STACKJACKING AND OTHER KERNEL NONSENSE

JON OBERHEIDE & DAN ROSENBERG
From arch/alpha/kernel/osf_sys.c:

SYSCALL_DEFINE4(osf_wait4, pid_t, pid, int __user *, ustatus, int, options, struct rusage32 __user *, ur) {
    struct rusage r;
    long ret, err;
    mm_segment_t old_fs;

    if (!ur)
        return sys_wait4(pid, ustatus, options, NULL);

    old_fs = get_fs();

    set_fs(KERNEL_DS);
    ret = sys_wait4(pid, ustatus, options, &r);
    set_fs(old_fs);

    ...
Btw, and you may not like this, since you are so focused on security, one reason I refuse to bother with the whole security circus is that I think it glorifies - and thus encourages - the wrong behavior.

It makes "heroes" out of security people, as if the people who don't just fix normal bugs aren't as important.

In fact, all the boring normal bugs are _way_ more important, just because there's a lot more of them. I don't think some spectacular security hole should be glorified or cared about as being any more "special" than a random spectacular crash due to bad locking.

Security is hard when upstream ignores the problems...and solutions!
“I get excited every time I see a conference add requirements to their talk selection along the lines of 'exploitation presentations must be against grsecurity/PaX' -- but then there never ends up being any presentations of this kind.”

– spender pratt
PENULTIMATE MOTIVATION
AGENDA

• A review of Linux kernel security

• Exploiting grsecurity/PaX kernels

• Stackjacking 2: Electric Boogaloo
DECADE OF KERNEL SECURITY

Vulnerabilities by CVSS severity

- 2000: 14
- 2001: 9
- 2002: 9
- 2003: 26
- 2004: 16
- 2005: 52
- 2006: 28
- 2007: 13
- 2008: 34
- 2009: 57

Stackjacking and Other Kernel Nonsense – Jon Oberheide / Dan Rosenberg – SummerCon 2011
HOW ABOUT LAST YEAR?

- 142 CVE's assigned
  - 30% worse than the previous worst year (2009)
  - Based on public CVE requests, issues tracked at Red Hat Bugzilla, and Eugene's tagged git tree
  - Missing dozens of non-CVE vulnerabilities (i.e. the “Dan Carpenter factor”)
- 61 (43%) discovered by six people
  - Kees (4), Brad (3), Tavis (7), Vasily (4), Dan (37), Nelson (6)
DAN EATS KERNEL BUGS LIKE...
BREAKDOWN BY IMPACT

- Bypass: 1
- DOS: 65
- Info: 30
- Priv Esc?: 7
- Priv Esc: 26
- Nothing: 13
static int inet_diag_bc_audit(const void *bytecode, int bytecode_len)
{
    const unsigned char *bc = bytecode;
    int len = bytecode_len;

    while (len > 0) {
        struct inet_diag_bc_op *op = (struct inet_diag_bc_op *)bc;

        switch (op->code) {
            ...
            case INET_DIAG_BC_JMP:
                if (op->no < 4 || op->no > len + 4)
                    return -EINVAL;
                if (op->no < len && !valid_cc(bytecode, bytecode_len, len - op->no))
                    return -EINVAL;
                break;
            ...
        }

        bc += op->yes;
        len -= op->yes;
    }

    return len == 0 ? 0 : -EINVAL;
}
FUN EXPLOITS OF 2010

- full-nelson.c
  - Combined three vulns to get a NULL write

- half-nelson.c
  - First Linux kernel stack overflow (not buffer overflow) exploit

- linux-rds-exploit.c
  - Arbitrary write in RDS packet family

- i-CAN-haz-MODHARDEN.c
  - SLUB overflow in CAN packet family

- american-sign-language.c
  - Exploit payload written in ACPI's ASL/AML
VANILLA EXPLOITATION

- Writes to known addresses (IDT)
- Function pointer overwrites
- Redirecting control flow to userspace
- Influencing privesc-related kernel data (eg. credentials structures)
- Relying on kallsyms and other info
MOST EXPLOITED VANILLA
AGENDA

• A review of Linux kernel security

• Exploiting grsecurity/PaX kernels

• Stackjacking 2: Electric Boogaloo
GRSECURITY/PAX

- grsecurity/PaX
  - KERNEXEC
    - Prevent the introduction of new executable code
  - UDEREF
    - Prevent invalid userspace pointer dereferences
  - HIDESYM
    - Hide info that may be useful to an attacker (kallsyms, slabinfo, kernel address leaks, etc)
  - MODHARDEN
    - Prevent auto-loading of crappy unused packet families (CAN, RDS, econet, etc)
THE MAIN EVENT

- A technique we call stackjacking

- Enables the bypass of common grsecurity/PaX configurations with common exploit primitives

- Independently discovered, collaboratively exploited, with slightly different techniques
PLAN OF ATTACK

STACK JACKING

OVERVIEW

PRIMITIVES

ROOT

???

???

???
TARGET KERNEL ASSUMPTIONS

- Hardened kernel with grsec/PaX
  - Config level GRKERNSEC_HIGH
  - KERNEXEC
  - UDEREF
  - HIDESYM
  - MODHARDEN
  - Etc...
STRONG ASSUMPTIONS

• Let's make some extra assumptions
  • We like a challenge, and these are assumptions that may possibly be obtainable now or in the future

• Stronger target assumptions
  • Zero knowledge of kernel address space
  • Fully randomized kernel text/data
  • Cannot introduce new code into kernel address space
  • Cannot modify kernel control flow (eg. data-only)
ATTACK ASSUMPTION #1

• Assumption: arbitrary kmem write
  • A common kernel exploitation primitive
  • Examples: RDS, MCAST_MSFILTER
  • Other vulns can be turned into writes, e.g. overflowing into a pointer that's written to

• Wut?
  • “You mean I can't escalate privs with an arbitrary kernel memory write normally?” NOPE.
ARBITRARY WRITE WHERE?

No clue where to write! Exploitation is infeasible.
NEED TO KNOW SOMETHING

- One way: arbitrary kmem disclosure
  - procfs (2005)
  - sctp (2008)
  - move_pages (2009)
  - pktcdvd (2010)

- Just dump entire address space!
  - But these are rare!
  - And in many instances, mitigated by grsec/PaX
SYSCALL_DEFINE3(osf_sysinfo, int, command, char __user *, buf, long, count)
{

...  
  long len, err = -EINVAL;
...

  len = strlen(res)+1;
  if (len > count)
      len = count;
  if (copy_to_user(buf, res, len))
      err = -EFAULT;

...  
}
SYSCALL_DEFINE5(osf_getsysinfo, unsigned long, op, void __user *,
buffer,
unsigned long, nbytes, int __user *, start, void __user *, arg)
{
...
switch (op) {
...
    case GSI_GET_HWRPB:
        if (nbytes < sizeof(*hwrpb))
            return -EINVAL;
        if (copy_to_user(buffer, hwrpb, nbytes) != 0)
            return -EFAULT;
        return 1;
    ...
}
static __noinline int ieee80211_ioctl_getchaninfo(struct ieee80211vap *vap, struct ieee80211req *ireq)
{
    struct ieee80211com *ic = vap->iv_ic;
    int space;

    space = __offsetof(struct ieee80211req_chaninfo, ic_chans[ic->ic_nchans]);
    if (space > ireq->i_len)
        space = ireq->i_len;
    /* XXX assumes compatible layout */
    return copyout(&ic->ic_nchans, ireq->i_data, space);
}
SOMETHING MORE COMMON?

• How about a more common vuln?
• Hints…
  • Widely considered to be a useless vulnerability
  • Commonly assigned a CVSS score of 1.9 (low)
  • 25+ such vulnerabilities reported in 2010
  • Often referred to as a Dan Rosenbug
• Can you guess it???
THE ANSWER IS...

KERNEL STACK MEMORY DISCLOSURES!

Initializing memory makes it a predictable value for attackers. Keep memory uninitialized for extra randomization and obfuscation.
THANKS FOR THE TIP DDZ!
LINUX KERNEL STACKS

- Each userspace thread is allocated a kernel stack
- Stores stack frames for kernel syscalls and other metadata
- Most commonly 8k, some distros use 4k
  - \( \text{THREAD SIZE} = 2 \times \text{PAGE SIZE} = 2 \times 4086 = 8192 \)
STACK MEM DISCLOSURES

• **Kstack mem disclosures**
  - Leak of memory from the kernel stack to userspace

• **Common cause**
  - Copying a struct on the kstack back to userspace with uninitialized fields
  - Improper initialization/memset, forgetting member assignment, structure padding/holes
  - A frequent occurrence, especially in compat
STACK MEM DISCLOSURES

1) process makes syscall and leaves sensitive data on kstack

2) kstack is reused on subsequent syscall and struct overlaps with sensitive data

3) foo struct is copied to userspace, leaking 4 bytes of kstack through uninitialized foo.leak member

```c
struct foo {
    uint32_t bar;
    uint32_t leak;
    uint32_t baz;
};

syscall() {
    struct foo;
    foo.bar = 1;
    foo.baz = 2;
    copy_to_user(foo);
}
```
PLAN OF ATTACK

STACK JACKING OVERVIEW

Arbitrary write
Kstack disclosure

???

ROOT

???

???
WHAT'S USEFUL ON KSTACK?

• Leak data off kstack?
  • Sensitive data left behind? Not really...

• Leak addresses off kstack?
  • Sensitive addresses left behind? Maybe...
    • Pointers to known structures could be exploited
    • *** Too specific of an attack! ***

• Need something more general
  • kstack disclosures differ widely in size/offsets
• How about a leaking an address that:
  • Is stored on the stack; and
  • Points to an address on the stack

• These are pretty common
  • Eg. pointers to local stack vars, saved ebp, etc

• But what does this gain us?
KSTACK SELF-DISCOVERY

- If we can leak an pointer to the kstack off the kstack, we can calculate the base address of the kstack

```
kstack_base = addr & ~(THREAD_SIZE - 1);
kstack_base = 0xcdef1234 & ~(8192 - 1)
kstack_base = 0xcdef0000
```

We call this **kstack self-discovery**
• Not all kstack disclosures are alike
  • May only leak a few bytes, non-consecutive
  • How do we effectively self-discover?

• Manual analysis
  • Figure out where kstack leak overlaps addresses

• Automatic analysis
  • libkstack
PLAN OF ATTACK

STACK JACKING OVERVIEW

STACK SELF-DISCOVERY
- Manual analysis
- Auto with libkstack

Arbitrary write
Kstack disclosure

ROOT

???

???
NO LONGER DARKNESS

We can self-discover kstack address! Exploitation is...maybe feasible?

A random pinpoint of light!
BEFORE SOLAR GOT HIS FLAIR
THE NEXT STEP

- We now have a tiny island
  - Use arbitrary write to modify anything on kstack
- Where to write?
  - Pointers, data, metadata on kstack
- What to write?
  - No userspace addrs (UDEREF), limited kernel
- Game over? Not yet!
Anything else of interest on the kstack???

start of stack

stack pointer

grows down

unused

thread_info

current_thread_info

4k/8k stack

low address

high address

thread_info struct stashed at base of kstack!
THREAD_INFO CANDIDATES

```c
struct thread_info {
    struct task_struct *task;
    struct exec_domain *exec_domain;
    __u32 flags;
    __u32 status;
    __u32 cpu;
    int preempt_count;
    mm_segment_t addr_limit;
    struct restart_block restart_block;
    void __user *sysenter_return;

    #ifdef CONFIG_X86_32
    unsigned long previous_esp;
    __u8 supervisor_stack;
    #endif

    int uaccess_err;
};
```

- What can we modify within `thread_info` to escalate privs?
ATTACKING TASK_STRUCT

struct thread_info {
    struct task_struct *task;
    ...
};

struct task_struct {
    ...
    const struct cred *real_cred;
    const struct cred *cred;
    ...
};

struct cred {
    ...
    uid_t uid;
    gid_t gid;
    ...
};

• task_struct->creds?
  • Modify creds of our process directly to escalate privileges?
  • But in order to write task_struct->creds, we need to know the address of task_struct!
  • If we could read the address of task_struct off the end of the kstack, we might win!
CONNECTING THE DOTS

Expanding our visibility
If we can read off the kstack, we can find task_struct/creds!
ATTACKING TASK_STRUCT

• We have write+kleak
  • Can we turn this into an arbitrary read?

• If we can get arbitrary read:
  • Read base of kstack to find address of task_struct
  • Read task_struct to find address of creds struct
  • Write into creds struct to set uids/gids/caps
  • Spawn a root shell!
PLAN OF ATTACK

STACK JACKING OVERVIEW

STACK SELF-DISCOVERY
- Manual analysis
- Auto with libkstack

STACK GROPING
- ???

STACK JACKING
- Read thread / task
- Overwrite creds

ROOT

Arbitrary write
Kstack disclosure
struct thread_info {
  struct task_struct  *task;
  struct exec_domain  *exec_domain;
  __u32 flags;
  __u32 status;
  __u32 cpu;
  int preempt_count;
  mm_segment_t addr_limit;
  struct restart_block restart_block;
  void __user *sysenter_return;
  #ifdef CONFIG_X86_32
  unsigned long previous_esp;
  __u8 supervisor_stack;
  #endif
  int uaccess_err;
};
PAX_UDEREF

- Strict user/kernel separation using segmentation
- Reload segment registers at kernel traps, used during copy operations
  - Fault on invalid access
- Use `%gs` register to keep track of segment for source/dest of copy
- `set_fs(KERNEL_DS)` sets `addr_limit` and reloads `%gs` register to contain `__KERNEL_DS` segment selector
NO MORE EASY ROOT

- Writing KERNEL_DS to addr_limit is no longer sufficient
- Access checks on pointers will pass, but we'll still fault in copy functions because of incorrect segment registers
- But, %gs register is reloaded on context switch (necessary to keep track of thread state)
- Reloaded based on contents of addr_limit!
USING KERNEL_DS TRICK

- Write KERNEL_DS into addr_limit of current thread

- Loop on write(pipefd, addr, size)
  - Eventually, thread will be scheduled out at right moment (before copy_from_user)
  - When thread resumes, %gs register will be reloaded with __KERNEL_DS, and read target will be copied into pipe buffer (kernel-to-kernel copying)

- Restore addr_limit and read
PLAN OF ATTACK

STACK JACKING OVERVIEW

STACK SELF-DISCOVERY
- Manual analysis
- Auto with libkstack

STACK GROPING
- Rosengrope technique

STACK JACKING
- Read thread / task
- Overwrite creds

Arbitrary write
Kstack disclosure
• The Rosengrope technique
  • Pros: clean, simple, generic method to obtain arbitrary read from write+kleak
  • Cons: depends on knowing the location of addr_limit member of thread_info
    • It's possible to move thread_info out of the kstack!

• Any alternatives?
  • Let's get a bit crazier...
ATTACKING KSTACK FRAMES

• The Obergrope technique
  • Don't attack the thread_info metadata on kstack
  • Attack the kstack frames themselves!

• End goal is a read
  • How to read data by writing a kstack frame?
OBSERVATIONS

- Lots of kernel codepaths copy data to userland, via `copy_to_user()`, `put_user()`, etc
- There may be `copy_to_user()` calls that use a source address argument that is, at some point, stored on the kernel stack
- If we can overwrite that source address on the kstack, we can control source of the `copy_to_user()` and leak data to userspace
PROBLEM

• How can we write to stack reliably?
• We have a tricky race to win:
  • Parent needs to write into child's kstack between when the copy_to_user() source register is pushed and popped from the kstack
• This is a very small race window..
• We need something that we can sleep arbitrarily to win the race!
SLEEPY SYSCALLS

• Any of these sleepy syscalls have our required conditions?

• Needs to:
  • Push a register to the stack
  • Go to sleep for an arbitrary amount of time
  • Pop that register off the stack
  • Use that register as the source for copy_to_user()
asmlinkage long compat_sys_waitid(int which, compat_pid_t pid, 
    struct compat_siginfo __user *uinfo, int options, 
    struct compat_rusage __user *uru)
{
    struct rUSAGE ru;
    ...
    ret = sys_waitid(which, pid, (siginfo_t __user *)&info,
        uru ? (struct rUSAGE __user *)&ru : NULL);
    ...
    ret = put_compat_rUSAGE(&ru, uru);
    ...
}

int put_compat_rUSAGE(const struct rUSAGE *r, struct compat_rUSAGE __user *ru)
{
    if (!access_ok(VERIFY_WRITE, ru, sizeof(*ru)) ||
        __put_user(r->ru_utime.tv_sec, &ru->ru_utime.tv_sec) ||
        ...
}
COMPAT_SYS_WAITID

Dump of assembler code for function compat_sys_waitid:

... 0xffffffff810aba4e <+62>: lea -0x140(%rbp),%r14 ...
... 0xffffffff810aba8b <+123>: callq 0xffffffff81063b70 <sys_waitid>
... 0xffffffff810abae <+158>: mov %r14,%rdi
0xffffffff810abab1 <+161>: callq 0xffffffff810aa700 <put_compat_rusage>
...

Dump of assembler code for function sys_waitid:

... 0xffffffff81063bf9 <+137>: callq 0xffffffff810637e0 <do_wait>
...

Dump of assembler code for function do_wait:

... 0xffffffff810637e6 <+6>: push %r14 ...
... PROCESS GOES TO SLEEP HERE ...
... 0xffffffff810639fb <+539>: pop %r14 ...

1) compat_sys_waitid() stores address of ru in r14
2) compat_sys_waitid() calls sys_waitid()
3) sys_waitid() calls do_wait()
4) do_wait() pushes r14 on kstack
5) do_wait() sleeps indefinitely
6) we clobber the saved r14 reg on the kstack
7) do_wait() wakes up
8) do_wait() pops r14 off the kstack
9) do_wait() returns
10) sys_waitid() returns
11) compat_sys_waitid() calls put_compat_rusage()
12) put_compat_rusage() uses clobbered source addr
13) put_user() copies from source addr to userspace
### HIGH-LEVEL EXPLOIT FLOW

1. jacker forks/execs groper
2. groper gets its own kstack addr
3. groper passes kstack addr up to jacker
4. groper forks/execs helper
5. helper goes to sleep for a bit
6. groper calls waitid on helper
7. jacker overwrites the required offset on groper's stack
8. helper wakes up from sleep
9. groper returns from waitid
10. groper leaks task_struct address back to userspace
11. groper passes leaked address back up with jacker
12. steps 4-11 are repeated to leak task/cred addresses
13. jacker modifies groper's cred struct in-place
14. groper forks off a root shell
PLAN OF ATTACK

STACK JACKING OVERVIEW

STACK SELF-DISCOVERY
- Manual analysis
- Auto with libkstack

STACK GROPING
- Rosengrope technique
- Obergrope technique

STACK JACKING
- Read thread / task
- Overwrite creds

ROOT

Arbitrary write
Kstack disclosure
LIVE SPLOITZ

DEMO TIME
AGENDA

• A review of Linux kernel security

• Exploiting grsecurity/PaX kernels

• Stackjacking 2: Electric Boogaloo
STACKJACKING TIMELINE

- HES preso
- Spender angry blog post
- Fixes round #1
- Banana cognac preso
- RANDKSTACK considered harmful
- Fixes round #2
- Present time!
- Future?
RECOMMENDED FIXES

Defenses?

- Mitigate the exploitation vectors?
  - Remove thread_info metadata from kstack
  - RANDKSTACK?
- Eliminate all kstack disclosures?
  - Clear kstack between syscalls?
  - Compiler/toolchain magic?
- ???
Last Friday at the ES conference in France, a presentation titled "Stackjacking: Your Way to gseeurity/Fax Bypass" was given. Since then, it has been tweeted and discussed on Twitter, with many people expressing interest in the techniques presented. However, the slides were not made available publicly, so I will use my own notes from the presentation and some additional information to illustrate the concepts.

I have a number of issues with how this was handled, which I believe are:

1. Lack of transparency: The legal and ethical aspects of stackjacking were not addressed.
2. Lack of context: The presentation was not grounded in the broader context of security and privacy.
3. Lack of detail: The technical details were not provided.
4. Lack of ownership: The credit for the presentation was not given to the presenters.

For instance, the slide showing the attack on the Firefox browser was marked as "Confidential". However, the details were not provided, making it difficult to understand the implications.

I believe that more transparency and context are needed in the presentation to fully understand the implications of stackjacking.

In conclusion, while the presentation was interesting, it lacked depth and context. More information and transparency are needed to fully understand the implications of stackjacking.

---

Stackjacking and Other Kernel Nonsense – Jon Oberheide / Dan Rosenberg – SummerCon 2011 Slide #68
• thread_info
  • thread_info struct moved off the kstack base
  • Kills Rosengrope technique

• RANDKSTACK
  • Randomizes kesp on each syscall
  • Make Obergrope a bit unreliable

• USERCOPY
  • Hardened to prevent task_struct → userspace
  • Makes any groping more difficult
STACK JACKING OVERVIEW

ROOT

STACK JACKING
- Read thread / task
- Overwrite creds

STACK GROPING
- Rosengrope technique
- Obergrope technique

STACK SELF-DISCOVERY
- Manual analysis
- Auto with libkstack

Arbitrary write
Kstack disclosure
“Enjoy presenting a dead technique at Infiltrate; I hope the 15 minutes of fame from last week was worth it. If your path to infosec famedom involves screwing over friends for a free plane ride and hotel, you picked the wrong people.”

– spender pratt

= dan and jono
RIP STACKJACKING???
DEAD TECHNIQUE?

STACK JACKING OVERVIEW

STACK SELF-DISCOVERY

- Manual analysis
- Auto with libkstack

ROOT

STACK JACKING

- Read thread / task
- Overwrite creds
REMEMBER?

WHAT'S USEFUL ON KSTACK?

- Leak data off kstack?
  - Sensitive data left behind? Not really...
- Leak addresses off kstack?
  - Sensitive addresses left behind? Maybe...
    - Pointers to known structures could be exploited
    - *** Too specific of an attack! ***
- Need something more general
  - kstack disclosures differ widely in size/offsets
RANDKSTACK = BAD NEWS

Target syscall

Syscall call x

Syscall call y

Syscall call z

RANDKSTACK CONSIDERED HARMFUL!
STRAIGHT TO CRED'S!

• With RANDKSTACK, stackjacking is even easier
  • Instead of leaking at a SINGLE offset
  • We can leak at a CRAPLOAD of offsets

• Between a rock...
  • Don't use RANDKSTACK, get OBERGROPED!
  • Use RANDKSTACK, get credjacked!
**SHORTCUT**

**STACK JACKING OVERVIEW**

- Arbitrary write
- Kstack disclosure

**STACK SELF-DISCOVERY**

- Manual analysis
- Auto with libkstack

**STACKJACKING 2.0**

- Overwrite creds

**ROOT**
FIXES ROUND #2

+config PAX_MEMORY_STACKLEAK
+     bool "Sanitize kernel stack"
+     depends on X86
+     help
+     By saying Y here the kernel will erase the kernel stack before it
+     returns from a system call. This in turn reduces the information
+     that a kernel stack leak bug can reveal.
+
+     Note that such a bug can still leak information that was put on
+     the stack by the current system call (the one eventually triggering
+     the bug) but traces of earlier system calls on the kernel stack
+     cannot leak anymore.
+
+     The tradeoff is performance impact: on a single CPU system kernel
+     compilation sees a 1% slowdown, other systems and workloads may vary
+     and you are advised to test this feature on your expected workload
+     before deploying it.
+
+     Note: full support for this feature requires gcc with plugin support
+     so make sure your compiler is at least gcc 4.5.0 (cross compilation
+     is not supported). Using older gcc versions means that functions
+     with large enough stack frames may leave uninitialized memory behind
+     that may be exposed to a later syscall leaking the stack.
BUT IN REALITY...

- Not enabled by default
- Similar to PAX_MEM_SANITIZE
  - aka no one uses it
- Additional caveats

- STACKJACKING LIVES ON!
ENTER SMEP
ALMOST THE END
● #busticati

● $1$kk1q85Xp$ld.gAcJOG7uelf36VQwJQ/

● ;PpPppPPpPpPPPpP
QUESTIONS?

Jon Oberheide
jon@oberheide.org
Duo Security

Dan Rosenberg
dan.j.rosenberg@gmail.com
Virtual Security Research