RandCompile Removing Forensic Gadgets from the Linux Kernel to Combat its Analysis

Fabian Franzen, Andreas Chris Wilhelmer, Jens Grossklags
Technical University of Munich

December 7, 2023
Outline

1. Current State of Memory Forensics
2. How to combat Modern Forensic Tools: RandCompile
3. Evaluation
4. Future Research & Conclusion
Current State of Memory Forensics
Memory-Forensic: The science of deducting information about an operating system state out of a memory dump

- Allows to reason about
  - Process List
  - (Cryptographic-)Secrets
  - IPs/MAC-Addresses of devices in proximity
  - ...

- Complexity depends on available information.
  - Debugging Symbols of operating system
Recent Developments in Linux Memory Forensics

New Challenges for analysts:

- **Structure Layout Randomization** (since 2017)
  - Binary Layout of data structures is modified at compile time.
  - Primarily a Binary Exploitation defense, but effective against forensic tools
Recent Developments in Linux Memory Forensics

New Challenges for analysts:

- **Structure Layout Randomization** (since 2017)
  - Binary Layout of data structures is modified at compile time.
  - Primarily a Binary Exploitation defense, but effective against forensic tools

Research Progress:

- Tools are capable to deal with **Structure Layout Randomization**
- OS-agnostic tools
  - Certain implementation characteristics are shared between OSes
  - Operate with minimal additional information on MacOS, Linux, Windows, and other operating systems
We systematized and grouped last-generation tools by the essential OS artifacts used to enable their analysis (Forensic Gadgets):

- **Special Comm** - "Special" strings allowing easy identification of the process list in the memory dump.
We systematized and grouped last-generation tools by the essential OS artifacts used to enable their analysis (Forensic Gadgets):

- **Special Comm** - ”Special” strings allowing easy identification of the process list in the memory dump.

- **Symbol Tables** - The OS maintains a list of its own functions/global variables for dynamic loading of drivers and or additional functionality (eBPF, ftrace, ...).
We systematized and grouped last-generation tools by the essential OS artifacts used to enable their analysis (Forensic Gadgets):

▶ **Special Comm** - ”Special” strings allowing easy identification of the process list in the memory dump.

▶ **Symbol Tables** - The OS maintaines a list of its own functions/global variables for dynamic loading of drivers and or additional functionality (eBPF, ftrace, ...).

▶ **ABI Constraints** - The compiled kernel code follows predictable patterns revealing location/layout of data structures (i.e. using offset revealing instructions).
We systematized and grouped last-generation tools by the essential OS artifacts used to enable their analysis (Forensic Gadgets):

- **Special Comm** - ”Special” strings allowing easy identification of the process list in the memory dump.
- **Symbol Tables** - The OS maintains a list of its own functions/global variables for dynamic loading of drivers and or additional functionality (eBPF, ftrace, …).
- **ABI Constraints** - The compiled kernel code follows predictable patterns revealing location/layout of data structures (i.e. using offset revealing instructions).
- **Order of Fields** - The data structure layout (especially without Structure Layout Randomization) is foreseeable.
We systematized and grouped last-generation tools by the essential OS artifacts used to enable their analysis (Forensic Gadgets):

- **Special Comm** - ”Special” strings allowing easy identification of the process list in the memory dump.

- **Symbol Tables** - The OS maintaines a list of its own functions/global variables for dynamic loading of drivers and or additional functionality (eBPF, ftrace, ...).

- **ABI Constraints** - The compiled kernel code follows predictable patterns revealing location/layout of data structures (i.e. using offset revealing instructions).

- **Order of Fields** - The data structure layout (especially without Structure Layout Randomization) is foreseeable.

- **Pointer Graph** - The pointers between the kernel objects form a characteristic graph revealing e.g. the process list uniquely out of the set of objects.
## Systematization of Last Generation Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Year</th>
<th>Analysis Subject</th>
<th>FG 1: Special comm</th>
<th>FG 2: Symbol Tables</th>
<th>FG 3: ABI Constraints</th>
<th>FG 4: Order of Fields</th>
<th>FG 5: Pointer Graph</th>
<th>Recovery Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linux-specific</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Katana</td>
<td>2022</td>
<td>Offset Revealing Instructions</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>All structures</td>
</tr>
<tr>
<td>Trustzone Rootkit</td>
<td>2022</td>
<td>Kernel Runtime Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Selected structures</td>
</tr>
<tr>
<td>LogicMem</td>
<td>2022</td>
<td>Kernel Runtime Data</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>Selected structures</td>
</tr>
<tr>
<td>AutoProfile</td>
<td>2021</td>
<td>Offset Revealing Instructions</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>All structures</td>
</tr>
<tr>
<td><strong>OS-agnostic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fossil</td>
<td>2023</td>
<td>Kernel Runtime Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>All structures</td>
</tr>
<tr>
<td>HyperLink</td>
<td>2016</td>
<td>Kernel Runtime Data</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>Selected structures</td>
</tr>
</tbody>
</table>
How to Combat Modern Memory Forensic Tools?
Harden Linux systems against automated forensic analysis
### Design

<table>
<thead>
<tr>
<th>String and Pointer Encryption</th>
<th>FG-1</th>
<th>FG-2</th>
<th>FG-3</th>
<th>FG-4</th>
<th>FG-5</th>
<th>GCC Plugin</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Better Data-Order Randomization</td>
<td>✔</td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Externalize <code>printk</code> Format Strings</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adding Bogus Parameters with Artificial Memory Accesses</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Perform **selected transformations** on the kernel to remove **four out five** forensic gadgets.
  - **two** are **automatically** applied (by a compiler plugin)
  - **two** applied manually in form of **kernel patch**

- **Disclaimer:** Perfect Obfuscation is in general **not possible**! This is a hardening mechanism against **automated tools**.
**ABI Randomization**

**KATANA and AUTOPROFILE target FG 3**

- Offset Revealing Instructions reveal layout of data structures

- ABI mandates calling convention
  - Allows a structural matching of generated machine code with the source code

**Example:**

```plaintext
1  do_stuff(current->mm, current->
    cred, &g);

1  mov rdx, 0xffffffff82019c60
2  mov rax, QWORD PTR gs:0x16d00
3  mov rsi, QWORD PTR [rax+0x10]
4  mov rdi, QWORD PTR [rax+0x440]
5  call ffffffff811bacd0 <do_stuff>
```
**Countermeasures** by RandCompile

- **Shuffle** the order of the arguments at call site and implementation site
- Applied automatically to all functions through a compiler plugin.

**Issues**

- Functions with few parameters have few possibilities for randomization

**Example:**

```plaintext
1. do_stuff(current->cred₁, current->mm ↩ 2, &g₃);

1. mov rdx₆,0xffffffff82019c60
2. mov rax,QWORD PTR gs:0x16d00
3. mov rsi₂,QWORD PTR [rax+0x440]
4. mov rdi₀,QWORD PTR [rax+0x10]
5. call fffffffff811badc0 <do_stuff>
```
ABI Randomization

We can add bogus parameters to functions with few parameters

- This can be undone by an analysis tool that has access to the source code
- Also add bogus assembly code hurting performance

Example:

```c
int64_t bogusstuff[6];
do_stuff(current->cred₁, current->mm
  ↦ 2, bogusstuff[0]₃, &g₄,
  ↦ ⁶);
```

```assembly
1 mov  rcx₁,0xffffffff82019c60
2 mov  r8⁵, QWORD PTR [rsp+0x18]
3 mov  r9⁶, QWORD PTR [rsp+0x28]
4 mov  rax, QWORD PTR gs:0x16d00
5 mov  rsi₀, QWORD PTR [rax+0x440]
6 mov  rdx⁸, QWORD PTR [rsp]
7 mov  rdi₀, QWORD PTR [rax+0x10]
8 call  ffffffff811bacd0 <do_stuff>
```
HYPERLINK and FOSSIL analyse the pointer graph of kernel objects (FG 5).

- e.g. the process information objects are connected by a linked list.
- first process in list contains well-known string (FG 1).

```
init_task
```
```
tasks->next
comm: "swapper/0"
cred
```
```
tasks->next
comm: "init"
cred
```
```
tasks->next
comm: "sh"
cred
```
```
root cred
```

Encrypt Pointers and Strings in process information objects
Store Encryption Key as immediate value in the compiled machine code.
HYPERLINK and FOSSIL analyse the pointer graph of kernel objects (FG 5).

- e.g. the process information objects are connected by a linked list.
- first process in list contains well-known string (FG 1).
- **Encrypt** Pointers and Strings in process information objects

```
init_task
EK(tasks->next)
comm: EK("swapper/0")
cred

EK(tasks->next)
comm: EK("init")
cred

EK(tasks->next)
comm: EK("sh")
cred

root cred
```
HYPERLINK and FOSSIL analyse the pointer graph of kernel objects (FG 5).

- e.g. the process information objects are connected by a linked list.
- first process in list contains well-known string (FG 1).
- Encrypt Pointers and Strings in process information objects
HYPERLINK and FOSSIL analyse the pointer graph of kernel objects (FG 5).

- e.g. the process information objects are connected by a linked list.
- first process in list contains well-known string (FG 1).
- Encrypt Pointers and Strings in process information objects
  - Store Encryption Key as immediate value in the compiled machine code.
Evaluation
We perform the core analysis of Katana with and without RandCompile. Already a single fault during reconstruction causes a fault!
Effectiveness against Kernel Runtime Data Analysis

- Encryption of the string "swapper/0" (FG-1) is most effective.
  - Stops LOGICMEM, Trustzone Rootkit, and HYPERLINK from operating
  - FOSSIL analysis performance is degraded. It depends on the analysts queries.

- Pointer Encryption
  - Degrades analysis opportunities of LOGICMEM, Trustzone Rootkit, and HYPERLINK further
  - Further degrades attack possibilities of FOSSIL
  - Future Work: Encrypt also other kernel pointers
Results using the `lmbench` Microbenchmark (runtimes are normalized to 1):
Results using the \texttt{lmbench} Microbenchmark (runtimes are normalized to 1):

- Less than 1-3 percent overhead on average
Are you applying only sound transformations?
▶ Yes. RandCompile does not change the semantic/core functionality of the Linux kernel.

Does not confidential computing (CC) (like AMD-SEV) mitigate this problem?
▶ RandCompile complements protection of CC approaches. I.e. AMD-SEV expects a Linux kernel to not trust his drivers.

Can this be used as a binary exploitation defense?
▶ Yes. In combination with Control Flow Integrity protections, it makes abusing existing kernel functions in ROP chains harder.

Is it a problem that the defenses are applied at compile time?
▶ Partially. Applying them during runtime would allow for more widespread use. Applying them at compile time adds diversity to the binary layout.
Conclusion
Conclusion

- RandCompile is an obfuscation tool for the Linux Kernel to harden it against various memory forensic tools.
- It is effective against modern forensic analysis tools.
- It completes and extends the Structure Layout Randomization, a mainlined Linux kernel feature.

We have source code!