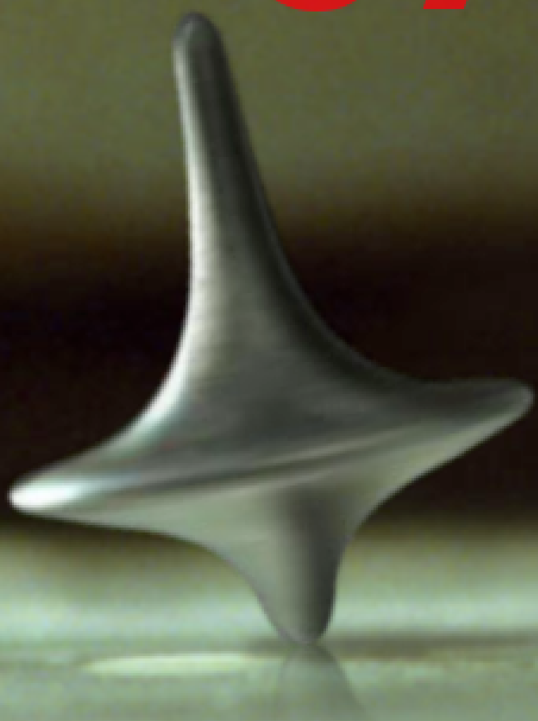
STACK JACKING

YOUR WAY TO GRSEC/PAX BYPASS



JON OBERHEIDE

DAN ROSENBERG

Introduction

- Jon Oberheide
- Dan Rosenberg

Introduction

In reference to the HES CFP:

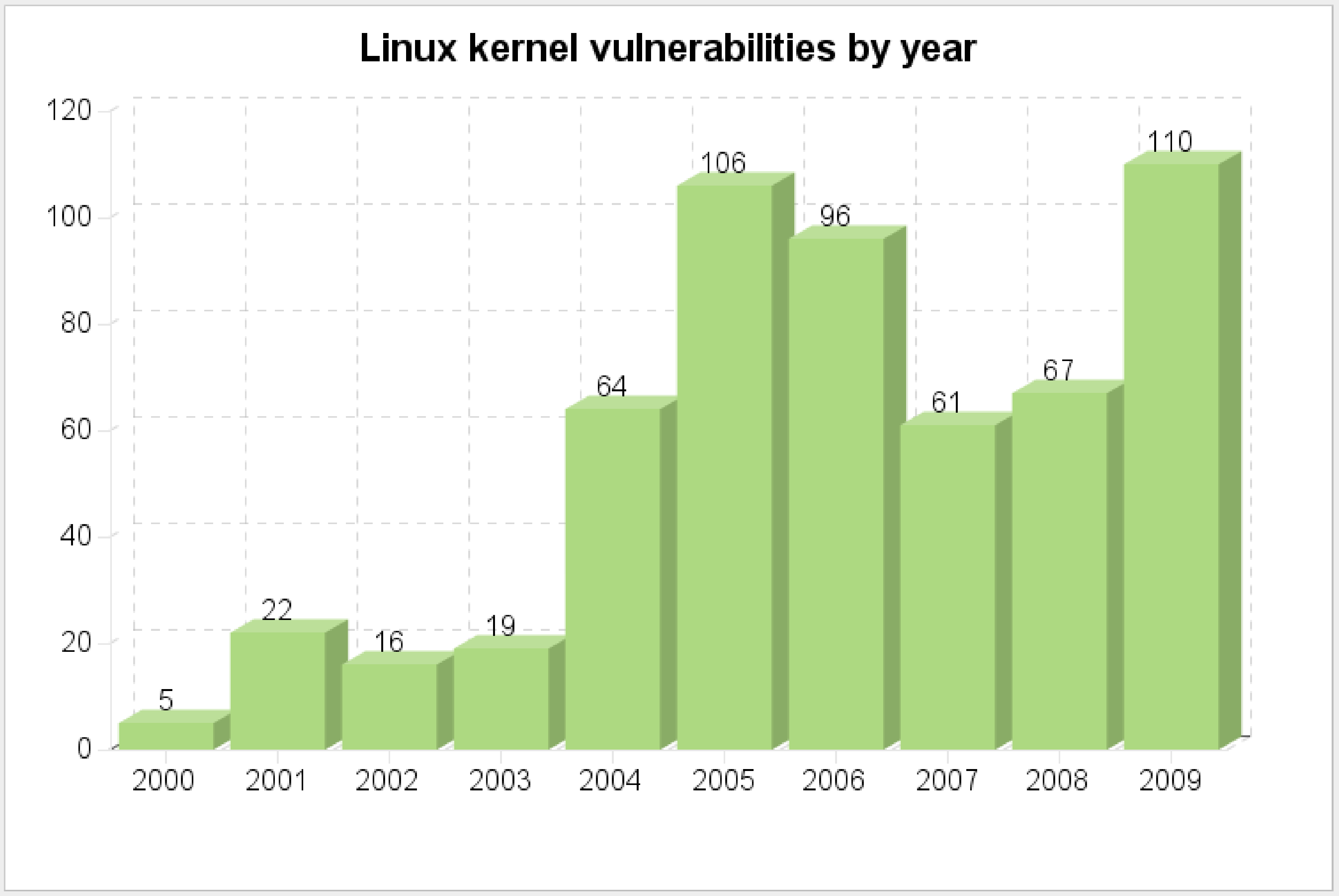
"I get excited every time I see a conference add requirements to their talk selection along the lines of 'exploitation presentations must be against grsecurity/PaX' -- but then there never ends up being any presentations of this kind."

– spender pratt

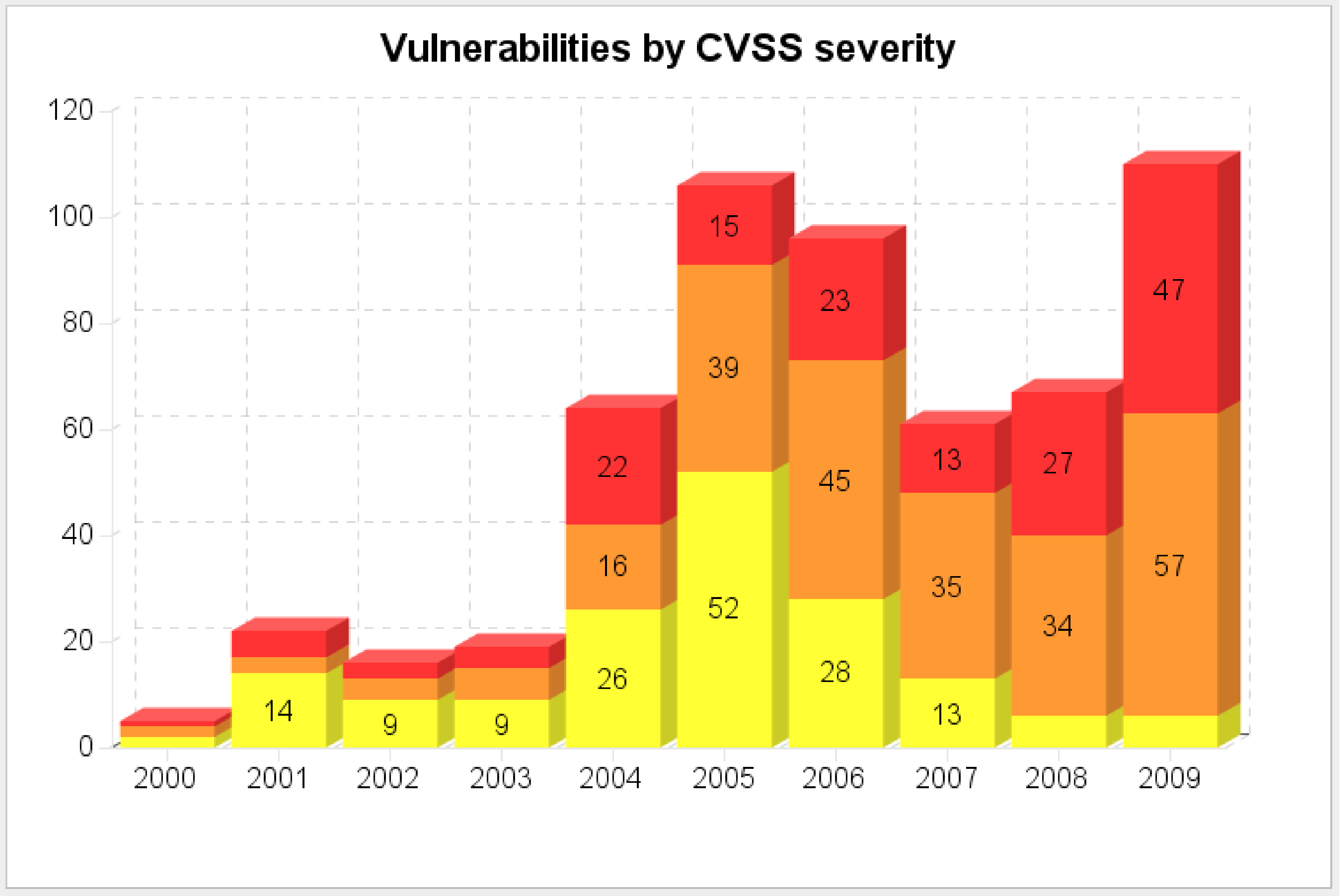
Agenda

- **A review of Linux kernel security**
- Exploitation vs. grsecurity/PaX
- Bypassing grsecurity/PaX

A decade of kernel security



A decade of kernel security



Upstream attitude

```
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
- Linux still hasn't had its “security awakening”

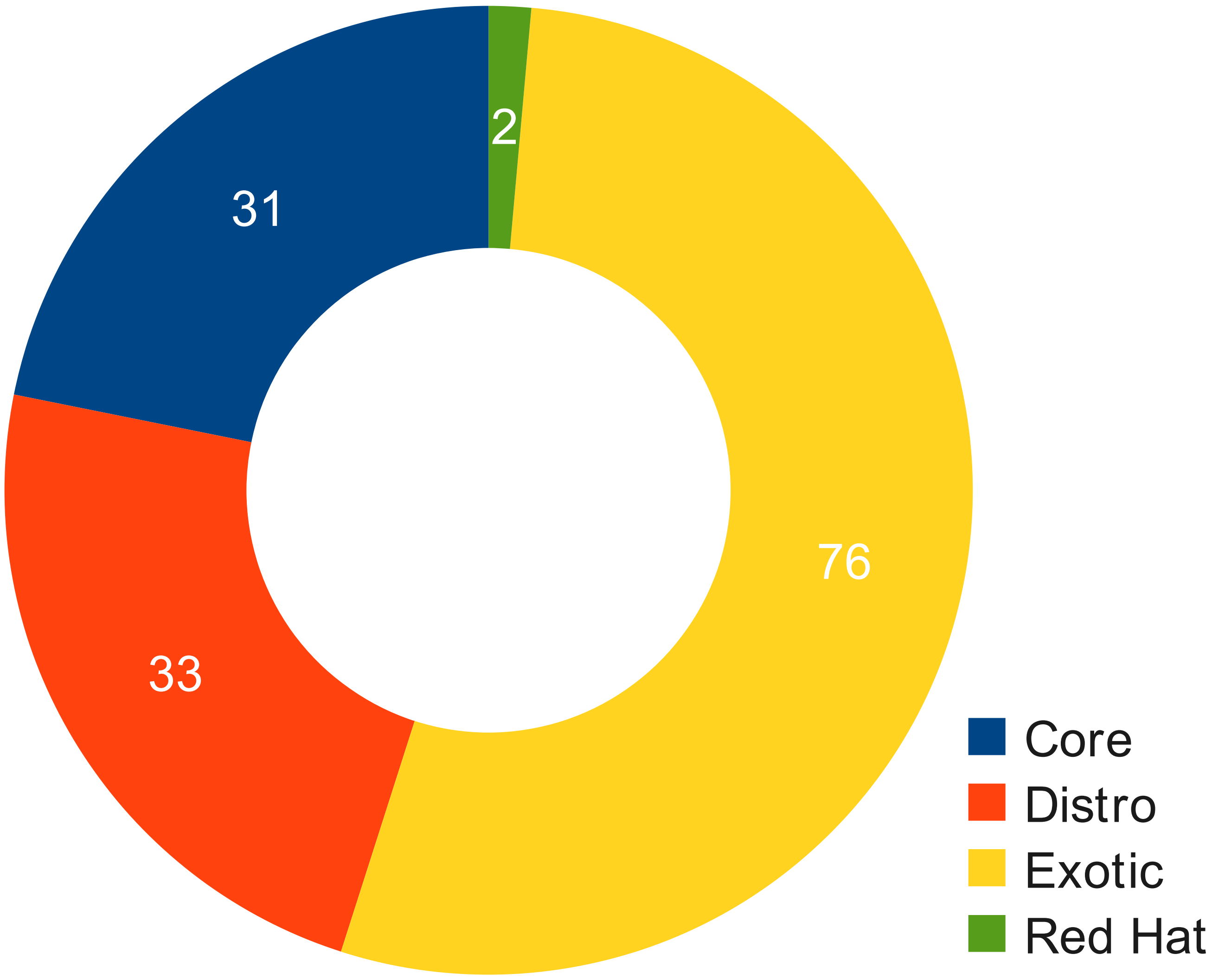
How about last year?

- 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)

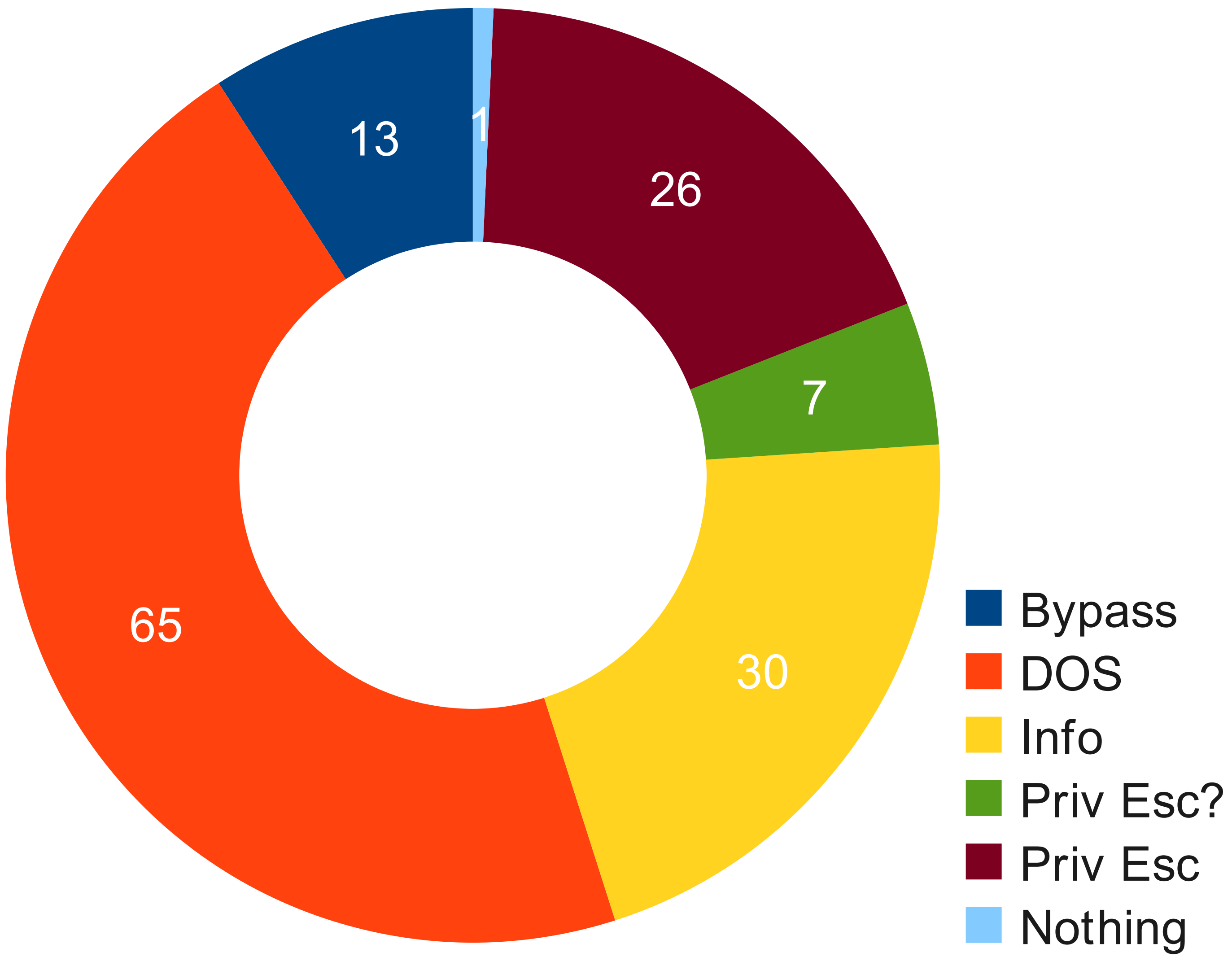
Kernel vulns in 2010

- 12 known exploits for local privilege escalation
- 13 remotely triggerable issues
- 33 potential privilege escalations

Breakdown by Target



Breakdown by Impact



Interesting 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

Agenda

- A review of Linux kernel security
- **Exploitation vs. grsecurity/PaX**
- Bypassing grsecurity/PaX

Traditional Linux exploitation

- Perhaps most general exploitation primitive is an arbitrary kernel write
- Sometimes occurs naturally, other times can be constructed (e.g. overwriting pointers in an overflow to trigger a write)

Linux exploitation examples

- 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

Overview of grsecurity/PaX

- grsecurity/PaX
  - Third-party patchset to harden Linux userspace/kernel security
- Attempts to prevent
  - Introduction/execution of arbitrary code
  - Execution of existing code out of original order
  - Execution of existing code in original order with arbitrary data

grsecurity/PaX hardening

- Kernel hardening features:
  - 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)

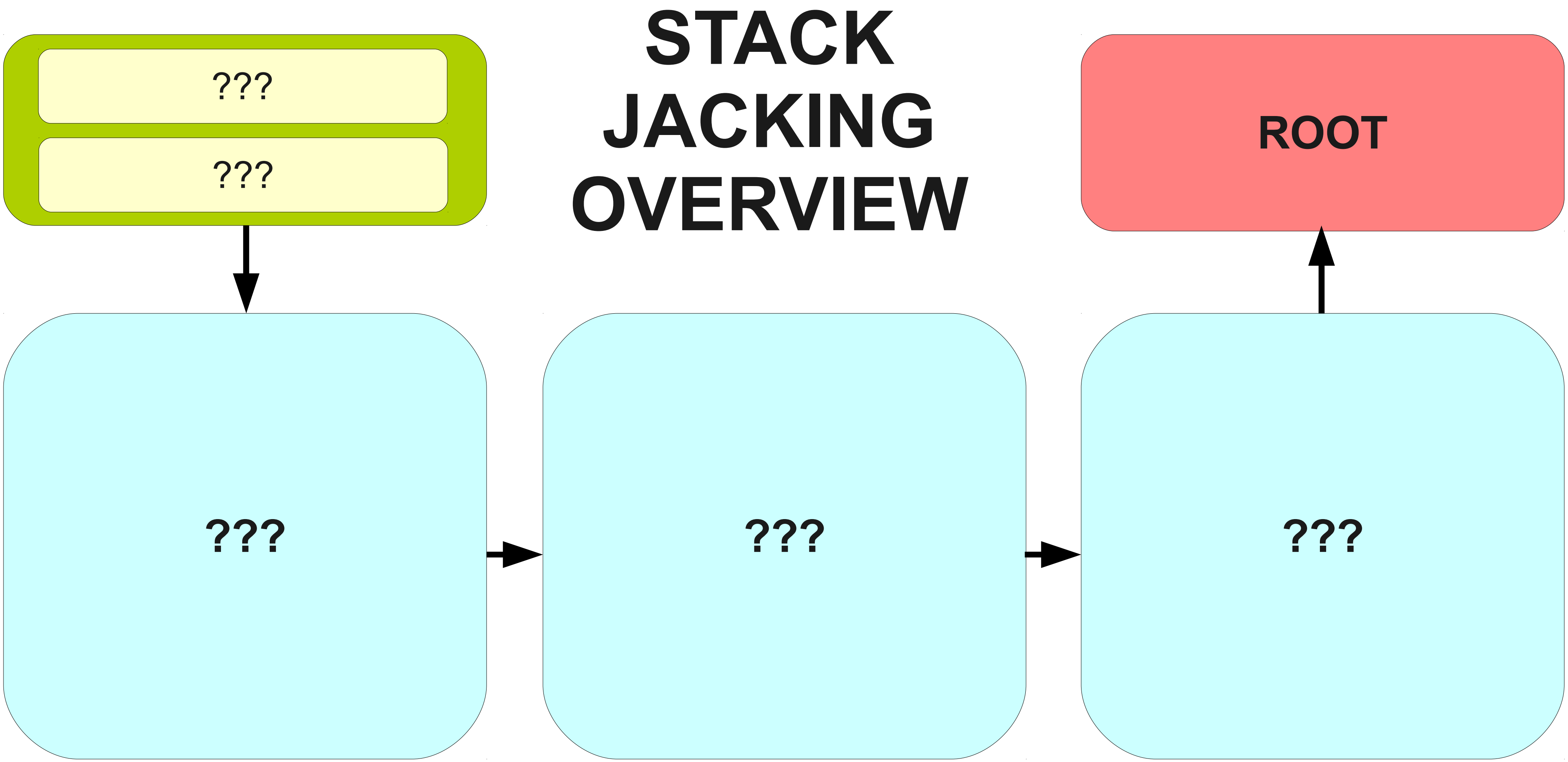
Agenda

- A review of Linux kernel security
- Exploitation vs. grsecurity/PaX
- **Bypassing grsecurity/PaX**

The main event

- A technique we call ***stackjacking***
  - Enables the bypass of common grsecurity/PaX configurations with common exploit primatives
  - 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...

Stronger target assumptions

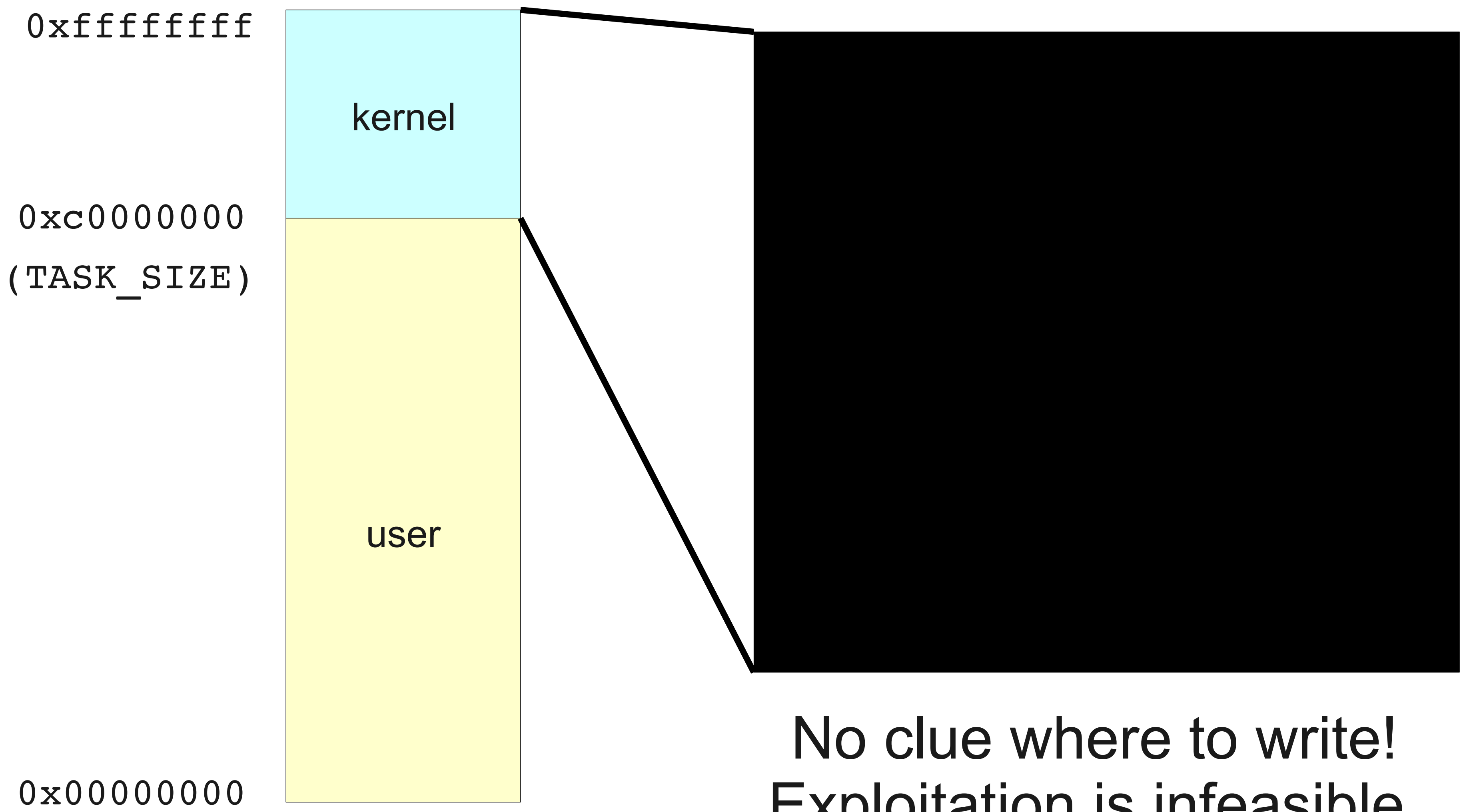
- Let's make some extra assumptions
  - We like a challenge, and these are assumptions that may possibly be obtainable now or in the future
- Stronger target assumptions
  - Zero knowledge of kernel address space
  - Fully randomized kernel text/data
  - Cannot introduce new code into kernel address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

Attacker 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 into the abyss

DARKNESS!



No clue where to write!
Exploitation is infeasible.

What's the secret sauce?

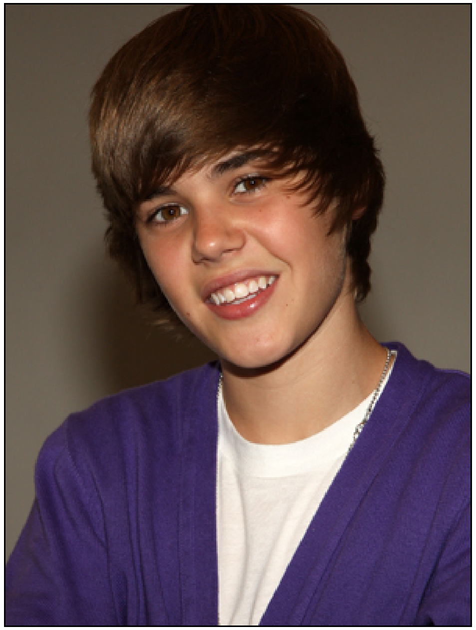
ARBITRARY WRITE + ? = <3

The Great Sandwich?

ARBITRARY WRITE +  = <3

msuiche?

Nah, he's taken



+

= <3

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

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???

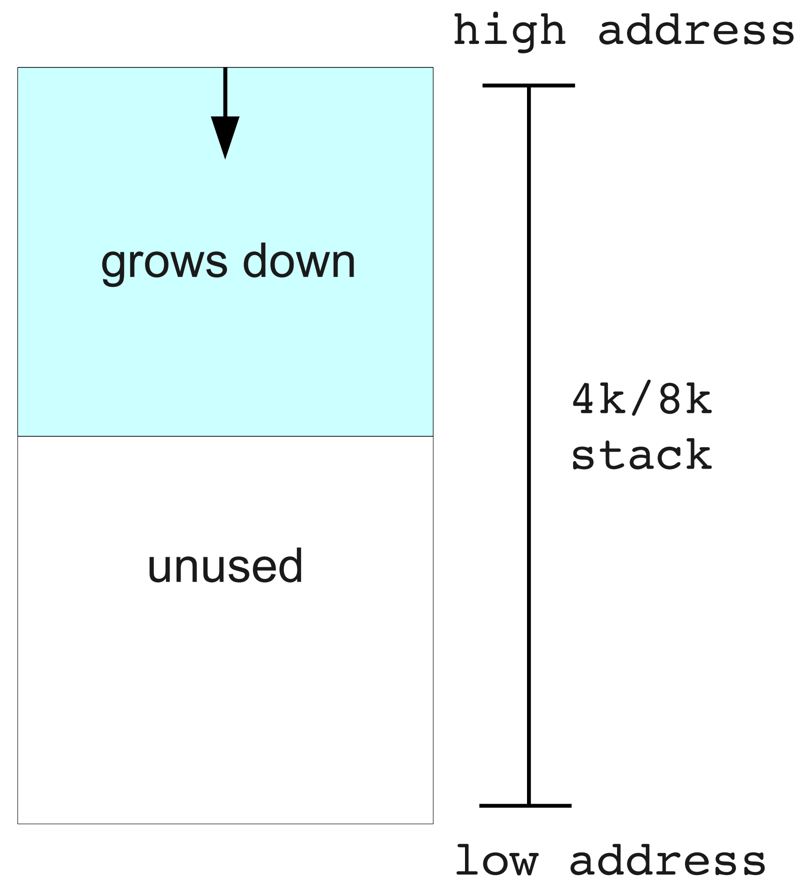
KSTACK MEM DISCLOSURE!

ARBITRARY WRITE + **KSTACK LEAK** = <3

How does kstack leak help?

A bit about Linux kernel stacks

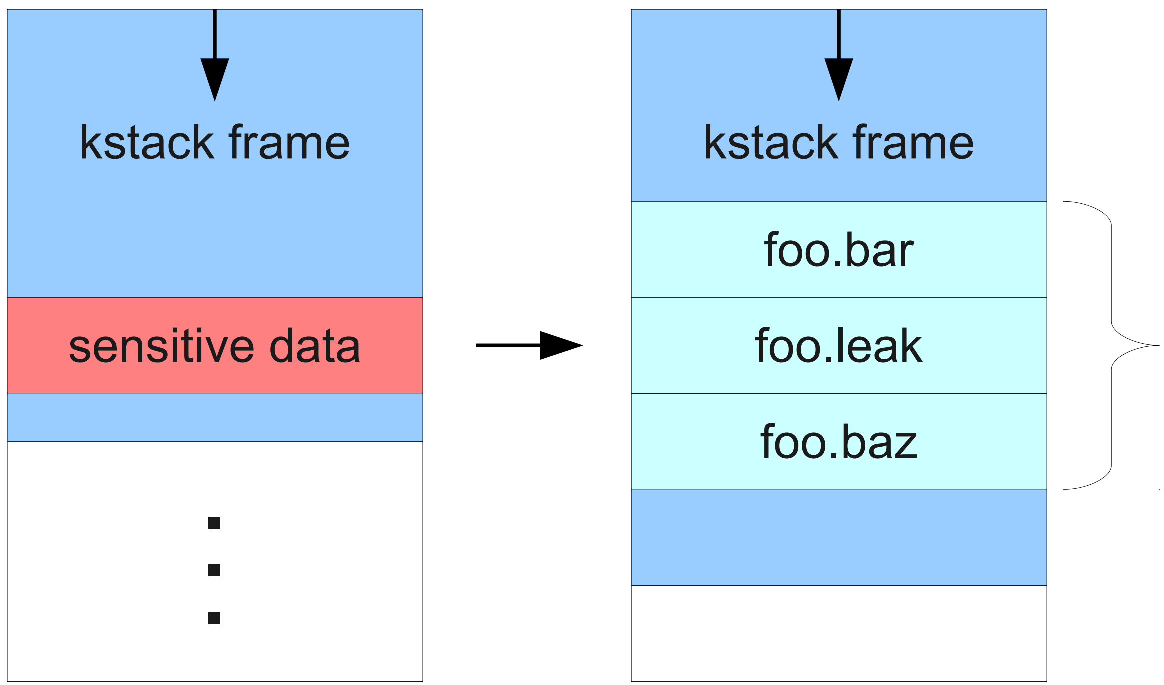
- Each userspace thread is allocated a kernel stack
- Stores stack frames for kernel syscalls and other metadata
- Most commonly 8k, some distros use 4k
  - THREAD_SIZE = 2*PAGE_SIZE = 2*4086 = 8192



Kernel 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

Kernel stack mem disclosures



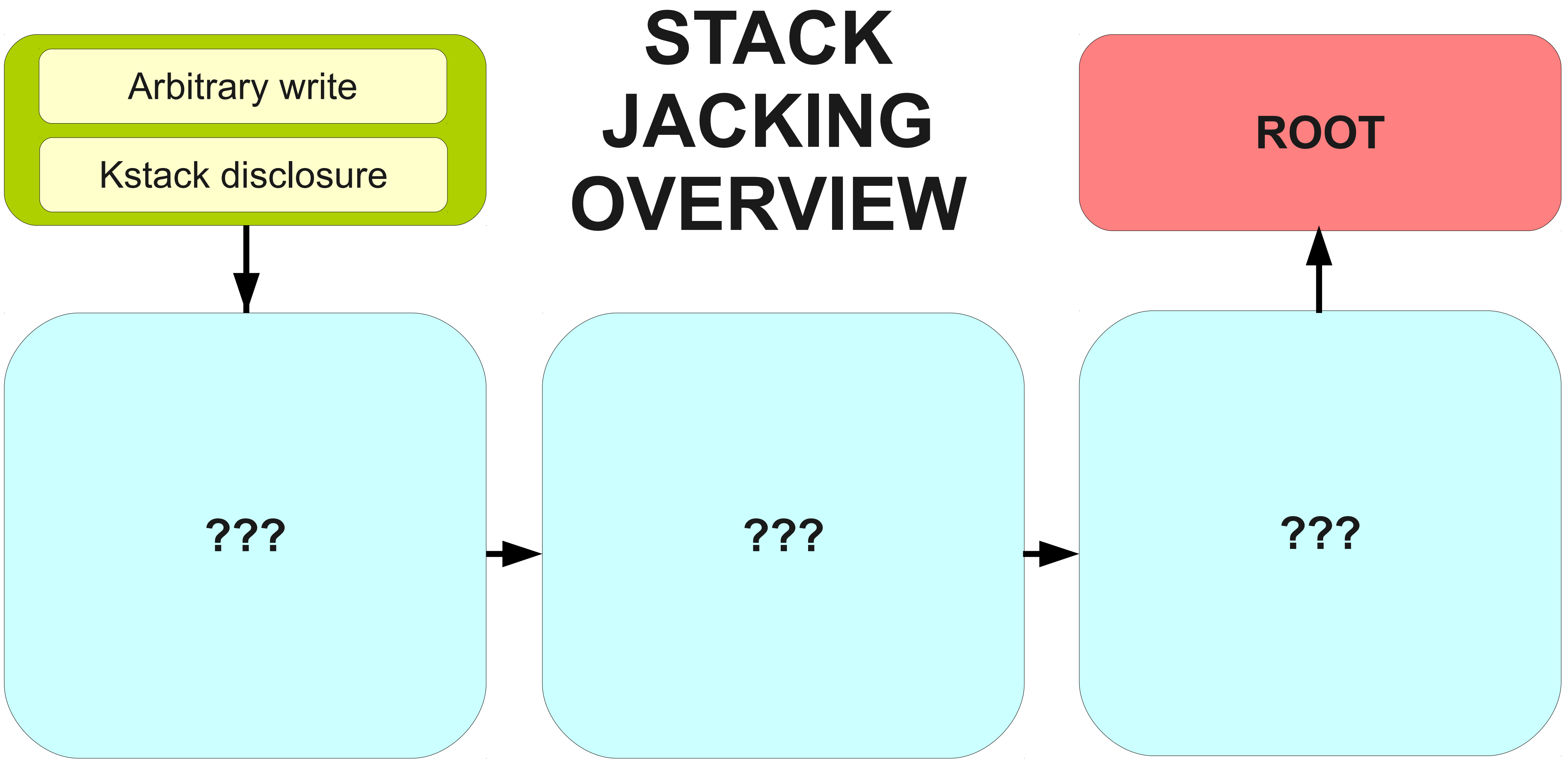

```
struct foo {
  uint32_t bar;
  uint32_t leak;
  uint32_t baz;
};

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 the kstack?

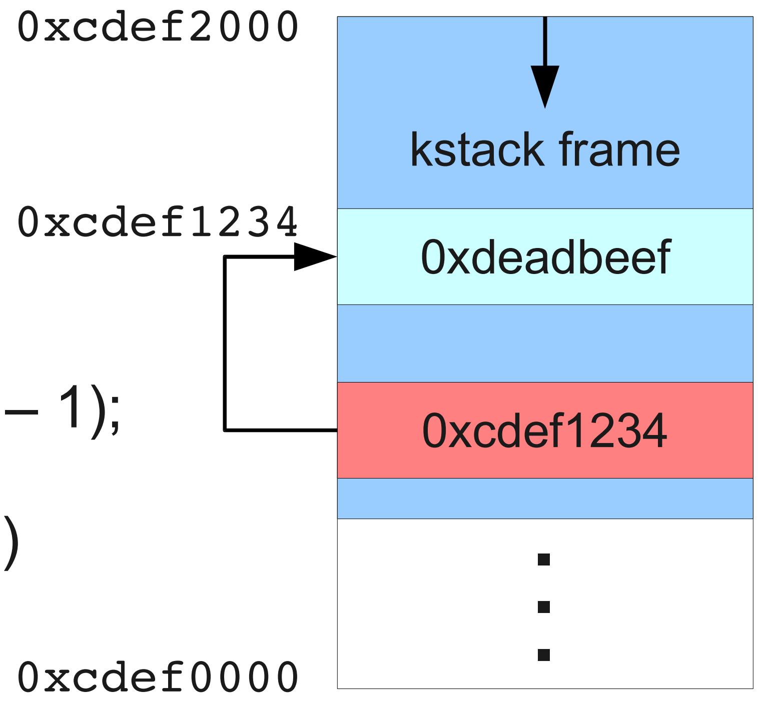
- Leak data off kstack?
  - Sensitive data left behind? Not really...
- Leak addresses off kstack?
  - Sensitive addresses left behind? Maybe...
    - Pointers to known structures could be exploited
    - Too specific of an attack!
- Need something more general
  - kstack disclosures differ widely in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

Kernel stack 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***

Effective kstack 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

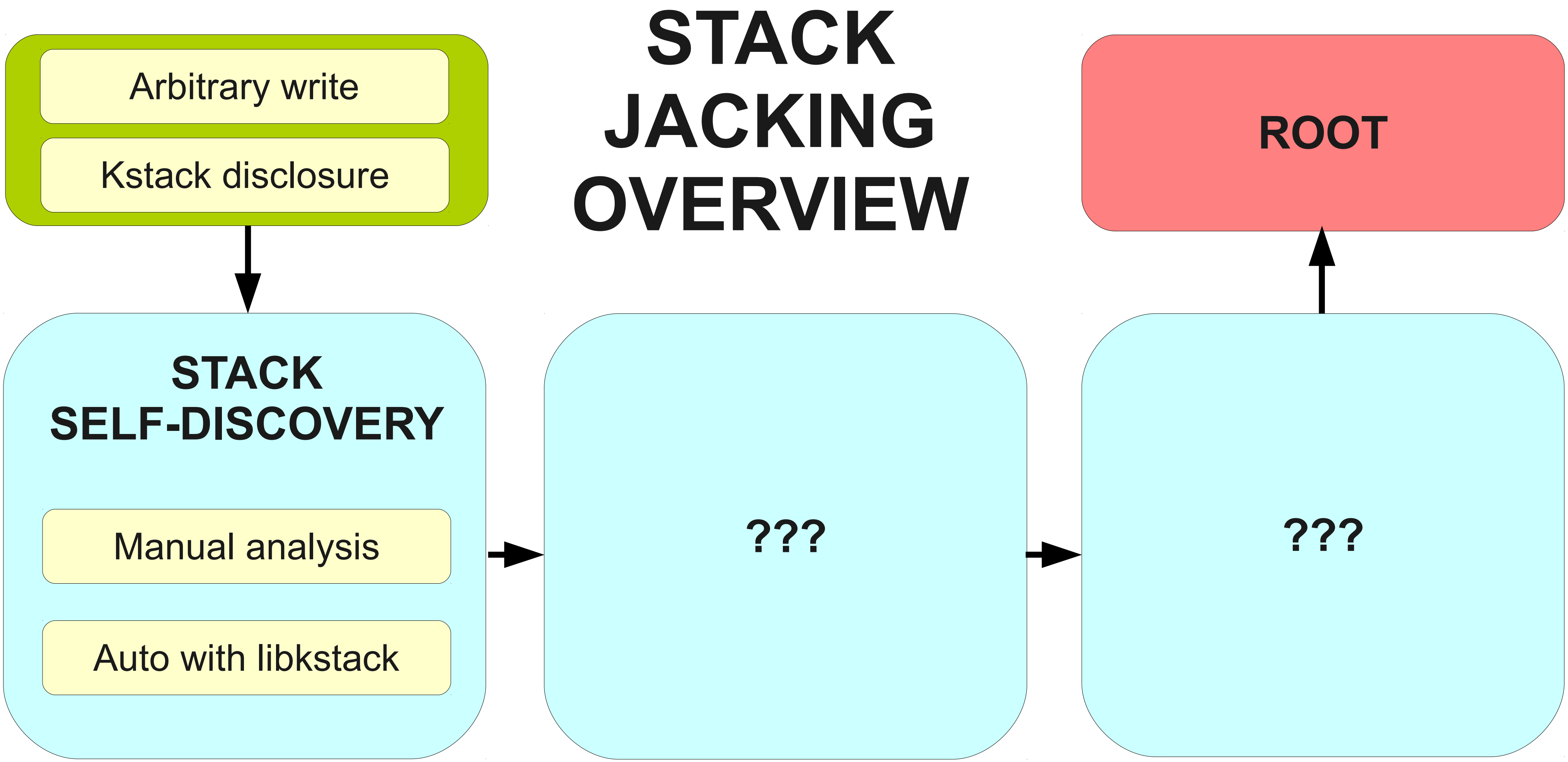
Manual kstack self-discovery

- Manual, offline analysis
  - 1. prime stack with random syscall
  - 2. leak bytes, see if any leaks match real kstack
  - 3. repeat until we've collected enough bytes
  - 4. construct list of priming syscalls needed for the particular leak to spill the beans

Automatic with libkstack

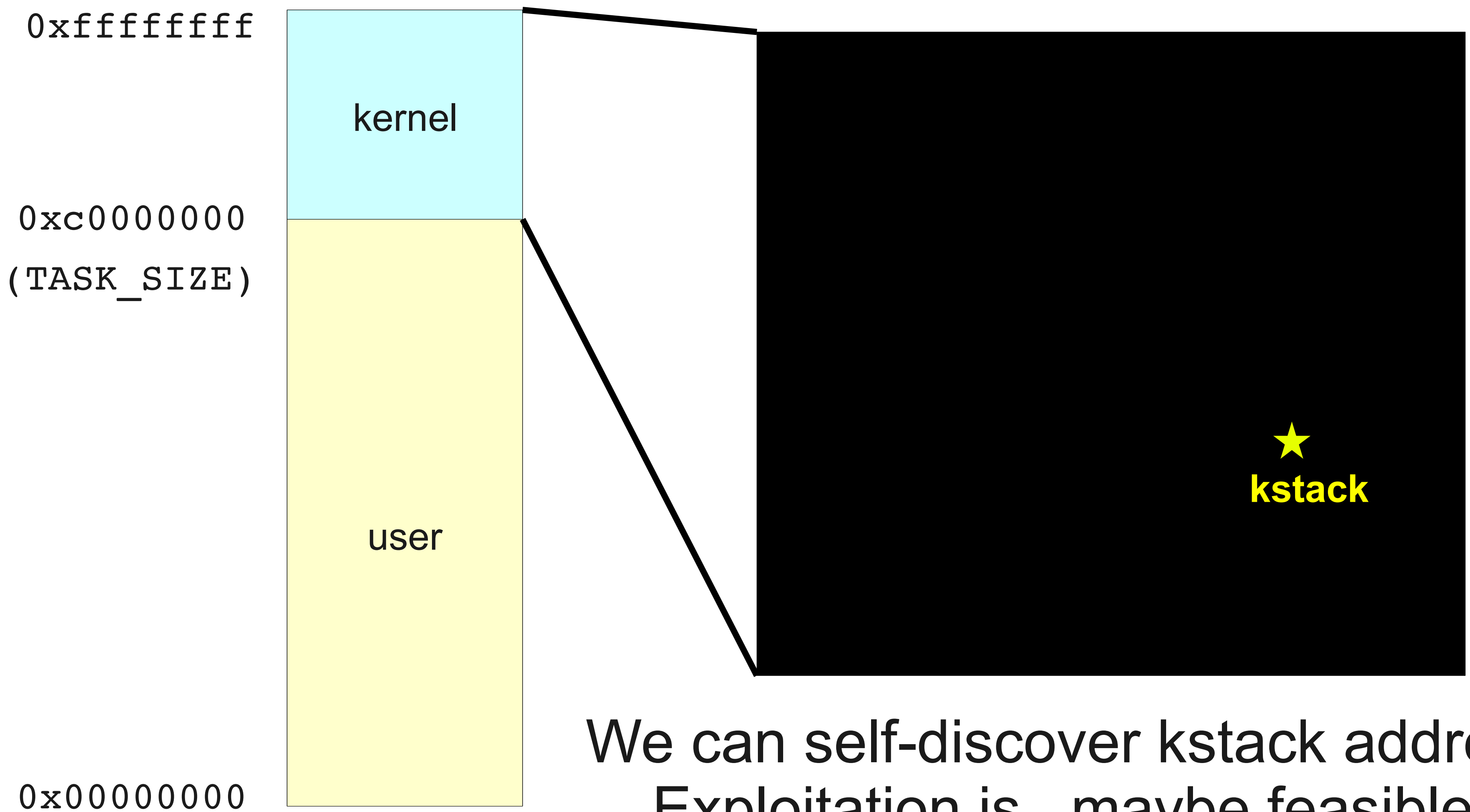
- We can automate this process for runtime self-discovery with libkstack
  - 1. prime stack with random syscall
  - 2. leak bytes, infer whether bytes belong to a kstack addr
  - 3. repeat until we have sufficient confidence to calculate the kstack base addr

Plan of attack!



No longer complete darkness

A random pinpoint of light!



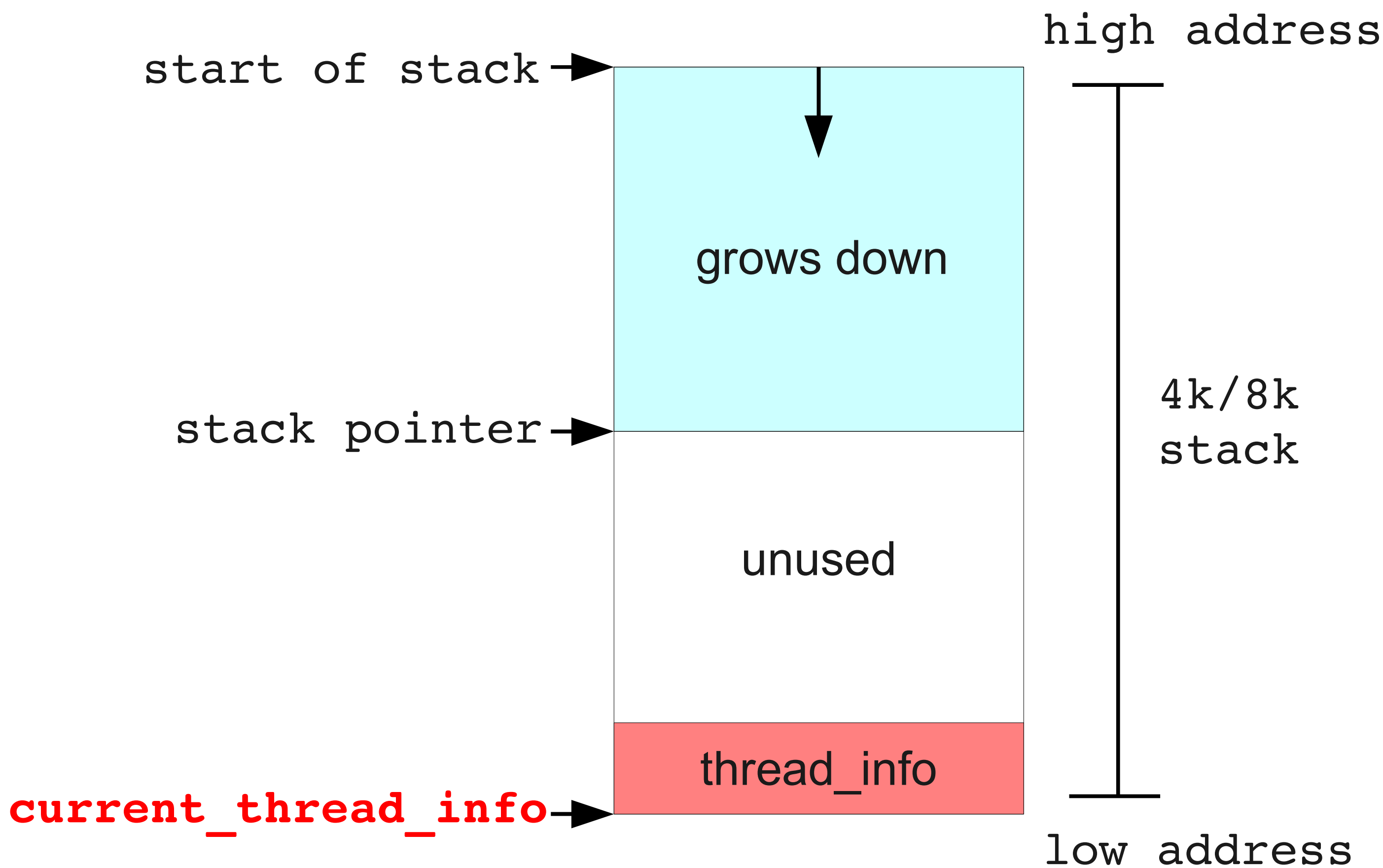
We can self-discover kstack address!
Exploitation is...maybe feasible?

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 kernel stack

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?

restart_block func ptr?

```
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;
};
```

- restart_block?
  - Has a func ptr we can overwrite and invoke via userspace!
  - Can't point to userspace (UDEREF)
  - Can't point to kmem (blackbox)
  - Plus assuming no control flow mod

task_struct pointer?

```
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;
};
```

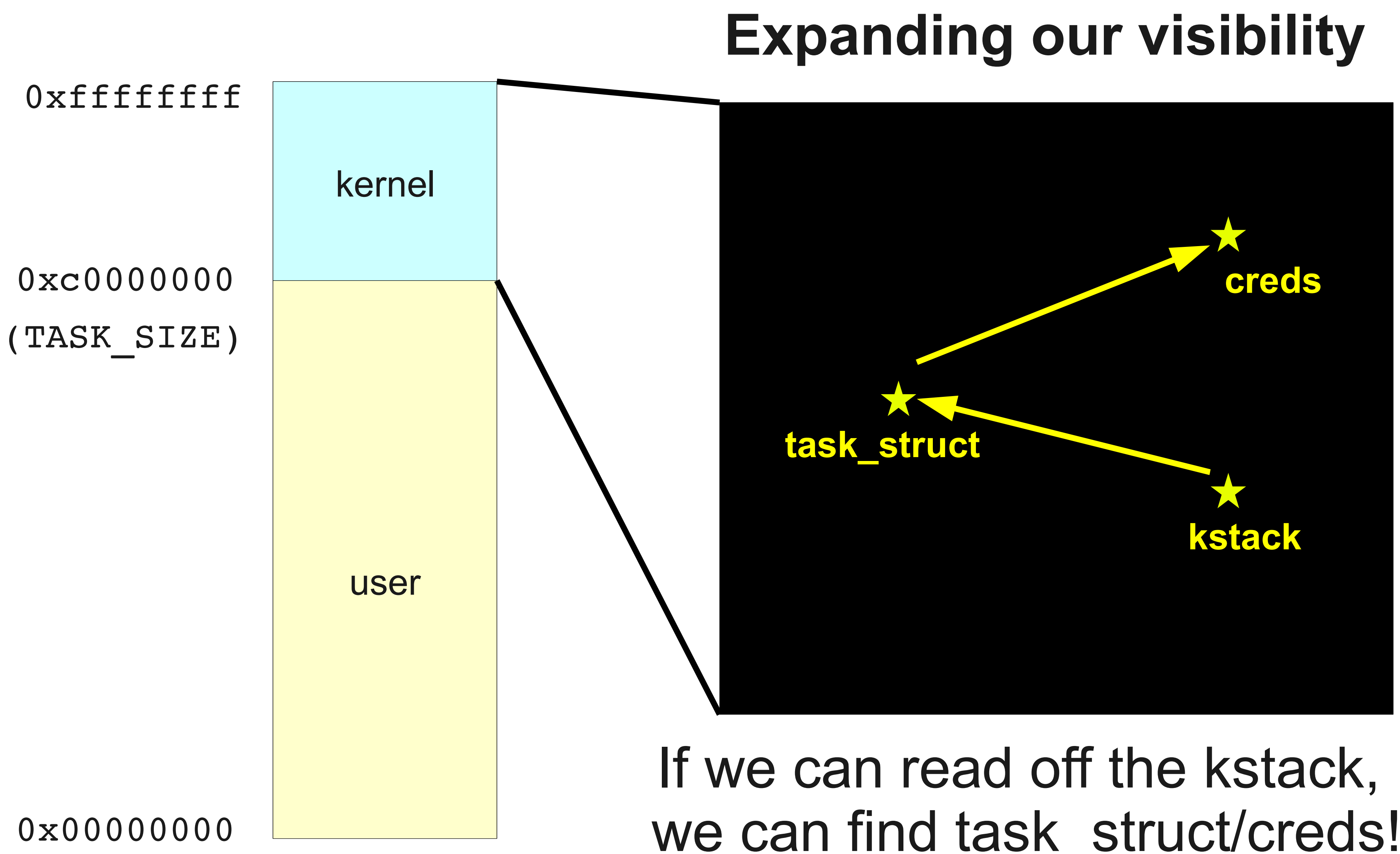
- task_struct?
  - Could point it at init_task_struct for getting creds/caps of the init task
  - But we don't know the address of init_task_struct!

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

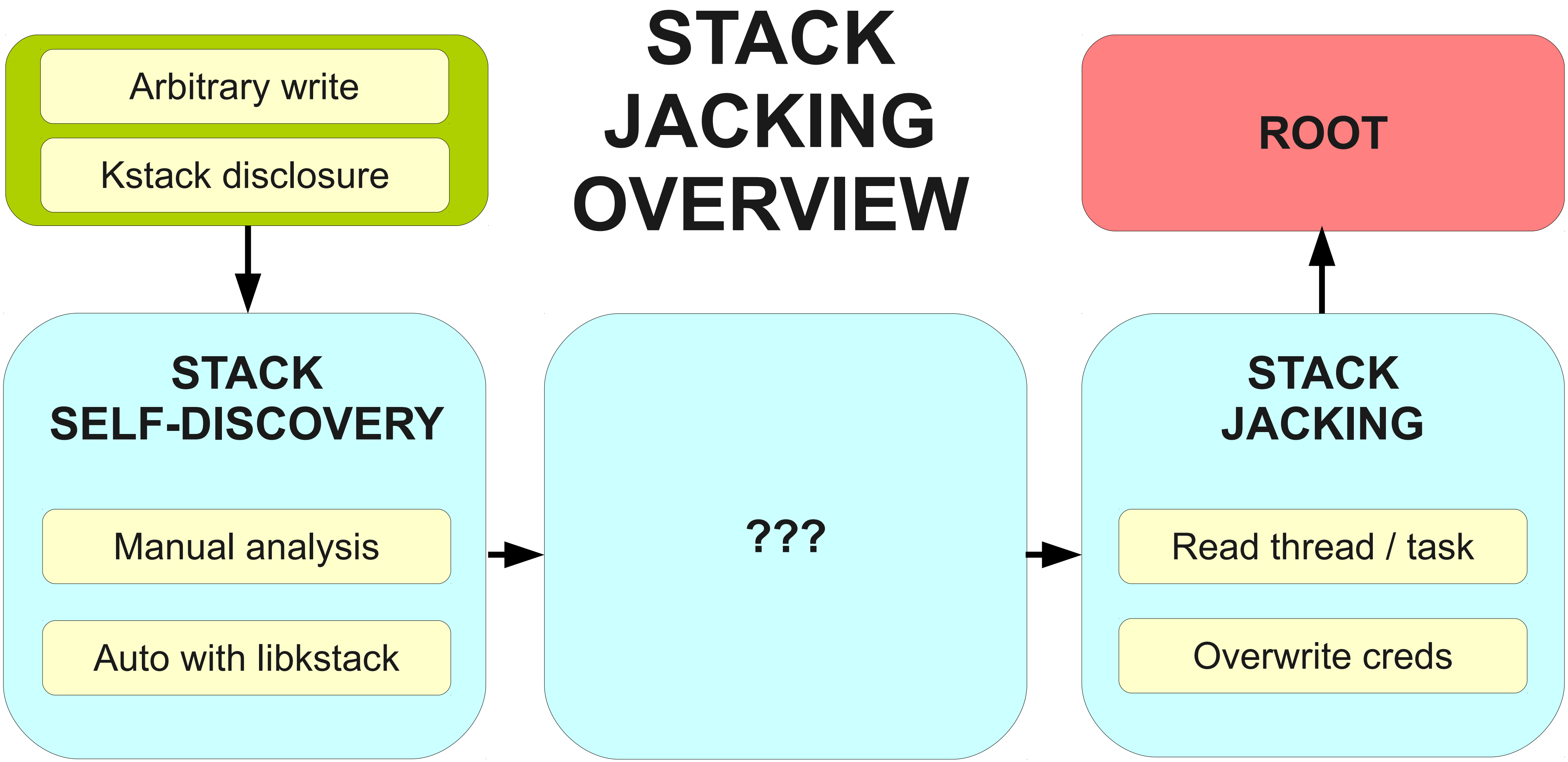


If we can read off the kstack, we can find task_struct/creds!

Attacking task_struct

- We have an arbitrary write on kstack
  - 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!



The Rosengrope Technique

Vanilla kernel

- No segmentation, user/kernel separation enforced by paging
- copy_*_user functions check user pointers against addr_limit (per-thread variable in thread_info struct)
- On vanilla, setting addr_limit to KERNEL_DS (ULONG_MAX) gives arbitrary read/write (all checks pass)

set_fs()

- Sometimes kernel wants to reuse code with kernel pointer arguments
  - kernel_sendmsg, kernel_recvmsg, etc.
- Calls set_fs(KERNEL_DS) to set addr_limit and allow copy_*_user functions to copy kernel-to-kernel
- Careful to make sure no user-influenced pointers are used

PAX_UDEREF

- Strict user/kernel separation using segmentation
- Reload segment registers at kernel traps, used during copy operations
  - Fault on invalid access

PAX_UDEREF and KERNEL_DS

- 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

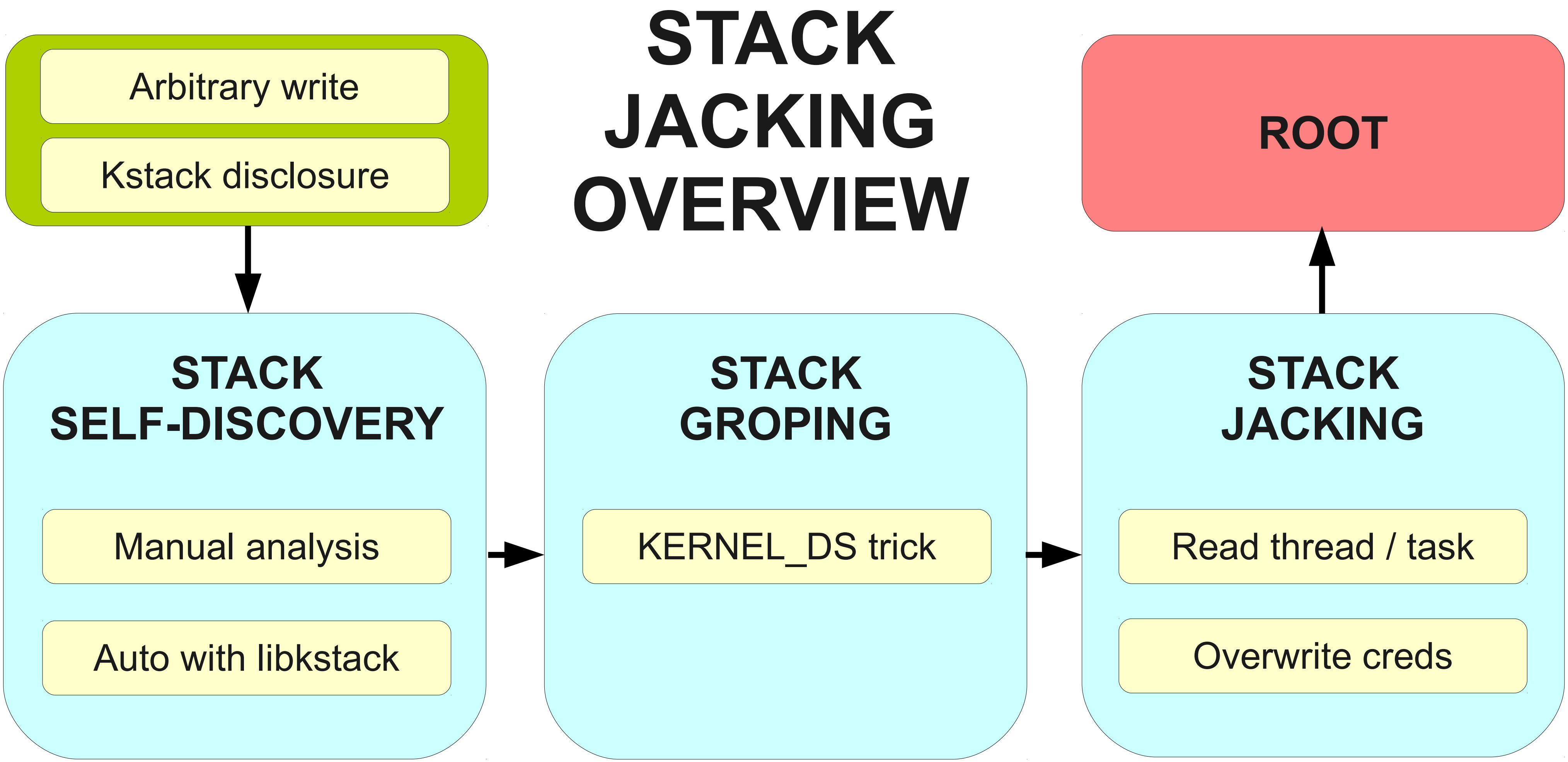
But...

- %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!

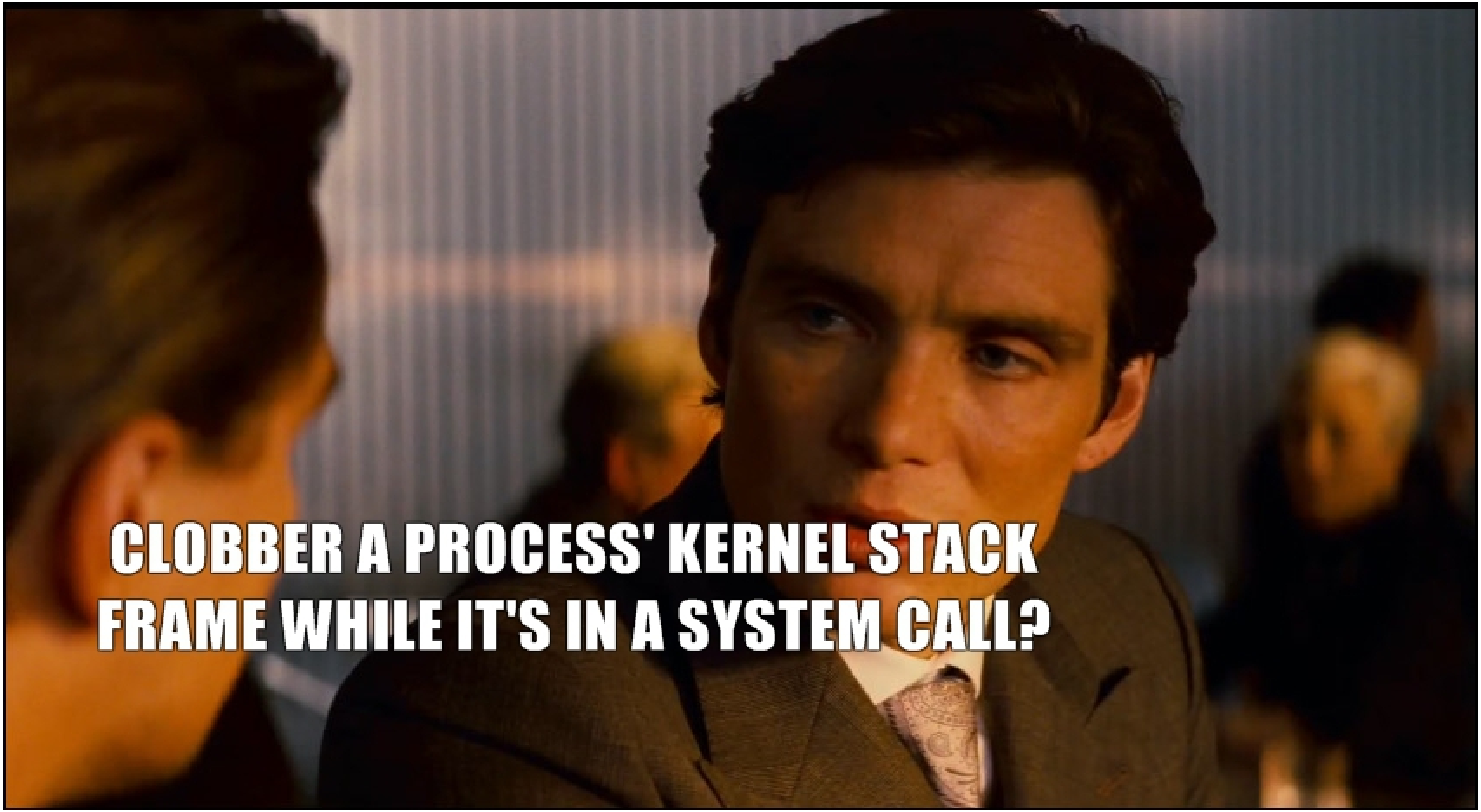


Pros and cons of KERNEL_DS

- KERNEL_DS
  - 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...

The Obergrope Technique

The Obergrope Technique



The Obergrope Technique

Attacking the kstack frames

- A different approach
  - 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 stack 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

A problem

- How can we write to our own kstack?
  - Unlikely to be able to write into our own stack while exploiting the vulnerability for our arbitrary write
- Use parent/child processes
  - Child self-discovers kstack addr
  - Passes kstack addr to parent
  - Parent writes into child while child is in syscall

More problems

- 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...

Winning Linux kernel races

- How to win Linux kernel races
  - Get very lucky w/scheduling on SMP machine
  - Cause a resource to be in contention (eg. locks)
  - Cause kernel to page in from slow I/O device (sgrakkyu)
- Ehhh...
  - We might hose the kernel if we lose the race
  - Anything better?

A twist on winning races

- This isn't a “standard” race though
  - We can have child execute ANY codepath that performs copy_to_user() with a src arg on kstack
- Enter, sleepy syscalls!
  - Syscalls that allow us to put process to sleep for an arbitrary amount of time
  - nanosleep, wait, select, etc

Sleepy syscall conditions

- 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 disasm

```
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
   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

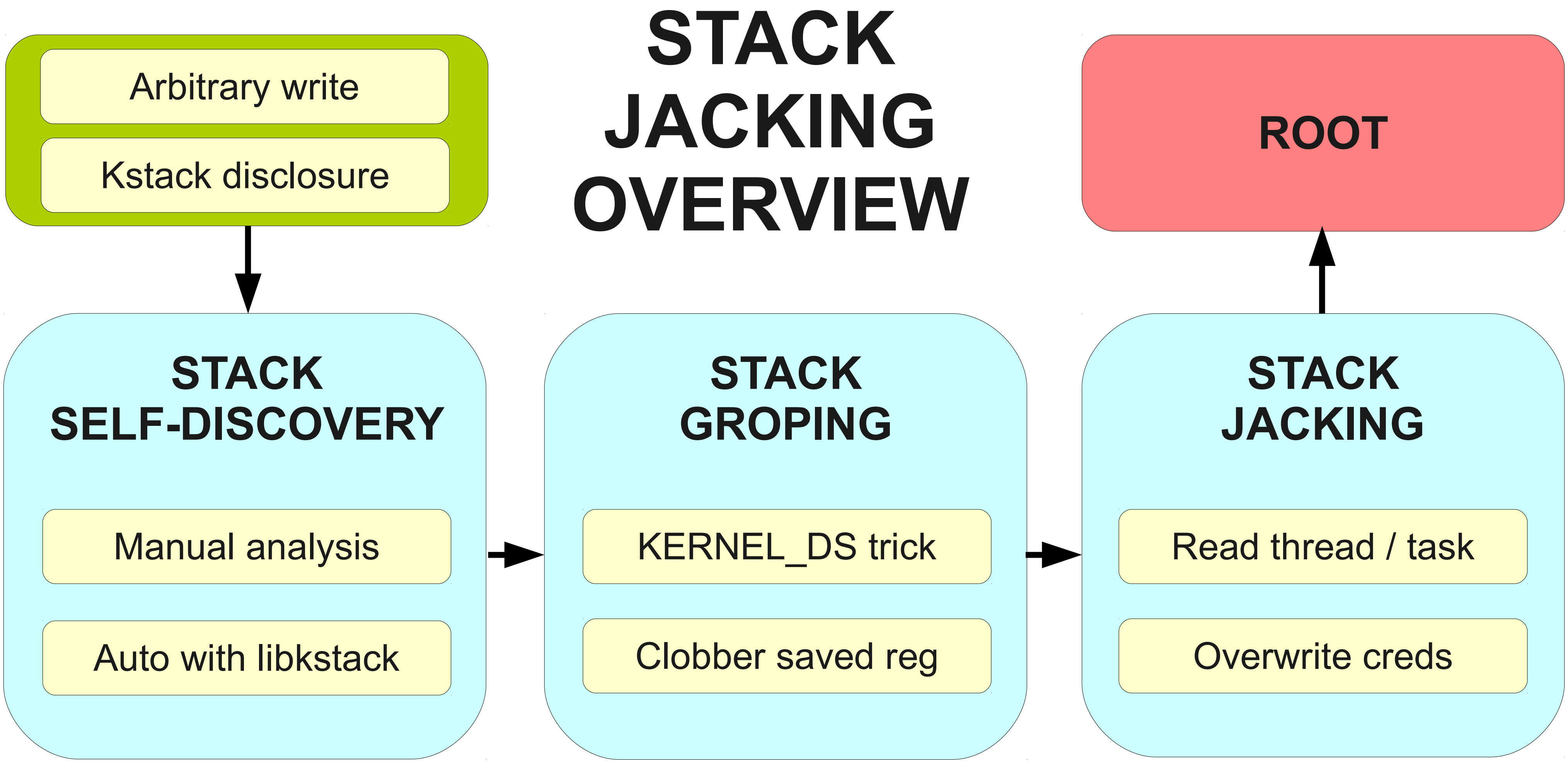
compat_sys_waitid reliability

- Is this reliable across kernel versions?
  - Yes, tested on:
    - Lucid default build vmlinuz-2.6.32-24-generic
    - Lucid custom build vmlinuz-2.6.32.26+drm33.12
    - Vanilla build vmlinuz-2.6.36.3
    - Vanilla build + grsec vmlinuz-2.6.36.3-grsec
- How about compilers?
  - Across most gcc 4.x? Needs more investigation
  - Potentially could runtime fingerprint compiler

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!



Live demo!

- Exploit against live hardened system...

Defenses?

- Mitigate the exploitation vectors?
  - Remove thread_info metadata from kstack
  - RANDKSTACK
- Eliminate all kstack disclosures?
  - Clear kstack between syscalls?
  - Compiler/toolchain foo?
- ???

Greetz

- #busticati
- 1kk1q85Xp$Id.gAcJOg7uelf36VQwJQ/
- Those who were already aware of this bypass vector ;-)

Q&A

QUESTIONS?

Jon Oberheide
jon@oberheide.org
Duo Security

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