Rootkit Programming - Final Presentation

LiveDM — Proof of Concept

Tizian Leonhardt, Max van Deurzen

Wintersemester 20/21
February 8, 2021

Agenda

1. Background
    - Dynamic Kernel Memory
    - LiveDM
2. Approach
3. Results
4. Discussion / Questions

- Dynamic kernel memory is...

- Dynamic kernel memory is...
  - ...hard to make sense of – usually, no type information is available

- Dynamic kernel memory is...
  - ...hard to make sense of – usually, no type information is available
  - ...constantly changing – it's dynamic, after all

- Dynamic kernel memory is...
  - ...hard to make sense of – usually, no type information is available
  - ...constantly changing – it's dynamic, after all
  - ...difficult to analyze!

- Dynamic kernel memory is...
  - ...hard to make sense of – usually, no type information is available
  - ...constantly changing – it’s dynamic, after all
  - ...difficult to analyze!
- How can we make analysis easier?

- LiveDM seeks to overcome these issues through Virtual Machine Introspection (VMI)

Background

- LiveDM seeks to overcome these issues through Virtual Machine Introspection (VMI)
  - Monitor the runtime state of a VM

- LiveDM seeks to overcome these issues through Virtual Machine Introspection (VMI)
  - Monitor the runtime state of a VM
  - Without altering the guest OS

Technische Universität München

- LiveDM seeks to overcome these issues through Virtual Machine Introspection (VMI)
  - Monitor the runtime state of a VM
  - Without altering the guest OS
- Memory allocation events can be intercepted

- LiveDM seeks to overcome these issues through Virtual Machine Introspection (VMI)
  - Monitor the runtime state of a VM
  - Without altering the guest OS
- Memory allocation events can be intercepted
- Going from there, LiveDM is able to create a memory map

- LiveDM seeks to overcome these issues through Virtual Machine Introspection (VMI)
  - Monitor the runtime state of a VM
  - Without altering the guest OS
- Memory allocation events can be intercepted
- Going from there, LiveDM is able to create a memory map
  - This map includes type information!

- Three distinct stages to create the mapping:

Background

LiveDM — Overview

- Three distinct stages to create the mapping:
  1. Gathering of necessary values

- Three distinct stages to create the mapping:
  1. Gathering of necessary values
  2. Determining the scope of memory monitoring

- Three distinct stages to create the mapping:
  1. Gathering of necessary values
  2. Determining the scope of memory monitoring
  3. Performing type interpretation

- Stage 1 is comprised of...
  - ...intercepting a set of memory allocation/deallocation functions

Technische Universität München

- Stage 1 is comprised of...
  - ...intercepting a set of memory allocation/deallocation functions
  - ...retrieving the requested allocation size, as well as the return value

- Stage 1 is comprised of...
  - ...intercepting a set of memory allocation/deallocation functions
  - ...retrieving the requested allocation size, as well as the return value
  - ...identifying the caller (call site) through the stack's return address

- In stage 2, the scope of memory monitoring is chosen

- In stage 2, the scope of memory monitoring is chosen
  - Offer snapshots of the memory map (containing type and size for allocated memory)
    - We offer this in our PoC (`rk-print-mem` and `rk-data <address>`)

Technische Universität München

- In stage 2, the scope of memory monitoring is chosen
  - Offer snapshots of the memory map (containing type and size for allocated memory)
    - We offer this in our PoC (`rk-print-mem` and `rk-data <address>`)
  - Trace every memory (write) access on known (vulnerable) memory blocks
    - We are able to showcase this in a small demo

- In stage 3, the caller's address is translated into a type

- In stage 3, the caller's address is translated into a type
  - Relies on instrumenting GCC to retrieve abstract syntax tree (AST)

- Why do we need this information? Possible answers include...

- Why do we need this information? Possible answers include...
    1. To make dynamic memory less transparent

- Why do we need this information? Possible answers include...
  1. To make dynamic memory less transparent
  2. To utilize this information for debugging

- Why do we need this information? Possible answers include...
  1. To make dynamic memory less transparent
  2. To utilize this information for debugging
  3. To utilize this information for rootkit detection

Approach

1. Background
2. Approach
   - Tools
   - Implementing stage 1 — Gathering of necessary values
   - Implementing stage 3 — Performing type interpretation
   - Implementing stage 2 — Determining the scope of memory monitoring
3. Results
4. Discussion / Questions

- Since introspection techniques are required, we need a VMM

- Since introspection techniques are required, we need a VMM
  - Xen
  - KVM
  - QEMU (in vivo introspection using GDB)
  - ...

- Intercepting allocations is easy: non-blocking **breakpoints**

Approach

Implementing stage 1 — Gathering of necessary values

- Intercepting allocations is easy: non-blocking **breakpoints**
  - Break on function entry and exit

Approach

Implementing stage 1 — Gathering of necessary values

- Intercepting allocations is easy: non-blocking **breakpoints**
  - Break on function entry and exit
    - At entry we extract allocation size
    - At exit we extract return value (base address of allocation)

Approach

Implementing stage 1 — Gathering of necessary values

- Intercepting allocations is easy: non-blocking **breakpoints**
    - Break on function entry and exit
        - At entry we extract allocation size
        - At exit we extract return value (base address of allocation)
    - Has a significant performance overhead, but system is still usable

Approach

Implementing stage 1 — Gathering of necessary values

- Intercepting allocations is easy: non-blocking **breakpoints**
  - Break on function entry and exit
    - At entry we extract allocation size
    - At exit we extract return value (base address of allocation)
  - Has a significant performance overhead, but system is still usable
  - Possible improvement: hardware breakpoints

Approach

Implementing stage 1 — Gathering of necessary values

- Intercepting allocations is easy: non-blocking **breakpoints**
  - Break on function entry and exit
    - At entry we extract allocation size
    - At exit we extract return value (base address of allocation)
  - Has a significant performance overhead, but system is still usable
  - Possible improvement: hardware breakpoints
    - Limited to a small amount

Approach

Implementing stage 1 — Gathering of necessary values

- To retrieve the size and return value of each allocation, we can rely on the System V calling convention

- To retrieve the size and return value of each allocation, we can rely on the System V calling convention
  - As the size is not always the first argument, we build a dictionary:

```
    break_arg = {
        "kmem_cache_alloc_trace": "rdx",
        "kmalloc_order": "rdi"
        [...]
    }
```

Approach

Implementing stage 1 — Gathering of necessary values

- To retrieve the size and return value of each allocation, we can rely on the System V calling convention
  - As the size is not always the first argument, we build a dictionary:

```
break_arg = {
    "kmem_cache_alloc_trace": "rdx",
    "kmalloc_order": "rdi"
    [...]
}
```

  - Return values are gathered by additionally breaking on return instructions

- To retrieve the size and return value of each allocation, we can rely on the System V calling convention
  - As the size is not always the first argument, we build a dictionary:

```
    break_arg = {
        "kmem_cache_alloc_trace": "rdx",
        "kmalloc_order": "rdi"
        [...]
    }
```

  - Return values are gathered by additionally breaking on return instructions
    - Look for `ret{,q}` instruction's offset from function entry in the disassembly

Approach

Implementing stage 1 — Gathering of necessary values

- To retrieve the size and return value of each allocation, we can rely on the System V calling convention
  - As the size is not always the first argument, we build a dictionary:

```
    break_arg = {
        "kmem_cache_alloc_trace": "rdx",
        "kmalloc_order": "rdi"
        [...]
    }
```

- Return values are gathered by additionally breaking on return instructions
  - Look for `ret{,q}` instruction's offset from function entry in the disassembly
  - Break on `<function entry> + <ret offset>`

Approach

Implementing stage 1 — Gathering of necessary values

- To retrieve the size and return value of each allocation, we can rely on the System V calling convention
  - As the size is not always the first argument, we build a dictionary:

```
break_arg = {
    "kmem_cache_alloc_trace": "rdx",
    "kmalloc_order": "rdi"
    [...]
}
```

  - Return values are gathered by additionally breaking on return instructions
    - Look for `ret{,q}` instruction's offset from function entry in the disassembly
    - Break on `<function entry> + <ret offset>`
    - Retrieve return value from `$rax`

Approach

Implementing stage 3 — Performing type interpretation

- Translation of call sites to types

Approach

Implementing stage 3 — Performing type interpretation

- Translation of call sites to types
- Possible approaches:

Approach

Implementing stage 3 — Performing type interpretation

- Translation of call sites to types
- Possible approaches:
  - Instrumenting `gcc` to extract AST (LiveDM)

Approach

Implementing stage 3 — Performing type interpretation

- Translation of call sites to types
- Possible approaches:
  - Instrumenting `gcc` to extract AST (LiveDM)
  - Use `clang` to generate an AST

Approach

Implementing stage 3 — Performing type interpretation

- Translation of call sites to types
- Possible approaches:
  - Instrumenting `gcc` to extract AST (LiveDM)
  - Use `clang` to generate an AST
  - Utilize GDB's `whatis` command to statically pre-compute type dictionary

- Process for statically generating the type dictionary: [1]

---

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences
  3. Extract the assigned-to symbol name at call site

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences
  3. Extract the assigned-to symbol name at call site
  4. Execute `whatis` on every assigned-to symbol

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences
  3. Extract the assigned-to symbol name at call site
  4. Execute `whatis` on every assigned-to symbol
     - Assumption: debug symbols for current kernel sources are available

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences
  3. Extract the assigned-to symbol name at call site
  4. Execute `whatis` on every assigned-to symbol
     - Assumption: debug symbols for current kernel sources are available
     - Compound type access chains (e.g., `desc->inbuf`) have to be recursively resolved

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences
  3. Extract the assigned-to symbol name at call site
  4. Execute `whatis` on every assigned-to symbol
     - Assumption: debug symbols for current kernel sources are available
     - Compound type access chains (e.g., `desc->inbuf`) have to be recursively resolved
     - We only require the type of the last dereferenced field, as that is what's being assigned to

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Technische Universität München

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences
  3. Extract the assigned-to symbol name at call site
  4. Execute `whatis` on every assigned-to symbol
     - Assumption: debug symbols for current kernel sources are available
     - Compound type access chains (e.g., `desc->inbuf`) have to be recursively resolved
     - We only require the type of the last dereferenced field, as that is what's being assigned to
  5. Store the results in a dictionary

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences
  3. Extract the assigned-to symbol name at call site
  4. Execute `whatis` on every assigned-to symbol
     - Assumption: debug symbols for current kernel sources are available
     - Compound type access chains (e.g., `desc->inbuf`) have to be recursively resolved
     - We only require the type of the last dereferenced field, as that is what's being assigned to
  5. Store the results in a dictionary
  6. Use this precompiled type dictionary in our runtime script

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Process for statically generating the type dictionary: [1]
  1. Find all occurences of function calls we are interested in using `cscope`
  2. Iterate over the generated occurences
  3. Extract the assigned-to symbol name at call site
  4. Execute `whatis` on every assigned-to symbol
     - Assumption: debug symbols for current kernel sources are available
     - Compound type access chains (e.g., `desc->inbuf`) have to be recursively resolved
     - We only require the type of the last dereferenced field, as that is what's being assigned to
  5. Store the results in a dictionary
  6. Use this precompiled type dictionary in our runtime script

```
    "./arch/x86/kernel/e820.c:675": "type = struct e820_table *",
    "./arch/x86/kernel/e820.c:681": "type = struct e820_table *",
  [...]
```

[1] Fully automated, since specific to kernel sources version, build options, and compiler optimizations

Approach

Implementing stage 3 — Performing type interpretation

- Once a breakpoint is encountered, we can walk the stack with gdb...

- Once a breakpoint is encountered, we can walk the stack with gdb...

```
    #0  __kmalloc (size=168, flags=6291456) at ./mm/slub.c:3784
    #1  0xffffffffa9384095 in kmalloc (flags=<optimized out>, size=<optimized out>) at ./include/linux/slab.h:520
    #2  bio_alloc_bioset (gfp_mask=6291456, nr_iovecs=<optimized out>, bs=0x0) at ./block/bio.c:452
```

Approach

Implementing stage 3 — Performing type interpretation

- Once a breakpoint is encountered, we can walk the stack with gdb...

```
#0  __kmalloc (size=168, flags=6291456) at ./mm/slub.c:3784
#1  0xffffffffa9384095 in kmalloc (flags=<optimized out>, size=<optimized out>) at ./include/linux/slab.h:520
#2  bio_alloc_bioset (gfp_mask=6291456, nr_iovecs=<optimized out>, bs=0x0) at ./block/bio.c:452
```

- ...and match the `file:line` descriptor to a type without expensive computations

Approach

Implementing stage 2 — Determining the scope of memory monitoring

1. Snapshot-based approach

Approach

Implementing stage 2 — Determining the scope of memory monitoring

1. Snapshot-based approach
   - Since we already store everything gathered, this is readily available

Approach

Implementing stage 2 — Determining the scope of memory monitoring

1. Snapshot-based approach
   - Since we already store everything gathered, this is readily available
   - Live allocations can be listed with `rk-print-mem` and interpreted with `rk-data <address>`:

```
  > rk-print-mem
  type: struct task_struct *, size: 3776 B, address: 0xffff8e72b87ce740, call site: ./kernel/fork.c:807
  type: struct fdtable *, size: 56 B, address: 0xffff8e72b84104c0, call site: ./fs/file.c:111
```

```
> rk-data 0xffff8e72b84104c0
  resolving 0xffff8e72b84104c0 to type = struct fdtable *

  $17 = {
    max_fds = 256,
    fd = 0xffff8e72b8ea4800,
    close_on_exec = 0xffff8e72b8411800,
    open_fds = 0xffff8e72b84117e0,
    full_fds_bits = 0xffff8e72b8411820,
    rcu = {
    next = 0x0,
    func = 0x0
    }
  }
```

Approach

Implementing stage 2 — Determining the scope of memory monitoring

2. Memory-access tracing

2. Memory-access tracing
    - Would require some advanced techniques (e.g., page unmapping) for full coverage

Approach

Implementing stage 2 — Determining the scope of memory monitoring

2. Memory-access tracing
   - Would require some advanced techniques (e.g., page unmapping) for full coverage
   - Not feasible within the given time frame

Technische Universität München

2. Memory-access tracing
    - Would require some advanced techniques (e.g., page unmapping) for full coverage
    - Not feasible within the given time frame
    - Instead, we will demonstrate a small example based on *hardware* watchpoints
        - Warn when critical values are written to traced blocks

- We will demonstrate the output in a running system now:

```
Allocating ('type = struct elf64_phdr *', 616, './fs/binfmt_elf.c:441') at 0xffff8d96b8857000
Allocating ('type = char *', 28, './fs/binfmt_elf.c:762') at 0xffff8d96ba5d98e0
Allocating ('type = struct elf64_phdr *', 504, './fs/binfmt_elf.c:441') at 0xffff8d96bb4b1e00
Allocating ('type = void *', 168, './block/bio.c:452') at 0xffff8d96ba14bcc0
```

Technische Universität München

- We will demonstrate the rootkit detection in a running system now:

```
//inside the vm, rootkit is loaded
> make_me_root
```

```
((((struct task_struct *)0xffff8d96bb6849c0)->real_cred)->uid) changed from val = 1000 to val = 0
WARNING: critical value 0 set to ((((struct task_struct *)0xffff8d96bb6849c0)->real_cred)->uid)
```

Discussion / Questions

Technische Universität München

...