Detecting Stack based kernel Information leaks

S. Peiró*, M. Muñoz, M. Masmano, and A. Crespo.

Instituto de Automática e Informática Industrial (AI2)
Universitat Politècnica de València, Spain
{speiro, mmunoz, mmasmano, acrespo}@ai2.upv.es

Abstract. The Linux kernel has become widely adopted in the mobile devices and cloud services, parallel to this has grown its abuse and 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, 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, that are a class of vulnerabilities that permit access to the kernel memory layout and contents. Information leaks can improve the attack reliability allowing the attacker to read sensitive kernel data to bypass kernel based protections.
In this work, we aim at the detection of stack based kernel information leaks to secure kernels. We analyse the problem of stack based kernel infoleaks, then we perform a classification of the causes of information disclosure vulnerabilities. Next, we propose an approach for the detection of stack based kernel infoleaks using static analysis techniques, and last we evaluate our approach applying it to the Linux kernel.

1 Introduction

With the wide adoption of the Linux kernel as the operating system used in Android mobile devices, embedded systems, and cloud services, the Linux kernel has become an interesting target for attackers. Recent advances in hardening userland with protection mechanisms as ASRL [25], StackGuard [8] and DEP [6, 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 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 [4] are information leaks vulnerabilities (infoleaks) [5] that allow to access kernel data from a malicious user

* The author wants to thank all the people that contributed to make this work possible.
The final publication is available at Springer via:
http://dx.doi.org/10.1007/978-3-319-07995-0_32

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 kernel code. First, we analyse the problem of kernel infoleaks in [Section 2]. Then, we present a technique for detecting kernel based infoleaks through static analysis [Section 3]. Next, we evaluate our technique by applying it to the Linux kernel [Section 4]. Last, we discuss the applications and limitations of our work [Section 4.3] and drawn our final conclusions.

Motivation for our work We analyse the security of current kernel protection mechanisms [8,25]. Discuss how these protection mechanisms can be circumvented by leveraging on information disclosure vulnerabilities [5] 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 1.1].
- **Classification of kernel information leaks**, following from our analysis we perform a classification of information leaks vulnerabilities [Section 2].
- **Detection of kernel stack based information leaks** present and evaluate a technique for the detection of stack based information leaks [Sections 3,4].

1.1 Protection Mechanisms

We start reviewing the kernel protection mechanisms.

StackGuard. StackGuard [8] is a compiler technique that thwarts buffer overflows 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) [25] 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.

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 [10,11]. 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 [6]. However, in the last stage is the task of the OS to ensure the confidentiality of its memory.

1.2 Related Work

The detection of software vulnerabilities is a classic topic of computer security, various techniques have been applied to the vulnerability detection. Next, we review and compare related approaches for the detection of infoleaks.

Static and Data flow Analysis Sparse [26] is a Semantic Checker/Parser for C used for kernel code checking and static analysis. Coccinelle [17] 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 [17] as well as finding defects in Linux and open source projects [22]. **Type Inference** Taint Analysis and Type Inference [15] as a variants of static analysis have also been performed on kernel. **Fuzzing** Kernel API fuzzing [16] is actively used to test the kernel API for unexpected vulnerabilities. **Real-time detection**: Real-time Intrusion detection techniques (IDS) are prevalent as attack prevention technique [7].

2 Information Disclosure Vulnerabilities

Information disclosure vulnerabilities [5] are the consequence of other kinds of vulnerabilities that lead to disclose the memory layout or contents of the running program. Infoleaks are relevant as they allow for the undesired disclosure of information that circumvents the confidentiality enforced by the operating system [24, 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* [20] 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 [10,11].

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 2.1], next a classification of the different types of infoleaks [Section 2.2]. We summarize the classification of infoleak vulnerabilities in figure 1, where the type of infoleaks we target appear grayed out.

2.1 The Anatomy Of An Infoleak

To focus our discussion we start off with the discussion of a real world infoleak vulnerability CVE-2013-2147 [3]. The CVE-2013-2147 [3] 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 1.1, 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 1.1. Example of infoleak code from CVE-2013-2147 (edited to fit)

The listing 1.1 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.

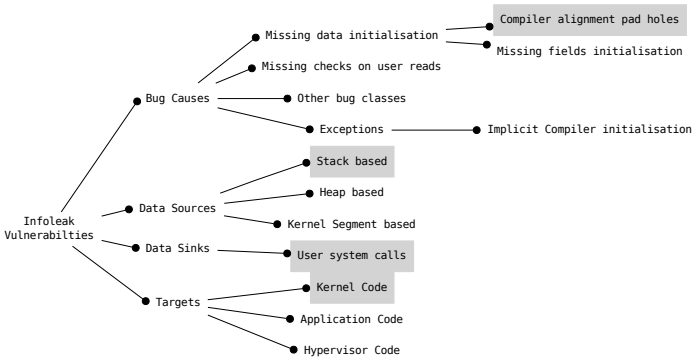

Fig. 1. Identification and Classification of Infoleak vulnerabilities.

2.2 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 [2].
- **Kernel infoleaks**: These disclose kernel memory as in CVE-2013-2147 [3].
- **Hypervisor infoleaks**: Disclose hypervisor data to guest CVE-2010-4525 [1].

In the case of kernel code, infoleaks have a high impact these disclose sensitive kernel data to user processes breaking the data confidentiality property enforced by the OS [24, 9.1.1]. The above reasoning motivates us to focus on stack based kernel infoleaks as we consider these the most critical case.

2.3 Infoleaks Bug Causes

As defined in [Section 2], 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 [6]. 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 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 is 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 [14, 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* [11] allow to read data that was not intended to be accessed.
- **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 these are already covered in Linux Kernel `kptr_restrict` mechanism [18]. A broader source of information disclosure flaws are covert and side channels, such as cache and TLB timing attacks [10] 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 `= {0}` all fields get initialized with zeros. The behaviour mandated by the C99 Standard [14, Sect. 6.7.8/19] is implemented by the compiler we have used during our analysis GCC. The GCC performs the implicit variable zeroing preventing the occurrence of infoleaks, even in the above cases of padding holes or missing initialisation.

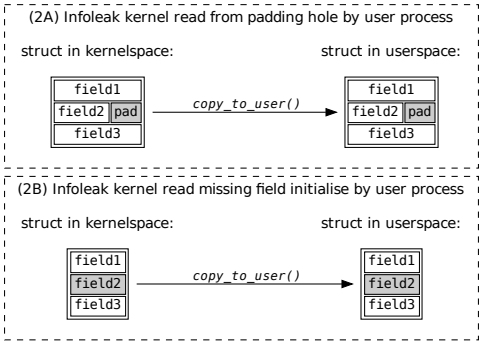

Fig. 2. Directions of data flow in kernel information leaks.

2.4 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 [12].

- **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 [8] 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 [20].
- **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.

3 Infoleak Detection Technique

In this section we present our technique, the main steps of the technique are outlined here: (3.1) Analysis of the attack and vulnerability model. (3.2) Design a semantic patch of the vulnerability model. (3.3) Filter and rank the code matches to remove false positives. (3.4) Review and correct the vulnerabilities detected.

3.1 Infoleak Vulnerability Model

We analyse infoleaks vulnerabilities in order to model them as a first step towards the detection of infoleaks. In our model of stack based kernel infoleak vulnerabilities we adopt the notions of taint analysis [9]. 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 2.1] 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*.

3.2 Semantic Patch Preparation

Based on the vulnerability model developed in our previous analysis, we prepare a semantic patch [17] to perform control-flow program static analysis to detect vulnerable code sites matching the vulnerability model. To this end we select Coccinelle [17] that is an open-source developer-friendly static analysis tool widely used in open source projects to perform automated API evolutions.

```
1 handler(...) {
2 <...
3     T ID;
4     ... when != memset(&ID, 0, ...)
5         when != ID = ...
6 *   copy_to_user(EV, &ID, EN)
7 ...> }
```

Listing 1.2. Semantic patch (SmPL) for stack based infoleak detection (edited to fit)

For our analysis we develop a Coccinelle semantic patch depicted at listing 1.2 that matches the infoleak vulnerability model discussed above.

- **Data Source**: The local variable `ID` of `handler()` declared at line 3.
- **Data Sink**: The local variable `ID` is copied to the user pointer `EV` at line 6.
- **Taint Property**: The property we want to ensure is that memory contents of `ID` remain uninitialised, therefore, we restrict to the situations where no `memset()` or initialisation operations occur at line 4.

3.3 Filter and Rank of matches

The results of the execution of the semantic patch discussed at the step 3.2 contain the potential vulnerabilities ranked according to its likelihood of being a real vulnerability. The ranking is performed to 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 3.3 calculates the size of the infoleak in bytes as the size of the padding holes in the struct. The equation 3.3 determines the relevance of the infoleak and allows to order the results giving a higher relevance to those leaking more bytes.

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

3.4 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.

4 Empirical Evaluation

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 4.1]. Last, we study how our approach applies to discovery of new vulnerabilities [Section 4.2].

4.1 Existing Infoleak Detection

To evaluate the performance of our detection technique, we prepare an experiment targeting known infoleak vulnerabilities present in the Linux kernel v3.0 series. For this we first review the MITRE Vulnerabilities CVE database, and select several stack based infoleak vulnerabilities in Linux kernel, such as CVE-2013-2147 [3]. With this set of infoleak vulnerabilities we prepare a kernel source tree containing the unpatched vulnerabilities, then target our detection approach towards it to verify the approach detects the infoleak vulnerabilities introduced. With this we can evaluate the detection performance of our technique.

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 1. Statistical performance of stack infoleak detection per kernel version.

The table 1 shows the statistical performance measures of the infoleak detection for stack based kernel infoleaks with $leaksize(struct) > 0$, i.e., those where the bug cause are compiler compiler padding holes. The detection performance presents a high sensitivity (TPR) and high specificity (SPC) both close to 100%, while the false positive rate (FPR) is close to zero. This enables analysts to perform security code audits to verify and correct the vulnerabilities detected.

4.2 Discovery of Vulnerabilities

We have applied our detection technique to the Linux kernel v3.12, as a result five new infoleak vulnerabilities have been uncovered disclosing 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 that correct the infoleaks have been applied to the kernel development tree [19].

4.3 Applications and Limitations of our Approach

We believe that our approach improves kernel security, we base our discussion on the Linux kernel, however, the approach is applicable to systems presenting 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 [13]. 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 [21, 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.

5 Conclusions and Further Work

In this work, we presented an analysis and classification of information leaks causes and its impact on security. 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. We focused on infoleaks at operating system level, however, infoleaks are present in hypervisors as well, where a malicious guest virtual machine can use infoleaks to compromise the security of the remaining guests. Further work covers detection of infoleaks in hypervisor to improve the confidentiality of the guest virtual machines [23] and overcome our approach limitations [Section 4.3].

References

1. CVE-2010-4525. kvm: x86: zero kvm_vcpu_events->interrupt.pad infoleak.
2. CVE-2012-0053: Apache information disclosure on response to Bad HTTP Request.
3. CVE-2013-2147. fix info leak in cciss_ioctl32_passthru(). https://git.kernel.org.
4. Haogang Chen, Yandong Mao, and Xil Wang. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *APSYS '11*. ACM.
5. MITRE. Common Weakness Enumeration. CWE-200: Information Exposure.
6. Intel Corp. *IA-32 Architecture Software Developer's Manual - Volume 3A*, 2007.
7. A. Herrero et al. RT-MOVICAB-IDS: Addressing real-time intrusion detection. *FGCS '13*.
8. C. Cowan et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX-SEC*, 1998.
9. D. E. Denning et al. Certification of Programs for Secure Information Flow. *C. ACM*, 1977.
10. R. Hund et al. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE SSP*, 2013.
11. R. Strackx et al. Breaking the Memory Secrecy Assumption. EUROSEC '09.
12. M. Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall.
13. J. E Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. 2008.
14. ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.
15. R. Johnson. Finding user/kernel pointer bugs with type inference. USENIX-SEC.
16. D. Jones. The Trinity system call fuzzer, Linux Kernel, 2013.
17. J. L. Lawall, J. Brunel, N. Palix, and R. Rydhof Hansen. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. DSN'09, IEEE, 2009.
18. Linux. kptr_restrict: disclosure of kernel pointers. Documentation/sysctl/kernel.txt.
19. S. Peiró. CVE request: Assorted kernel infoleak security fixes. CVE-2014-1444.
20. D. Rosenberg and J. Oberheide. Stackjacking: A PaX exploit framework, 2011.
21. J. Saltzer. The protection of information in computer systems. *IEEE Proc.*, 1975.
22. Henrik Stuart. *Hunting Bugs with Coccinelle*. PhD thesis, Diku, 2008.
23. 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*, 2012.
24. A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 3rd edition, 2007.
25. PAX Team. Address Space Layout Randomization (ASLR). 2001.
26. Linus Torvalds. Sparse: A semantic parser for C, 2006. http://sparse.wiki.kernel.org.