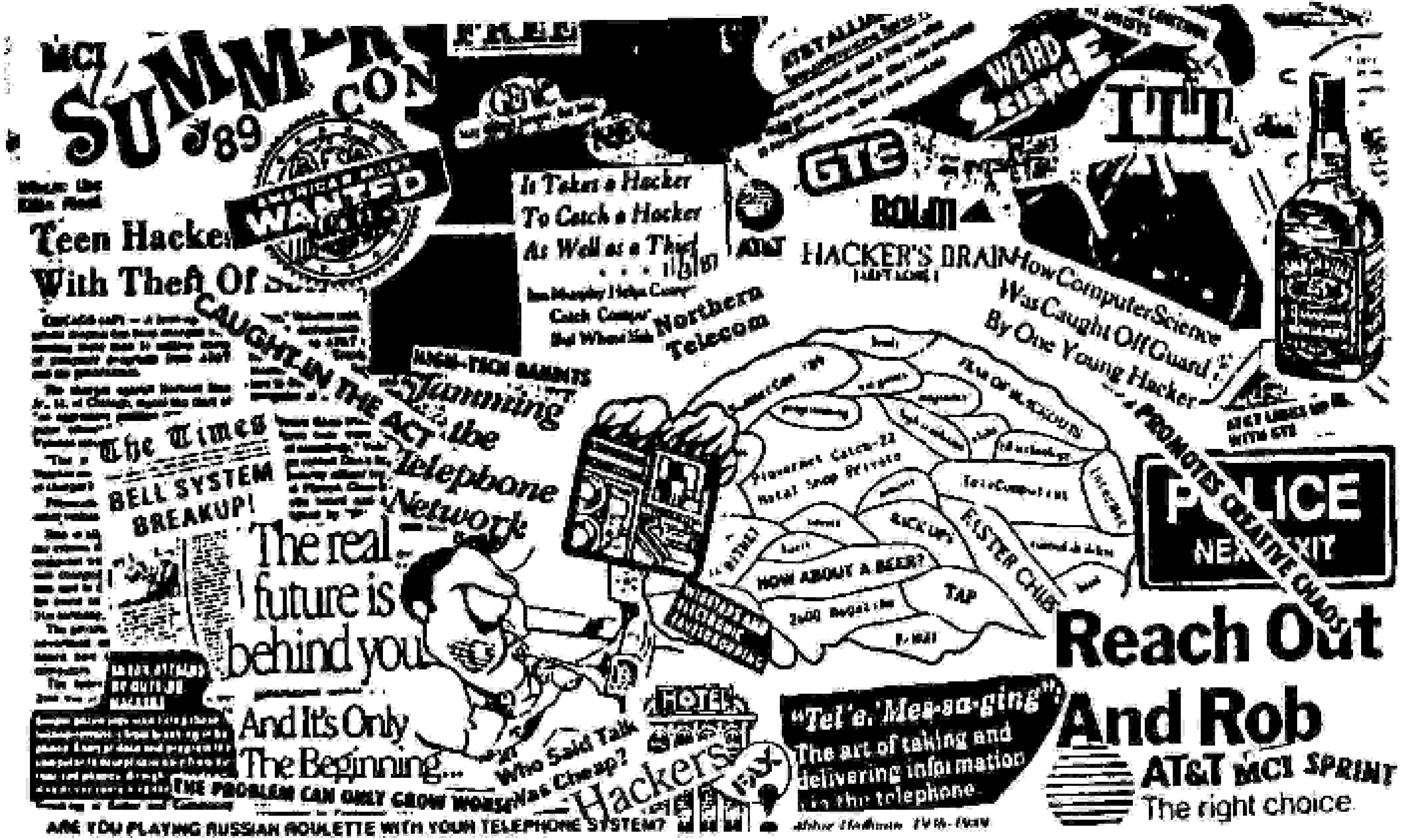
STACKJACKING AND OTHER KERNEL NONSENSE



JON OBERHEIDE & DAN ROSENBERG

MANDATORY FIRST SLIDE 0DAY

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);
...
```

PRIMARY MOTIVATION

```
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!

ADDITIONAL MOTIVATION

"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."

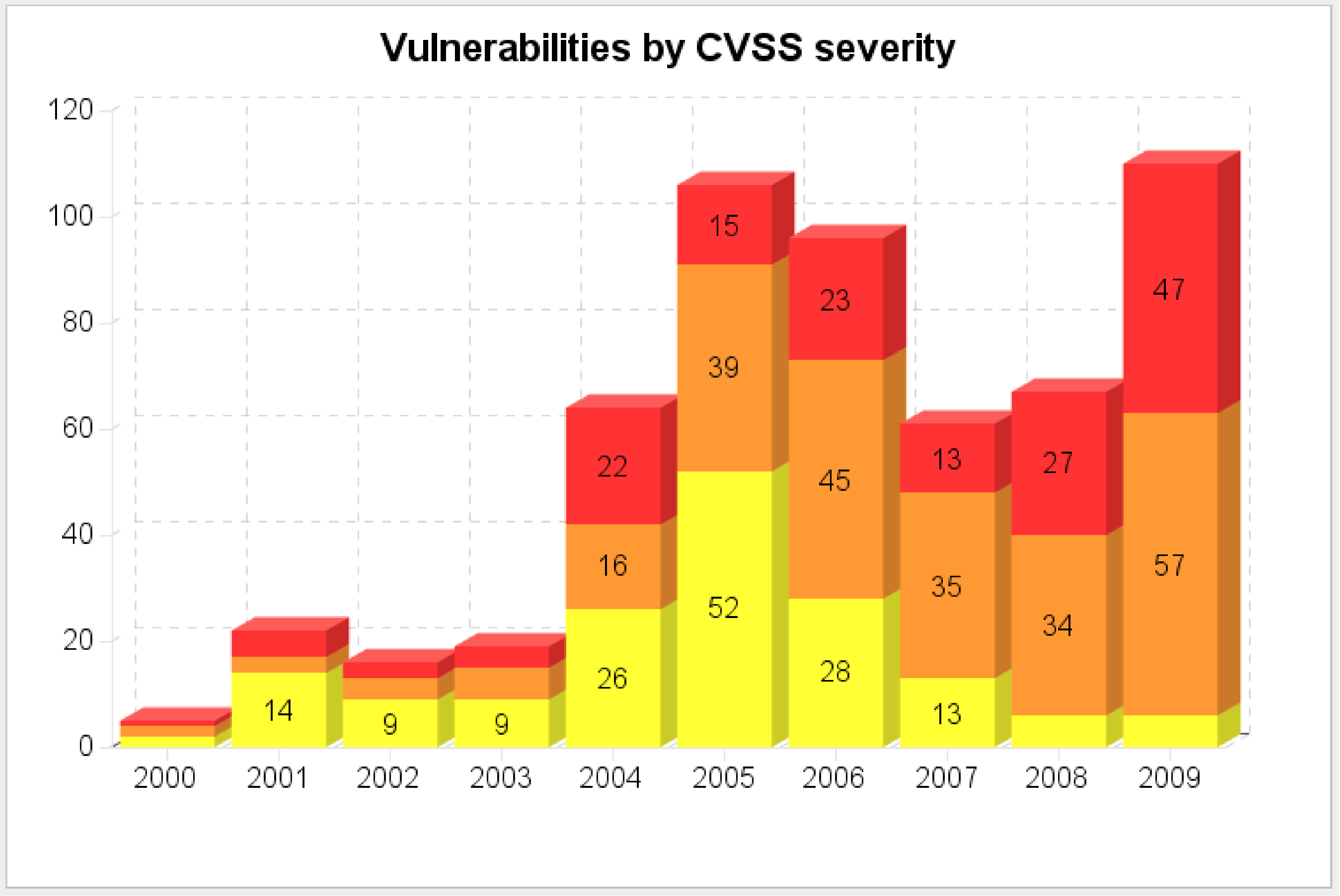
– spender pratt

PENULTIMATE MOTIVATION

AGENDA

- **A review of Linux kernel security**
- Exploiting grsecurity/PaX kernels
- Stackjacking 2: Electric Boogaloo

DECADE OF KERNEL SECURITY

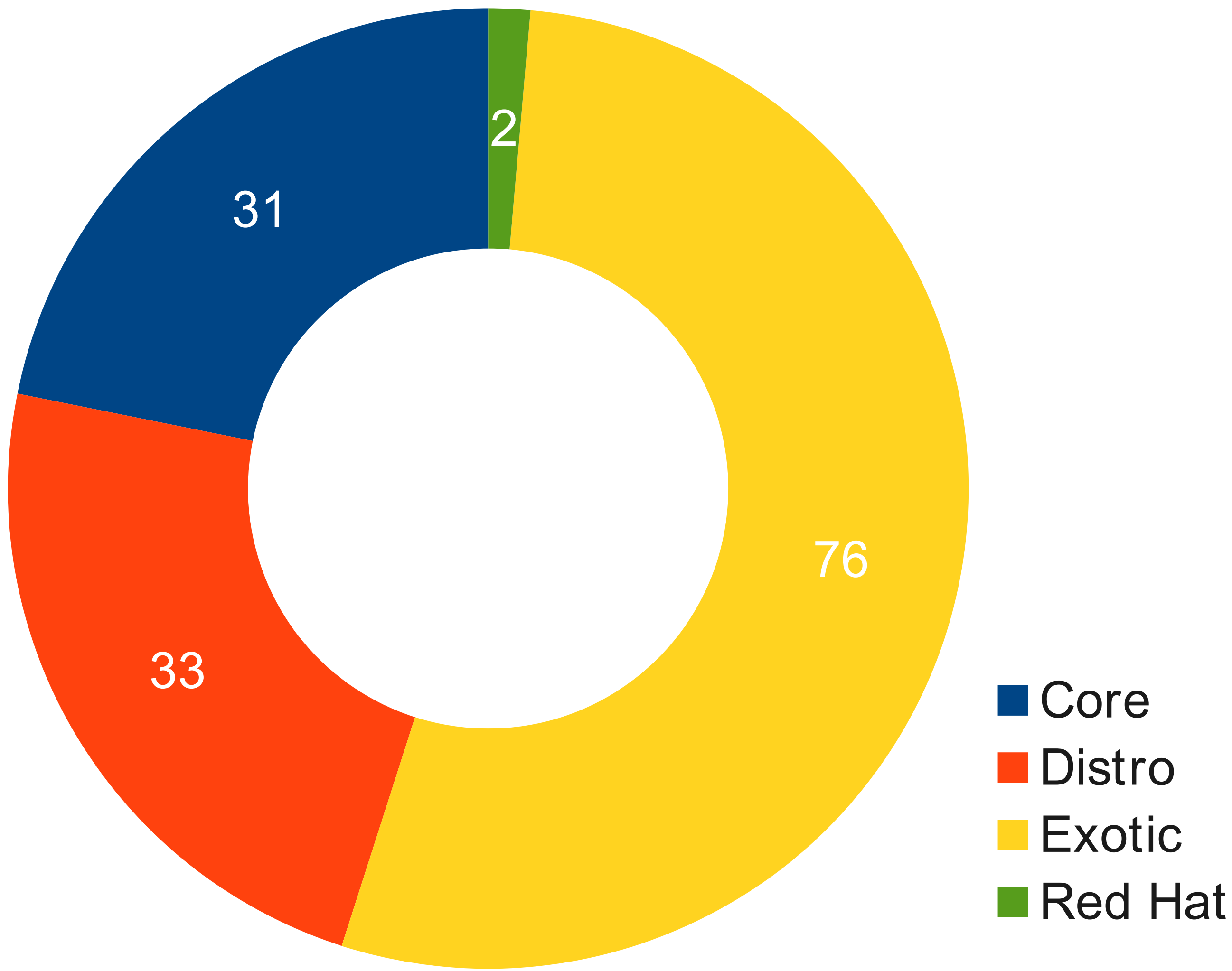


HOW ABOUT LAST YEAR?

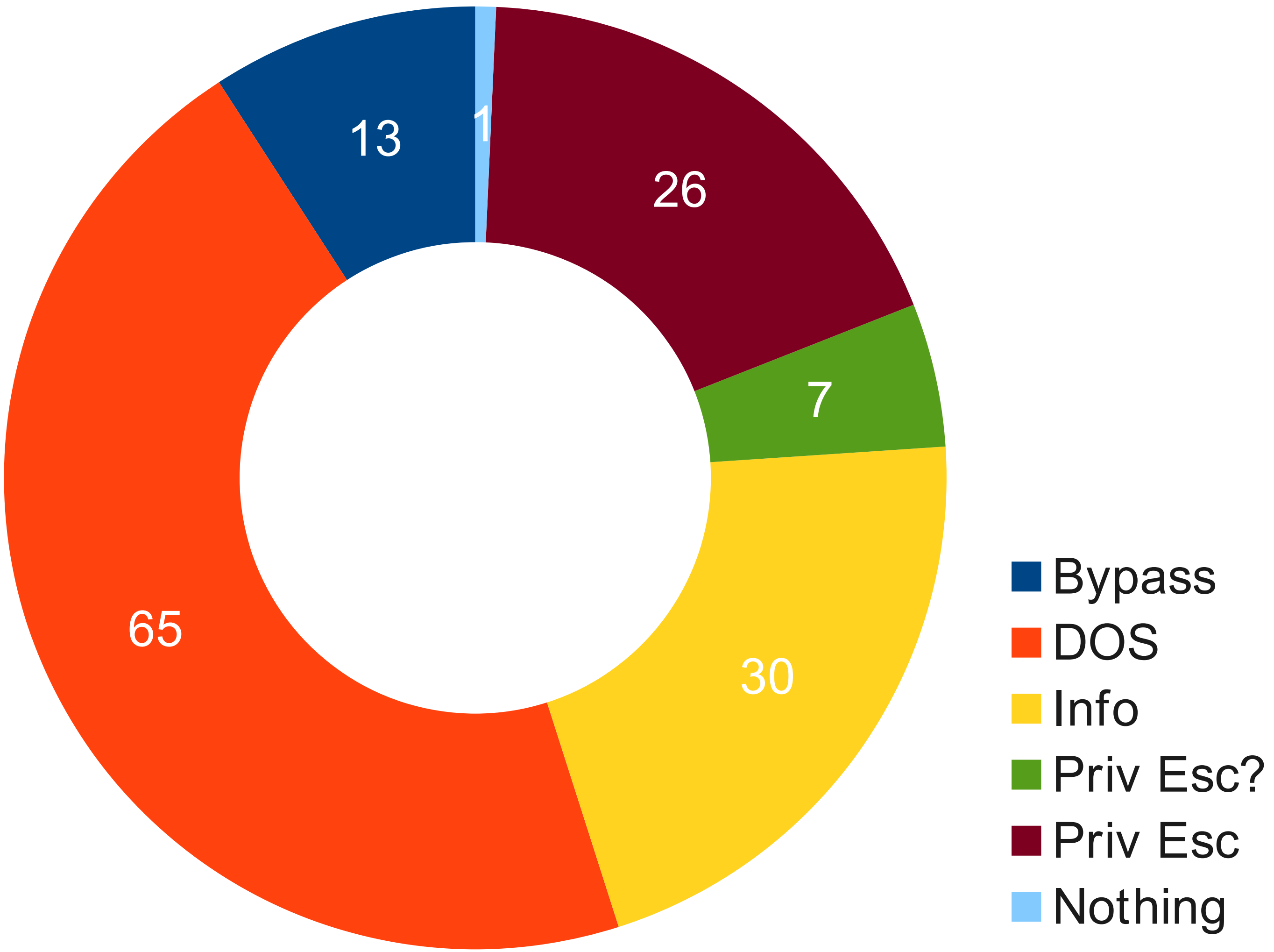
- 142 CVE's assigned
  - 30% worse than the previous worst year (2009)
  - Based on public CVE requests, issues tracked at Red Hat Bugzilla, and Eugene's tagged git tree
  - Missing dozens of non-CVE vulnerabilities (i.e. the “Dan Carpenter factor”)
- 61 (43%) discovered by six people
  - Kees (4), Brad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

DAN EATS KERNEL BUGS LIKE...

BREAKDOWN BY TARGET



BREAKDOWN BY IMPACT



LINUX DOS

```
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

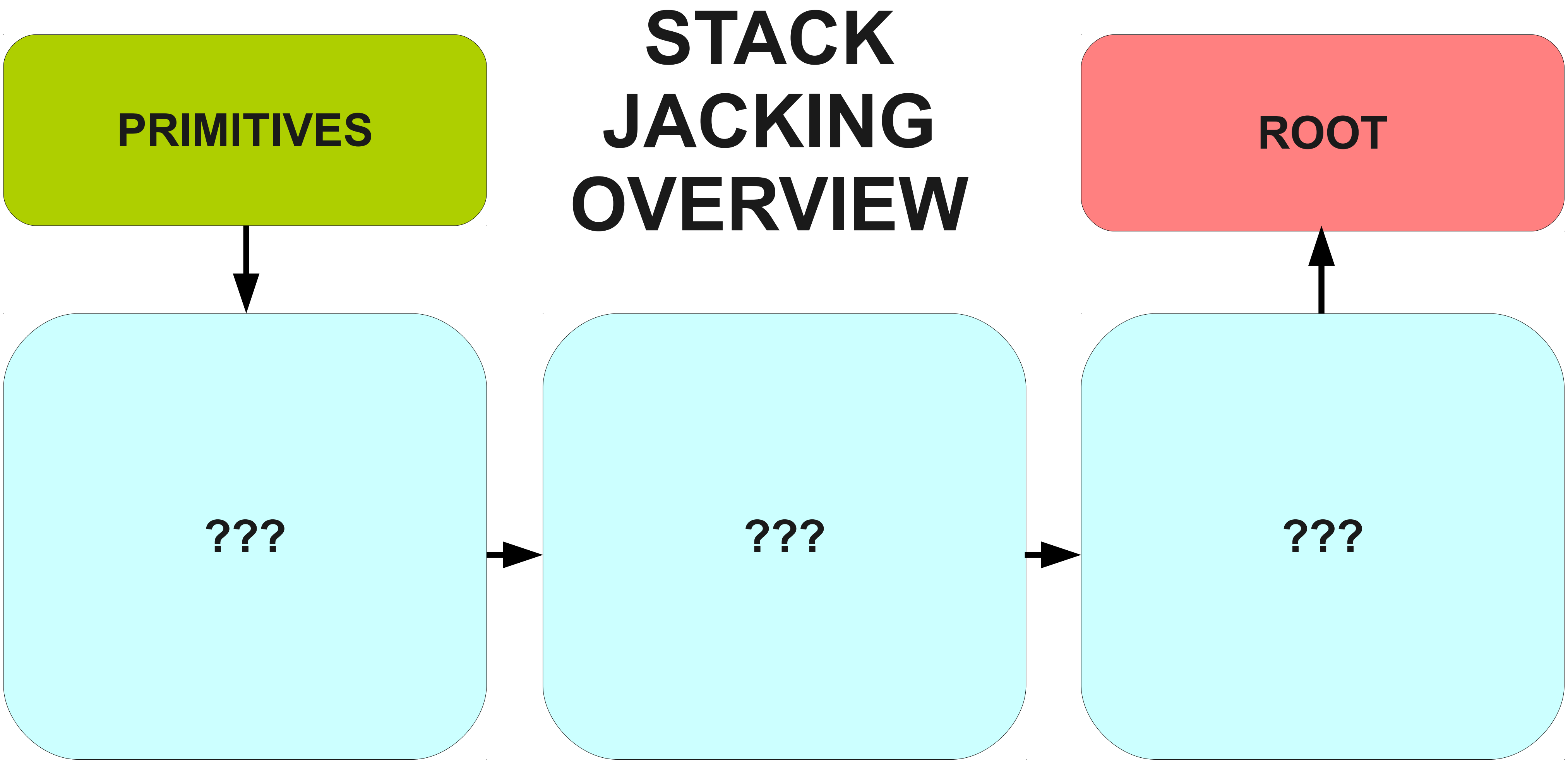
- A review of Linux kernel security
- **Exploiting grsecurity/PaX kernels**
- Stackjacking 2: Electric Boogaloo

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)

- 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



TARGET KERNEL ASSUMPTIONS

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

STRONG ASSUMPTIONS

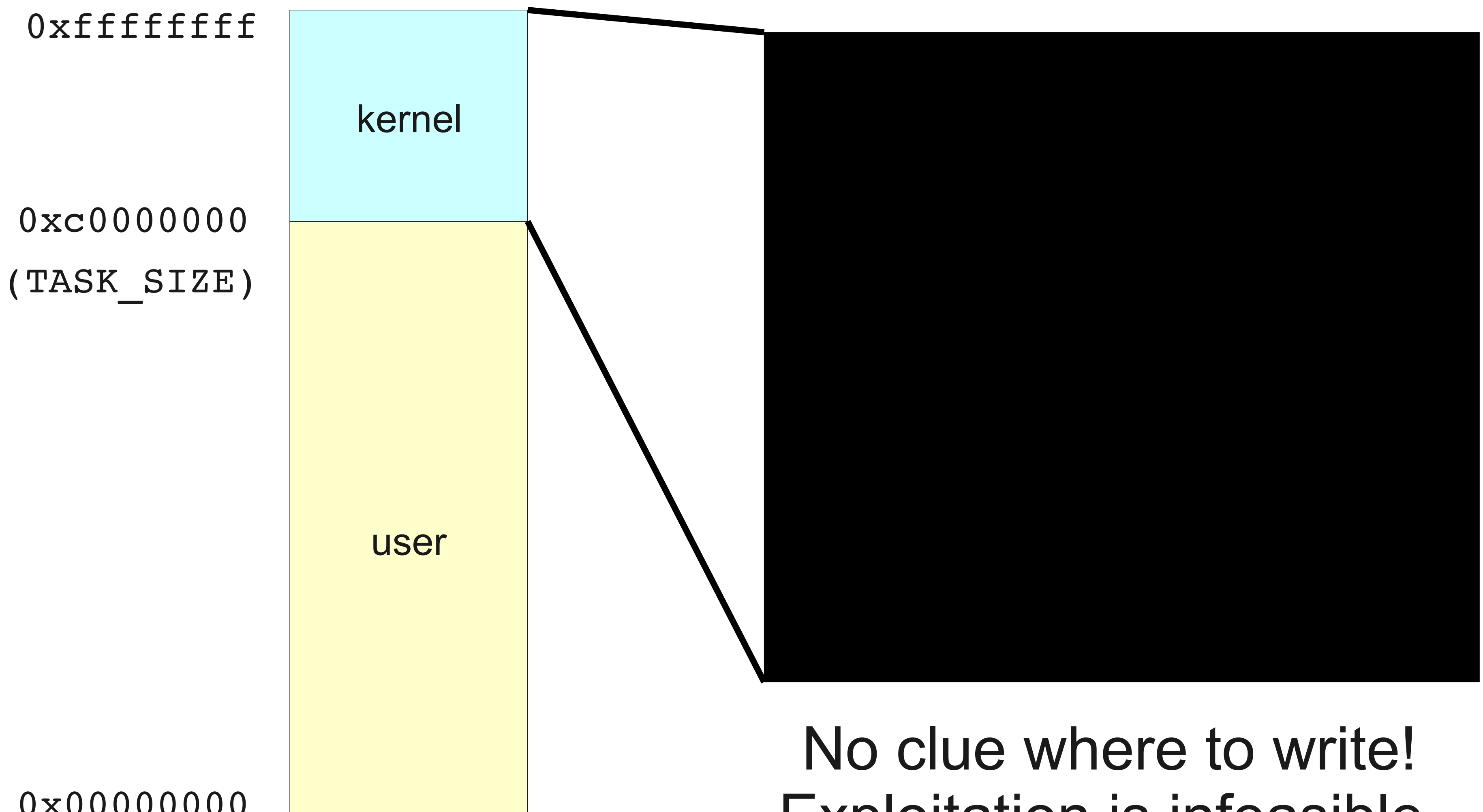
- Let's make some extra assumptions
  - We like a challenge, and these are assumptions that may possibly be obtainable now or in the future
- Stronger target assumptions
  - Zero knowledge of kernel address space
  - Fully randomized kernel text/data
  - Cannot introduce new code into kernel address 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?

DARKNESS!



No clue where to write!
Exploitation is infeasible.

- 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

ALPHA KMEM DISCLOSURE #1

```
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;
...
}
```

ALPHA KMEM DISCLOSURE #2

```
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;
...
}
```

FREEBSD KMEM DISCLOSURE

```
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);
}
```

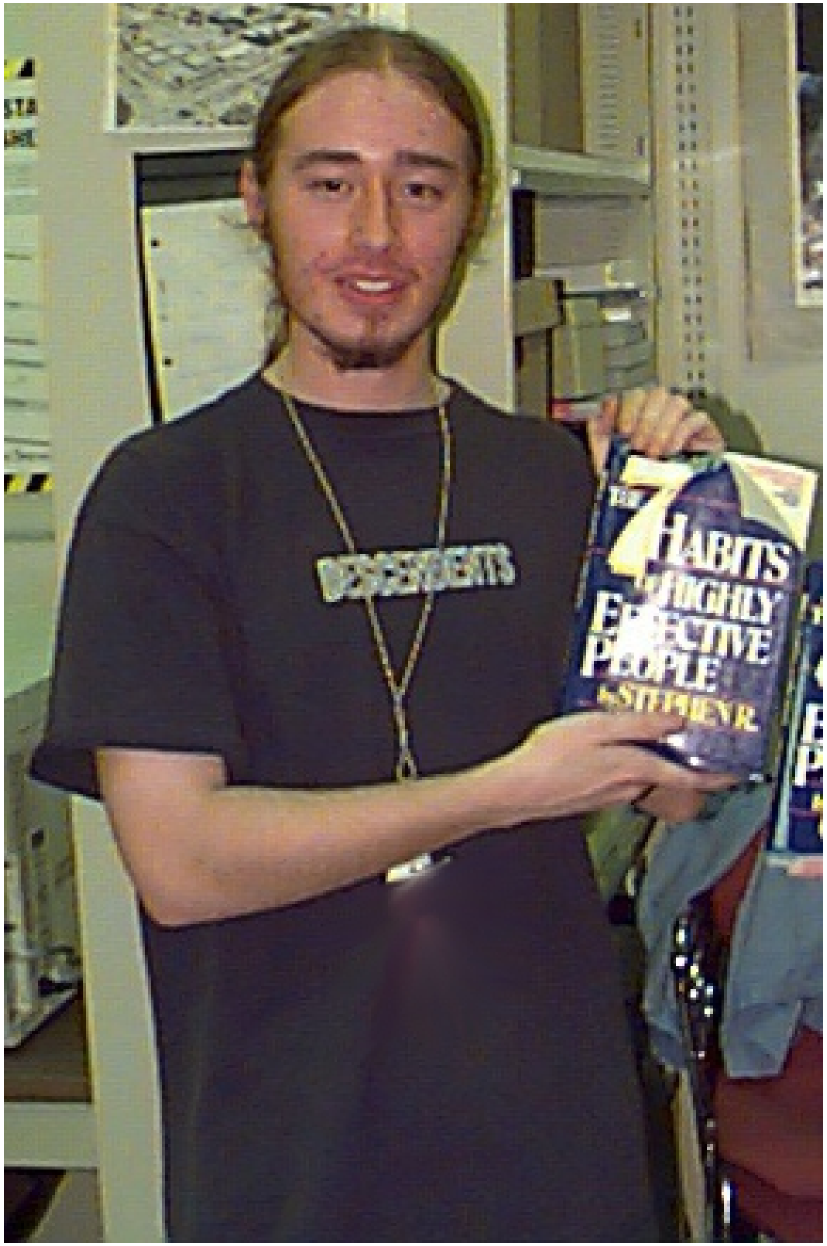
SOMETHING MORE COMMON?

- 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...

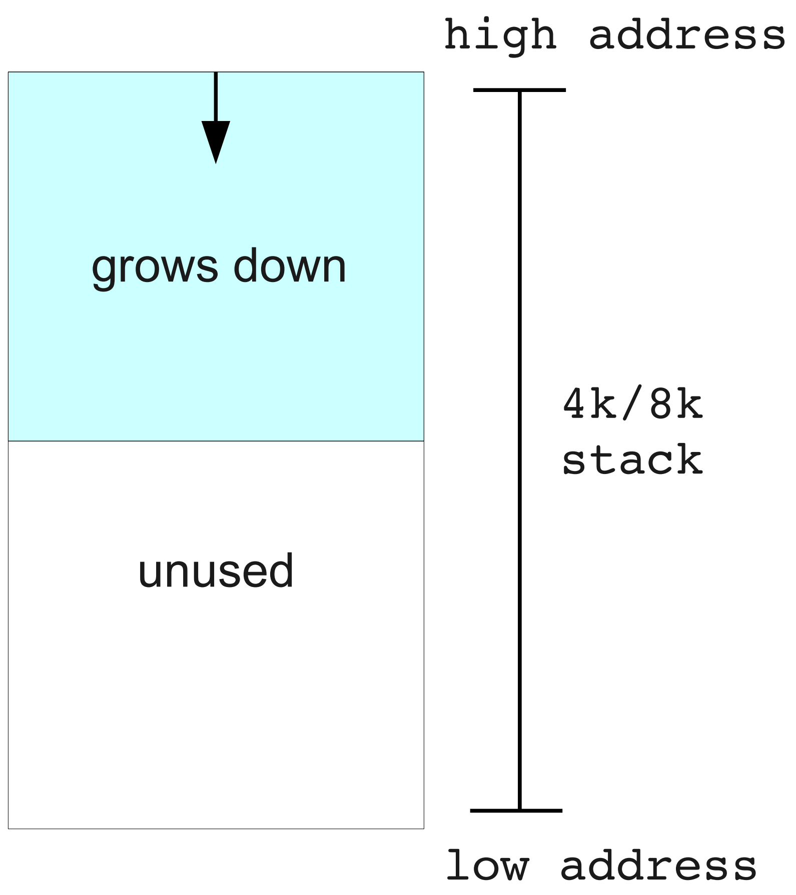
KERNEL STACK MEMORY DISCLOSURES!

THANKS FOR THE TIP DDZ!



LINUX KERNEL STACKS

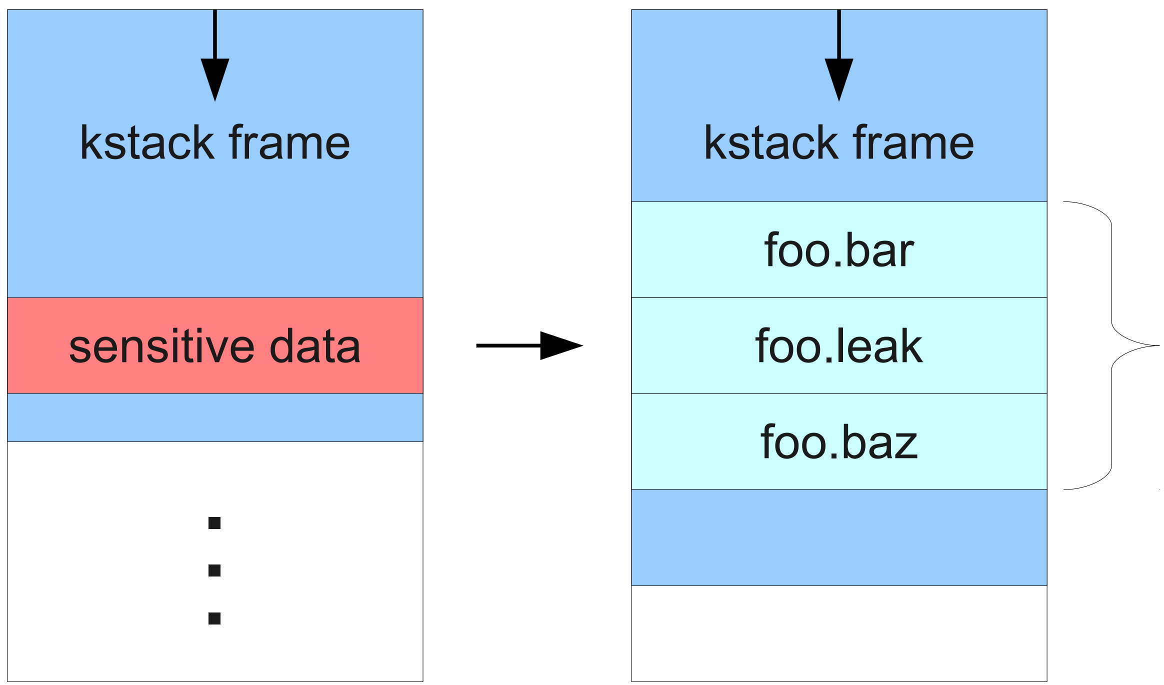
- Each userspace thread is allocated a kernel stack
- Stores stack frames for kernel syscalls and other metadata
- Most commonly 8k, some distros use 4k
  - THREAD_SIZE = 2*PAGE_SIZE = 2*4086 = 8192



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



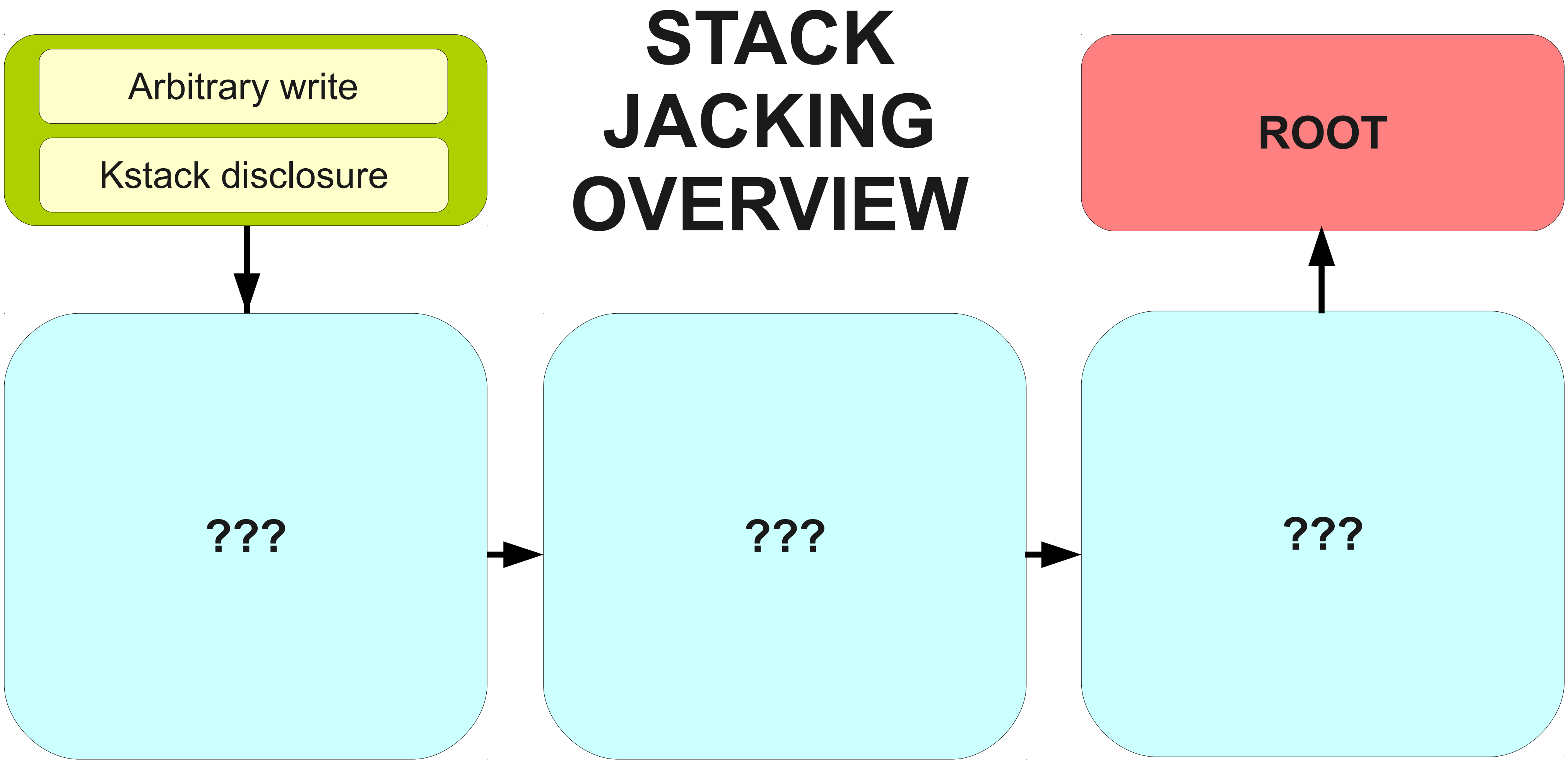

```
struct foo {
  uint32_t bar;
  uint32_t leak;
  uint32_t baz;
};

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

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

PLAN OF ATTACK



WHAT'S USEFUL ON KSTACK?

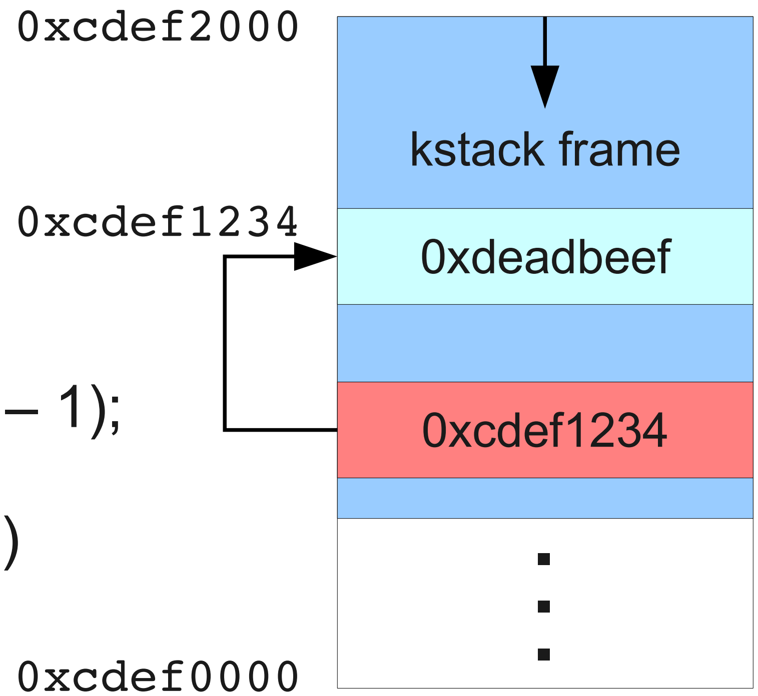
- Leak data off kstack?
  - Sensitive data left behind? Not really...
- Leak addresses off kstack?
  - Sensitive addresses left behind? Maybe...
    - Pointers to known structures could be exploited
    - ***** Too specific of an attack! *****
- Need something more general
  - kstack disclosures differ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

KERNEL STACK ADDRESSES

- 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)

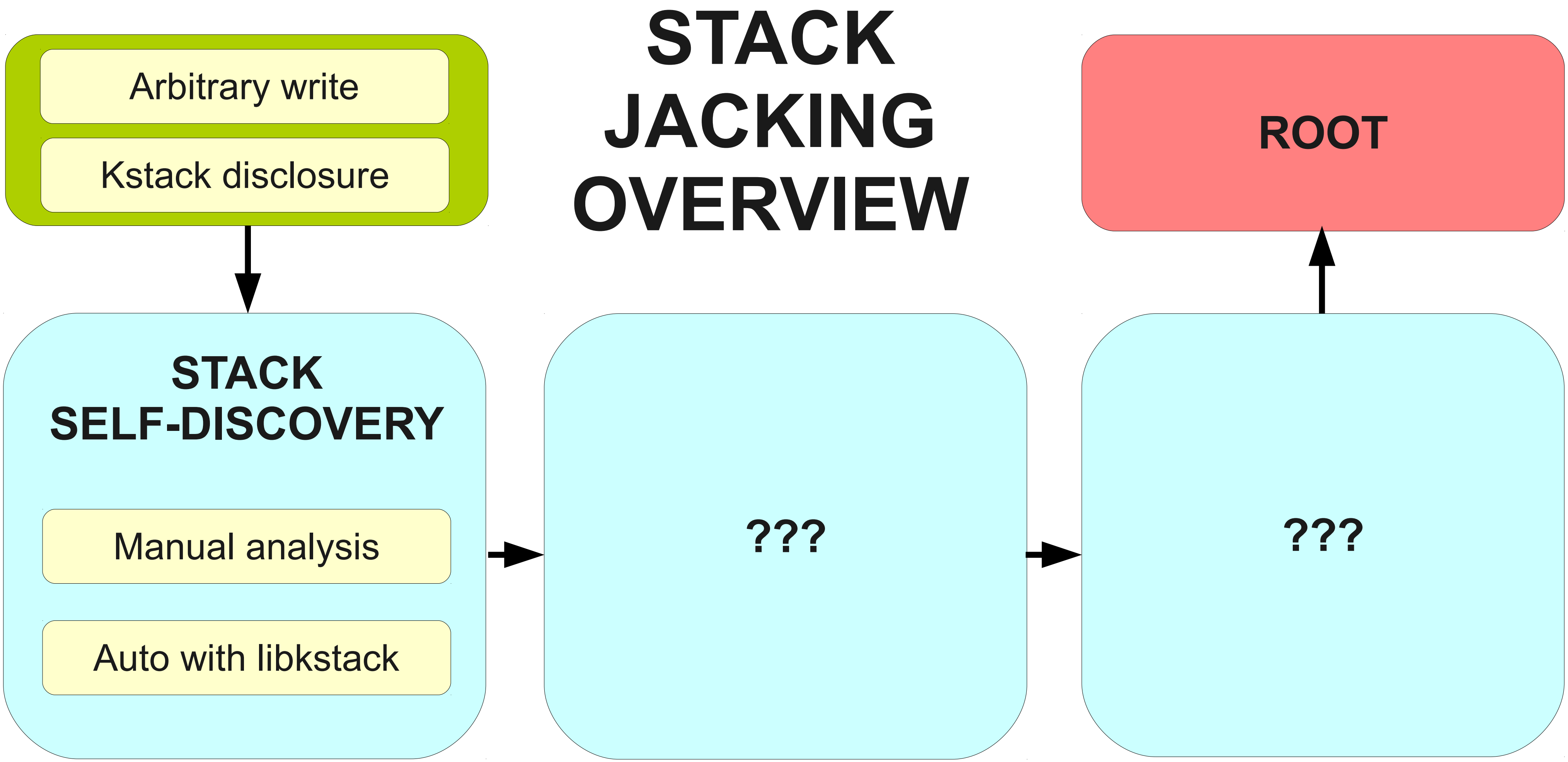
kstack_base = 0xcdef0000

We call this ***kstack self-discovery***

HOW TO SELF-DISCOVER

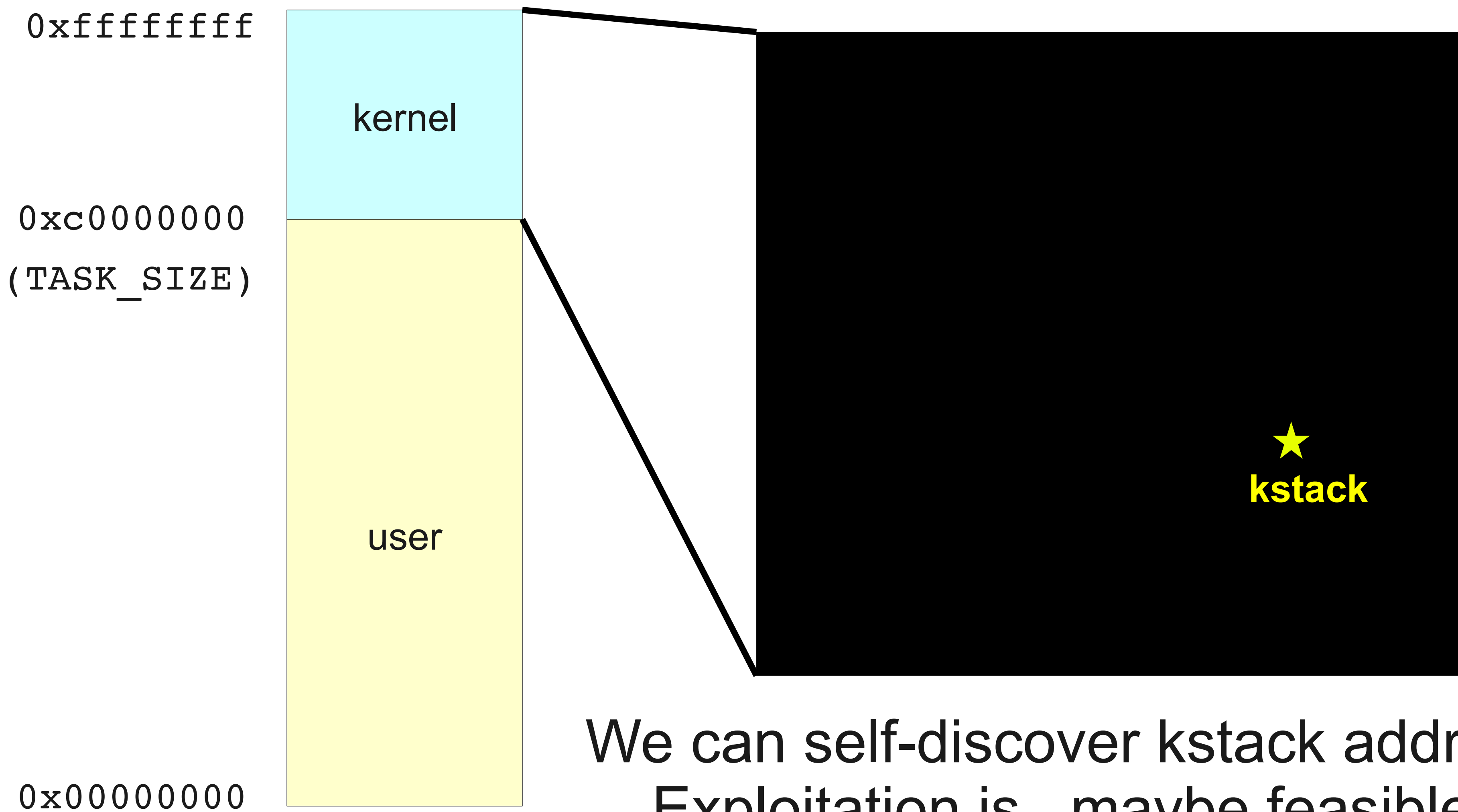
- Not all kstack disclosures are alike
  - May only leak a few bytes, non-consecutive
  - How do we effectively self-discover?
- Manual analysis
  - Figure out where kstack leak overlaps addresses
- Automatic analysis
  - libkstack

PLAN OF ATTACK



NO LONGER DARKNESS

A random pinpoint of light!



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

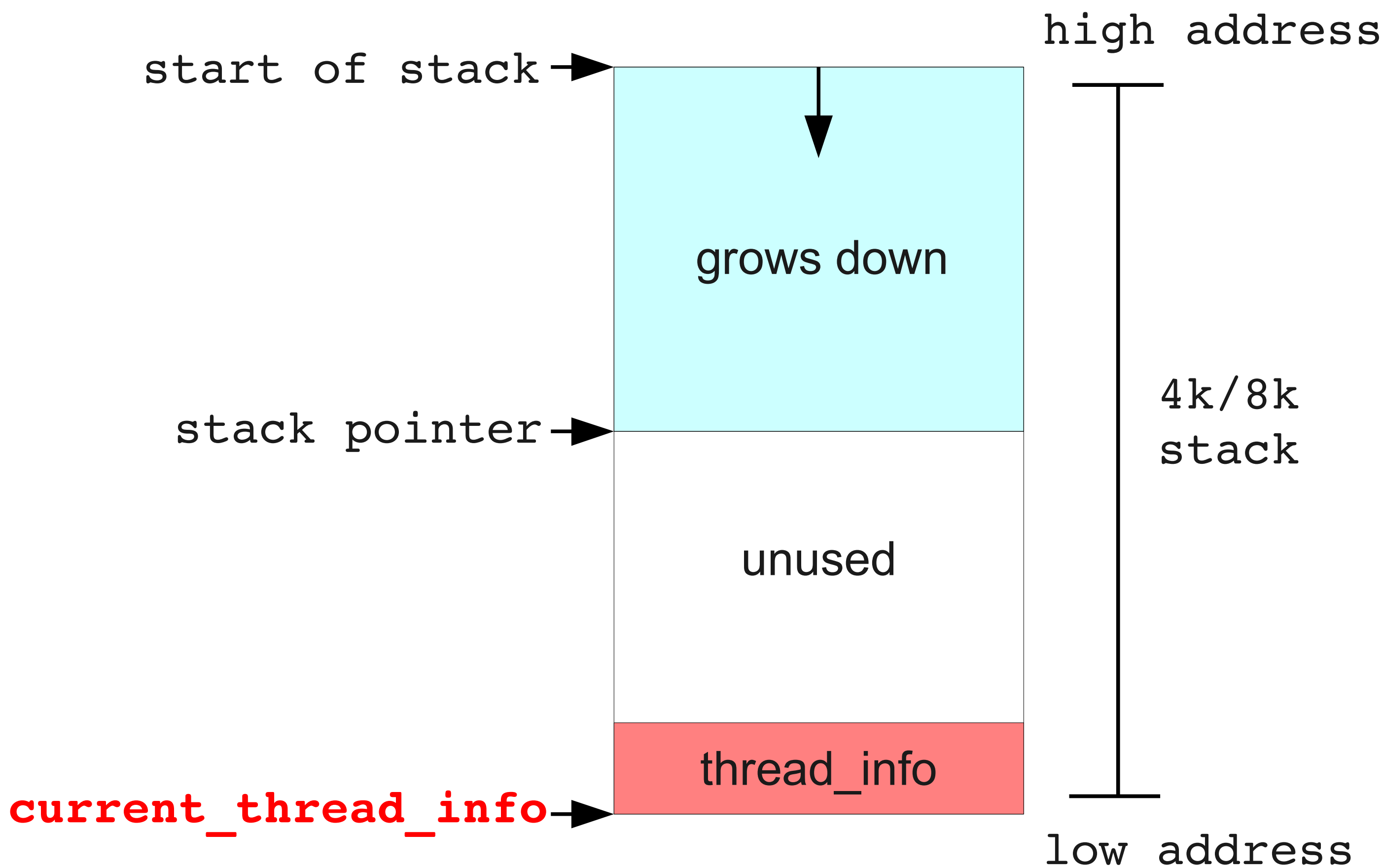
BEFORE SOLAR GOT HIS FLAIR

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!

METADATA ON KSTACK

Anything else of interest on the kstack???



thread_info struct stashed at base of kstack!

THREAD_INFO CANDIDATES

```
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?

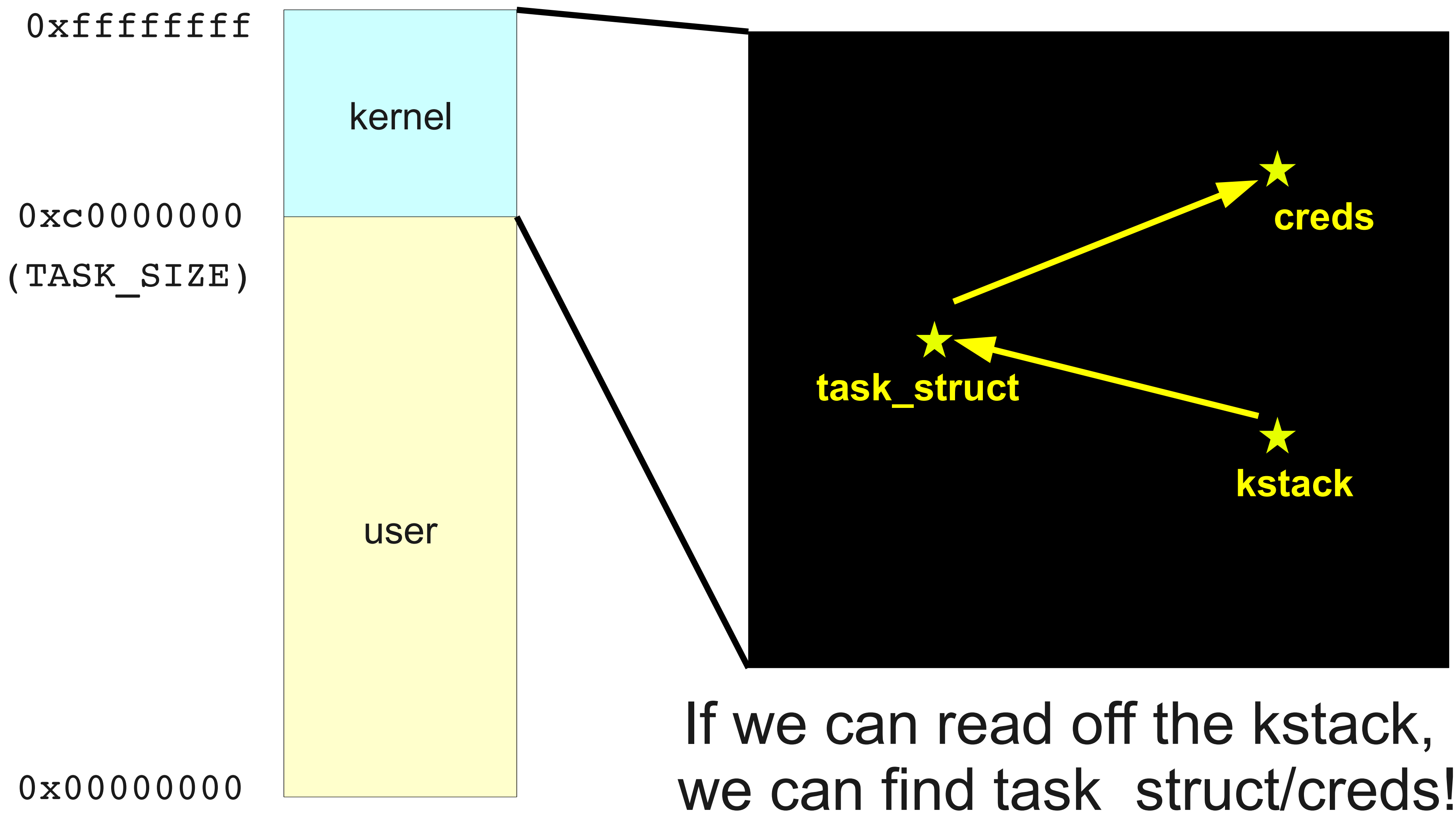
ATTACKING TASK_STRUCT

```
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

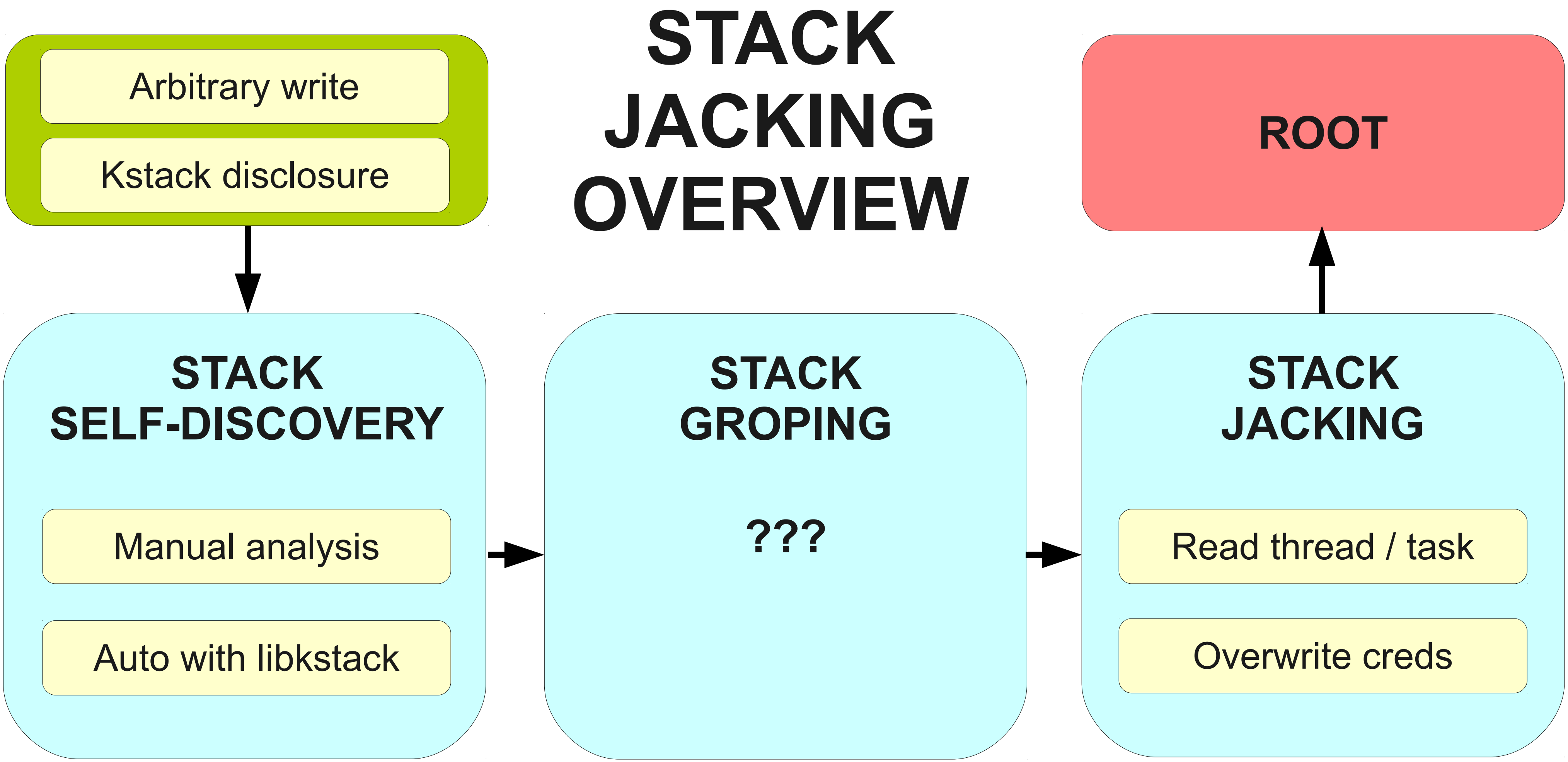
Expanding our visibility



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!


STACK JACKING OVERVIEW
Arbitrary write
Kstack disclosure
ROOT
STACK SELF-DISCOVERY
Manual analysis
Auto with libkstack
STACK GROPING
???
STACK JACKING
Read thread / task
Overwrite creds

ROSENGROPE TECHNIQUE

```
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

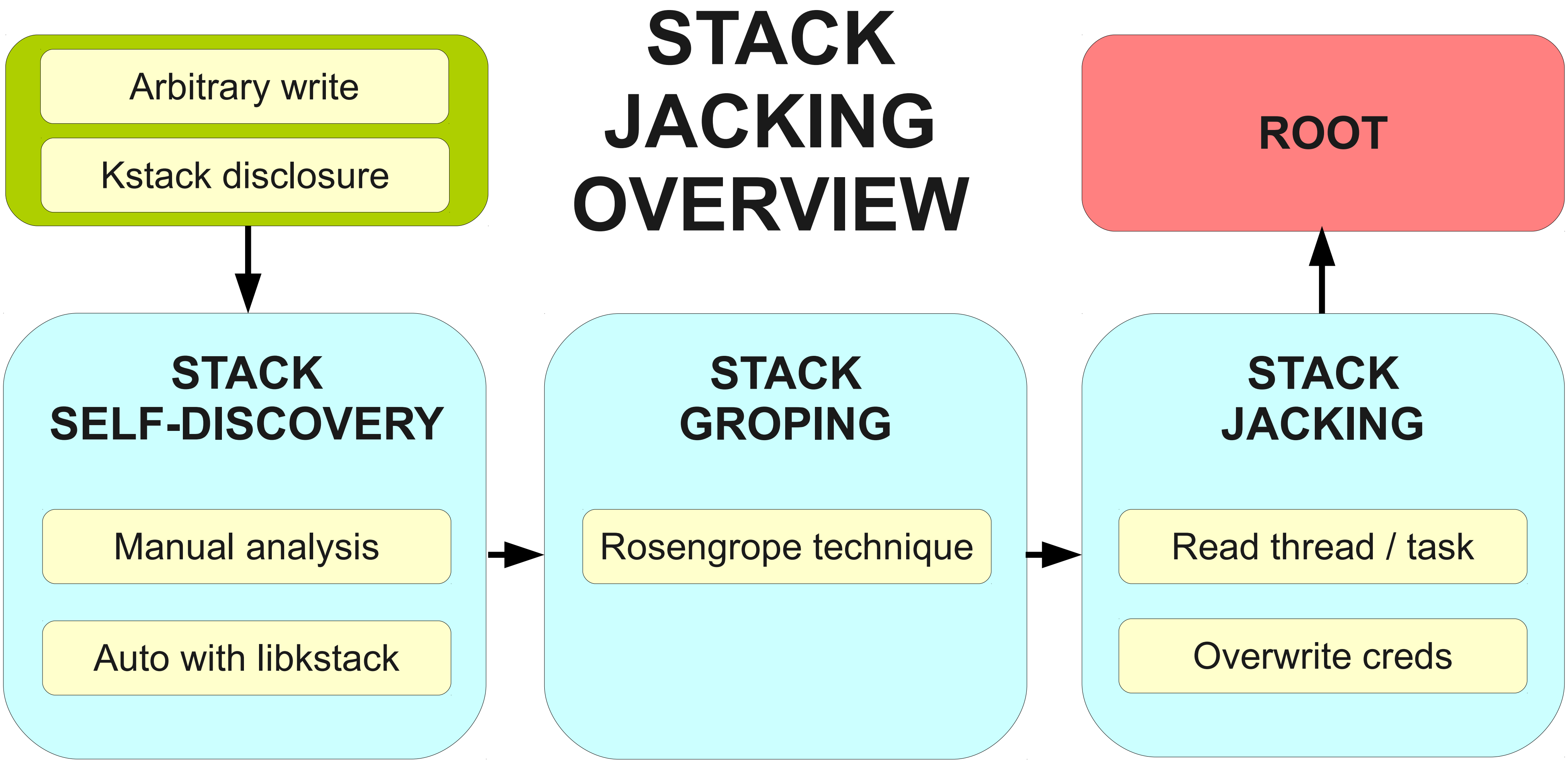
NO MORE EASY ROOT

- 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



PROS/CONS OF KERNEL_DS

- 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...

PROS/CONS OF THING2

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()

COMPAT_SYS_WAITID

```
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

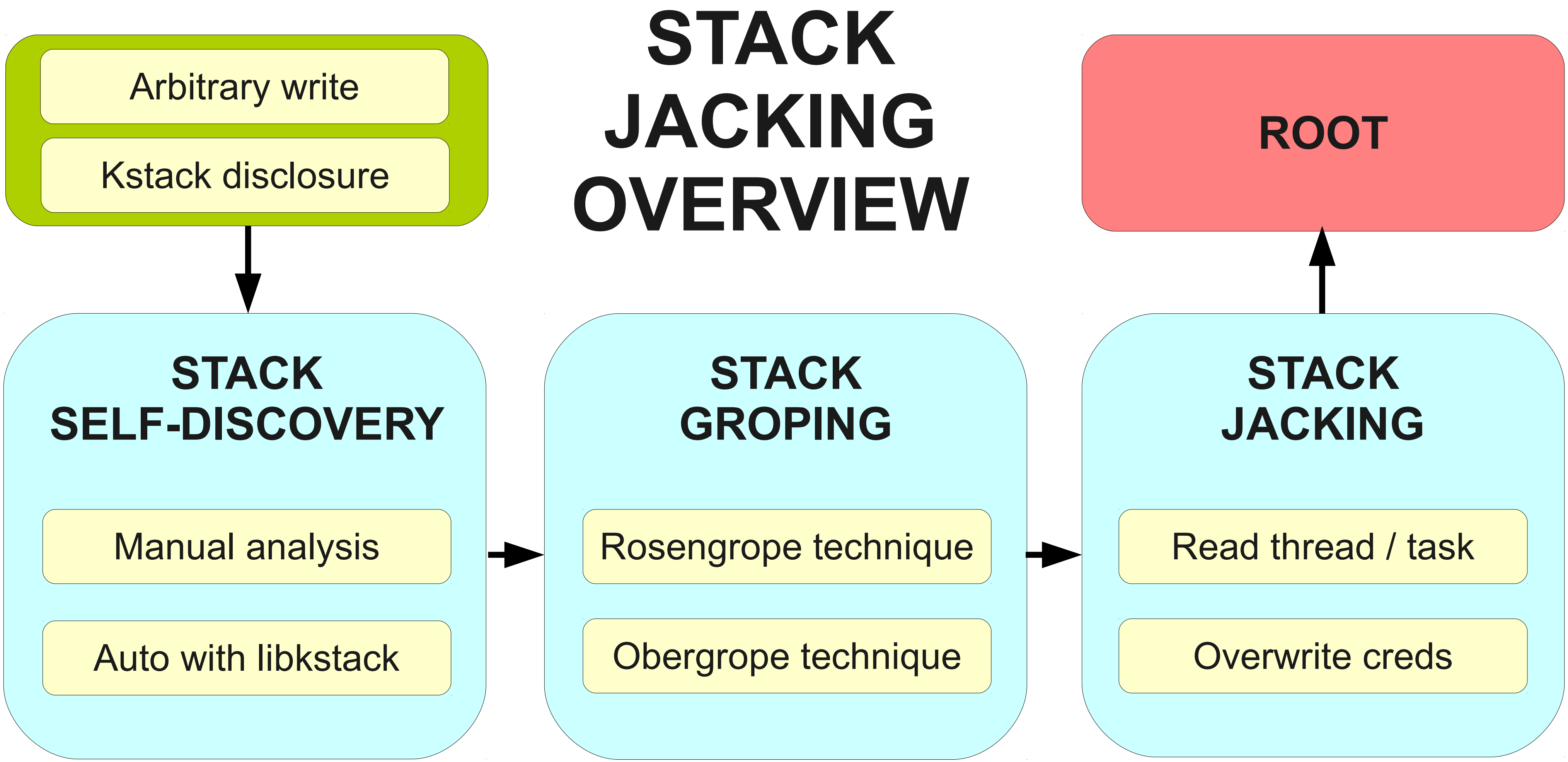
```
Dump of assembler code for function compat_sys_waitid:
...
   0xffffffff810aba4e <+62>:  lea    -0x140(%rbp),%r14
...
   0xffffffff810aba8b <+123>: callq  0xffffffff81063b70
                                     <sys_waitid>
...
   0xffffffff810abaae <+158>: mov    %r14,%rdi
   0xffffffff810ababl <+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



DEMO TIME

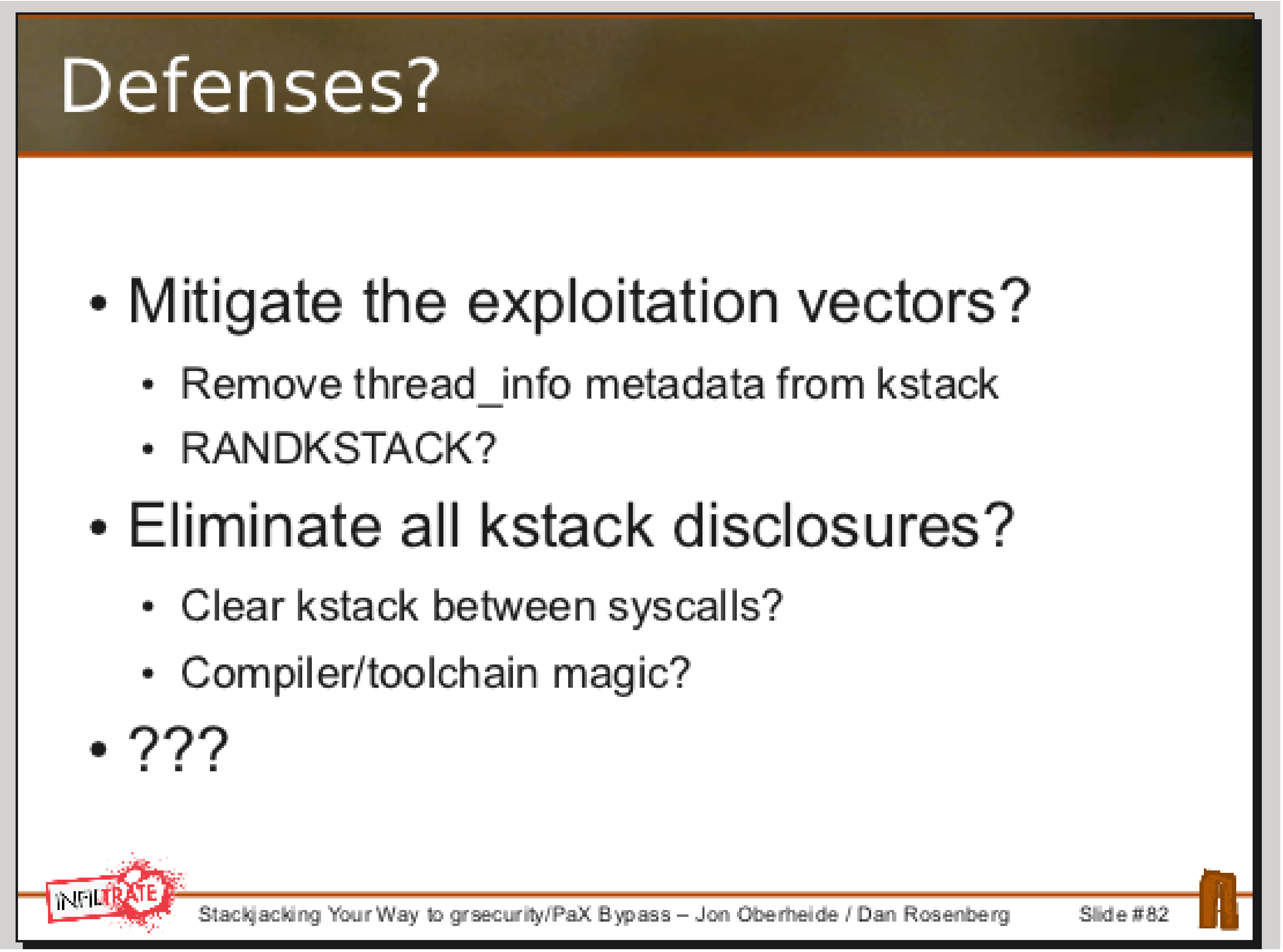
AGENDA

- 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



SPENDER BLOG POST

Much Ado About Nothing: A Response in Text and Code

by **spender** » Sat Apr 16, 2011 5:34 pm

Last Friday at the HES conference in France, a presentation entitled "Stackjacking Your Way to grsecurity/PaX Bypass" was given. Soon after, an image was retweeted frequently on Twitter, supposedly of the presenters getting "root on a grsecurity/PaX kernel on-stage". No other details were provided to those retweeting the image (except for those in attendance at HES) and it was mentioned that the slides/code for the attack wouldn't be published until after a repeat presentation at Infiltrate in Miami this Sunday.

I have a number of issues with how this was handled, which I will elucidate here.

For starters, the PaX Team and myself got about 30 minutes advance notice of the slides for the presentation. Though this in itself would not be out of the ordinary for people we don't know at all, the presenters in this case have used my research in their own presentations and created patches "loosely based" on grsecurity features and submitted them to the Linux Kernel.

Up to the presentation (and even still on the Infiltrate website), the presentation was entitled "TBA Kernel Fun", setting itself apart from all other presentations as being completely unnecessarily secretive. The presentation either went through a secret approval process through the HES committee (of which I was a member) or underwent no form of committee approval with no submitted abstract. The reason for such unusual secrecy is puzzling.

We knew from seeing the slides 30 minutes before the presentation that the attack demonstrated was apparently against a "2.6.36.3-grsec" kernel. We didn't learn until after the presentation was given that the kernel was modified to add a fake arbitrary-write vulnerability. I understand the case where one would do this to demonstrate a technique alone without having to kill any valuable bugs, but as the presenters themselves brag at the beginning of the presentation, they've discovered plenty of bugs in the classes required for their attack, arbitrary-write and arbitrary-read or stack infoleak. Yet not only was a 6 month old kernel used (to take advantage of the only core-kernel stack infoleak published in the past year, out of the dozens applicable to vanilla Linux kernels), but the arbitrary-write bug had to be fabricated. This seems disingenuous, since at least from a reading of the slides themselves, as I have no knowledge of the way in which the presentation was given, the introductory slides set up a picture of how buggy the Linux kernel is (which is of course true) and mentions how prevalent the infoleaks, etc are, creating the suggestion that the conditions for a "bypass" of a grsecurity kernel are anything but rare. The slides don't mention at all how specific features of grsecurity can drastically reduce the attack surface of the kernel and limit the possible damage from rarely-used modules and certain classes of vulnerabilities.

I object to the use of "bypass" when referring to a security system with dozens of features. An example of a bypass would be the various mmap_min_addr bypasses that existed: in these cases it was always possible to achieve the same goal and abuse the same vulnerabilities. In the case of this presentation, what is being bypassed exactly? The features with defined protections worked as intended, preventing them from abusing certain vulnerabilities that would have been usable on vanilla kernels. The exploit was done within a known attack space mentioned specifically in the PaX documentation (nothing can currently stop an arbitrary read+write). Because it's very difficult to do anything about this class (especially when still meeting usability and performance requirements) I've been focusing on, and I believe have been successful at, reducing the reachability of vulns in general (including those of this class) and in reducing their scope (turning a linear overflow into a write bounded by the heap object itself, for instance). Bugs of this class are getting more rare as they're eventually weeded out. Arbitrary read/write bugs often arise out of the non-existence of bounds/sanity checking, which in many cases are easier to spot than the more prevalent bugs of some form of read/write with added constraints, due to the deceiving nature of any existing (flawed) bounds/sanity checking.

So I'm left wondering why an attack with so little real-life application compared to the other things we deal with in grsecurity required so much hype-inducing secrecy. Did they know the technique could be killed and wanted to make sure they could give both presentations with a still-relevant attack?

For people I consider friends and colleagues, this was a slap in the face. I don't find it to be acceptable and have received no apology. So today, prior to their repeat presentation, I'm announcing the death of every single technique they presented at HES. I hope the message will be clear for others in the future that working with us will be much less painful than the alternative. It should also be clear to those interested in secrecy and hype-at-any-cost that a subsequent "reality adjustment" will be swift. This release time wasn't chosen intentionally: the PaX Team and I have been using all of our spare time since last Friday to write this up, test it, and port it to the kernels we support. In fact, we've taken great pains to make sure it was complete long before I return home on Sunday, ironically about 30 minutes before their repeat presentation would begin.

Enough talk. Here's what we've done:

Moved thread_info off the kernel stack completely for both i386 and amd64 -- it's now located in the task struct, which is located in its own slab cache.

Implemented PAX_RANDKSTACK for amd64 (without requiring MSR access, performance hit is the same as the i386 version since it calls the same function). This is applied per-syscall, making it essentially immune to infoleaks against the stack pointer itself, the same as on the i386 version.

Implemented additional protection in PAX_USERCOPY by identifying which slab caches required direct reads/writes from the userland accessor functions. Slab cache information is free within the current PAX_USERCOPY framework. We implemented a whitelist-based approach, and rewrote several areas of code to remove the need for direct reads/writes to certain structures so that the associated slab caches can be protected against direct reads/writes to/from userland. The technique allows for individual allocations within the whitelisted caches to be marked in the future, so that certain sensitive structures can be better protected. Of particular note is that task structs are protected under this mechanism.

Implemented active response against kernel exploitation. Attacks by unprivileged users will result in a permanent ban of the user (all processes killed, no new processes allowed) until system reboot. Attacks by root (either by real uid-0 or as fallout from buggy post-exploitation cleanup) or while in interrupt context result in a system panic.

Extended automatic bruteforce deterrence to apply to suid/sgid binaries -- detected PaX terminations or crashes will result in a 15 minute user ban (the amount of time intended to help offset entropy reduction attacks).

Improved MODHARDEN -- removed the fallback in netdev code to allow auto-loading any module when CAP_SYS_MODULE is present. The PaX Team came up with the beautiful idea, and I implemented a system by which we can control the entire asynchronous procedure of module auto-loading, along with correlating it with the original requester. Through this system, we can ensure (for instance) that mount -t can only cause filesystem modules to be loaded (verified through symbol inspection at load time). This system also has the side-effect of removing unnecessary reports for non-existent modules, as was the case in the previous system (since at request time we couldn't know whether the module existed or not). All these changes allow us to make stronger guarantees about the feature, regardless of any buggy privileged user code that performs certain actions on behalf of unprivileged users through dbus or the like.

TL;DR: Lack of coordination works both ways. 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. In case you'd like to continue this game, we're more than capable and willing to kill anything else you come up with :) Thanks for playing.

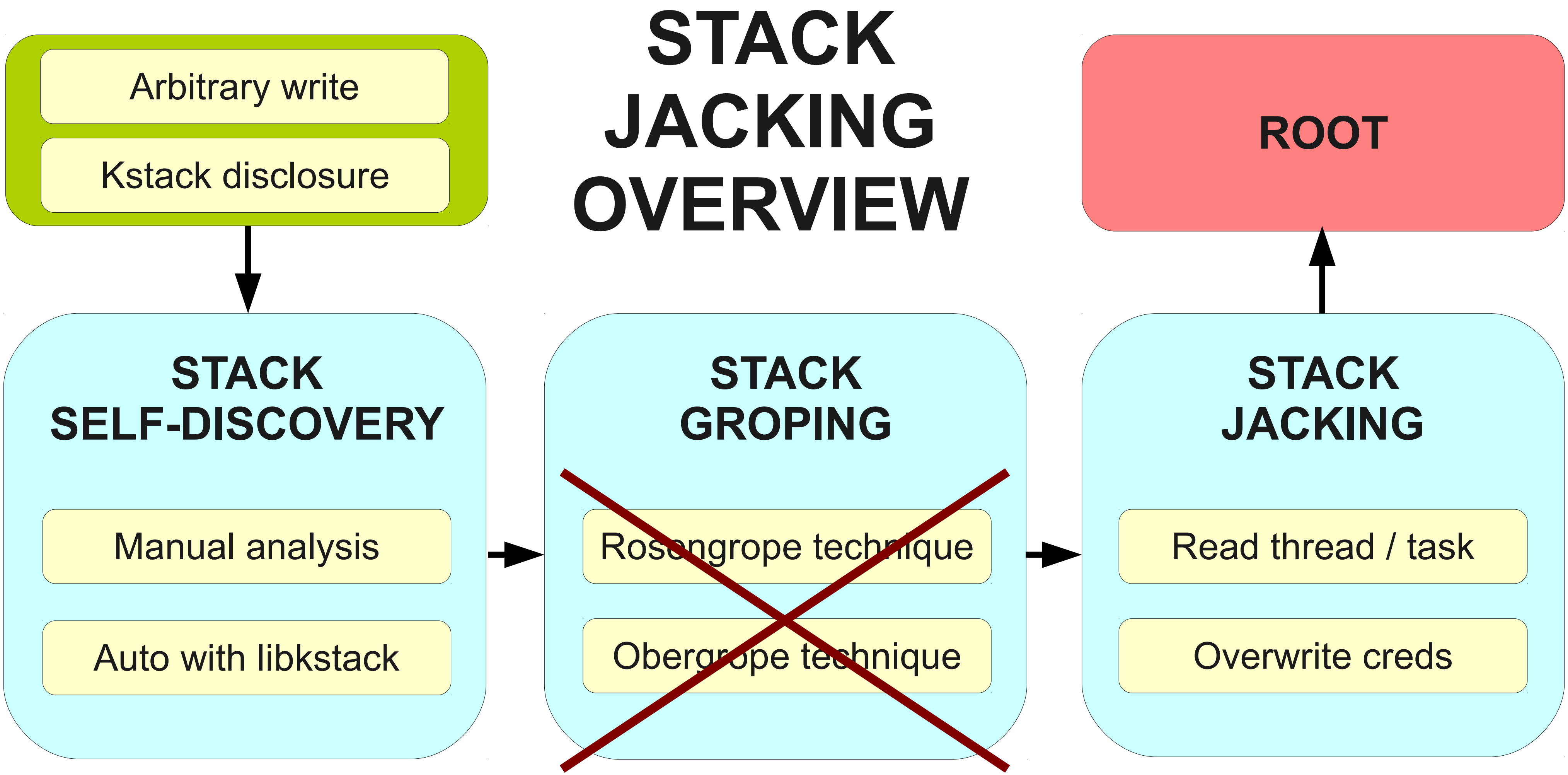
-Brad

Flattr this!

FIXES ROUND #1

- 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

FIXES ROUND #1



SPENDER ANGRY

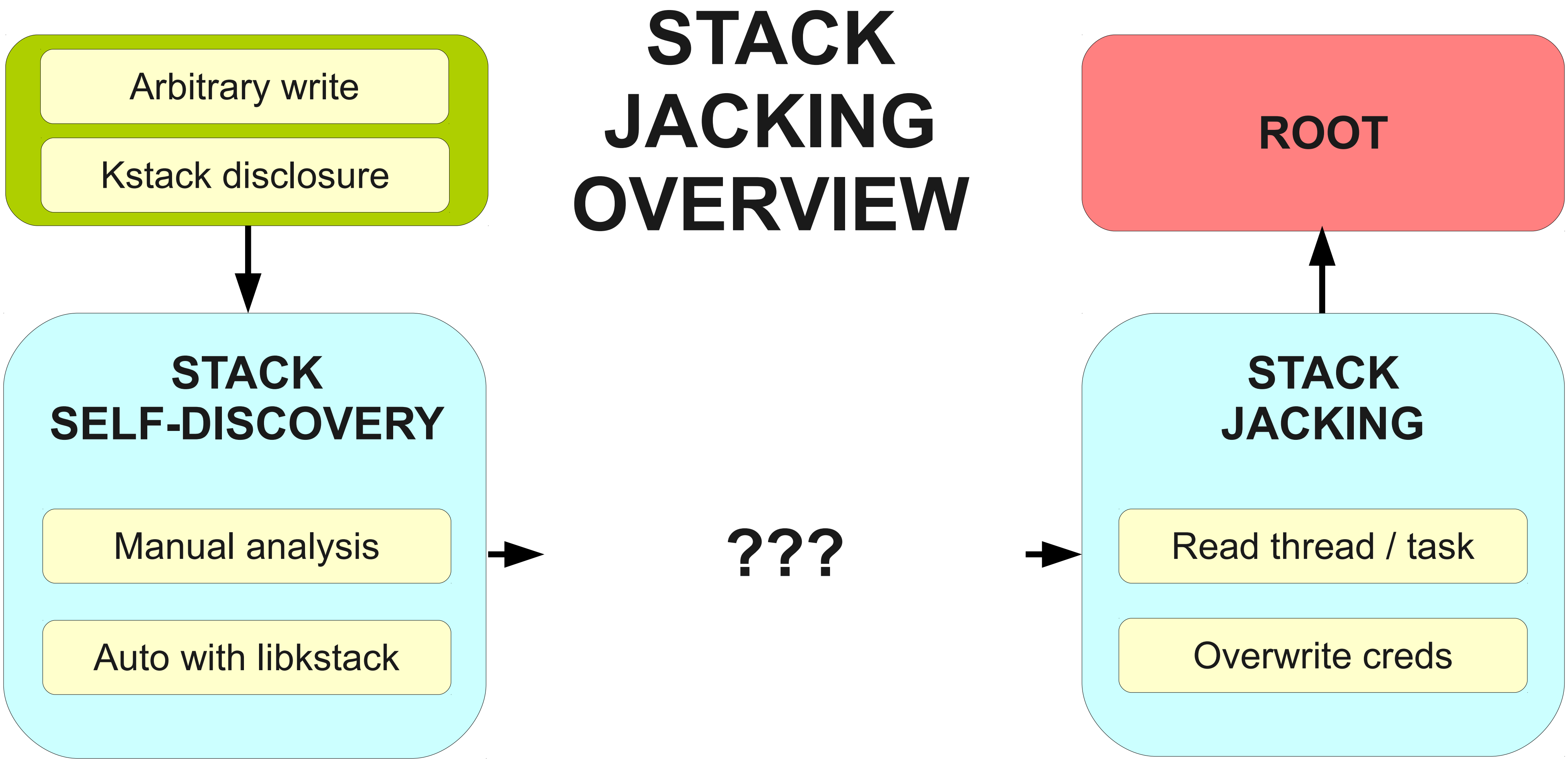
“Enjoy presenting a dead technique at Infiltrate; I hope the 15 minutes of fame from last week was worth it. If your path to infosec famedom involves screwing over friends for a free plane ride and hotel, you picked the wrong people.”

– spender pratt

= dan and jono

RIP STACKJACKING???

DEAD TECHNIQUE?



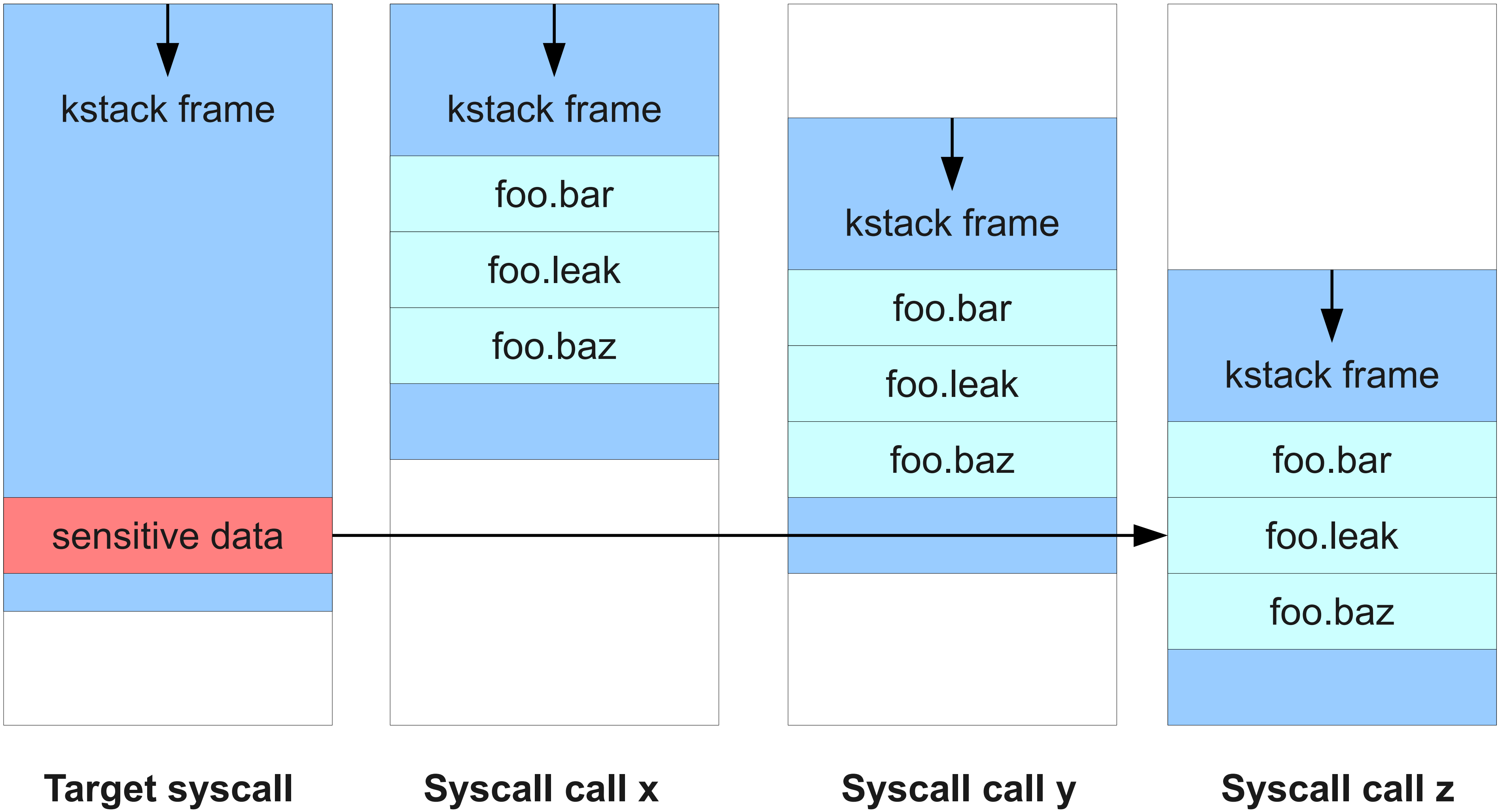
REMEMBER?

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

Stackjacking and Other Kernel Nonsense – Jon Oberheide / Dan Rosenberg – SummerCon 2011 Slide #32

RANDKSTACK = BAD NEWS

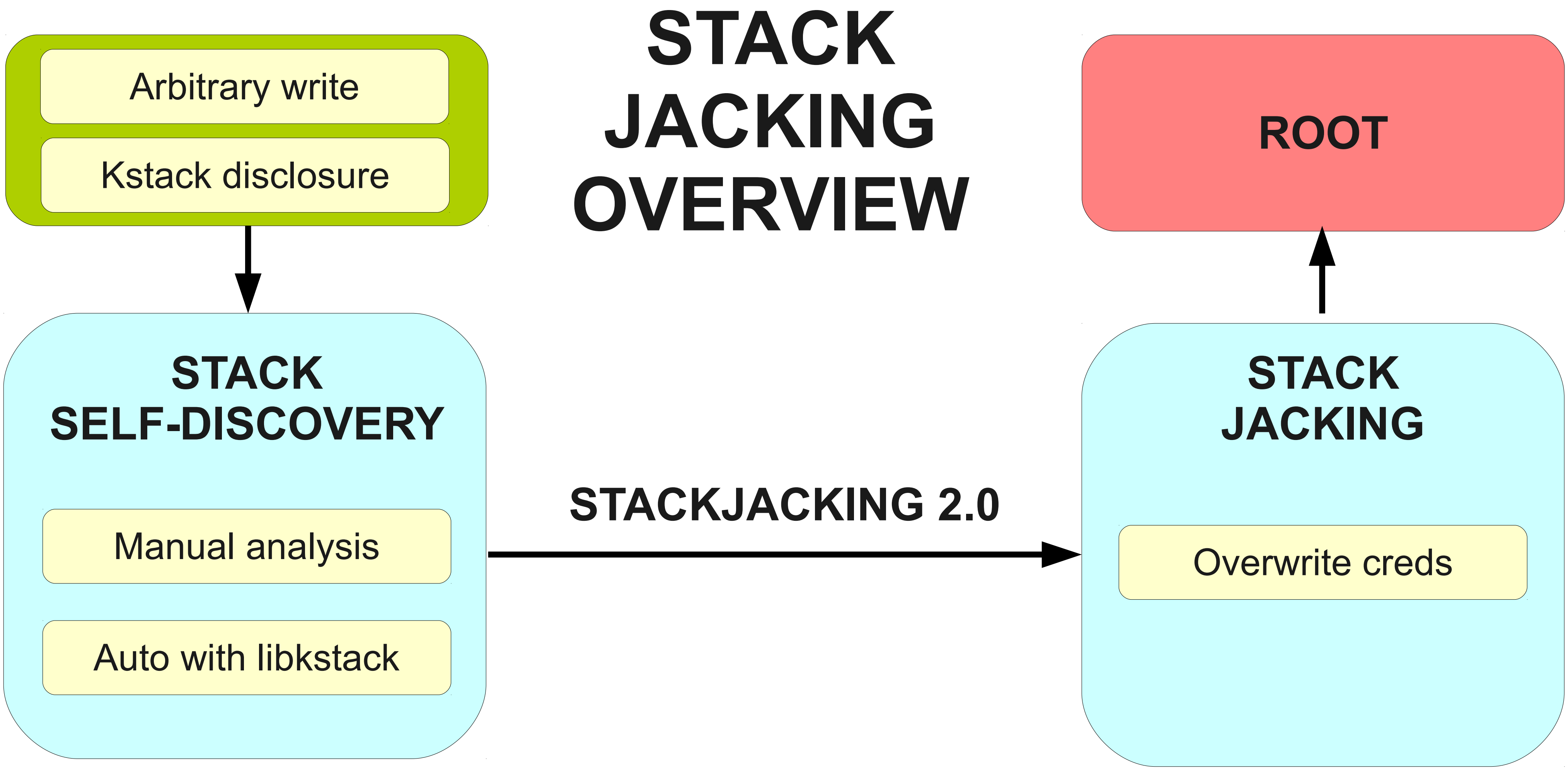


RANDKSTACK CONSIDERED HARMFUL!

STRAIGHT TO CREDS!

- 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


Arbitrary write
Kstack disclosure
STACK
JACKING
OVERVIEW
ROOT
STACK
SELF-DISCOVERY
Manual analysis
Auto with libkstack
STACKJACKING 2.0
STACK
JACKING
Overwrite creds

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

SMEP

ALMOST THE END

GREETZ

- #busticati
- ~~1kk1q85Xp$Id.gAcJOg7uelf36VQwJQ/~~
- ;PpPppPpPpPPPpP

QUESTIONS?

Jon Oberheide
jon@oberheide.org
Duo Security

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