.00. http://dx.doi.org/10.1145/2660267.2660331
.

within any hardened function, it can only transfer control to a hardened function. Hence, we only need to branch once per execution thread (e.g., on system call entry for system calls). This makes it possible to remain with a largely unmodified base kernel code.

Three distinct hardening features are added into the hardened version of each function, by making use of assembly rewriting as an assembler pre-processor step during kernel compilation. These mechanisms (stack exhaustion protection, stack zeroing, function pointer protection) can prevent or mitigate numerous attacks in practice. However, we implement these mechanisms primarily as a demonstration of the advantages of using SPLIT KERNEL when in the presence of a kernel hardening mechanism that benefits security but incurs significant performance costs.

Our evaluation shows a vanilla kernel hardened with these protection mechanisms will incur significant overhead: up to 200% in micro-benchmarks, and ranging from 3% to 60% on real-world workloads. Results show that, under SPLIT KERNEL, real-world benchmark overhead can be significantly improved (e.g., 1% instead of 33% overhead with OpenSSH), because the hardened code is only run when necessary.

More importantly, SPLIT KERNEL defers the decision of which processes should be "running a hardened kernel" to the run-time. System administrators, distribution providers, or users can choose, depending on the threat model, which processes/network interfaces should be used with a hardened kernel, if any. Similarly, one may also decide to run all non-kernel-intensive workloads (such as web browsing) on the hardened kernel, as the overhead would be negligible. Finally, on all benchmarks, running SPLIT KERNEL in base mode incurs no significant overhead over a vanilla kernel.

The main contributions of this paper are summarized as follows.

- We propose SPLIT KERNEL, a build-time, extensible, operating system enhancement tool allowing to run some processes (or hardware) under *hardened kernel* code, while simultaneously running others on *base kernel* code. This allows to select at run-time when the hardened kernel code should be used. As a consequence, it greatly improves overall system performance. In the most extreme case, SPLIT KERNEL allows to opt-out of kernel hardening at run-time without performance costs when compared to running an unmodified kernel.

- We implement SPLIT KERNEL on a recent Linux kernel, and describe challenges we have encountered. We port the implementation to two distinct Linux variants and hardware (Ubuntu on an x86-64 server, OpenWrt on a MIPS32-based home router).

- We implement three assembly-rewrite-based kernel hardening mechanisms. We evaluate results on different real-world use cases and benchmarks.

2. SPLIT KERNEL

We start with the high-level design and goals of SPLIT KERNEL. We then describe our modifications to a standard build system to obtain *split objects* from source files, i.e., objects which contain two versions of the code, with the data being shared. These modifications are largely generic to any ELF-based build system, but we also describe Linux-kernel-specific changes that need to be made. Next, we define *bindings* that we implemented to enable the use of hardened kernel functions dynamically at run-time (e.g., to configure which process will run in hardened mode). Finally, we showcase three kernel hardening mechanisms we have implemented with SPLIT KERNEL.

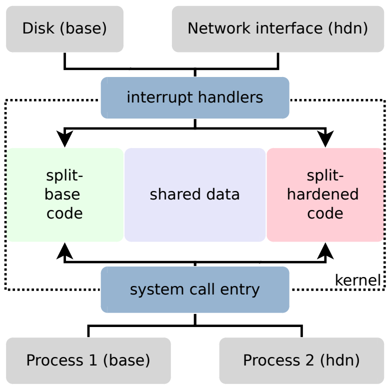

Figure 1: A SPLIT KERNEL at run-time. Process 2 and the network interface are set to use hardened kernel code.

2.1 Overview and Goals

SPLIT KERNEL modifies the kernel's build process to generate two copies of each kernel function: a hardened version which is compiled with additional processing to add kernel hardening code, and a mostly unmodified, base version.

To distinguish between a base kernel (e.g., a Linux distribution or vanilla kernel), a fully-hardened kernel, and SPLIT KERNEL, we refer to the hardened version of the SPLIT KERNEL code as the *split-hardened* code in the remainder of this paper. Similarly, the largely unmodified SPLIT KERNEL code is referred to as *split-base*.

Some processes are solely restricted to use the split-hardened kernel code (see Figure 1). We say they are *bound to hardened mode*. A similar restriction is also applied to hardware devices: for instance, the kernel can run in hardened mode when parsing packets on behalf of a network device. During the kernel build, the data sections are not copied or modified: they remain unchanged when compared to a base kernel. In particular, the data relocations in the split-base and split-hardened code must refer to the same data.

Hence, there is only one kernel running: no security isolation exists between split-base and split-hardened kernel code, and if the kernel is compromised in either of these execution contexts, it is compromised in both. However, compromising it while running in split-hardened mode is more difficult than in split-base mode.

As such, the threat models in which SPLIT KERNEL can be used should essentially assume that the attacker is restricted to using only the split-hardened context. We will see more detailed threat models in Section 3 for each use case, but this is essentially a reasonable assumption. For instance, in the case of a sandboxed process, if the process is bound by the kernel to execute in split-hardened context, the attacker needs to first compromise another process that executes in split-base mode. However, in doing so, the attacker needs to escape the sandbox first, which is typically their initial motivation for using a kernel exploit [15].

SPLIT KERNEL is designed with the following **goals** in mind:

No split-base overhead. Because SPLIT KERNEL aims to provide the choice between performance and security, we need the performance of running a split-base kernel to closely match that of a base kernel under most common workloads. This translates into as little modifications to split-base code as possible: for instance it would not be acceptable to instrument every split-base function.

No control-flow from split-hardened to split-base. Once the kernel control-flow has been bound to the split-hardened code (e.g., during system call entry) it should never accidentally switch to split-base code. Failing this would mean that under some code paths the

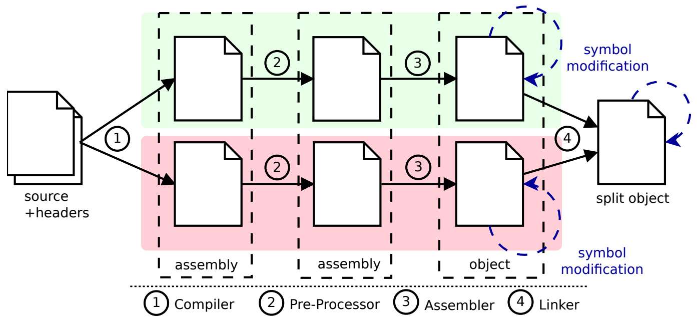

Figure 2: Split build of a translation unit.

kernel code that is executed would not be hardened. This would mean vulnerabilities in kernel code in such code paths would remain as exploitable as in the base kernel. The converse (accidentally switching from split-base to split-hardened code) is less of a problem because it would only impact performance (and this is tolerable if it happens in rare cases).

Run-time configurability. We need to be able to decide dynamically which entities (e.g., processes, hardware-triggered interrupts) will be bound to split-hardened kernel code.

Maintainability. We do not want to perform intrusive modifications to the kernel sources, as that would hinder maintainability of the kernel code. Indeed, the Linux kernel is updated at a very fast pace, which would make our approach quickly obsolete if it depended on many patches to the kernel sources. In addition, since the kernel is written in C and assembly language (specific to each supported architecture), our changes must apply equally well to code generated from C or assembler source files.

2.2 Split Kernel Builds: Split Objects

SPLIT KERNEL modifies Linux kernel builds mostly at the level of the compilation of individual translation units (i.e., compilation of "atomic" object code, usually from a single source file and multiple header files). Figure 2 summarizes these modifications. We add an assembler pre-processing step which applies to both the split-base object and split-hardened object. Then the split-base, and split-hardened undergo symbol modifications in preparation for a linking step. After linking, we obtain a single split-object containing both versions (split-base and split-hardened) of each function. These split-objects can then be used transparently by the existing linking phase of the kernel build. We then obtain the final kernel binary and loadable kernel module (LKM) binaries for SPLIT KERNEL.

The goals of these build modifications are: (*a*) creating two copies of each kernel function which also reference the same data locations (*b*) when in split-hardened code, restricting control-flow to the split-hardened code, i.e. fulfill the second goal described in Section 2.1.

Symbol modification. We leverage the flexibility provided by ELF symbols and relocations to achieve most of these two goals.

ELF symbols can either be local (i.e., scoped to a translation unit), global (i.e., they may be referred from another translation unit, and are unique), or weak (i.e., they will be ignored and superseded if another symbol with the same name exists). They can also correspond to code (functions or other executable locations), or objects (i.e., read-only or read-write data). Relocations instruct the linker (or loader) where and how to fill in code and data addresses once symbols are resolved (i.e., their addresses are known).

As shown in Figure 2, we first compile each translation unit two times, and obtain two object files. For the split-hardened code: (*a*) we rename all code symbols by adding an `__hdn` suffix to their names, (*b*) we temporarily globalize all local data symbols, (*c*) we weaken all data symbols. For the split-base kernel, we only need to make all data symbols temporarily global.

Then, we link the split-hardened and split-base objects. The linker ignores and replaces all split-hardened data symbols, because they are superseded by the non-weakened split-base data symbols. However, because of their distinct names, the split-hardened and split-base functions co-exist in the split object. Because the split-hardened code symbols have all been renamed, any direct intra-split-object function call originating from a split-hardened function now targets a split-hardened function.

Finally, the last symbol modification step makes all data symbols that were made temporarily global once again local.

These steps are not sufficient to attain our goals. Mainly, we still need to cope with: (*a*) indirect calls (through function pointers), (*b*) calls to external functions (which are undefined once the split object is obtained), (*c*) relocations which reference section symbols instead of data symbols.

Assembler pre-processing. The pre-processor is comprised of two halves: the first includes hardening features. This only applies to the split-hardened build, and we detail this step in Section 2.4. The second, that we detail here, performs indirect call instrumentation to keep run-time control-flow contained within split-hardened code. It also helps the symbol modification phase with respect to undefined functions.

Indeed, function pointers are stored in the data sections which are shared by both split-base and split-hardened code. By default, these point to split-base functions. Hence, any function pointer dereference would bring execution back in split-base mode, which contradicts our goals. Therefore, we rewrite indirect calls in split-hardened assembly to first lookup the corresponding split-hardened function pointer.

Although the split-hardened code can tolerate some overhead, this lookup needs to be efficient because function pointers are heavily used in large code bases such as the Linux kernel. To achieve this goal, we prefix an *alternative function address* to the prologue of each kernel function, both in split-base and split-hardened code. Then, instead of dereferencing function pointers directly, indirect calls dereference the alternative function address, which points to the split-hardened version of the function it precedes.

Listing 1: Modification to the declaration of a base function (line 1 will contain the address of `func__hdn`).

```
1+ .quad   func__hdn
2   .type   func, @function
3  func:
4 ...
```

Listing 2: Indirect call instrumentation to use an alternative function addresses.

```
1-    call   *0x10(%rax)
2+    movq   0x10(%rax), %rax
3+    movq   -0x8(%rax), %rax
4+    call   *%rax
```

This is implemented by adding a symbol address relocation, in the code section, right before the prologue of each function (see Listing 1; note that all listings are *unified diffs* of GAS-syntax x86-64 assembly). In Listing 2, line 2 loads the address of `func`, and line 3 loads the address of `func__hdn` into register `%rax`. By merely adding one memory reference, this makes the lookup for the indirect functions efficient. (In addition to this, in the case of local functions, we need to make use of the temporary globalization explained previously, except that this time it applies to code symbols.)

Another benefit of the assembler pre-processing is the ability to easily identify function calls to deal with undefined symbols during split-object generation. The pre-processor makes a list of all function symbols that are called, and this list is intersected, in the symbol modification phase, with undefined symbols. Hence we obtain a list of references to external functions, which are then renamed in the split-hardened symbol modification phase.

Section-relative relocations. For local symbols, by default, the GNU assembler (GAS) produces relocations that are relative to a section symbol. For uninitialized data for instance, the relocation might be of the form `.bss+8`, indicating that the value computed by the loader should be the address of the `.bss` section, incremented by an 8 byte addend. Therefore, the data symbol weakening approach previously explained would not work for such local data, and the corresponding local data symbol would be allocated at two different locations in the `.bss` section.

To prevent such cases, we patch GAS to emit relocations that are data-symbol-based, also for local data symbols. This results in an increase of the number of symbols in the final kernel image. However, in practice, this only impacts linking and kernel load time.

Object metadata for LKM support. The Linux kernel build creates a number of special ELF sections in each object file. `ksymtab` and `kstrtab` are used for locating so-called exported symbols. The kernel exports functions and data for use by LKMs: non-exported should never be used in kernel modules. `kstrtab` is simply an array of strings, whereas `ksymtab` is an array of (symbol address, `kstrtab` address) couples.

Because these are created using GCC directives and CPP macros, the symbol modification step is "too low-level" to automatically trigger a modification of the corresponding strings for split-hardened exported functions. This causes a problem when loading LKMs. To fix this, the assembler pre-processor detects `kstrtab` section strings and appends the `__hdn` suffix to each string. In addition, we also need to rename `ksymtab`-related section names. (The explanation in the previous paragraph is in fact simplified: each object initially contains multiple sections of the format `__ksymtab+symbol_name` for each exported symbol, they are only merged into a single section when linking the kernel image). This is also part of the symbol modification step.

We made a similar modification for `kcrctab`, which is used for LKM module version information.

Init and exit sections discarded (optimization). Initialization code used in the Linux kernel is stored in a separate section. This is to optimize the size of the kernel: after initialization completes, this section is unmapped from memory. As the initialization must only be executed once, the linker creating the split object simply discards init (and exit) sections from the split-hardened object (this is implemented via a custom linker script). Note that this also applies to LKMs.

Code segregation (optimization). Because SPLIT KERNEL builds double kernel code size (without accounting for code increase due to hardening), we optimize the code layout. This consists of a simple kernel linker script modification to ensure split-base code is grouped together in the `.text` section, and followed by the remaining split-hardened code. Without this optimization, when executing split-base code, split-hardened code could be fetched (unnecessarily) into the CPU instruction cache due to spatial locality.

2.3 Bindings

We now explain the modifications we have made to provide users and administrators with the option of making use of the split-hardened kernel code. These modifications are performed directly on the kernel source files (as opposed to the kernel build system as in Section 2.2).

System call entry modifications. We have shown the SPLIT KERNEL build will ensure that the kernel will remain in split-hardened mode once it starts executing any split-hardened function. Hence, to switch to the split-hardened kernel code from the system call interface, it is sufficient to call the split-hardened version of each system call.

This requires modifying the system call entry code. Addresses of system calls are stored in the system call table. The entry code prepares registers and calls an offset in the system call table. The offset corresponds to the system call number passed by the process requesting the system call.

For the split-hardened system calls, we add a second system call table. It contains addresses corresponding to the hardened counterpart of split-base system calls, in the same order as the split-base system call table.

Listing 3: Modifications for one system call entry point

```
 1 system_call_fastpath:
 2     cmpq $__NR_syscall_max,%rax
 3     ja badsys
 4+    movq %gs:current_task, %rcx
 5+    movq SPLITMODE-TASKOFFSET(%rcx), %rcx
 6+    cmpq split_val, %rcx
 7     movq %r10,%rcx
 8-    call *sys_call_table(,%rax,8)
 9+    jne hdn_mode_syscall_fastpath
10+    call *sys_call_table(,%rax,8)
11+    jmp after_sys_call_fastpath
12+hdn_mode_syscall_fastpath:
13+    call *sys_call_table_hdn(,%rax,8)
14+after_sys_call_fastpath:
15     movq %rax,RAX-ARGOFFSET(%rsp)
```

As shown in Listing 3, we check that a value in `task_struct` is equal to a non-zero `split_val` value (lines 4-6). If the value matches, we execute base system calls (line 10). If it does not, we execute hardened system calls (line 13).

The `task_struct` structure is essentially created by the kernel for each process and stores process-specific information. We add into this structure a `split_mode` field. For the first process to execute in user-space, `init`, this field is initialized to `split_val`, indicating that it will run in base mode.

As an implementation note, we cover all system call entry points: the Linux x86-64 kernel has support for interrupt-based system calls (`int 0x80`), fast system calls (`syscall`, shown in Listing 3), and 32-bit compatibility mode system calls for 32-bit processes running on a 64-bit kernel. In addition to the system call interface, another kernel entry point for processes are exception handlers. Exceptions can occur on operations such as a division by zero, a page fault, or a general protection fault when accessing privileged registers in user-space. The kernel registers handlers for each of these exceptions. Although this interface exposes far less attack surface to user-space when compared to the system call interface, we also make a modification similar to Listing 3 for exception entry points: hence, we treat exceptions in the same way as system calls.

Process binding. We make use of process boundaries and allow binding a given process to a split-hardened kernel. This means all system calls performed by a process must be routed to the split-hardened system call table, i.e. replace the split mode address entry in `task_struct` must point to the hardened system call table.

Since this is a per-process parameter, we extend the existing `/proc/pid` interface which contains parameters and information on a process with a given process ID (`pid`). We add a `/proc/pid/split_harden` pseudo-file which can only be accessed by the owner of the current process and root. On reading the file, "1" is returned if the process is bound to the split-hardened kernel code, and "0" otherwise. On writing a "1" to this file, the process is bound to the split-hardened mode by the kernel setting its `task_struct` `split_mode` entry to `0`. Because this value is different from `split_val`, the process will execute in hardened mode.

Any other values written to the pseudo-file, in particular "0", has no effect: for security reasons, it is not possible to switch a process back to split-base mode. Similarly, the hardening binding is left unmodified across `fork`, `clone` and `execve` system calls. In particular, the `task_struct` entry is essentially copied on `fork`. Hence, an attacker cannot simply fork or execute another process to escape the binding.

The process binding functionality is particularly useful for developers: e.g., when a sandboxed browser process is run, the developers merely need to write to the pseudo-file, which is a simple change to implement. In fact, this can even be performed by a wrapper shell script, without any modifications to the application.

User ID binding. Process binding can be insufficient for system administrators: typically, they do not have the possibility of modifying programs, and writing to the `/proc/pid/split_harden` pseudo-file (as root) after the target program starts would be prone to race conditions. Indeed, the process may already be compromised, and an attacker may have already executed a vulnerable split-base system call.

Therefore, we provide an additional interface that makes use of existing trust boundaries in the OS. The simplest is the traditional UID (User ID). Indeed, many daemons, such as SSH, change UID to drop privileges. When changing UIDs, e.g., through a `setresuid()` call, We bind to split-hardened mode any process whose owner matches a given set of UIDs

This list of UIDs is shared with user-space through `sysfs`. An administrator can simply set these UIDs by writing to `/sys/kernel /split/hdn_uid_list`.

Interrupt binding (for networking). Attackers can also target the kernel when it is not running on behalf of a user-process (i.e., not in *process context*). In particular, remote attackers can take advantage of the complexity of the kernel network stack. In the past, network drivers as well as network stacks such as SCTP have been prone to remotely exploitable kernel vulnerabilities. Hence, we also provide the possibility of binding, to split-hardened mode, the treatment by the kernel of network packets.

Most recent network drivers in Linux make use of NAPI: the new, fast networking API. Upon packet reception, the device at first raises an interrupt. Then, the interrupt handler schedules a polling kernel thread (a `softirq`) and masks interrupts from the network device. This thread then parses received network packets. The polling thread is scheduled in a loop, parsing a given amount of packets at a time. When no more packets are available, the driver re-enables interrupts and the `softirq` is removed from the scheduling queue.

The initial interrupt handling code does not have much, if any, attack surface: it does not perform any parsing. Hence, it would not be beneficial to bind it to split-hardened mode (although it is possible: we also implemented an additional `sysfs` interface to allow binding specific interrupt handlers, based on their interrupt vector identifier). However, the `softirq` parses network packets. Therefore, we switch to hardened mode only the `softirq` polling function.

We opted to make the configuration of this binding as easy as possible, while maintaining the flexibility of only binding for packets coming from specific network devices. Hence, we use the `/sys/kernel/split /hdn_napi_if_list` pseudo-file to allow the system administrator to list all network devices (such as `wlan0`) for which incoming network packet parsing should be done in the split-hardened kernel.

Upon writing to this file, the kernel goes through the list of registered NAPI polling functions. If any polling function pointer corresponds to a device with the given interface name, it is replaced with the split-hardened function pointer. When the polling thread is subsequently scheduled, it will execute in hardened mode.

2.4 Kernel Hardening

We now detail three hardening mechanisms that we implement and that are used when the kernel operates in hardened mode. These mechanisms serve as a demonstration of the advantages and limitations of SPLIT KERNEL. Each mechanism is implemented in the assembly pre-processor, essentially rewriting parts of the assembly. The assembly input to the pre-processor is either that obtained after compiling a translation unit (here, using GCC), or simply a pure-assembler file.

For each hardening mechanism, we briefly describe the classes of vulnerabilities that it mitigates, the general idea of the mitigation, and give relevant details on our implementation. We only provide examples of the instrumented listings in the Annex. Indeed, these hardening mechanisms need many instructions and branches when implemented thoroughly, and their implementation is not the main contribution of this paper. We have deliberately tried to choose costly hardening features to better showcase the benefits of SPLIT KERNEL.

Stack exhaustion checks. Each Linux process has, in addition to its user-space stack, its own kernel stack. This kernel stack is used when the kernel is executing a system call on behalf of the process. It is limited to a pre-defined size (8 KB on most modern Linux kernels). In reality, the kernel also keeps process-specific information, `thread_info`, at the end (lowest addresses) of this stack, hence the stack size available to the kernel when executing a system call is slightly less than 8 KB. If the call stack exceeds the available size, (because of a particularly long call stack or because of a function using a large stack frame), `thread_info` can be overwritten by the kernel, possibly with attacker-controlled content. Since `thread_info` contains sensitive data such as function pointers, this is an interesting target for attackers. For instance, such a vulnerability [6] was shown to be exploitable for local privilege escalation by Nelson Elhage [12].

The Linux kernel sources include a static binary analysis tool, `checkstack.pl`, that identifies the most stack-consuming kernel functions (for the kernel configuration used for the build). In addition, kernel developers are discouraged from making use of recursive calls or variable-length arrays. A build-time configuration option, `CONFIG_DEBUG_STACKOVERFLOW`, also allows kernel developers to enable kernel stack size checks at run-time. However, these checks only occur when an interrupt preempts the kernel for performance reasons (and, for performance reasons again, this configuration option is in any case disabled in distribution kernels that most users run, as it is intended as a kernel developer debugging feature only). Hence, these measures only alleviate the problem, and kernel stack exhaustion can still occur.

To prevent stack exhaustion and `thread_info` overwrites, the pre-processor instruments each function prologue (i.e., function entry) and stack allocation (stack decrement) instruction. It verifies that the current stack pointer is not less than the top of `thread_info`, plus a *guard zone*. In the common case where a function makes use of stack variables, the check is added after stack allocation is performed (but before writing to the stack). The guard zone (128 bytes) is used as a safety net to prevent `thread_info` overwrites that could occur due to registers that are saved on the stack prior to function calls.

We instrument not only each function prologue, but also each stack decrement instruction. In the common case where these two operation follow each other (prior to a call), the check is done only once. In other cases where stack decrement occurs again, for example when using inner-C-block stack variables, we instrument again. We instrument function prologues because it is possible to also exhaust the stack by merely performing recursive calls with no stack allocation (due to spilled registers). In addition, this instrumentation is not valid for the interrupt stack. Hence, the instrumentation contains a check in case the function is executing in interrupt mode: if so, the check always passes.

Stack clearance. Missing initialization is a widespread C programming error [29], including among Linux kernel programmers [7, 30]. For instance, a data structure can be padded with optional bytes, and these bytes are not zeroed before being copied to user-space or sent over the network. This means that, potentially sensitive data from previous stack-frames, such as cryptographic keys that should remain protected in the kernel can be leaked. Leaking kernel addresses and offsets can also be useful to attackers: for instance, this can defeat randomization-based exploit mitigation techniques [18]. Such vulnerabilities have been exploited in the Linux kernel recently (e.g., CVE-2010-4158).

Alternatively, a function can make use of an uninitialized pointer. Of course, this uninitialized data has in fact the value that was assigned to it in a previous use. For instance, when the uninitialized data is on the stack, it corresponds to data from a previous stack frame of another function. With careful sequencing of functions, an attacker may place attacker-chosen data at the correct location on the stack. When the vulnerable function is executed, it will use the attacker-provided pointer. If it is a function pointer, this directly leads to code execution in kernel mode for the attacker [8].

A simple approach can mitigate most vulnerabilities based on use of uninitialized stack data: zeroing the current stack frame at each function call, after it is allocated (by decrementing the stack pointer on systems where the stack grows towards low addresses). An alternative approach is to zero-out the kernel stack corresponding to the executing process, on each system call. The latter approach provides better performance, but will still allow information leaks or uninitialized data use originating from a function executing within the same system call. In the spirit of SPLIT KERNEL, we have implemented the former, more secure approach; the latter is implemented in the grsecurity-patched Linux kernel [35].

The assembly pre-processor simply identifies a stack decrement assembly instruction. It then inserts instructions zeroing the allocated stack. On x86-64, we make use of the `stosq` instruction to zero eight bytes at a time when possible. Stack allocations can either be static (i.e., compiler-computed) or dynamic (i.e., use of C99 variable-length arrays, or `alloca()`). In the common case where the allocation is static, the size computation is done in the pre-processor. In the dynamic case, the pre-processor adds instructions to compute the adjusted count from the register holding the size.

Function pointer protection. After an arbitrary kernel write vulnerability is found, attackers will often overwrite a function pointer used by the kernel. In case of local privilege escalation exploits, attackers will point the function pointer to attacker-crafted code in user-space. The attacker will then trigger this function (through an appropriate system call) and achieve kernel-mode arbitrary code execution. This is a common pattern in many publicly available kernel privilege escalation exploits.

We add a verification that all function-pointer-dereferencing calls target a valid kernel function (much like a limited form of control flow integrity (CFI) [1]). Because the pre-processor already instruments each indirect function call, this modification is added on top of it. First, to identify valid function pointer targets, we add an 8-byte verification label prior to the prologue of each split-hardened function (right before the 8 bytes of the alternate function address described in Section 2.2, as seen in Listing 4). Then, on each indirect call, we insert a compare instruction that checks that the verification value matches.

Listing 4: Pre-function prologue modifications for a hardened function. Note that `func` will be renamed to `func__hdn` during symbol modification.

```
+ .quad   0x13660e4b12b74208
+ .quad   func__hdn
  .type   func, @function
 func:
```

The verification value is the same for all split-hardened functions. It is generated randomly at build-time, after checking that this byte sequence does not occur in other kernel object files. The value does not need to be secret: an attacker can learn the verification value, and attempt to insert data (either in user-space or in kernel data sections) with an attacker-chosen pointer value followed by the verification value. However, we also check that the verification value is located in the kernel text section, or within a module. These kernel segments are all mapped either read-only or non-executable, hence an attacker cannot bypass this check with the aforementioned technique.

2.5 Implementation Summary

Table 1 summarizes our modifications: (*a*) split builds consist in about 600 lines of architecture-independent additions; (*b*) bindings result in about 1000 lines of kernel source changes, with about 300 lines of architecture-independent changes; (*c*) the three hardening mechanisms and the pre-processor additions for the split builds consists of about 800 lines of Perl code for x86-64.

To demonstrate the generality of SPLIT KERNEL, we port this vanilla-Linux 3.2.48 based implementation to OpenWrt (based on Linux 2.6.39.4). OpenWrt is a Linux kernel distribution for embedded systems and also contains the toolchain necessary to cross-compile and flash a new firmware on routers. Our port to the MIPS32 architecture results in the following modifications: (*a*) split builds only need a different patch for GAS (50 lines), (*b*) bindings incur about 500 lines of kernel source changes, mostly in MIPS system call entries (*c*) we only port function pointer protection as hardening mechanism, it consists in about 250 lines of Perl code.

Finally, for purposes of evaluation and benchmarking, we create a fully hardened kernel. This consists in only keeping the hardening mechanisms: no split builds or bindings (we also remove the alternative function values prior to each function call). As an implementation note, it also requires to add verification values to functions in the init sections: this is not required for SPLIT KERNEL because kernel initialization is performed in base mode.

3. USE CASES AND EVALUATION

3.1 Use Cases

For each use case, we first provide a description, then explain the threat model we consider, and finally describe the corresponding binding configuration we have made.

All three first use cases are run on an x86-64 Intel Xeon E5440 server with 20 GB of RAM running Ubuntu 12.04 with the Linux 3.2.48 vanilla kernel (compiled with Ubuntu configuration options).

In cases where the benchmark involves networking, we consider that attackers may be in a position to exploit a remote kernel vulnerability in either the network driver or the network stack of the

Source file	Modification	Description
`include/asm-generic/vmlinux.lds.h`	4 lines added	Segregate base and hardened code when linking
`scripts/Makefile.build`	147 lines added/modified	Split build modifications
`scripts/symbol/globalize.sh`	39 lines	Globalize data/relocation symbols
`scripts/symbol/metadata.sh`	109 lines	Section renaming for `ksymtab`-like sections
`scripts/symbol/renamesym.sh`	97 lines	Renaming text symbols
`scripts/symbol/preproc.pl`	64 lines	Pre-processor for build modifications
`scripts/symbol/weaken.sh`	26 lines	Weaken split-hardened data symbols
`scripts/symbol/linker.lds`	72 lines	Linker script for split-object linker
GNU assembler patch	37 lines added	Symbol-relative relocations only
`arch/x86/kernel/entry_64.S`	166 lines modified/added	Split system call entry and exception handlers
`arch/x86/kernel/syscall_64.c`	14 lines added	Additional system call table
`arch/x86/ia32/ia32entry.S`	392 lines (code + data)	32-bit compatibility system-call modifications
`kernel/cred.c`	8 lines added	UID binding: check on UID change
`fs/proc/base.c`	44 lines added	`/proc/pid/` binding
`include/linux/split.h`	1 file added (31 lines)	Header file for SPLIT KERNEL
`include/linux/init_task.h`	2 lines modified/added	`init` starts in split-base mode
`kernel/split/kobject.c`	102 lines	`sysfs` binding interfaces
`kernel/split/hdn.c`	62 lines	Custom error handling functions (for hardening)
`scripts/asrewrite/func_base.pl`	105 lines	Pre-processor for base functions
`scripts/asrewrite/func_hdn.pl`	114 lines	Pre-processor for hardened functions
`scripts/asrewrite/call_hdn.pl`	230 lines	Pre-processor for indirect call instrumentation
`scripts/asrewrite/stack_hdn.pl`	348 lines	Pre-processor for stack clearance and exhaustion

Table 1: Breakdown of our modifications to the Linux kernel builds, sources and the assembler pre-processor.

kernel. Hence, we bind the network interface to hardened mode by simply writing the interface name to `/sys/kernel/split /hdn_napi_if_list`. We perform benchmarks both with this binding enabled and disabled. Indeed, an administrator may decide to disable this binding for improving performance, based on the observation that remote kernel exploits are rare compared to local ones.

OpenSSH. We use the distribution-provided OpenSSH daemon and run two benchmarks. The first measures the remote login wall-clock time (this is run from a client machine with low-latency network access to the server). The second measures the time taken to transfer a 100 MB file over `scp`.

The OpenSSH daemon is privilege separated [31]. Although SSH needs root privileges to spawn user-privileged shells, the part of the authentication process receiving direct input from an unauthenticated remote client is performed in a forked process, after dropping privileges. This unprivileged, sandboxed process runs as user `sshd`. Therefore, it would not make sense to harden the kernel from attacks coming from the main ssh daemon running as `root`. However, it makes sense to bind the unprivileged process to hardened mode in order to prevent an attacker that has gained code execution in the sandboxed process to escape via a kernel exploit.

This requires no changes to the OpenSSH daemon: the admin can simply add a line in system initialization scripts to execute `id -u sshd >> /sys/kernel /split/hdn_uid_list`.

Apache. We set-up an Apache web server in its default settings (however, we disable access logging to prevent filling up storage space during extended benchmarks), and run *ab* (Apache Benchmark) with 100 requesting threads while serving static web page.

We consider an attacker that gained code execution with the privileges of the Apache process (e.g., through a vulnerable PHP script), and attempts to elevate his privileges through a kernel exploit.

Therefore, we bind the apache daemon to split-hardened kernel. We achieve this simply by writing the UID corresponding to Apache to the `/sys/kernel /split/hdn_uid_list` file exposed through sysfs on system initialization.

Kernel	Code Size
Vanilla (Linux 3.2.48)	6475k
– with SPLIT KERNEL	16644k
– with full hardening	10003k
OpenWrt (Linux 2.6.39)	1849k
– with SPLIT KERNEL	3829k

Table 2: Comparison of kernel code sizes at load time (excluding init sections).

Firefox. We use Firefox to browse popular websites (Amazon, BBC, CNN, Craigslist, eBay, Google, MSN, Slashdot, Twitter, YouTube) with the BBench benchmark [20].

We consider an attacker that aims to gain execution of the browser. This can be due, for example, to a Flash plugin exploit. We consider two bindings. The first binds the entire Firefox process to hardened mode. The second binds solely Firefox's flash player process to hardened mode. This requires no changes to Firefox: we simply rename the flash player process and create a shell wrapper that writes to its PID pseudo-file before executing the flash plugin process.

3.2 Evaluation

Micro-benchmarks. We measure performance for system calls with the LMBENCH 3 benchmarking suite. We perform between 50 and 80 runs for each kernel, and calculate 95% confidence intervals. Results are shown in Figure 3, normalized to the base kernel. Split-base mode incurs no measurable overhead: indeed, the only difference in executed code is a branch and memory reference on system call entry, and there are no cache effects due to code segregation. The fully hardened and split-hardened (we set the process performing to hardened mode before starting the benchmark) results are comparable, and both incur high overheads on micro-benchmarks (between 1.2x and 3x slower). This is explained by the significant amount of additional instrumenting code that is executed due to the three hardening mechanisms we selected (average of 55% increase in code size, see Table 2).

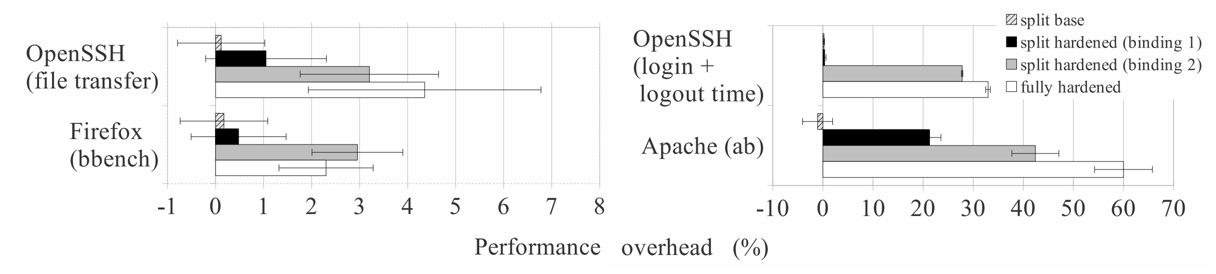

Figure 4: Use-case benchmark results.

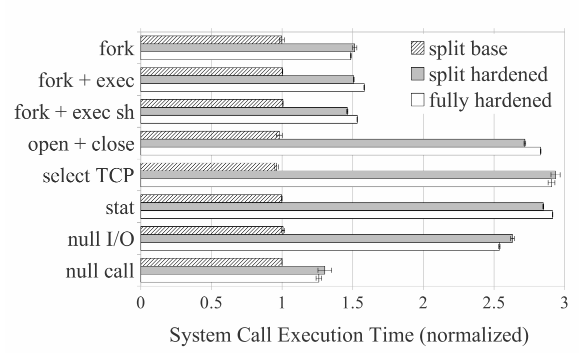

Figure 3: Micro-benchmark results.

Use case benchmarks. We run each macro benchmark at least 50 times, and report 95% confidence intervals.

We consider two binding settings for each SSH and Apache benchmark: *binding 1* is only with the UID sandboxing, *binding 2* is with UID sandboxing and network interface binding. For Firefox, *binding 1* is with flash plugin sandboxing only, while *binding 2* is with full browser sandboxing.

In Figure 4, we observe two groups of results: less than 5% fully-hardened overhead benchmarks on the left, and more than 30% overhead on the right. Indeed, some workloads are more kernel-intesive than others: here, the file transfer and Firefox benchmark spend comparatively more time waiting for I/O, or performing user-space processing, than the SSH login and Apache benchmarks which are more kernel-intensive. Hence, a potential deployment strategy for a performance-conscious system administrator can be to disable hardening only on all kernel-intensive applications.

As predicted by the micro-benchmarks, split-base incurs no significant overhead on any benchmark. Split-hardened benchmarks demonstrate that, depending on the threat-model one adopts, one can modulate performance overhead between split-base results and the fully-hardened results. In particular, on the right, the difference between binding 1 and 2 is only the handling of network `softirq` polling threads in hardened mode. This indicates that most of the overhead incurred by the fully-hardened kernel is in fact in interrupts, especially for the SSH login benchmark. Finally, split-hardened under binding 2 is still significantly faster than the fully hardened kernel. This is due to other kernel execution threads (e.g., the kernel scheduler or kernel threads such as `kflushd` which writes back dirty pages) that are not running in hardened mode for SPLIT KERNEL, but are solicitated heavily for this workload.

One may consider whether switching back and forth between base and hardened kernel code may create a significant performance loss due to cache effects. In fact, this is reflected in the split-hardened benchmarks of Firefox and SSH with binding 1. Indeed, in these benchmarks, some of the processes run in base mode while others in hardened mode. In particular, in the case of SSH with binding 1 the performance is comparable to the split-base performance. This is because the the SSH authentication process (which runs in split-hardened mode) performs two orders of magnitude less system calls that the main SSH process (1K vs. 100K system calls). Because the split-base and split-hardened performance are very close, we conclude that the performance impact of such kernel code-caching effects is insignificant in practice, especially when compared to running in full-hardened mode.

Testing. To test our implementation, we made use of the Linux testing project (LTP) [27], in addition to all aforementioned benchmarks. We did not observe any crashes or errors specific to the SPLIT KERNEL and fully hardened kernels. To test that no control flow from hardened to base occurs, we also collected kernel stack traces from hardened-mode system calls and verified that all called functions are hardened.

3.3 Security Considerations

We discuss here first the ways in which an attacker can potentially bypass SPLIT KERNEL or the implemented hardening mechanisms.

Stack exhaustion checks. The stack exhaustion check covers all possible cases (with the exception of exhaustions of an interrupt stack, however this is not in scope). Indeed, we have covered the case of recursive calls (by instrumentation function prologues, even in the absence of stack allocation), the case of variable length arrays, the case of inner-block local C variables, and register spills on the stack due to stack switches.

Stack clearance. As explained in Section 2.4, for stack clearance, there are two types of vulnerabilities to mitigate: information disclosure, and use of uninitialized data. We zero the stack at each stack allocation. This means that all uninitialized stack variables will be initialized to 0: there can be no information disclosure. However, in the case where the uninitialized stack variable is made use of in the kernel, in some rare cases, this value could be detrimental: e.g., a variable that control whether access control should be performed. In the case of a function pointer, this may result in a NULL-pointer dereference. Although NULL-pointer dereference vulnerabilities were still easily exploitable in 2009 on Linux, the restrictions on mmap made them much less likely to be so (an attacker needs to find, in addition, a way to bypass this mechanism). Note also this is caught by the function pointer protection.

Function pointer protection. The function pointer protection implemented here only considers the case where an attacker has a vulnerability that permits them to overwrite a specific function pointer. Therefore, this protection does not apply in case the attacker has a vulnerability that directly allows him to arbitrarily modify other control data (such as return addresses on the stack) or sensitive non-control data (such as the UID of the current process).

There are a few indirect calls we do not instrument, because we manually identified it to be unnecessary. Some kernel function pointers are declared as constant, and are therefore mapped in a read-only kernel page. For instance, in the system call entry (Listing 3), we do not check before dereferencing in the system call table. However, both system call tables there are read-only: they cannot be overwritten directly by the attacker.

When compared to return-to-user protections [16, 22, 35], this protection has advantages. This protection mitigates to some extent code-injection-based and return-oriented-programming (ROP)-based exploit payloads (but only when the attacker overwrites a function pointer). This is achieved in two steps. First, the check that the address where control is transferred is either in the core kernel text section or in an address mapped for kernel modules guarantees that the target address and the verification value are either in a non-executable or read-only page. A simple check that the target address is within the kernel [16, 22] is not sufficient to achieve this. Indeed, we have verified that the recent kernel we use maps a few kernel pages (such as the BIOS) read-write and executable [25]. This means this protection works even if an attacker could inject code into these pages (potentially through the same, or another vulnerability), and then overwrite the function pointer with the injected code's address. In addition, we essentially implemented a simpler form of CFI by using verification values. This means that an attacker can only transfer control to "valid" function entry points. As a consequence, ROP in its general form becomes impossible (e.g., the attacker cannot make use of a *stack pivot* gadget to make the switch the stack pointer to its ROP payload, because no legitimate function is likely to perform a stack pivot). However, return-to-libc-type attacks (but with kernel functions) may remain possible. Indeed, in contrast to proper CFI [1, 14], we do not perform any control-flow-graph analysis to narrow down possible branch targets at a fine granularity.

The above are standard considerations [37]. We now discuss SPLIT KERNEL-specific weaknesses: we implemented this mechanism also to illustrate limitations of SPLIT KERNEL. The function pointer that is overwritten by the attacker can be in a (writable) data section shared with base kernel code. Note that this is not the case if the function pointer is on the kernel stack, since each process has its own and can only be either in base or hardened mode. In such cases, the attacker-provided address can be dereferenced by another process, in base mode, with no instrumentation. Nevertheless, recall that by assumption, the attacker does not control any base-mode processes. Hence, in such cases, the attacker remains with a challenge: it can redirect the control-flow in kernel mode via a specific function pointer, but does not control either the process address space, the system calls that the process performs, or even the inputs to the system calls.

To generalize, if an attacker is powerful enough to arbitrarily access kernel data shared between hardened and base, hardening mechanisms implemented within SPLIT KERNEL may not be effective. Fortunately, many hardening mechanisms, such as the two others we implement, do not assume such a strong attacker. In such cases, hardening mechanisms implemented with SPLIT KERNEL are as useful as their full-kernel implementations.

Bypassing SPLIT KERNEL. Because there are two sets of kernel functions, we consider whether an attacker that controls an unprivileged process running in hardened mode may instead execute base kernel code, potentially exploiting vulnerabilities without hardening mechanisms. Assume the attacker aims to exploit such a kernel vulnerability to escape from its current security domain (e.g., ssh authentication sandbox). As explained in Section 2.3, a fork or execution of a new process cannot result in a transition to base mode. Similarly, the `sysfs` and `procfs` interfaces cannot be used to transition to base mode. The attacker may attempt to gain code execution in an existing base-mode running process. However, this already corresponds to bypassing the security domain of the current process, and, would often correspond to the end goal of the attacker (e.g., escaping the ssh authentication sandbox, and gaining code execution in the main `sshd` process running as root). Finally, an attacker may exploit a kernel vulnerability to disable SPLIT KERNEL. This kernel vulnerability may either be unexploitable due to the hardening mechanisms implemented in hardened mode, or not. In the latter case, the vulnerability needs to allow the attacker to modify entries in `task_struct` (`split_mode`, more precisely), which contains other sensitive data. Hence, such powerful vulnerabilities would also be exploitable for sandbox escapes in a fully-hardened kernel.

4. DISCUSSION AND FUTURE WORK

Portability across architectures. Parts of our implementation have CPU architecture-dependent components. However, as demonstrated by our effort to port to OpenWrt, porting SPLIT KERNEL builds to other architectures is possible. In particular, bindings affecting architecture-dependent code (such as the system call entry) require extra work. Clearly, compiler hardening mechanism that are inherently portable would not need such adaptation, and one can envision making use of such approaches in the future, also for better optimizing the generated assembly code.

Split builds on other OSes. Our implementation is Linux-specific, but the core idea of SPLIT KERNEL, building two versions of kernel code and deferring the decision of which to use to run-time, can be applied to any OS. In fact, it could also be applied to user-space applications, provided that there are motivations for doing so. In terms of the implementation, only part of SPLIT KERNEL builds is Linux specific: e.g., when dealing with LKMs and discarding `init` sections. The rest of the implementation depends on the ELF format, which is used in most current UNIX-like OSes (OS X and AIX are notable exceptions). For example, a port to a BSD kernel would not require any major overhaul.

Other hardening mechanisms. Not all hardening mechanisms can be used, especially unmodified, with SPLIT KERNEL. As we discuss in Section 3.3, the guarantees of CFI-like mechanisms are weakened because control-data is shared between the base and hardened kernels. However, mechanisms that protect data accesses (such as stack exhaustion and stack clearance) are well adapted. As an example, mechanisms that protect out-of-bounds accesses, such as AddressSanitizer for the Linux kernel [19], could be implemented in future work. However, this may require instrumenting some base code as well: namely for each kernel heap (de-)allocation from the base kernel, to update associated metadata. Because data accesses would not be instrumented in the base kernel and only allocations would, this should limit the impact on overall split-base performance.

Alternative bindings. The bindings we show (process, UID, network interface) can be extended to the many other flavors of process trust boundaries available on Linux. LSM [39] framework-based access control mechanisms such as SELinux [34] provide "security contexts" which are dynamically computed (depending on policies and labels on programs), and it would be possible to extend these frameworks to bind specific security contexts to split-hardened mode. For instance, any process with a given SELinux *type* can be bound to a split-hardened kernel. Closer to the spirit of such access control systems, running in base mode can be a new permission. By default, processes could then run in hardened mode, except when they have the base permission. Finally, SPLIT KERNEL could be used together with an anomaly detection mechanism, such as kRazor [23], to transition execution to split-hardened mode when an unusual kernel function is executed.

5. RELATED WORK

To the best of our knowledge, Split Kernel-modified Linux is the first kernel that can be dynamically configured, at a system-call- or interrupt-granularity, to be run in hardened mode or not. However, it is inspired from and builds on research in areas we describe below.

OS and VMM research. Micro-kernels [5, 13, 26] aim for extensibility and configurability by design: much of the functionality provided in kernel-mode in monolithic OSes is modularized and is run as verified [5] or deprivileged "processes" known as services. In exokernels [13], applications can make use of specialized *library OSes*. Such specialization enables sophisticated optimizations (e.g., different page cache management or thread scheduling algorithms for each process).

Closer to our commodity OS-based approach, Proxos [36] splits the Linux kernel interface in two for a group of private processes: private and traditional system calls. A VMM is used to spawn a private OS, and private processes will see their private system calls routed to the private OS, while its other system calls will be routed to the untrusted commodity OS. Because the two OSes are isolated, compromising the untrusted OS is not sufficient to access private data. In contrast, Split Kernel is in a single protection domain: it does not aim for isolation between split-hardened and split-base code. Rather, it makes the exploitation of vulnerabilities more difficult for a configurable group of processes (or interrupts). One can consider that it trades-off isolation for better performance and maintainability: no latencies due to VM context switches, no dependencies on a VMM or micro-kernel, no modification of applications.

Kernel hardening. Hardening techniques (such as NX, ASLR, SSP, format string and heap hardening, see references in [38]) emerged as a successful counter to memory corruption exploits against privileged processes. Because this made privilege escalation through user-land attacks more challenging, some attackers shifted their focus towards OS kernels which remained easier to exploit. Hence, kernel hardening only recently attracted interest, despite kernel memory corruption vulnerabilities such as unsafe user pointer dereferences being known as early as 1972 [2].

Kemerlis, Portokalidis, and Keromytis [22] discuss return-to-user attacks in depth, and propose instrumentation of indirect branches (using GCC plugins) to prevent them. In contrast to the x86-64 Supervisor Mode Execution Protection (SMEP) [16] feature available since Ivy Bridge CPUs, their approach is easily applicable to other architectures. Grsecurity [35] also provides Linux kernel patches that prevent return-to-user attacks (KERNEXEC) on x86, x86-64 and ARM architectures. In addition, it provides numerous kernel hardening features including: kernel stack randomization at each system call entry, removal of read-write-execute kernel mappings, prevention of unsafe user pointer dereferences (similar to the recent Supervisor Mode Access Protection (SMAP) [10] feature to appear in future x86-64 CPUs), prevention of reference-counting overflows, and, recently, a stack exhaustion prevention feature.

Liakh, Grace, and Jiang [25] analyze the enforcement of the *write exclusive-or execute* regime for Linux kernel pages, and propose fixes for each violation. However, on the 3.2.48 x86-64 vanilla Linux kernel we used, we were able to verify (by performing a page table dump) that writable and executable kernel pages remain. Li et al. [24] take a compiler-based approach to generate a kernel without any return opcodes to mitigate return-oriented programming [32]. Secure Virtual Architecture (SVA) [11] compiles the kernel sources into a safe instruction set architecture using LLVM, which is translated to native instructions by the SVA VM. This provides among other guarantees, a variant of type safety and control flow integrity for the Linux kernel. Giuffrida, Kuijsten, and Tanenbaum [18] proposes randomization and periodic live re-randomization of kernel data structures, kernel stack, static and dynamic data, and basic code blocks to mitigate ROP payloads and the impact of information leakage vulnerabilities. They also leverage LLVM and apply the approach to Minix 3 [21]. AddressSanitizer [33] is a GCC and LLVM extension that instruments memory accesses to detect object-based out-of-bounds reads and writes. It has recently been adapted to work with the Linux kernel and was used to uncover many vulnerabilities [19].

Compiler function cloning. In compiler research and development, function cloning [4, 9, 17] consists of the compiler generating a copy of a function to optimize it. For instance, that copy could be used by some callers after the compiler performs constant propagation on it, while keeping the original version of the function intact for the rest of the callers (because other callers do not call the function with a constant parameter). In existing research, the aim of function cloning is to optimize binaries (e.g., for performance or power-consumption), but not to provide a second set of security hardened functions. This explains the differences between Split Kernel and compiler function cloning. Indeed, Split Kernel does not merely perform function cloning for each function: it also ensures that control flow remains within the base set of functions or the hardened set of functions (through renaming function symbols at compile-time and instrumenting indirect calls).

Kernel live patching. Kernel live-patching techniques [3, 28] have been proposed to prevent rebooting kernels on updates. Because live-patching techniques create a patched copy of kernel functions (and redirect control-flow to the patched version), they bear similarities to the techniques used in Split Kernel builds. In addition, such techniques could be investigated to live-patch a distribution kernel into a split kernel by merely loading a kernel module (without needing to reboot or recompile the kernel).

6. CONCLUSION

This paper challenges the established wisdom that OS kernels are too sensitive to performance and can only include lightweight kernel self-protection mechanisms. We show it is possible (at least in some cases) to have a best-of-both-worlds approach by allowing kernel hardening to be enabled selectively at run-time.

The design of Split Kernel is simple and effective: we provide two versions of the kernel code. By design, we do not instrument any code in the base version of the kernel code by making it the hardened version's responsibility to ensure control is not transferred from hardened code to base.

We implement on Linux x86-64, and then port our implementation to an embedded router. We implement three distinct hardening mechanisms. We implement a fully-hardened kernel without Split Kernel, but with all hardening mechanisms. We consider three use cases with different threat models, and evaluate performance. Our implementations are fully functional, and the resulting kernels work with any existing application.

Our results demonstrate Split Kernel is a useful approach: it can be used in practice to put the decision of using hardening in the hands of OS distributors, system administrators or users. Under different real-world workloads, Split Kernel running in base mode incurs no significant overhead. We also show that in some cases the split-hardened kernel code has negligible overhead, unlike a fully-hardened kernel.

Finally, we hope Split Kernel will pave the way for new kernel self-protection mechanisms that may have been disregarded for being unlikely to be performant and practical.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow Integrity Principles, Implementations, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (2009), 4:1–4:40. ISSN: 1094-9224.

[2] James P Anderson. *Computer Security Technology Planning Study. Volume 2*. Technical report. DTIC Document, 1972.

[3] Jeff Arnold and M. Frans Kaashoek. "Ksplice: Automatic Rebootless Kernel Updates". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. 2009, pages 187–198. ISBN: 978-1-60558-482-9.

[4] Andrew Ayers, Richard Schooler, and Robert Gottlieb. "Aggressive Inlining". In: *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*. PLDI '97. 1997, pages 134–145. ISBN: 0-89791-907-6.

[5] Brian N. Bershad, Craig Chambers, Susan J. Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. "SPIN — An Extensible Microkernel for Application-specific Operating System Services". In: *ACM SIGOPS European Workshop*. 1994, pages 68–71.

[6] Phil Blundell. *Econet: fix CVE-2010-3848*. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=a27e13d370415add34879`.

[7] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. "Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys '11. 2011, 5:1–5:5. ISBN: 978-1-4503-1179-3.

[8] Kees Cook. *Kernel exploitation via uninitialized stack*. DefCon 19 `https://www.defcon.org/images/defcon-19/dc-19-presentations/Cook/DEFCON-19-Cook-Kernel-Exploitation.pdf`.

[9] Keith D Cooper, Mary W Hall, and Ken Kennedy. "A Methodology for Procedure Cloning". In: *Comput. Lang.* 19.2 (1993), pages 105–117. ISSN: 0096-0551.

[10] Intel Corp. *Intel Architecture Instruction Set Extensions Programming Reference*. `http://software.intel.com/sites/default/files/319433-014.pdf`. 2012.

[11] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. "Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*. (Stevenson, WA, USA). 2007, pages 351–366. ISBN: 978-1-59593-591-5.

[12] Nelson Elhage. *Econet local privilege escalation*. `http://www.exploit-db.com/exploits/15704/`.

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. 1995, pages 251–266.

[14] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. "XFI: Software Guards for System Address Spaces". In: *7th Symposium on Operating System Design and Implementation (OSDI '06)*. (Seattle, WA, USA). 2006, pages 75–88. ISBN: 1-931971-47-1.

[15] Chris Evans. *Pwnium 3 and Pwn2Own Results*. `http://blog.chromium.org/2013/03/pwnium-3-and-pwn2own-results.html`. 2012.

[16] Stephen Fischer. *Supervisor Mode Execution Protection*. NSA Trusted Computing Conference and Exposition. `https://www.ncsi.com/nsatc11/presentations/wednesday/emerging_technologies/fischer.pdf`. 2011.

[17] Grigori Fursin, Cupertino Miranda, Sebastian Pop, Albert Cohen, and Olivier Temam. "Practical Run-time Adaptation with Procedure Cloning to Enable Continuous Collective Compilation". In: *GCC Developers' Summit*. 2007, page 39.

[18] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization". In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security'12. 2012, pages 40–40.

[19] Google. *AddressSanitizer for the Linux kernel*. `http://bit.ly/TdRoab`. 2014.

[20] A. Gutierrez, R.G. Dreslinski, T.F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. "Full-System Analysis and Characterization of Interactive Smartphone Applications". In: *the proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*. 2011, pages 81–90.

[21] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. "MINIX 3: a highly reliable, self-repairing operating system". In: *SIGOPS Oper. Syst. Rev.* 40.3 (2006), pages 80–89. ISSN: 0163-5980.

[22] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. "kGuard: Lightweight Kernel Protection Against Return-to-user Attacks". In: *Proceedings of the 21st USENIX Conference on Security Symposium*. Security'12. 2012, pages 39–39.

[23] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. "Quantifiable Run-Time Kernel Attack Surface Reduction". In: *Proceedings of the 11th DIMVA Conference*. 2014, pages 212–234.

[24] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. "Defeating Return-oriented Rootkits with "Return-Less" Kernels". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. 2010, pages 195–208. ISBN: 978-1-60558-577-2.

[25] Siarhei Liakh, Michael Grace, and Xuxian Jiang. "Analyzing and Improving Linux Kernel Memory Protection: A Model Checking Approach". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. 2010, pages 271–280. ISBN: 978-1-4503-0133-6.

[26] Jochen Liedtke. "On μ-Kernel Construction". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM SIGOPS Operating Systems Review. 1995.

[27] *LTP – Linux Test Project*. https://linux-test-project.github.io/.

[28] Kristis Makris and Kyung Dong Ryu. "Dynamic and Adaptive Updates of Non-quiescent Subsystems in Commodity Operating System Kernels". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. 2007, pages 327–340. ISBN: 978-1-59593-636-3.

[29] MITRE. *CWE-457: Use of Uninitialized Variable*. `https://cwe.mitre.org/data/definitions/457.html`.

[30] Vegard Nossum. *Documentation of the Linux kernel kmemcheck configuration option*. `https://www.kernel.org/doc/Documentation/kmemcheck.txt`.

[31] Niels Provos, Markus Friedl, and Peter Honeyman. "Preventing Privilege Escalation". In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. 2003, pages 16–16.

[32] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012), 2:1–2:34. ISSN: 1094-9224.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. 2012, pages 28–28.

[34] Stephen Smalley, Chris Vance, and Wayne Salamon. *Implementing SELinux as a Linux security module*. Technical report. NAI Labs Report, 2001.

[35] Brad Spengler and PaX team. *grsecurity kernel patches*. `www.grsecurity.net`.

[36] Richard Ta-Min, Lionel Litty, and David Lie. "Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. 2006, pages 279–292. ISBN: 1-931971-47-1.

[37] PaX team. *Future direction of PaX*. `pax.grsecurity.net/docs/pax-future.txt`.

[38] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. "Memory Errors: The Past, the Present, and the Future". In: *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*. RAID'12. 2012, pages 86–106. ISBN: 978-3-642-33337-8.

[39] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. "Linux security module framework". In: *Ottawa Linux Symposium*. Volume 8032. 2002.

Annex

Here, we provide a step-by-step explanation of the pre-processor-generated instrumentation for each hardening mechanism.

Listing 5: Stack exhaustion check with stack clearance

```
1    movq  %rsp, %rbp
2
3    subq  $0x50, %rsp
4+   pushq %rdi
5+   pushq %rax
```

```
 6+   pushq %rcx
 7+   pushf
 8+   movq  $0x2000, %rdi
 9+   dec   %rdi
10+   not   %rdi
11+   andq  %rsp, %rdi
12+   movq  %rdi, %rax
13+   addq  $0x68, %rdi
14+   addq  $0x80, %rdi
15+   cmpq  %rsp, %rdi
16+   jl    .hdn_stack_check_ok_6
17+   movq  %gs:current_task, %rax
18+   movq  16(%rax), %rdi
19+   movq  28(%rdi), %rdi
20+   movq  $0x100, %rax
21+   andq  $0xff00, %rax
22+   orq   $0x3ff0000, %rax
23+   orq   $0x4000000, %rax
24+   andq  %rax, %rdi
25+   jnz   .hdn_stack_check_ok_6
26+   call   __hdn_stack_chk_failed
27+.hdn_stack_check_ok_6:
28+   cld
29+   movq  %rsp, %rdi
30+   addq  $0x20, %rdi
31+   xorq  %rax, %rax
32+   movq  $0xa, %rcx
33+   rep stosq
34+   popf
35+   popq  %rcx
36+   popq  %rax
37+   popq  %rdi
38    movq  %rbx, -0x28(%rbp)
39    movq  %r12, -0x20(%rbp)
40    movq  %r13, -0x18(%rbp)
```

In line 3, this function allocates 80 bytes on the stack; this is the instruction we instrument. In lines 4 to 7, we save registers we use in the next instructions. In lines 15 and 16, we check that the stack pointer is lower than a calculated address, to make sure the stack pointer points to a safe address. Hence, we calculate first the lowest possible address of the bottom of the stack, `thread_info`+size of `thread_info`+guard zone size: In lines 9-11, we calculate the address of the current process's `thread_info` in the same way it is performed in the Linux kernel: aligning the stack pointer `%rsp` to 0x2000 byte boundaries (`THREAD_SIZE` constant in the Linux kernel, 8 KB). In line 13, the size of the `thread_info` structure is added. In line 14, the size of the guard zone is added. If the stack pointer is lower than the guard zone, we continue by performing the stack zeroing (line 27). Otherwise, we check if the kernel is executing in interrupt mode and is therefore on the interrupt stack. To perform the check we use a bitmask against `preempt_count` (retrieved from in lines 17-19). The bitmask is created on lines 20-23; `SORTIRQ_MASK` (line 20), `SOFTIRQ_OFFSET` (line 21), `HARDIRQ_MASK` (line 22) and `NMI_MASK` (line 23). In lines 24-25, we check if any of these bits are set. If so, we are in interrupt mode and the previous check's result is irrelevant: we proceed with stack zeroing (line 27). If we are not in interrupt mode, we call our custom error handler function in line 24.

In lines 28-33, we set up and perform stack zeroing. In line 28, we clear the direction flag to perform the zeroing in the correct direction. In line 29, we take the current stack pointer as the position to start the zeroing. We add 32 bytes to skip the four registers we saved on the stack at the beginning (line 30). These do not need to be zeroed afterwards, because they are outside the stack frame of the current function. In line 31, we zero `%rax`, the value the memory is going to be overwritten with. The number of repetitions in `%rcx` (line 32). In line 33, the `rep stosq` instruction writes repeatedly 8 bytes at once. In lines 34 to 36, we need to restore the registers to continue execution.

(Although this is not figured in this listing, if in presence of VLAs or if the static stack allocation is greater than the size of `thread_info`, we check in addition whether the stack pointers, before and after the stack allocation, are within the same stack. Indeed, if we would not cover these cases, an attacker could directly pass below the guard zone and `thread_info`, passing the checks and overwrite data from another stack, or `thread_info`.)

Listing 6: Function pointer protection

```
 1-   call  *0x10(%rax)
 2+   pushq %r13
 3+   pushq %rdi
 4+   movq  0x10(%rax), %rax
 5+   movq  $0x13660e4b12b74207, %rdi
 6+   inc   %rdi
 7+   movq  -0x8(%rax), %rax
 8+   movq  -0x10(%rax), %r13
 9+   cmpq  %rdi, %r13
10+   jne   .hdn_fptr_err_1
11+   leaq  _shdntext, %r13
12+   leaq  _etext, %rdi
13+   cmpq  %r13, %rax
14+   jb    .hdn_check_if_module_1
15+   cmpq  %rdi, %rax
16+   ja    .hdn_check_if_module_1
17+.hdn_do_the_call_1:
18+   popq  %rdi
19+   popq  %r13
20+   call  *%rax
21+   jmp   .hdn_fptr_done_1
22+.hdn_check_if_module_1:
23+   movq  _modulestart, %r13
24+   movq  _moduleend, %rdi
25+   cmpq  %r13, %rax
26+   jb    .hdn_fptr_err_1
27+   cmpq  %rdi, %rax
28+   ja    .hdn_fptr_err_1
29+   jmp   .hdn_do_the_call_1
30+.hdn_fptr_err_1:
31+   movq  %r13, %rdi
32+   movq  %rax, %rsi
33+   call  __hdn_function_pointer_err
34+.hdn_fptr_done_1:
```

The indirect call in line 1 is instrumented as follows: In lines 2 and 3, we save the registers we use for calculation. In line 4, we dereference the indirect address and store the destination address in `%rax`. In lines 5 and 6, we put the verification value (minus one) in a different register and increment this value (this prevents creating the verification value in the generated machine code). In line 7, we overwrite the destination address in `%rax` by the corresponding hardened function address. Note that, in case `%rax` already stores an address to a hardened function, it is overwritten by the same value again. In line 8, we fetch the verification value from the pre-function prologue and compare it in line 9 to the previously (line 5) stored value. We call the error handler function in lines 31 to 33 if the values don't match. In lines 11 to 16, the new destination address (see line 7) is checked. If it is in the hardened kernel text section, we proceed to line 18 to make the call. If not, we additionally check in lines 23 to 29 if the call address is in a kernel module (this range can also contain kernel module data, however this check is far more efficient and short). In lines 31 to 33 we call the error handling function if the previous test was negative. If all tests pass, the call is done by restoring the pushed registers (lines 18 and 19) followed by the indirect call in line 20. Finally, in line 21, a jump is performed to continue execution after the call returns.