An Analysis on the Impact and Detection of Kernel Stack Infoleaks

S. Peiró*, M. Muñoz, and A. Crespo.
Instituto de Automática e Informática Industrial (AI2)
Universitat Politècnica de València, Spain
{speiro, mmunoz, acrespo}@ai2.upv.es

March 14, 2016†

Abstract

The Linux kernel has become a fundamental component of mainstream computing solutions, now being used in a wide range of applications ranging from consumer electronics to cloud and server solutions. Being expected to continue its growth, especially in the mission-critical workloads.

Parallel to the Linux adoption has increased its misuse by attackers and malicious users. This has increased attention paid to kernel security through the deployment of kernel protection mechanisms. Kernel based attacks require reliability, where kernel attack reliability is achieved through the information gathering stage, where the attacker is able to gather enough information about the target to succeed. The taxonomy of kernel vulnerabilities includes information leaks (CWE-200), that are a class of vulnerabilities that permit access to the kernel memory layout and contents. Information leaks can improve the attack reliability enabling the attacker to read sensitive kernel data to bypass kernel based protections.

In this work we aim at the analysis and detection of stack based information leaks to harden the security of the kernel. First, we analyse the problem of kernel infoleaks in [Section 3], next, we examine the impact of infoleaks attacks on the security of the kernel in [Section 4]. Then, we present a technique for detecting kernel based infoleaks through static analysis [Section 5]. Next, we evaluate our technique by applying it to the Linux kernel [Section 6]. Last, we discuss the applications and limitations of our work [Section 6.3] and finally we draw our concluding remarks.

Keywords
- Confidentiality, Information Security, Information Disclosure (Infoleak), Operating System, Kernel

1 Introduction

The Linux kernel has become a fundamental component of mainstream computing solutions, now being used in a wide range of applications ranging from consumer electronics to cloud and server solutions. And is expected to continue its growth, especially in the mission-critical workloads. Parallel to this growth the Linux kernel has become an interesting target for attackers. Recent advances in hardening userland with protection mechanisms as ASLR [Tea01], StackGuard [CPM+98] and DEP [Cor07, 5.13] have increased the difficulty of userland based attacks, moving the attacker focus to the kernel. The main difference between userland and kernel based attacks is the consequence of the attack failure that leads to system panic/halt on kernel attacks, while on userland attacks, failure is more benign as implies a process crash/restart. Therefore, the reliability of the attack is critical when targeting the

*The author wants to thank all the people that contributed to make this work possible.
†This article has been submitted to the Logic Journal of the IGPL. Published by Oxford University Press.

kernel, kernel attack reliability is achieved through the information gathering stage where the attacker is able to gather enough information about the target to succeed in its purposes.
Among the taxonomy of kernel vulnerabilities [CMW+11] are information leaks vulnerabilities (infoleaks), identified as CWE-200 by MITRE [CWE08b], that allow to access kernel data from malicious user process. Information leaks are often underestimated as they can improve the attack efficiency allowing the attacker to access sensitive kernel data to bypass kernel based protection mechanisms.
In this work we aim at the detection of stack based information leaks to harden the security of the kernel. First, we analyse the problem of kernel infoleaks in [Section 3]. Next, we examine the impact of infoleaks attacks on the security of the kernel in [Section 4]. Then, we present a technique for detecting kernel based infoleaks through static analysis [Section 5]. Next, we evaluate our technique by applying it to the Linux kernel [Section 6]. Last, we discuss the applications and limitations of our work [Section 6.3] and draw our concluding remarks in [section 7].
Motivation for our work. We analyse the security of current kernel protection mechanisms [CPM+98, Tea01]. Discuss how these protection mechanisms can be circumvented by leveraging on information disclosure vulnerabilities [CWE08b] to access sensitive data of the protection mechanisms. This motivates our work on the detection stack based kernel information leaks. **Contributions.** The overall contribution is a systematic approach for the detection of stack based infoleak vulnerabilities, in more detail, we make the following contributions:
- **Analysis of kernel information leak vulnerabilities**, focusing on its impact on the security of kernel protection mechanisms [Section 2.1 and 4].
- **Classification of kernel information leaks**, based on our analysis we perform a classification of information leaks vulnerabilities [Section 3].
- **Detection of kernel stack based information leaks** present and evaluate a technique for the detection of stack based information leaks [Sections 5, 6].

2 Related Work

Related work addressing the security of the kernel is organised in two sections: First, subsection 2.1 discusses the current security protection mechanisms implemented by the Linux Kernel and their limitations. Next, subsection 2.2 discusses the techniques to address security vulnerabilities on the Linux Kernel, focusing on vulnerability detection and prevention techniques.

2.1 Protection Mechanisms

We start reviewing the kernel protection mechanisms against memory corruption vulnerabilities.
StackGuard. StackGuard [CPM+98] is a compiler technique that thwarts buffer overflows [CWE08a] vulnerabilities by placing a "canary" word next to the return address on the stack. If the canary is altered when the function returns a smashing attack has been attempted, and the program responds by emitting an intruder alert.
Address Space Layout Randomization. The goal of Address Space Layout Randomization (ASLR) [Tea01] is to introduce randomness into addresses used by a given task. This will make a class of exploit techniques fail with a quantifiable probability and also allow their detection since failed attempts will most likely crash the attacked task.
Data Execution Prevention (DEP) is a capability of processors to prevent data pages from being used by malicious software to execute code [Cor07, Part 1, Sec 5.13].

- NX features. Non executable and writable pages present W^X, known as No Execute (NX) Intel x86 technology. Carefully controlling the protection bits of the memory pages the OS can reduce the number of pages that are RWX, since these can be used to write code into a writable page and jump to the written code (execute). By setting the protection bits of the kernel memory pages to W^X, write or execute, but never both, hence mitigating this kind of attack.
  Still the kernel allows the user to mmap() user pages with RWX protection, therefore, enabling

the user to define a exploitable mapping, and redirecting the execution to the user mmap()ed memory pages. A common kind of attack is mmap()ing the NULL page, and causing NULL page dereference from the kernel.

- Supervisor Mode Execution Prevention (SMEP) introduces security checks to prevent the CPU to fetch user memory pages while the CPU is in CPL 0. This technique blocks the NULL page dereference attacks used to gain code execution. SMEP goes together with Supervisor Mode Access Prevention (SMAP) that controls when the CPU can access userland data pages:
  - SMEP detects when kernel is fetching instructions from userland
  - SMAP detects when kernel accessing data from userland.

Still writes from kernel to user still must be allowed at controlled points for the kernel to provide information to the userland and vice-versa (eg. ioctl interfaces), therefore, faults at these controlled points are still possible, and, its the sole responsibility of the kernel to prevent them, in case the kernel fails to prevent them, two kinds of vulnerabilities can occur:

- Arbitrary kernel memory write (kernel write), affecting the integrity of the kernel
- Arbitrary kernel memory read (infoleaks), affecting the confidentiality of the kernel.

The effectiveness of these protection mechanisms relies on the protection secrets remaining unknown to the attacker, i.e., canary value in the case of StackGuard and the randomized base address to load executable code in case of ASLR. Otherwise, revealing these secrets leads to circumvent these protection mechanisms [ea13, ea09]. The confidentiality property of the operating system is required for the protection mechanisms to remain effective, confidentiality is achieved through the hardware processors paging and memory management units [Cor07]. However, in the last stage is the task of the OS to ensure the confidentiality of its memory.

2.2 Protection Techniques

The detection of software vulnerabilities is a classic topic of computer security, various techniques have been applied to the vulnerability detection. Next, we review the existing approaches for the detection and prevention of infoleaks vulnerabilities.
Static and Data flow Analysis Sparse [Tor06] is a Semantic Checker/Parser for C used for kernel code checking and static analysis. The C programming language has some unsafe/unspecified/compiler-dependent behaviours. To solve this limitation Sparse provides C -> AST and then allows the developer to write code analyses to verify the kernel security properties, such as: kernel/user pointers, locks, integer over/under flows, endianness. Smatch [Dan13] is a rewrite of the Standford Checker (MC) using sparse, to provide the static analysis checks at the Linux kernel. Coccinelle [LBP+09] is a tool for performing control-flow based program searches and transformations in C code. Coccinelle is actively used to perform API evolutions of the Linux kernel [LBP+09] as well as finding defects in Linux and open source projects [Stu08]. **Type Inference** Taint Analysis and Type Inference [JW04] as variants of static analysis have also been performed on the kernel. **Fuzzing** Kernel API fuzzing [Jon13] is actively used to test the kernel API for unexpected vulnerabilities. **Operating system level techniques** PAX security features [Tea01] provide GCC compiler *stackleak* plugin that zeroes all automatic user structs that are allocated on the stack, providing an effective protection against infoleaks. **Coding standards** The safety critical MISRA-C standard [MIS13] mandates the initialization of all automatic variables: *"Rule 9.1: All automatic variables shall have been assigned a value before being used."* However, checking of the mandatory rules requires code reviews to enforce the coding standard, however this can be effective but time consuming and non exhaustive. **Hardware protection techniques** Hardware techniques such as SMAP (subsection 2.1) provide partial protection against infoleaks.

3 Classification of Information Disclosure Vulnerabilities

Information disclosure vulnerabilities [CWE08b] are the consequence of other kinds of vulnerabilities that lead to disclose the memory layout or contents of the running program [Ser12]. Infoleaks are relevant as they allow for the undesired disclosure of information that circumvents the confidentiality enforced by the operating system [Tan07, Security Threats 9.1.1]. The failure to protect confidentiality can be used by an attacker to increase the attack efficiency, an example of the latter are *stackjack attacks* [RO11] where infoleak vulnerabilities are employed to selectively control the stack values/contents disclosed in order to build a kernel read primitive. The kernel data read primitive is used afterwards to gain knowledge about the kernel protection mechanisms in place, for example as stack pointers, canary values and ASLR base addresses that lead to effective exploits [ea13, ea09].
In this section we analyse the problem of infoleaks to understand it as the first step towards its solution, we begin our discussion with a real-world infoleak example [Section 3.1], next a classification of the different types of infoleaks [Section 3.2]. We summarize the classification of infoleak vulnerabilities in figure 1, where the type of infoleaks we target appear grayed out.

3.1 The Anatomy Of An Infoleak

To focus our discussion we start off with the discussion of a real world infoleak vulnerability CVE-2013-2147 [MIT13]. The CVE-2013-2147 [MIT13] is a kernel stack infoleak that enables an attacker to read 2 bytes of uninitialised memory after field `->buf_size` of the `IOCTL_Command_Struct` where the memory contents are leaked from the kernel stack of the process. The relevant code displaying the vulnerability is depicted in Listing 1, along with an explanation of the vulnerability details.

```
1 static int cciss_ioctl32_passthru(
2     struct block_device *bdev, fmode_t mode, unsigned cmd, unsigned long arg) {
3     IOCTL_Command_struct arg64;
4     IOCTL_Command_struct __user *p = compat_alloc_user_space(sizeof(arg64));
5     int err;
6     err |= copy_to_user(p, &arg64, sizeof(arg64));
7     if (err)
8         return -EFAULT;
```

Listing 1: Example of infoleak code from CVE-2013-2147 (edited to fit)

The listing 1 contains an excerpt of function `cciss_ioctl32_passthru()` where the `arg64` local variable is declared at line 3 without explicit initialisation. At the compiler level the effect is that memory from the kernel stack is reserved for the `arg64` variable, however, the `arg64` memory is left uninitialised containing the data already present on the stack. This memory is afterwards copied to user space through the `copy_to_user()` at line 6 that allows an attacker to read the memory contents of the kernel stack. A detailed analysis on the security impact of stack infoleaks is presented at section 4.

3.2 Targets of Infoleaks

The previous example introduced kernel based infoleaks, however, infoleaks are also present in systems ranging from application to hypervisor level. The following examples give an idea of the targets of infoleaks:

- **Application infoleaks**: A common case of application infoleak is the disclosure of sensitive data by a server process to a remote client CVE-2012-0053 [MIT12].
- **Kernel infoleaks**: These disclose kernel memory as in CVE-2013-2147 [MIT13].
- **Hypervisor infoleaks**: Disclose hypervisor data to guest CVE-2010-4525 [MIT10].

In the case of kernel code, infoleaks have a high impact as disclose sensitive kernel data to user processes breaking the data confidentiality property enforced by the OS [Tan07, 9.1.1]. The above reasoning motivates us to focus on kernel infoleaks as we consider these the most critical case.

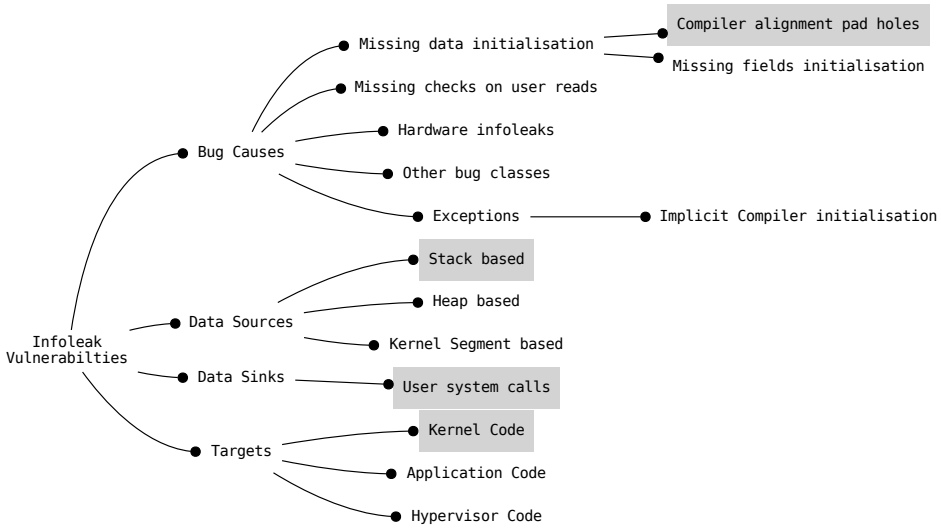

Figure 1: Identification and Classification of the Infoleak vulnerabilities.

3.3 Infoleaks Bug Causes

As defined in [Section 3], infoleaks are the consequence of other kinds of vulnerabilities that lead to disclose the memory layout or contents of the running program. This section analyses the causes that lead to information disclosure.

- **Compiler padding holes.** Compilers align data structures for performance reasons, this leads the compiler to introduce padding holes between structure fields in order to improve their memory access times [Cor07]. Therefore, when copying data to the userland the uninitialised struct holes leak kernel information. Padding holes in structures allow data to pass between user-kernel land without explicit checks, this can happen in both directions. Depending on the direction of the information flow, we can identify two situations:
  - **Writes from kernel to user**: Results in an infoleak to userland as depicted in sub-figures 2A and 2B, and is the case we target in our study.
  - **Writes from user to kernel**: Results in a kernel write from userland as depicted 2C. This can be regarded as a critical vulnerability, as represents a kernel write from user land, giving an attacker the ability to alter the contents of kernel memory. However, in this case the contents of the padding holes are usually discarded by the kernel and, are out of the scope of this work.
- **Missing memory initialisation.** When a local variable is declared on a kernel function without explicit initialization, according to the C99 Standard [ISO99, Sect. 6.7.8/10] its contents are indeterminate. In practice, however, the variable gets allocated on the stack, and its value is determined by the memory contents already present on the stack, that remain uncleared for performance reasons. When the variable is copied afterwards to userland it leads to an information leak of kernel memory, as depicted in sub-figure 2B.
- **Missing checks on user reads.** Missing or incorrect checks on buffer bounds (start, size) when copying data to user enable the user to read memory contents outside of the buffer. That kind of vulnerability named *buffer overreads* [ea09] allow to read data that was not intended to be accessed.

- **Hardware based infoleaks.** Hardware infoleaks belong to the kind of infoleak provided by the environment, this type of infoleaks is provided by sensitive instructions [PG74], i.e. instructions that modify or reveal CPU machine state to the user, break the confidentiality, and, enable to detect if a system is running under VM/VMM (pills). Examples of sensitive instructions on the x86 architecture are: `sidt/sgdt/sldt/smsw/str` [RI00], that enable to read supervisor related information, such as, the physical address of kernel interrupt descriptor tables.
- **Other bug classes leading to infoleaks.** Other sources of infoleaks not explored in this work, are those related to information already available in the environment, say for example the kernel pointer addresses provided by the `/proc/`, `/sys/` and `/boot/` file-systems that are secured in Linux Kernel by the `kptr_restrict` mechanism [Ros10]. A broader source of information disclosure flaws are covert and side channels, such as cache and TLB timing attacks [ea13] that exploit the shared nature of these hardware resources to infer information regarding memory addresses.
- **Exceptions.** There are exceptions to the infoleak bug causes discussed above, for example, variable declarations followed by a partial initialization, e.g. with `var = {0}` all fields get initialized with zeros. The behaviour mandated by the C99 Standard [ISO99, Sect. 6.7.8/19] is implemented by the compiler we have used during our analysis GCC [Pro14] The GCC performs the implicit variable zeroing preventing the occurrence of infoleaks, even in the above cases of padding holes or missing initialisation.

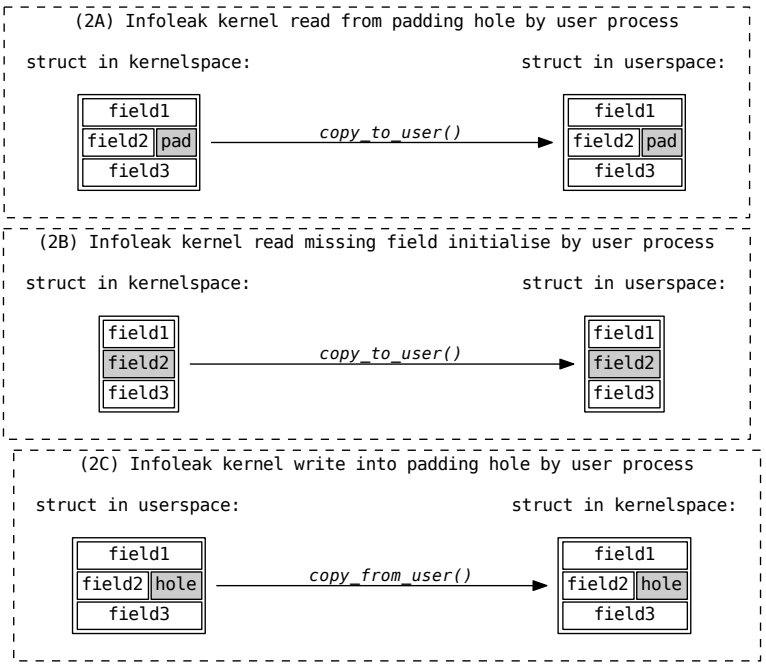

Figure 2: Directions of data flow in kernel information leaks and writes.

3.4 Infoleaks Data Sources

Information leaks disclose kernel memory contents, therefore, depending on the memory section affected, a leak can disclose different kinds of information. We focus on the three main sources from where kernel memory is allocated [Gor04].

- **Data segment**. The kernel data segment is the area that contains global kernel variables fixed during compilation time. A data segment leak can disclose the contents of static kernel symbols such as configuration variables.

- **Stack section**. The kernel stack is allocated at runtime and its operation is defined by the kernel C procedure call convention (ABI). Stack content leaks contain valuable information, as they can reveal return addresses, stack pointer, and other data contained in the stack; such as function call parameters, passed on through stack on x86-32 architecture. Other data that is kept on the stack are kernel protection mechanism secrets, such as canary values for StackGuard [CPM+98] protection. In addition, with non-randomized kernel process stacks, the stack layout remains unchanged and provides a predictable stack layout when the same kernel path is called repeatedly [RO11].
- **Heap section**. The kernel heap is managed by memory allocators employed by kernel subsystems when dynamically allocated memory is required. Due to the nature of kernel allocators, heap leaks can disclose memory around the object being allocated and its nearby objects, this can include leaks of object the type and contents, i.e., the values of its fields.

4 Analysis on the Security Impact of Stack Infoleaks

To understand why infoleak vulnerabilities are important to kernel security, we analyse the role played by infoleaks in the attack process, we start with the steps that compose a kernel attack [subsection 4.1], then, analyse the contents of the stack in [subsection 4.2] and last infoleak based attacks [subsection 4.3].

4.1 The Anatomy of An Attack

Kernel attacks have the goal of increasing the privilege level of the code run by the attacker. To do so, the attacker must gain controlled code execution capabilities to increase its privilege level, code execution is achieved by exploiting kernel vulnerabilities. An attack scenario with current kernel protection mechanisms [subsection 2.1] in place involves the following steps to break kernel security:

1) Perform identification of the system.
2) Bypass the kernel protection mechanisms: Find canary, return address.
   2.1) Find a kernel memory read vulnerability (infoleak).
3) Transfer kernel execution control to injected attacker code: Write canary, return address.
   3.1) Find an arbitrary kernel memory write vulnerability.

An attacker targeting the kernel performs the above steps, among them, the step (2.1) is required to gain enough information to ensure attack success, otherwise, the result of a failed attack is panic/halt the system. To succeed the attacker must find a way learn the canary value and return address. In user land attacks this is achieved using brute force to guess the secrets, where failed probe causes the server process to be restarted without major interference to the system. However, this no longer happens in kernel land, where a failed canary overwrite leads to a kernel panic.

4.2 The Contents of the Kernel Stack

The target of this work are kernel stack based infoleaks, to analyse the impact of infoleaks we examine the contents of the kernel stack when a kernel system call is performed in order to identify what kinds of information can be obtained by an attacker. The contents of the kernel stack on a function call are defined by the C API function call procedure which in our case is defined in the Intel x86-32 architecture [Cor07], and implemented by the C compiler GCC x86-32 [Pro14]. The Table 1 details the contents of the kernel process stack when a kernel system call is invoked, three main groups of data can be identified, starting from the top:

- The callee arguments to the function call: *callee(param1, param2, param3, ...)*

- The saved CPU registers stored by the caller to restore on return: *EIP, EBP, canary.*
- The local variables of the callee: *local1, local2, local3, ...*

Address	Value	Description
$ebp + 8$	params	Function parameters
$ebp + 4$	SEIP	Saved EIP
$ebp + 0$	SEBP	Saved EBP *(optional)*
$ebp - 4$	PAD	GCC Stack padding
$ebp - 8$	CAN	Saved Stack Canary
$ebp - 12$	locals	Local variables

Table 1: Stack Layout on function call() relative to %ebp (x86-32)

All kernel memory dumps are a security issue, however, the kernel stack dumps are more relevant due to the nature of stack, and the C99 language [ISO99] calling conventions, the following contents can be found:

- params: The argument/parameters passed to the function.
- SEIP: The return addresses to kernel code, enable to find the kernel load address.
  - Kernel address reveal the load address of the kernel image.
  - Module address reveal the load address of the module address.
- SEBP: The saved stack base pointer *(optional)*.
- CAN: The saved canary (per kernel process thread).
- locals: The local variables allocated by the function.

4.3 Infoleak Based Attacks

After an analysis of the contents of the stack in subsection 4.2, we outline the possible uses of the information obtained by an infoleak.

- **Precise system identification**

  Perform the identification of the system. Being able to leak kernel addresses, enables effective system identification of the exact kernel version running on the system, by fingerprinting the kernel function addresses, and building a table of tuples $(address, kernel\_version)$. This fact should not be overlooked since effective identification of the running kernel target, is the first step towards effective attacks.

- **StackGuard protection bypass**

  Obtain the canary values to bypass the kernel StackGuard [CPM+98] protection. As seen in Table 1, the canary value (CAN) resides on the stack, therefore, an info leak reveals the canary value of the current kernel thread process allowing to bypass the *StackGuard*.

- **KASLR protection bypass**

  Obtain the kernel text return addresses to bypass the kernel KASLR [Coo13] protection. As seen in Table 1, the return address value (SEIP) resides on the stack, therefore, an info leak reveals the SEIP $(aslr\_ktext\_addr)$ value of the current kernel thread process allowing to

bypass *KASLR*. Since both $aslr\_ktext\_addr$ and $ktext\_addr$ are known, the randomised kernel text section load offset introduced by the *KASLR* can be computed as: $aslr\_ktext\_offset = aslr\_ktext\_addr - ktext\_addr$ [Byp09].

- **Stack trace**

  A full/partial stack-trace of the call leading to the infoleak, depending on the size of the infoleak.

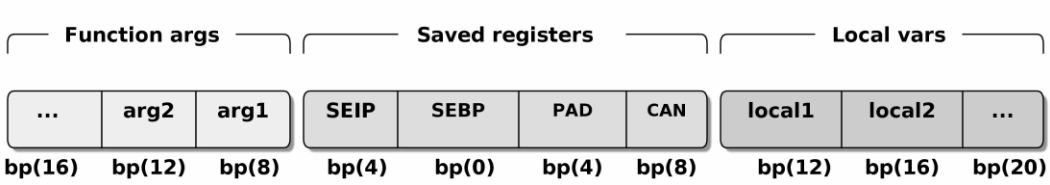

Figure 3: Stack Layout on function call relative to ebp (x86-32).

4.3.1 Infoleak Attack Variations

The stack infoleaks disclose only the uninitialised part of the current stack frame, therefore, acting as a window that enables to look at the contents of the stack. The infoleak window is described as $W = (offset, size, contents)$ where:

- $offset$: The $offset$ is measured from the top of the stack and its value fixed by the function call stack frame
- $size$: The $size$ is fixed and defined by the size of uninitialised stack frame section disclosed.
- $contents$: The contents of the stack revealed by the infoleak window.

The only parameter of W that is influenced by the attacker is the $W.content$, since calling different system calls leaves the contents of that $syscall()$ stack frame in memory, these contents can be later retrieved by invoking the infoleak during the triggering stage of the vulnerability.
The Figure 4 depicts the above scenario where the contents of the kernel process stack are shown after performing two different syscalls. In Figure 4(a) after invoking $syscall1()$ the kernel stack contains $W.contents = 0x11111111$. While in Figure 4(b) after invoking $syscall2()$ the kernel stack contains $W.contents = 0x22222222$. This scenario depicts how the attacker performs a preparation stage before triggering the infoleak, where he controls the *contents* of the kernel stack invoking selected *syscalls* that leave sensible data on the stack, that are later retrieved during the triggering stage of the infoleak. Therefore, the above scenario is important for system calls that leave security critical information on the kernel stack.

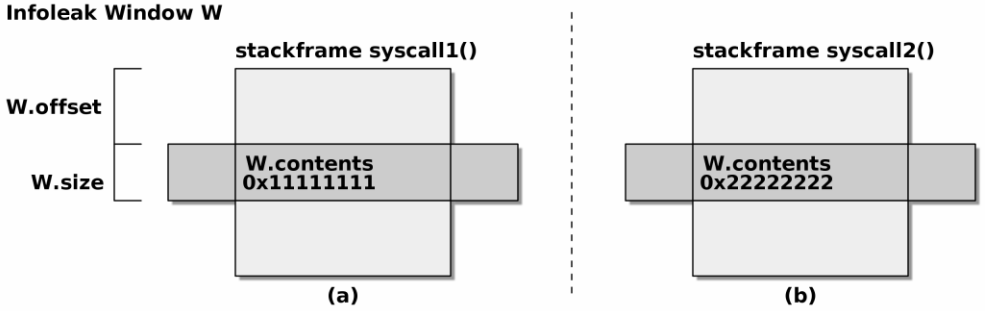

Figure 4: Kernel stackframe infoleak window.

5 The Detection of Information Leak vulnerabilities

This section discusses the proposed infoleak detection technique, the key idea is to perform static source code analysis, the main steps that compose our approach are outlined here:

1. Analysis of the Infoleak vulnerability model in [Section 5.1].
2. Design of the semantic patch for the vulnerability model in [Section 5.2].
3. Filter and rank the code matches to remove false positives in [Section 5.3].
4. Review and correct the vulnerabilities detected in [Section 5.4].

5.1 Analysis of the Infoleak vulnerability model

The first step is to analyse infoleaks vulnerabilities in order to model them. In order to model stack based kernel infoleak vulnerabilities we adopt the notions of taint analysis [DD77]. We focus on infoleaks of privileged kernel memory to userland as depicted in figure 2, and start with the identification of data sources, data sinks, and taint property:

- **Data Sources**: The interesting *data sources* for our analysis are the uninitialised kernel stack memory contents. As discussed in [Section 3.1] on source kernel data are uninitialised local variables declared on kernel functions.
- **Data Sinks**: The type of *data sinks* we are interested are those reachable from userland, these are part of the kernel API exposed through the system call interface. Examples of these are file-system *read()* operations these are interesting sinks for our analysis as they allow data to flow from kernel to user, here we focus on the `copy_to_user()` calls data sinks.
- **Taint Property**: The taint property we are interested in is the flow of uninitialised data from the identified kernel space *sources* to user space *sinks*.

5.2 Design of the semantic patch for the vulnerability model

Based on the vulnerability model developed in our previous step (subsection 5.1), we prepare a semantic patch [LBP+09] to perform control-flow program static analysis to detect vulnerable code sites matching the vulnerability model. To this end we select Coccinelle [LBP+09] that is an open-source developer-friendly static analysis tool widely used in open source projects to perform automated API evolutions.
The semantic patch depicted in Listing 2 is applied to the complete Linux kernel source tree using the semantic patch infrastructure available on the Linux kernel [PLM10] invoking the command `make coccicheck COCCI=infoleak.cocci` to detect the infoleak vulnerability model discussed above.
The semantic patch (Listing 2) matches the body of each kernel function at line 7, for each kernel function matched the following taint analysis conditions are checked:

1) The line 9 matches the declaration of any non-initialised local variable `ID` of function `handler` that is allocated on the kernel stack (**data source**).
2) The line 10 ensures that memory contents of `ID` remain uninitialised by restricting to situations where no `memset()` or initialisation operations of the local variable `ID` occur (**taint property**).
3) The line 12 matches the code location that copies the local variable `ID` to the user pointer `EV` (**data sink**).

Last, for each code location where the above taint conditions hold, the line 20 uses Python code scripting to calculate the size of the infoleak as detailed in subsection 5.3.

5.3 Filter and Rank of matches

The results of the execution of the semantic patch discussed at subsection 5.2 contain the potential vulnerabilities ranked according to its likelihood of being a real vulnerability. The ranking is performed to

```
1  @rule@
2  type T;
3  identifier ID;
4  function handler;
5  expression EV, EN;
6  @@
7  handler(...) {
8  <...
9  *   T ID;
10     ... when != memset(&ID, 0, ...)
11         when != ID = ...
12 *   copy_to_user(EV, &ID, EN)
13  ...>
14 }
15
16 @script:python@
17 t << rule.T;
18 @@
19 import leaklib
20 (sizeof, sumsizeof, leaksize, fields, ftypes) = leaklib.leaksize(t);
```

Listing 2: Semantic patch (SmPL) for stack based infoleak detection.

reduce the amount of required manual work during code audits of the infoleak detection results to increase the vulnerability detection rate. For each code location matched by the semantic patch, the following fields are extracted from the match to identify each vulnerability $vuln = (function, variable, struct)$. The filtering function in Equation 1 calculates the size of the infoleak in bytes as the size of the padding holes in the struct. This enables to determine the relevance of the infoleak and allows to order the results giving a higher relevance to those leaking more bytes.

$$leaksize(struct) = sizeof(struct) - \sum_{f \in struct} sizeof(struct.f) = \begin{cases} = 0 & \text{No padding leak.} \\ > 0 & \text{padding leak.} \end{cases} \quad (1)$$

The Equation 1 is calculated for each struct matched by the semantic patch at Listing 2, the calculation requires accessing the information about the fields and sizes of the struct emitted by the compiler during the kernel build. This information is already available in the DWARF debug section of the vmlinux ELF image [Too01] when the Linux kernel is compiled debug information enabled (`CONFIG_DEBUG_INFO`). Therefore, the $leaksize(struct)$ is implemented by querying the vmlinux DWARF information of the structures using the Python GDB bindings from the *{python script}* section of the semantic patch (Listing 2) at line 20.

5.4 Infoleak Code Review and Correction

The last step is to review the detected vulnerabilities, to triage the real bugs out of the potential vulnerabilities. This is the only step requiring manual intervention, but, can be partially automated by zeroing all the detected local variable declarations thus preventing the detected infoleaks, however, requires a compromise between performance impact and security implications.

6 Experimental Evaluation of the Detection Technique

To evaluate our approach we select the Linux kernel sources as the target for the detection potential infoleaks. First, we evaluate our approach using an experiment aimed at detecting already known vulnerabilities [Section 6.1]. Last, we study how our approach applies to discovery of new vulnerabilities [Section 6.2].

6.1 Existing Infoleak Detection

To evaluate the performance of our detection technique, we conduct a binary classification experiment [She04], where the proposed detection technique is evaluated as a binary classifier that given a code location classifies it as a vulnerability (Null hypothesis H_0) or not a vulnerability (Alternative hypothesis H_A). The experiment setup is performed as follows: (1) First, select the set of known stack based infoleak vulnerabilities of the Linux kernel listed at Table 4, such as CVE-2013-2147 [MIT13]. (2) Next, prepare a copy of the Linux kernel sources containing the selected set of vulnerabilities in Table 4. (3) Last, apply the detection technique to the full copy of the kernel source tree prepared at step 2 to evaluate the detection of infoleak vulnerabilities introduced.
The possible outcomes of the test are depicted in the confusion matrix at Table 2 [KP98], where each outcome is defined as:

- *True Positive (TP)*: A vulnerability correctly detected as vulnerability.
- *True Negative (TN)*: A non vulnerability correctly detected as non vulnerability.
- *False Positive (FP)*: A non vulnerability detected as a vulnerability (Error of Type I).
- *False Negative (FN)*: A vulnerability detected as non vulnerability (Error of Type II).

Confusion matrix	H_0 **True**	H_0 **False**
Detection Positive	True Positive	False Positive (Type I Error)
Detection Negative	False Negative (Type II Error)	True Negative

Table 2: Outcomes of the vulnerability detection classification.

The detection experiments are conducted on a recent Intel Dual-Core i5 CPU 3.3 GHz with 4 GB of RAM, the detection is applied to the whole Linux kernel source tree, where the experiments take an average of 15 minutes to complete. After conducting the experiments of the infoleak detection for stack based kernel infoleaks with $leaksize(struct) > 0$, the outcomes of the experiments are then statistically evaluated applying the Hypothesis testing techniques defined in [She04, p. 305].

Measure/Kernel ver	*v2.6*	*v3.0*	*v3.2*	*v3.4*	*v3.8*	*v3.14*
Vulns Detected/Present	13/8	14/8	12/6	12/6	11/4	9/4
True Positive (TPR%)	100.0	100.0	100.0	100.0	100.0	50.0
True Negative (SPC%)	99.2	99.2	99.3	99.3	99.4	99.5
Positive Pred (PPV%)	61.5	57.1	50.0	50.0	36.3	22.2
False Positive (FPR%)	0.8	0.8	0.7	0.7	0.6	0.5

Table 3: Statistical performance of stack infoleak detection per kernel version.

The results of the statistical evaluation are depicted in Table 3, where the detection performance results presents a high sensitivity (TPR) and high specificity (SPC) both close to 100%, while the false positive rate (FPR) is close to zero. That enables to carry security code audits to verify and correct the vulnerabilities detected.

6.2 Discovery of Vulnerabilities

We have applied our detection technique to the Linux kernel v3.12, as a result multiple new infoleak vulnerabilities have been uncovered listed in Table 5, that disclosed between two and four bytes of the kernel stack contents to userland. The affected device driver files are: `net/wan/{farsync.c,wanxl.c}`, `tty/{synclink.c,synclink_gt.c}`, and `net/hamradio/yam.c`. After preparation of the corresponding patches the corrections to the infoleaks have been incorporated into the kernel development tree [Pei14].
During 2015 we performed a second infoleak analysis to the Linux kernel v4.0, applying our detection technique that uncovered three new infoleak vulnerabilities that disclose between 16 and 200 bytes of kernel stack contents to userland, the corrections have been applied to the kernel development tree [Pei15]. Among the vulnerabilities we found [Pei15, CVE-2014-1739] that caused 200 bytes of kernel stack to be read by a local user on devices using the Linux media subsystem, and affected desktop and Android devices using Linux video devices and cameras (V4L).

Known Infoleaks	**Commit Fixes**	**Versions Affected**
copy_semid_to_user	982f7c2b	v2.6
cciss_ioctl32_passthru	58f09e00	v2.6 - v3.14
hiddev_ioctl	9561f7fa	v2.6 - v3.0
proc_connectinfo	886ccd45	v2.6 - v3.14
pg_read	a2c2a0e6	v2.6 - v3.0
vcc_getsockopt	e862f1a9	v2.6 - v3.4
ip_vs_get_dest_entries	a8241c63	v2.6 - v3.14
ip6_tnl_ioctl	206b495	v3.0 - v3.4

Table 4: Known infoleak vulnerabilities used for detection

New Infoleaks	**Commit Fixes**	**CVE identifier**
hdlcdev_ioctl	b19a47e0	(not assigned)
wanxl_ioctl	2b13d06c	CVE-2014-1445
yam_ioctl	8e3fbf87	CVE-2014-1446
fst_get_iface	96b34040	CVE-2014-1444
media_device	e6a62346	CVE-2014-1739
vivid_fb_ioctl	eda98796	CVE-2015-7884
dgnc_mgmt_ioctl	4b618433	CVE-2015-7885

Table 5: Infoleak vulnerabilities discovered with the proposed approach

6.3 Applications and Limitations of our Approach

We believe that our approach improves the kernel security, we base our discussion on the Linux kernel, however, the approach is applicable to systems that present a similar vulnerability model. The main

application of our technique is on conducting security audits at different stages of the development cycle: **(a) At Release stage** to ensure that less bugs get into the product release. **(b) At Development stage** to avoid introducing errors in early development stage. **(c) At Regression stage** to ensure a known bug is not re-introduced.
Similar to other methods for the discovery of security flaws, our approach cannot overcome the inherent limitations of vulnerability identification, i.e., vulnerability detection is an undecidable problem that stems from Rice's theorem [Hop08]. Our technique aims at finding known vulnerabilities at the source code level, therefore, unknown flaws not matching the vulnerable model remain undetected, for example our model only considers infoleaks at a single function level, therefore,infoleaks involving multiple functions are not covered by our approach. The result derives from the limitations of black-listing as a security measure [SS75, Fail-safe defaults], where blacklist detects only a subset of unallowed patterns. Therefore, a better approach is to enforce a white-list to detect all unallowed patterns.

7 Conclusions and Further Work

In this work, we presented an analysis and classification of information leaks causes and their impact on the security of the kernel. Then, we proposed a technique for the detection of the class of stack based kernel information leaks. Last, we evaluated our technique applying it to the Linux kernel, our evaluation results showed that the detection technique is effective to improve operating system security. This work has focused on stack infoleaks at the operating system level, however, there are other types of infoleaks, such as heap infoleaks (subsection 3.4). That is, infoleaks that target heap memory where the data source is provided by the memory allocator (eg. `kmalloc()`). In this case, the proposed semantic patch (Listing 2) can be updated by replacing the declaration of the local variable allocated on the stack at line 9 by the expression `ID = kmalloc(...)`, that matches variables allocated from the kernel heap. Another interesting line of work, is to address infoleaks at the hypervisor level, where a malicious guest virtual machine can use infoleaks to compromise the security of the remaining guests. Therefore, further work covers detection of infoleaks at the hypervisor to improve the confidentiality of the guest virtual machines [SPM+12] and to overcome our approach limitations [Section 6.3].

References

[Byp09] Bypassing PaX ASLR protection. Tyler Durden, 2009. http://www.phrack.com/issues.html?issue=5&id=9.

[CMW+11] 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*, page 5. ACM, 2011.

[Coo13] Kees Cook. Linux Kernel ASLR (KASLR). In *Linux Security Summit*, October 2013.

[Cor07] Intel Corp. *IA-32 Architecture Software Developer's Manual - Volume 3A*, 2007.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[CWE08a] MITRE. Common Weakness Enumeration. CWE-121: Stack-based Buffer Overflow., 2008. http://cwe.mitre.org/data/definitions/121.html.

[CWE08b] MITRE. Common Weakness Enumeration. CWE-200: Information Exposure., 2008. http://cwe.mitre.org/data/definitions/200.html.

[Dan13] Dan Carpenter. Smatch, Static analysis for C, 2013. http://repo.or.cz/w/smatch.git.

[DD77] Dorothy E Denning and Peter J Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[ea09] R. Strackx et al. Breaking the Memory Secrecy Assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.

[ea13] R. Hund et al. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE SSP*, 2013.

[Gor04] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River, 2004.

[Hop08] J. E Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education, 2008.

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[Jon13] D. Jones. The Trinity system call fuzzer, Linux Kernel, 2013.

[JW04] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, volume 2, page 0, 2004.

[KP98] R. Kohavi and F. Provost. Special Issue on Applications of Machine Learning and the Knowledge Discovery Process. 1998. http://robotics.stanford.edu/~ronnyk/glossary.pdf.

[LBP+09] Julia L Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 43–52. DSN'09, IEEE, 2009.

[MIS13] MISRA. MISRA C:2012. Guidelines for the use of the C language in critical systems, March 2013.

[MIT10] MITRE. CVE-2010-4525. kvm: x86: zero kvm_vcpu_events->interrupt.pad infoleak, 2010. CVE-2010-4525.

[MIT12] MITRE. CVE-2012-0053: Apache information disclosure on response to Bad HTTP Request, 2012. CVE-2012-0053.

[MIT13] MITRE. CVE-2013-2147. fix info leak in cciss_ioctl32_passthru()., 2013. https://git.kernel.org.

[Pei14] S. Peiró. CVE request: Assorted Kernel infoleak security fixes, 2014. CVE-2014-1444.

[Pei15] S. Peiró. CVE Request: Linux Kernel ioctl infoleaks fixes, 2015. CVE-2014-1739 and CVE-2015-7884.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[PLM10] Nicolas Palix, Julia Lawall, and Gilles Muller. Using Coccinelle on the Linux kernel. 2010. https://www.kernel.org/doc/Documentation/coccinelle.txt.

[Pro14] The GNU Project. The GNU C Compiler Collection (GCC) gcc-4.7, 2014. http://gcc.gnu.org/gcc-4.7/.

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 3–4, 2000.

[RO11] Dan Rosenberg and J. Oberheide. Stackjacking: A PaX exploit framework, 2011. https://github.com/jonoberheide/stackjacking/.

[Ros10] Dan Rosenberg. kptr_restrict for hiding kernel pointers, 2010. http://lwn.net/Articles/420403/.

[Ser12] Serna, Fermin J. The info leak era on software exploitation. *Black Hat USA*, 2012.

[She04] David J Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. CRC Press, 2004. 4th edition.

[SPM+12] J. Sánchez, S. Peiró, M. Masmano, J. Simó, and P. Balbastre. Linux porting to the XtratuM Hypervisor for x86 processors. In *14th Real Time Linux Workshop*, October 2012.

[SS75] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[Stu08] Henrik Stuart. *Hunting Bugs with Coccinelle.* PhD thesis, Diku, 2008.

[Tan07] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[Tea01] PAX Team. PaX address space layout randomization (ASLR), 2001. http://pax.grsecurity.net/docs/aslr.txt.

[Too01] Tool Interface Standards Committee. Executable and Linkable Format (ELF). *Specification, Unix System Laboratories*, 2001. version 1.2.

[Tor06] Linus Torvalds. Sparse: A semantic parser for C, 2006. http://sparse.wiki.kernel.org.