Detecting Stack Based kernel Information Leaks.

S. Peiró, M. Muñoz, M. Masmano, A. Crespo
Instituto de Automática e Informática Industrial
Universitat Politècnica de València, Spain
{speiro, mmuñoz, mmasmano, acrespo}@ai2.upv.es

July, 2014

Motivation

Objective Detection of infoleak vulnerabilities to harden against kernel attacks.
Why? Because infoleaks lead to bypass kernel protection mechanism.

Linux Attacks

- Linux has become an interesting target for attackers.
- Attack complexity increased by protection mechanisms: StackGuard, ASLR, DEP [8, 25, 6].

Kerneland Attacks

- Attacks have shifted from userland to the Linux kernel.
- Kernel attacks require high reliability (failure leads to system crash).
- Attack reliability is achieved by the information gathering stage.

Infoleak Vulnerabilities

Infoleaks vulnerabilities lead to build reliable kernel attacks [20, stackjacking]

- Leak pointers/addresses to find memory layout: Defeat ASLR.
- Leak stack contents to find stack canary, return addresses: Defeat StackGuard.
- Leak of keys and sensitive kernel data.

Analysis of the Kernel Vulnerabilities

A look at reported kernel vulnerabilities during 2011 [4] reveals that:

- In table 1 Infoleak vulnerabilities have a high occurrence rate.
  - Specifically infoleaks due to uninitialised memory have the highest occurrence.
- This motivates us to focus on kernel infoleaks due to uninitialised memory.

Vulnerability/Exploit	mem. corruption	policy violation	dos	info. disclosure
Missing pointer check	6	0	1	2
Missing permission check	0	15	3	0
Buffer overflow	13	1	1	2
Integer overflow	12	0	5	3
Uninitialized data	0	0	1	28
Null dereference	0	0	20	0
Divide by zero	0	0	4	0
Infinite loop	0	0	3	0
Data race / deadlock	1	0	7	0
Memory mismanagement	0	0	10	0
Miscellaneous	0	0	5	2
Total	**32**	**16**	**60**	**37**

Table: Reported kernel vulnerabilities during 2011. Data from H. Chen study [4].

Outline

Infoleak Vulnerabilities through Example: CVE-2014-1739

CVE-2014-1739 is one of the infoleaks detected using the technique discussed here.

- **Impact**: A local user can read 200 bytes from the kernel process stack.
- **Affected version**: Linux Kernel `media` subsystem from v2.6.38 ahead (3 years).
- **Affected systems**: Android phones and servers setups using affected versions.
- **Attack**: Read memory contents from kernel process stack offset controlling stack depth.
- **Reported**: April, 2014

```
1  static long media_device_enum_entities(struct media_device *mdev,
2      struct media_entity *ent;
3      struct media_entity_driversesc u_ent;                                          [3]
4
5    + memset(&u_ent, 0, sizeof(u_ent));                                              [5]
6      // ...
7      if (copy_to_user(uent, &u_ent, sizeof(u_ent)))                                 [7]
8         return -EFAULT;
9     return 0;
```

Listing 1: CVE-2014-1739 code from `drivers/media/media-device.c`

CVE-2014-1739 Infoleak Vulnerability Description

- At line 3 of listing 1 shows the `u_ent` local variable is declared without explicit initialisation.
- The `u_ent` memory is left uninitialised containing the data already present on the stack.
- At line 7 of listing 1 `u_ent` is copied to user space through the `copy_to_user()`. That allows an attacker to read the memory contents of the kernel stack.

Infoleak Vulnerabilities: Definition

Definition Infoleaks [5] are the consequence of other kinds of vulnerabilities that lead to disclose the memory layout or contents of the running program.

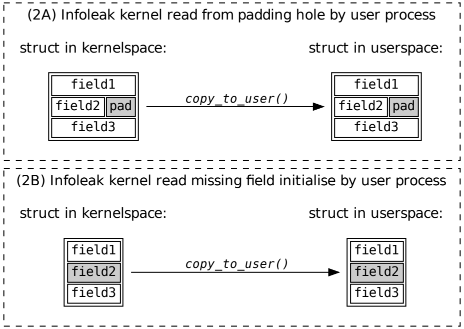


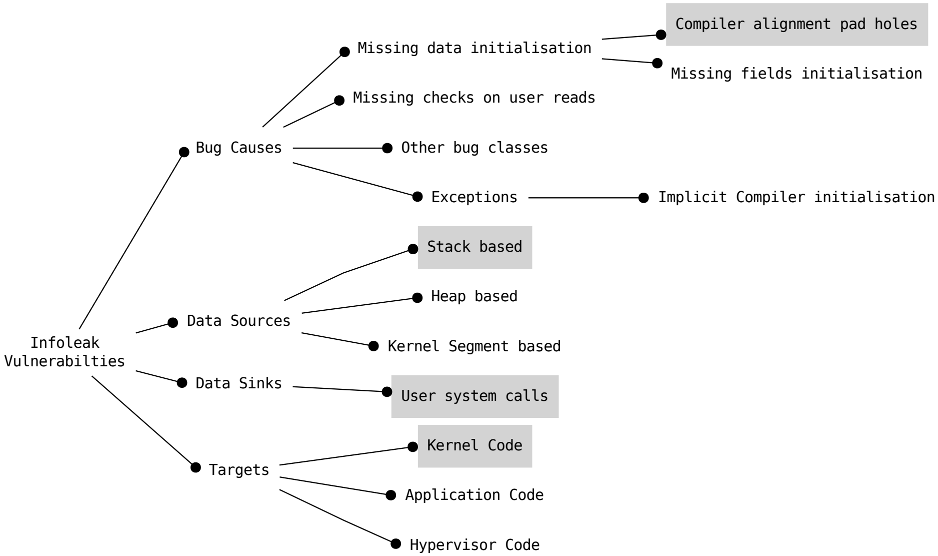
Figure: Directions of data flow in kernel information leaks.

The figure 1 shows a write from a kernel **data source** to a user **data sink** that results in an infoleak due to uninitialised memory.

- Figure 1A: uninitialised memory in compiler padding holes.
- Figure 1B: uninitialised memory in missing fields initialisation.

Classification of Infoleak Vulnerabilities

Our Objective: Stack based infoleaks of kernel code due to compiler alignment holes



Infoleak Analysis: Bug Causes

Infoleaks are the consequence of other kinds of vulnerabilities that lead to disclose the memory layout or contents of the running program. This analyses the causes of infoleaks.

- **Compiler padding holes.** Compilers align data structures for performance reasons [6], this leads the compiler to introduce padding holes between structure fields. Depending on the direction of the information flow, we can identify two situations:
  - **Writes from kernel to user**: Results in an infoleak shown in 1, are our case of study.
  - **Writes from user to kernel**: Results in a kernel write, and out of scope of our work.
- **Missing memory initialisation.** The contents of local variables declared without explicit initialization are indeterminate (C99 [14, Sect. 6.7.8/10]). In practice, local variables get allocated on the stack reusing the memory contents already present on the stack.
- **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].
- **Other bug classes leading to infoleaks.** Other sources of infoleaks not explored in this work, are those related to information available in the environment: `kptr_restrict` mechanism [18] and the hardware: cache and TLB timing attacks [10]
- **Exceptions.**: An example exception is partial variable initialization as in: `struct Type var = {0}` all fields get initialized with zeros (C99 [14, Sect. 6.7.8/19]).

Infoleak Analysis: 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.

Infoleak Detection Technique

Key idea: Semantic patches matching the infoleak vulnerability model.

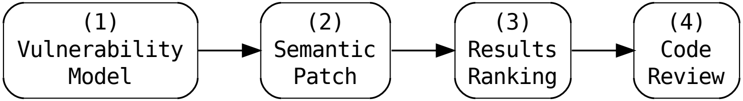


Figure: Infoleak Detection Technique Steps

The main steps of the technique are outlined here:

- Step 1: Analysis of the attack and definition of the vulnerability model.
- Step 2: Design a semantic patch of the vulnerability model to perform static analysis.
- Step 3: Filter and rank the results of static analysis to remove false positives.
- Step **??**: Review and correct the vulnerabilities detected.

Infoleak Detection Technique: 1. 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 1, 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. At source code level this 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*.

Infoleak Detection Technique: 2. Semantic Patch

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] open-source developer-friendly static analysis tool 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 2: Semantic patch (SmPL) for stack based infoleak detection (edited to fit)

For our analysis we develop a Coccinelle semantic patch depicted at listing 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.

Infoleak Detection Technique: 3. Ranking

The results of the execution of the semantic patch discussed at the step 2 contain the potential vulnerabilities ranked according to its likelihood of being a real vulnerability.

- The ranking improves the vulnerability detection rate
- Reduces the amount of required manual work during code audits.

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

For each code location matched by the semantic patch, the following fields are extracted from the match to identify each vulnerability $\mathit{vuln} = (\mathit{function}, \mathit{variable}, \mathit{struct})$.

- The filtering function in equation 3 calculates the size of the infoleak in bytes as the size of the padding holes in the struct.
- The equation 3 determines the relevance of the infoleak and allows to order the results giving a higher relevance to those leaking more bytes.

Evaluation of Infoleak Detection: Existing Infoleak Detection

We performed two evaluation approaches of the infoleak detection technique:

- Existing Infoleak Detection: Introduce known vulnerabilities to check detection performance
- Discovery of New Infoleaks: Evaluate the detection of previously unknown vulnerabilities

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: Statistical performance of stack infoleak detection per kernel version.

The table 2 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.

Evaluation of Infoleak Detection: Discovery of Infoleaks

For evaluation in real world we applied our detection technique to the Linux kernel v3.12.

As a result five new Infoleak vulnerabilities have been uncovered in the Linux kernel:

- CVE-2014-1739 200 bytes infoleak on the local media-device system of the kernel
- CVE-2014-1446 4 bytes infoleak on the local hamradio driver of the kernel
- CVE-2014-1445 2 bytes infoleak on the local wanxl driver of the kernel
- CVE-2014-1444 2 bytes infoleak on the local farsync driver of the kernel
- CVE-unassigned 2 bytes infoleak on the local synclink driver of the kernel

Systems affected by this CVEs

- Most of the above CVE's have been present for 3-5 years
- The wide use of the Linux in embedded systems and consumer appliances: think of Android, cloud services
- This results in large number of running systems affected by these infoleaks.

Applications and Limitations

- Applications
  - Reduce attacks reliability by removing infoleaks
  - Reduce costs involved with Fixes/Maintenance.
  - Error detection: the earlier the better.
- Application Stages
  - **At Release stage** to ensure that less bugs get into the product release.
  - **At Development stage** to avoid introducing errors in early development stage.
  - **At Regression stage** to ensure a known bug is not re-introduced.
- Limitations
  - Static analysis Vulnerability detection is an undecidable problem [13, Rice's theorem].
  - Vulnerabilities not capture by the vulnerability model [21, Fail-safe defaults]

Questions

Thanks, Questions?

Slides are online at:
http://speirofr.appspot.com/category/infoleaks/

References

CVE-2010-4525. kvm: x86: zero kvm_vcpu_events-¿interrupt.pad infoleak.

CVE-2012-0053: Apache information disclosure on response to Bad HTTP Request.

CVE-2013-2147. fix info leak in cciss_ioctl32_passthru(). https://git.kernel.org.

Haogang Chen, Yandong Mao, and Xil Wang. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *APSYS '11*. ACM.

MITRE. Common Weakness Enumeration. CWE-200: Information Exposure.

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

A. Herrero et al. RT-MOVICAB-IDS: Addressing real-time intrusion detection. *FGCS '13*.

C. Cowan et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX-SEC*, 1998.

D. E. Denning et al. Certification of Programs for Secure Information Flow. *C. ACM*, 1977.

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

R. Strackx et al. Breaking the Memory Secrecy Assumption. EUROSEC '09.

M. Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall.

J. E Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. 2008.

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