Address Watchpoints

Instrument data, not code.

Peter Goodman, Ashvin Goel

Advanced Host-Level Security (AHLS)
May 14, 2014

Goals of Project

- Goal is to protect operating system kernels
- Protect kernel against module code
  - Buggy modules
    - Expose kernel to attack
    - Need to detect disallowed behavior
  - Malicious modules (rootkits)
    - Often installed using social engineering
    - Have complete access to kernel code and data
    - Need to detect anomalous behavior
- Requires understanding module behavior
  - What they do, what they should be allowed to do

Approach

- Instrument all module code at runtime using Dynamic Binary Translation (DBT)
  - Rewrite module code on-the-fly during execution
  - No source code or debug information required
  - Operates at instruction / basic block granularity
  - Complete control over a module's execution
  - Built a prototype system called Granary
    - **Think "Valgrind", but for the Linux kernel**

- Two key ideas to securing modules
  - Interpose on module/kernel interface with **wrappers**
  - Verify memory accesses with **watchpoints**

Why DBT ~~is SCARY~~ Doesn't Always Fit the Problem

- Too low level
  - Hard to write instrumentation that is both *safe* and *efficient* for injection into module code
    - Often have to special-case tricky instructions
    - Need to worry about re-entrancy
    - Must maintain illusion that DBT system not there
- Wrong abstraction
  - In practice, don't care about instructions being executed, care about what/how data is accessed
  - E.g. data race detector, memory access bugs
- Binary means binary
  - All code instrumented or not... always in the same way

We Want Data-Centric Instrumentation

Types of applications that we want to make, but are hard to do with run of the mill DBT systems:

- Buffer overflow detectors
- Use-after-free, read-before-write, double-free, etc
- Selective shadow memory
- Object-specific invariant checking
- Memory leak detector
- Accurate working set estimation
- Access pattern detector / recorder

Ideally, we want

1. You tell the hardware what objects your tool cares about

2. The hardware tells your tool when the memory of those objects is accessed

Current Solutions

- Hardware watchpoints
  - Too scarce to be useful at a large scale
- Hardware protection domains
  - Only available on specialized hardware
- Page protection
  - Too coarse-grained
- Shadow memory
  - "All or nothing", even memory you don't care about needs to be shadowed

Key Insight

- Hard to track objects, easy to track addresses!
  - Taint object addresses so that accesses to "interesting" objects always raise a fault.
    - **"Address watchpoints"**
  - Relies on x86-64 48-bit address implementation in which 16 bits are "free" to be changed.
  - Kind of like getting a segfault when you read a bad pointer.
- Interpose on fault when object is accessed.
  - Use the tainted bits to identify i) what object is accessed, and ii) what do about it.

Example (1)

```
struct sk_buff *skb = alloc_skb(skb_size,
                                 GFP_KERNEL);




...
dma_map_single(..., skb->data, skb->len,
               DMA_TO_DEVICE);
```

Example (2)

```
struct sk_buff *skb = alloc_skb(skb_size,
                                GFP_KERNEL);

skb = add_watchpoint(skb, <meta-data>);



...
dma_map_single(..., skb->data, skb->len,
               DMA_TO_DEVICE);
```

Example (3)

```
struct sk_buff *skb = alloc_skb(skb_size,
                                GFP_KERNEL);
skb == 0xFFFFFFFFA092600
skb = add_watchpoint(skb, <meta-data>);



...
dma_map_single(..., skb->data, skb->len,
               DMA_TO_DEVICE);
```

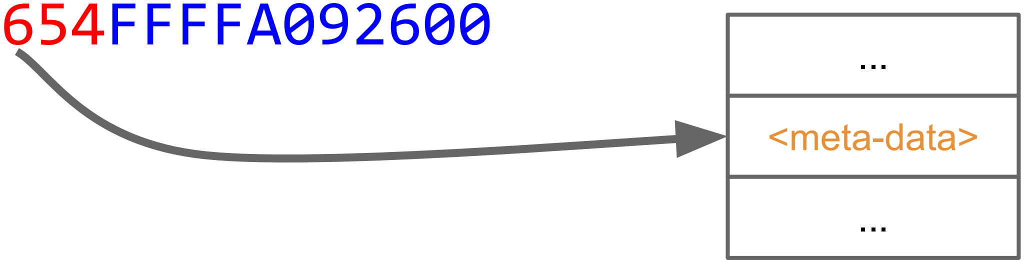
Example (4)

```
struct sk_buff *skb = alloc_skb(skb_size,
                                GFP_KERNEL);
skb == 0xFFFFFFFFA092600
skb = add_watchpoint(skb, <meta-data>);
skb == 0x7654FFFFA092600

...
dma_map_single(..., skb->data, skb->len,
               DMA_TO_DEVICE);
```

Example (5)

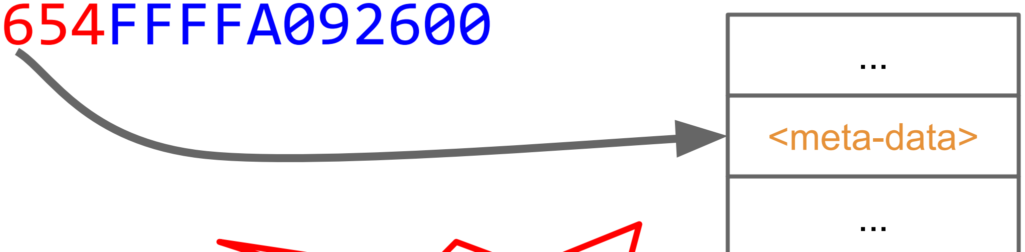
```
struct sk_buff *skb = alloc_skb(skb_size,
                                GFP_KERNEL);
skb == 0xFFFFFFFFA092600
skb = add_watchpoint(skb, <meta-data>);
skb == 0x7654FFFFA092600
```




```
...
dma_map_single(..., skb->data, skb->len,
               DMA_TO_DEVICE);
```

Example (6)

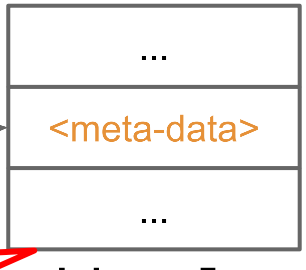
```
struct sk_buff *skb = alloc_skb(skb_size,
                                GFP_KERNEL);
skb == 0xFFFFFFFFA092600
skb = add_watchpoint(skb, <meta-data>);
skb == 0x7654FFFFA092600
```




```
...
dma_map_single(..., skb->data, skb->len,
               DMA_TO_DEVICE);
```

Example (7)

```
struct sk_buff *skb = alloc_skb(skb_size,
                                GFP_KERNEL);
skb == 0xFFFFFFFFA092600
skb = add_watchpoint(skb, <meta-data>);
skb == 0x7654FFFFA092600
```




```
...
dma_map_single(..., skb->data, skb->len,
               DMA_TO_DEVICE);
do_general_protection (GP fault handler)
... regs->regs[...] == 0x7654FFFFA0926E0
```

Challenges of Address Watchpoints

- Efficiency
  - Faults are expensive, how can we minimize them?
- Correctness
  - Need to temporarily "untaint" and then re-taint address to get control back.
  - Handle user addresses, physical addresses.
- Usage
  - When and how to insert calls to `add_watchpoint`?

Efficiency

- Strawman approach
  - Take fault on each watched address, very expensive
- Existing DBT approaches
  - Instrument all code, dispatch callback on watched address, avoids faults, but still expensive
- Address watchpoint approach
  - Take fault on first access to watched address
  - Turn on DBT, and then turn it off when watched addresses are not expected to be accessed
    - Take advantage of locality of accesses to provide efficiency

Correctness

- User addresses
  - Detect user addresses by using the kernel's "exception table" mechanism
  - Interesting benefit: can detect uses of user addresses that do not use the special `copy_to_user` / `copy_from_user` functions
- Physical addresses
  - Need to special case
    - Virtual-to-physical address translation
    - Things that hash virtual address
  - Open problem
    - Lose taint when going virt -> phys -> virt.

Usage

- When should you add an address watchpoint, and how do you do it?
  - Identify "sources" of objects, e.g.. type-specific allocators, calls to generic allocators.
  - Interpose and replace allocated address with a watched address.
  - Attach meta-data to the watched address, every time the tainted address or an address derived from it is accessed, we can get the meta-data back!
  - Create a callback function that operates on a watched address and its meta-data.

Implemented Address Watchpoints

Implemented address watchpoints [HotDep'13] using Granary DBT system.

Made some applications:

- Buffer overflow detector
- Use-after-free, read-before-write
- Memory leak detector

Still, Things Weren't Perfect

- Hard to implement the address watchpoints instrumentation
  - Granary didn't have the "right" interface for easily getting at the data being accessed
  - Had to special case some instructions
  - Poor user space support
  - Long-standing bug went undetected
- Infrastructure useful beyond watchpoints
  - Undergrad wanted to make shadow memory system, duplicated most of watchpoints code because the hard part of the memory access instrumentation was "done"

To The Future, And Beyond!

- Address watchpoints gives us data selectivity; we also want code selectivity:
  - Binary still means binary: watchpoint "fires" or it doesn't, regardless of where memory is accessed
  - Want context-specific firing
    - E.g. fire only when access inside critical section
- Better infrastructure
  - Throw away the prototype (Granary)
    - Started work on Granary+ in January 2014
  - Flexible "virtual registers" system
    - Makes all kinds of instrumentation easier™
    - Key success factor of PIN, Valgrind


THAT'S THE TALK
memegenerator.net