The Stack is Back

Jon Oberheide

The heap sucks

Heap vs. stack

Excerpt from “*Objective quantitative scientific comparison of the heap and stack*” by Dr. Jono, PhD from the journal of Useless Computer Science:

- ~~Heap~~:
  - Complicated
  - Requires skillz
  - Bad connotation: “heap of trash”
  - The 1%, elitist, pro-life, racist

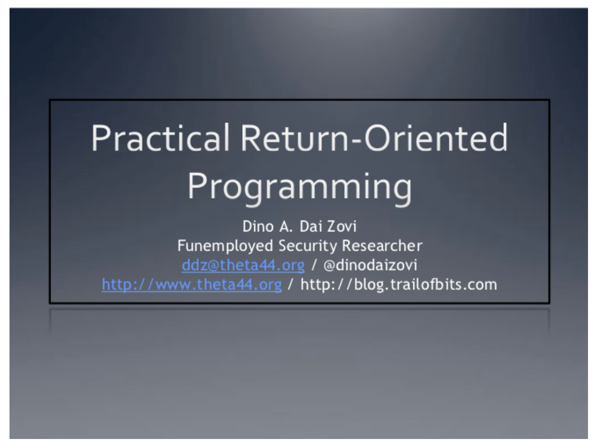
- Stack:
  - Easy
  - Doesn't
  - Good connotation: “stack of bills”
  - Saves kittens from burning buildings

Bringing the stack back

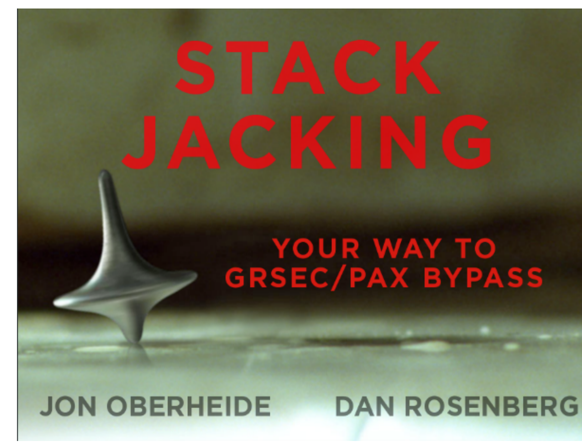
- What's left to exploit with the stack?

```
.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org
```

Smashing?



ROP'ing?



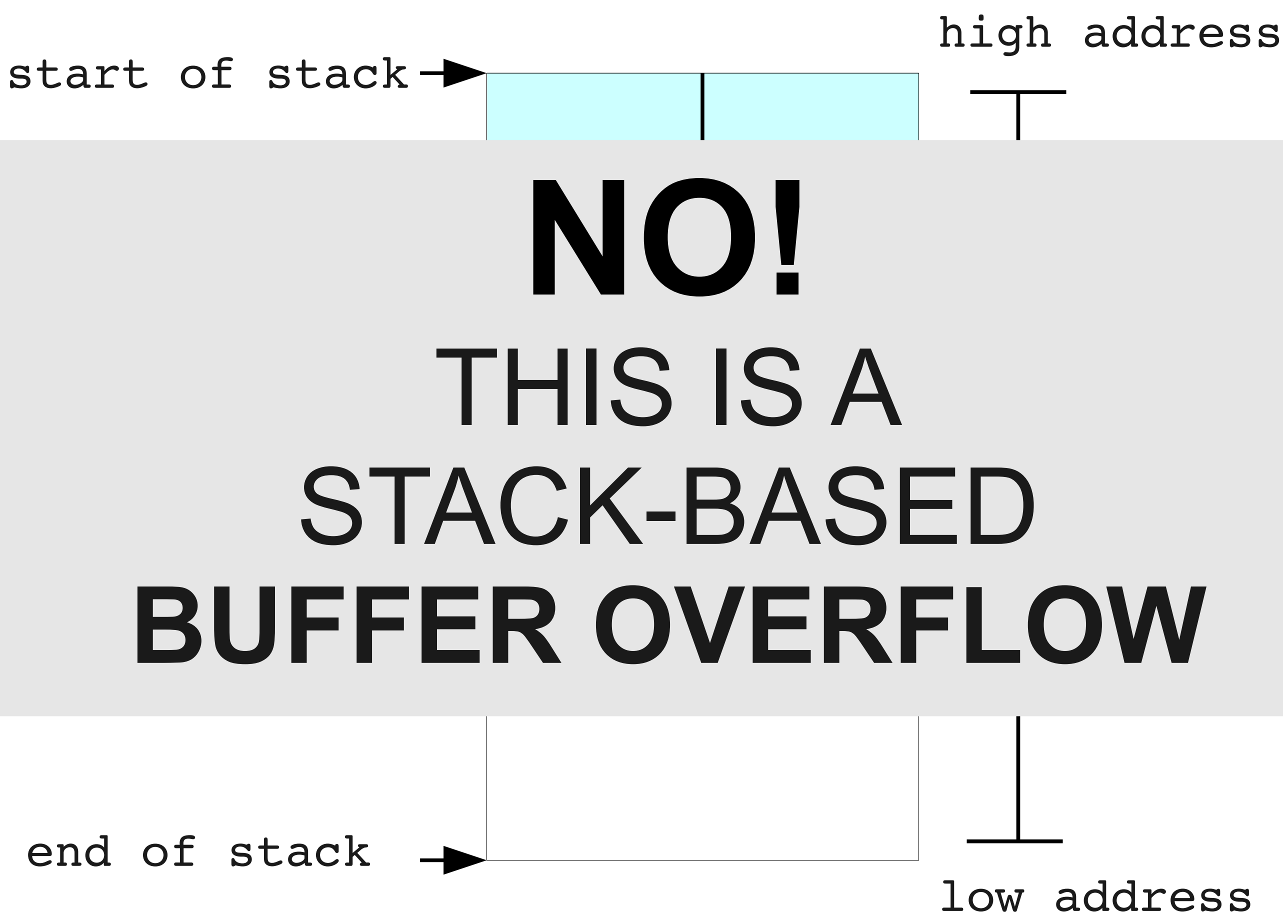
Jacking?

- Let's exploit ***stack overflows***!

The stack is back

- **A brief history of stack overflows**
- Stack overflows in the Linux kernel
- Exploiting exotic stack overflows
- Discovering and mitigating stack overflows

Fake stack overflows



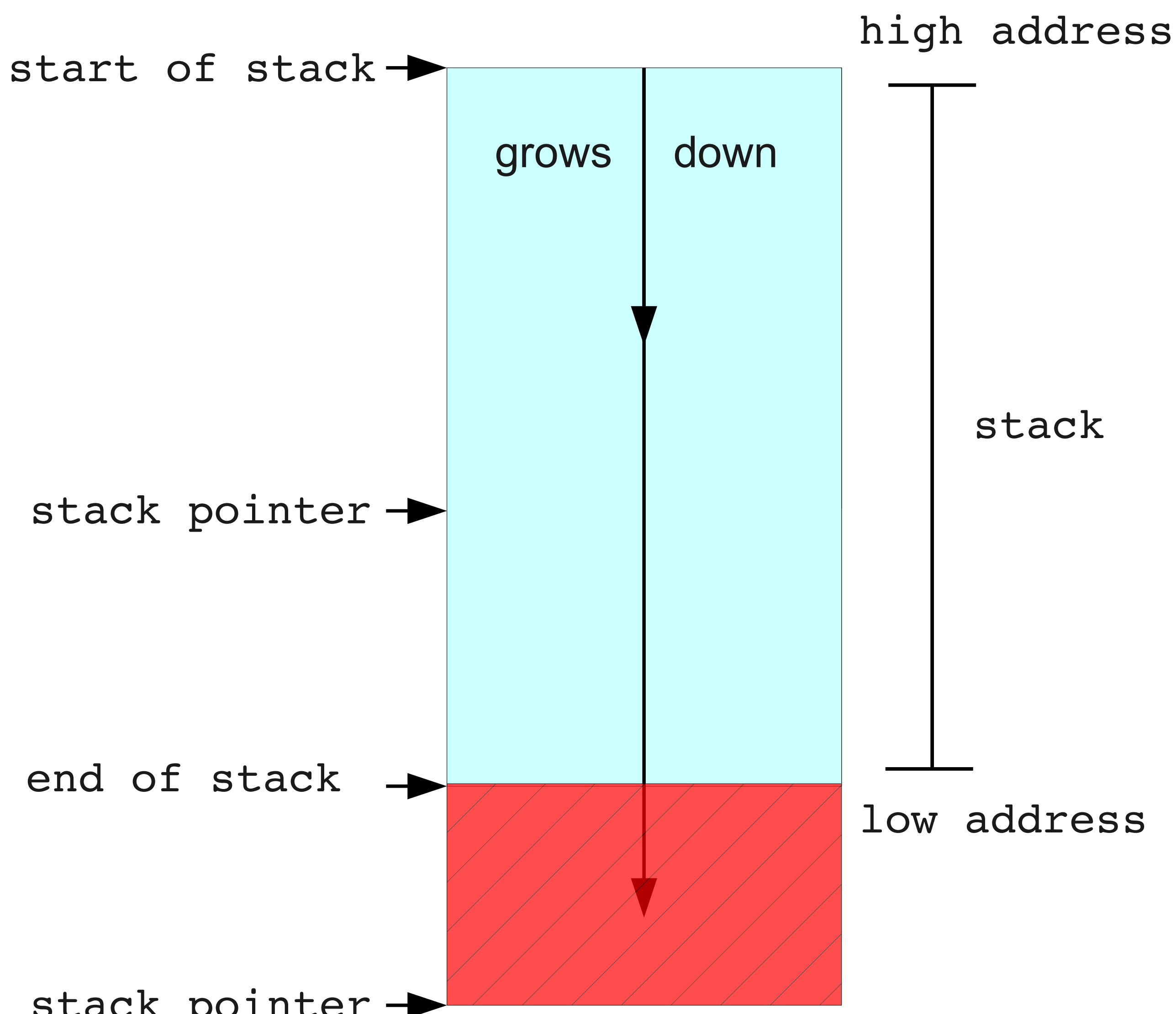
NO!

THIS IS A

STACK-BASED

BUFFER OVERFLOW

Real stack overflows



Stack overflows

- Stack overflows
  - Misuse of terminology
  - Jono's definition:

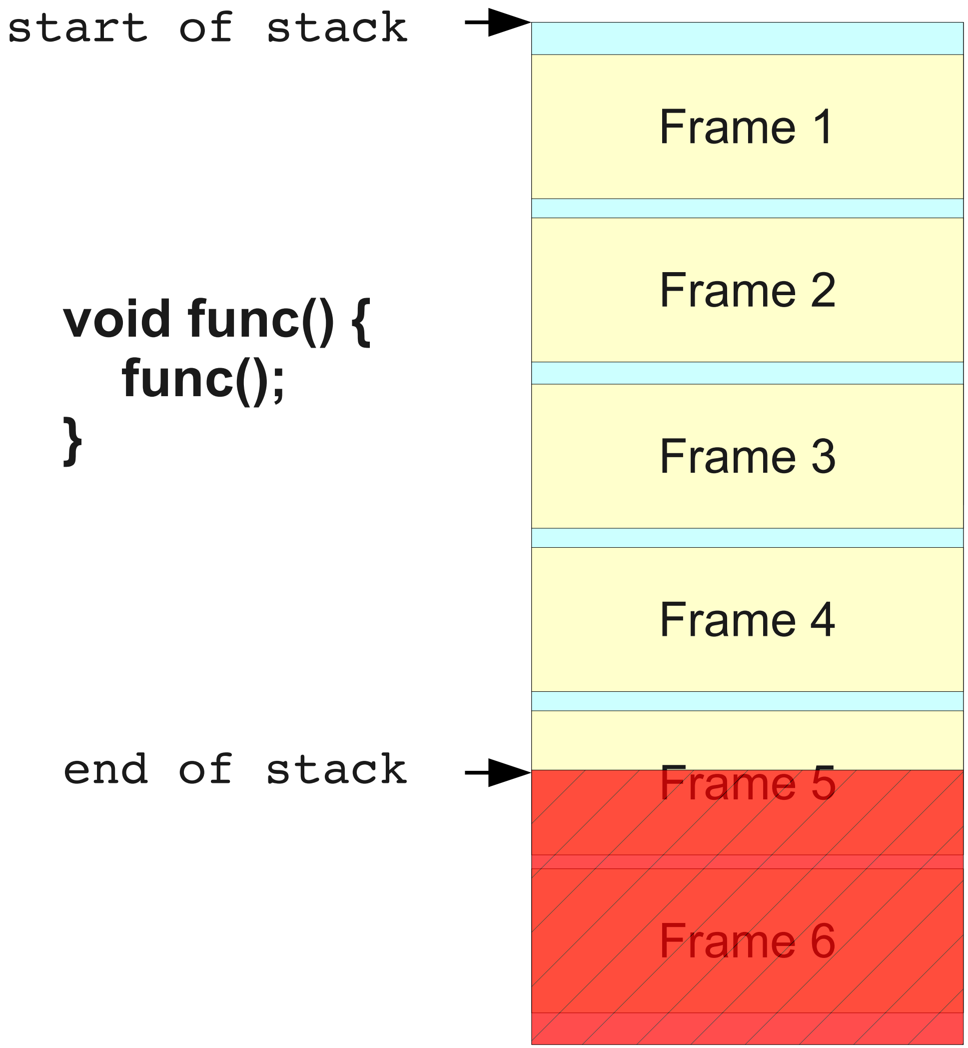
> Stack pointer decremented beyond the intended bounds of the stack's allocated VMA.

- Types of overflows
  - Incremental overflows
  - Allocation overflows

Incremental overflows

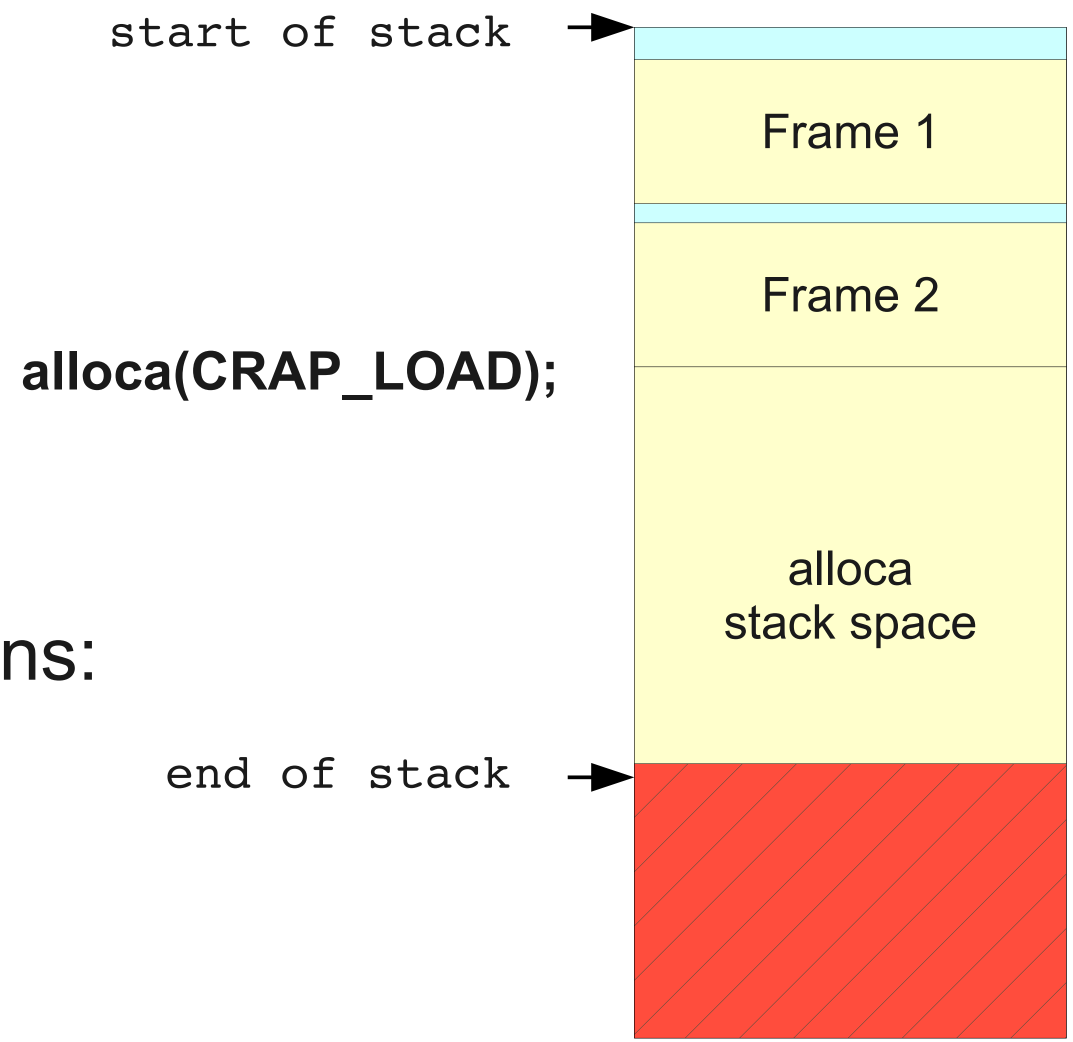
- Incremental overflows
  - Deep call chains
  - Recursion

```
void func() {
    func();
}
```



Allocation overflows

- Allocation overflows
  - Large local stack vars
  - Dynamic allocations: VLAs, alloca(3)



Exploiting stack overflows

- Stack overflows in userspace
  - Not uncommon
  - Lots of controllable (and uncontrollable) recursion
  - Some use of C99 VLAs and alloca(3)
- ***Exploitable*** stack overflows
  - Exploitable = more than DoS
  - Quite rare!

Trivia #1

What is one scenario where a userspace stack overflow might be exploitable?

Android Phone

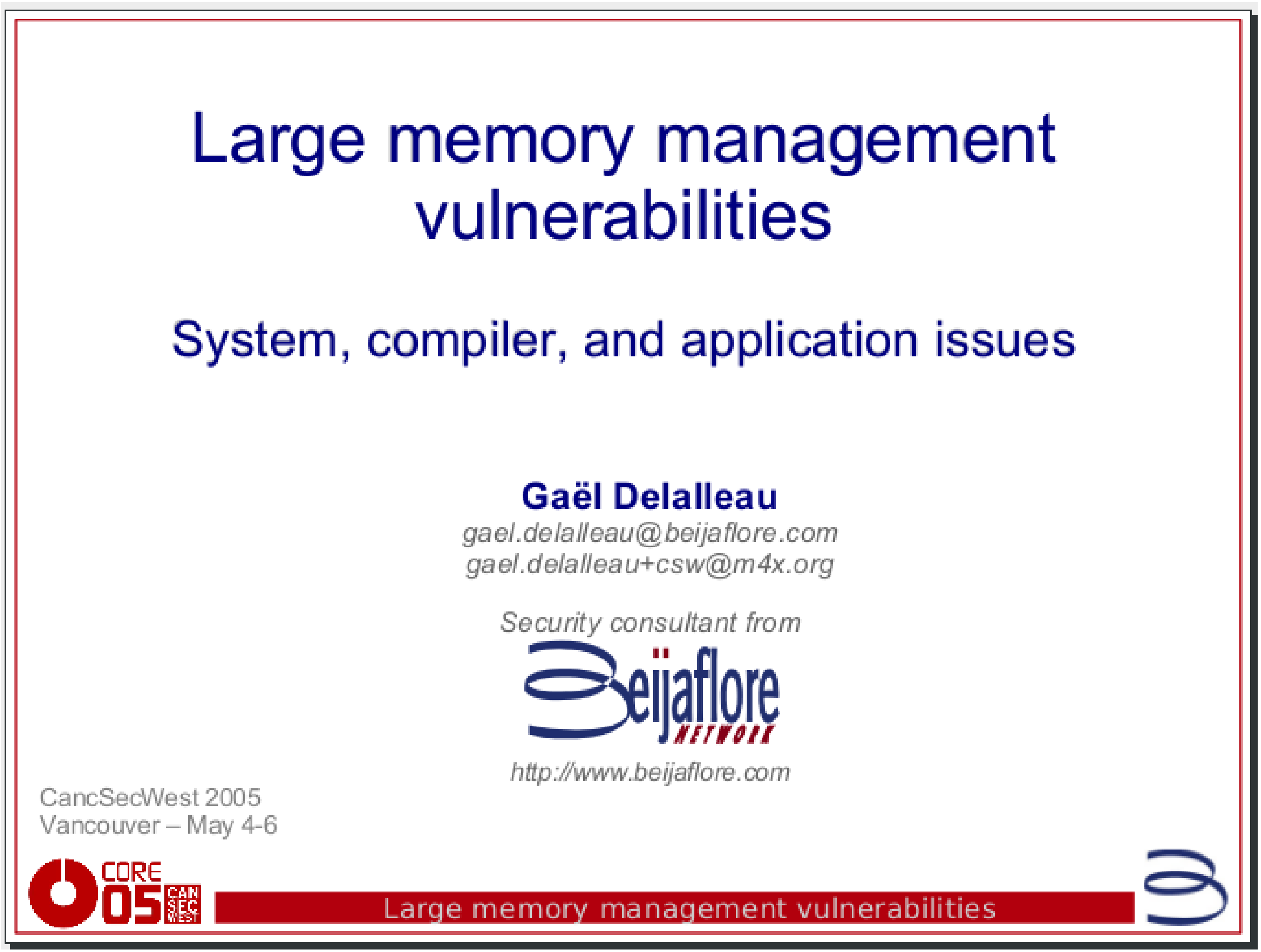
```
################################################
##::::::::::::::::::::::::::::::::::::::::::::##
##:'####::::::'#########:'##::::::::'#######::##
##'##  ##:'##: ##.....:: ##:::::::'##.... ##::##
##..::. ####:: ##::::::: ##::::::: ##:::: ##::##
##::::....::: #######::: ##:::::::: #######::##
##::::::::::: ##...:::: ##:::::::'##.... ##::##
##::::::::::: ##::::::: ##::::::: ##:::: ##::##
##:el8.n3.net: #########: #########:. #######::##
##:::::::::::.........::.........:::........::::##
################################################
```

T-Shirt

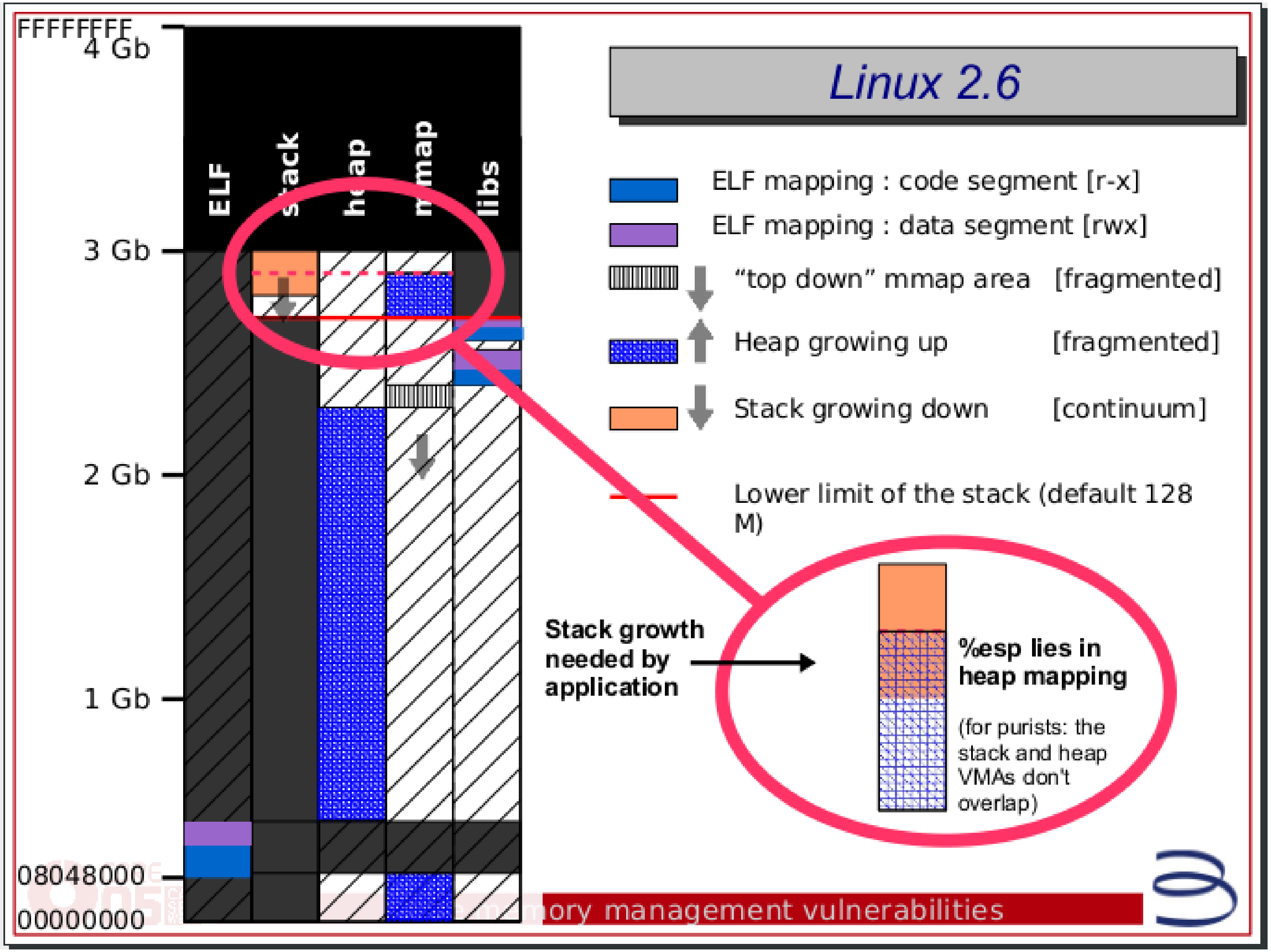
1-855-
FOR-0DAY

Phone Number

Large MM vulns



Stack overlap



Real-world stack overflows

Not a lot of real-world examples...maybe one?

- Xorg large MM vuln by Rafal @ ITL
  - No guard page at end of stack on <= Linux 2.6.36
  - Allocate large pixmaps to exhaust address space
  - Force a shm allocation adjacent to the stack
  - Call recursive function to cause stack/shm overlap
  - Write to the shm and therefore the Xorg stack

Embedded platforms

Limited memory → limited stack → stack overflows

Remote kernel overflows?

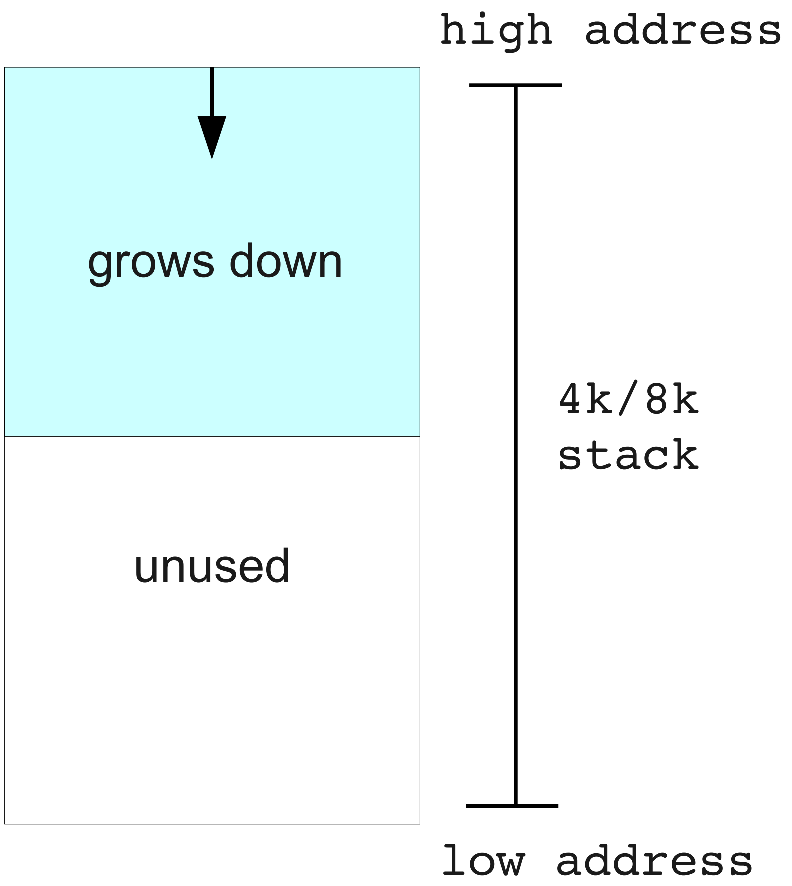
- BSD IPComp kernel stack overflow
  - Travis Normandy
  - Recursive decompression in IP stack
- Exploitable?
  - Ehhhh...

The stack is back

- A brief history of stack overflows
- **Stack overflows in the Linux kernel**
- Exploiting exotic stack overflows
- Discovering and mitigating stack overflows

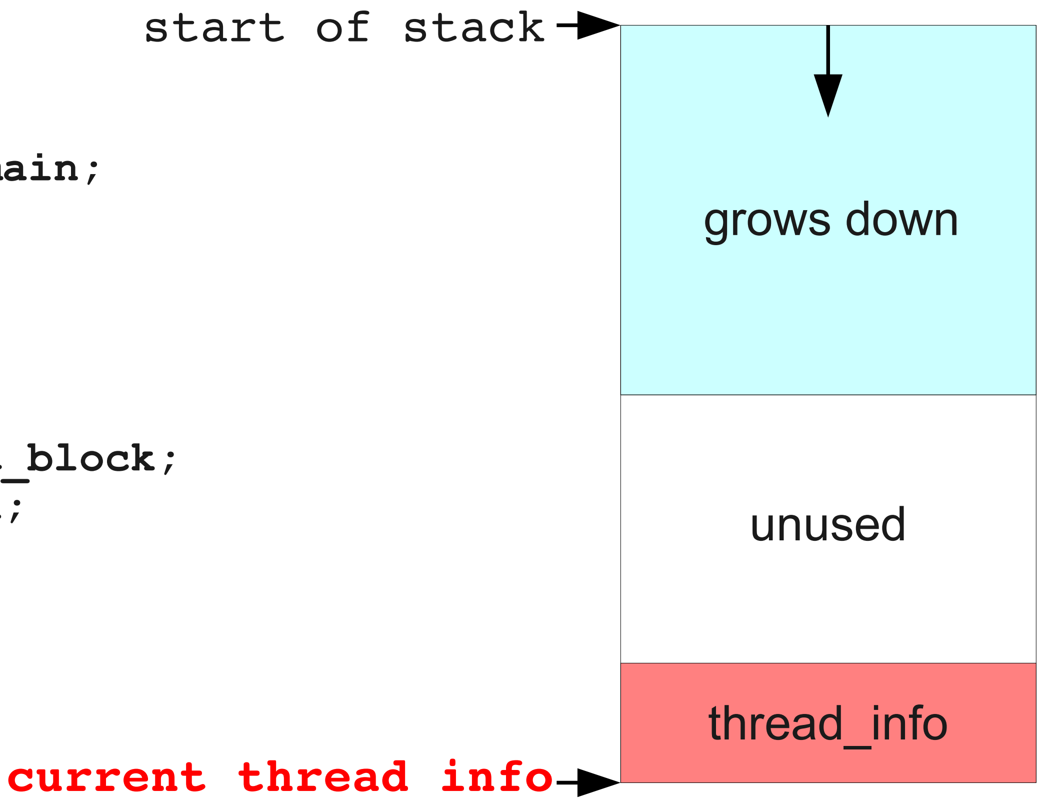
Linux kernel stacks

- 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



Metadata on kernel stack

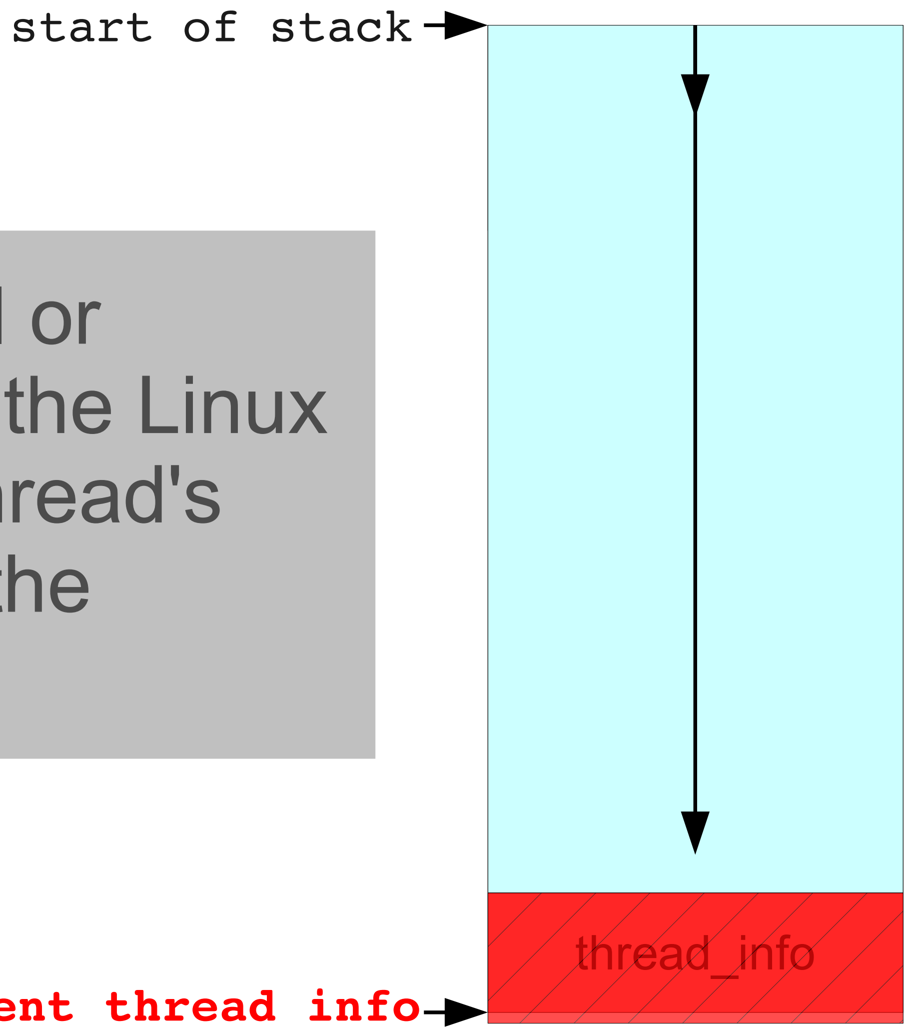
```
struct thread_info {
    struct task_struct *task;
    struct exec_domain *exec_domain;
    __u32 flags;
    __u32 status;
    __u32 cpu;
    int preempt_count;
    mm_segment_t addr_limit;
    struct restart_block restart_block;
    void __user *sysenter_return;
#ifdef CONFIG_X86_32
    unsigned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



thread_info struct is at the base of kstack!

Exploiting stack overflows

If we control an incremental or allocation stack overflow in the Linux kernel, we can cause our thread's kernel stack to collide with the thread_info structure.



Targeting thread_info

- What would the overflow collide with?
  - uaccess_err
    - No security impact, but safe to clobber
  - **restart_block**
    - A function pointer, BINGO!
  - **addr_limit**
    - Define u/k boundary, BONGO!
  - preempt_count .. task_struct
    - Pretty sensitive members, avoid clobbering

```
struct restart_block {
    long (*fn)(struct restart_block *);
    union {} /* safe to clobber */
};
```

```
access_ok()/__range_not_ok():

Test whether a block of memory
is a valid user space address.

addr + size > addr_limit.seg
```

Controlling the clobber

- Can we control the clobbering value?
  - Incremental overflow: tip of the stack, unlikely
  - Allocation overflow: VLA values, maybe
- Good news, don't need *much* control!
- Two categories:
  - Value represents a kernel space address
    - Value > TASK_SIZE
  - Value represents a user space address
    - Value < TASK_SIZE

Clobber → Code Exec

- If value < TASK_SIZE
  - Clobber restart_block fptr with userspace value
  - mmap privesc payload at that address in userspace
  - Trigger fptr via syscall(SYS_restart_syscall);
- If value > TASK_SIZE
  - Clobber addr_limit with a high kernel space value
  - You can now pass copy_from_user()/access_ok() checks up to that kernel address
  - So we can read(2) from a fd and write into kmem

Vanilla exploitation

We consider these “vanilla” stack overflows.

- thread_info clobbering technique
  - Will work in the common case for Linux kernel stack overflows
- Example vuln @ CSAW CTF
  - Controlled recursion with userspace value at tip of the stack

http://jon.oberheide.org/blog/2011/11/27/csaw-ctf-2011-kernel-exploitation-challenge/

Architecture specifics

- x86_64
  - Pretty clean, dedicated interrupt stacks
- x86_32
  - Interrupt stack shared with process stack
  - Less predictability, but more opportunity to trigger a stack overflow
- ARM, alpha, others
  - restart_block is on end of thread_info :-)

The stack is back

- A brief history of stack overflows
- Stack overflows in the Linux kernel
- **Exploiting exotic stack overflows**
- Discovering and mitigating stack overflows

Real world vulnerability

Let's look at a real-world Linux kernel stack overflow vulnerability.

- Two great bugs from Nelson
  - CVE-2010-3848
  - CVE-2010-3850
  - And a bonus bug that will come into play later
- Econet packet family
  - Stack overflow in econet_sendmsg()

Vulnerable code

```
int econet_sendmsg(struct kiocb *iocb, struct socket
*sock, struct msghdr *msg, size_t len)
{
...
    struct iovec iov[msg->msg_iovlen+1];
```

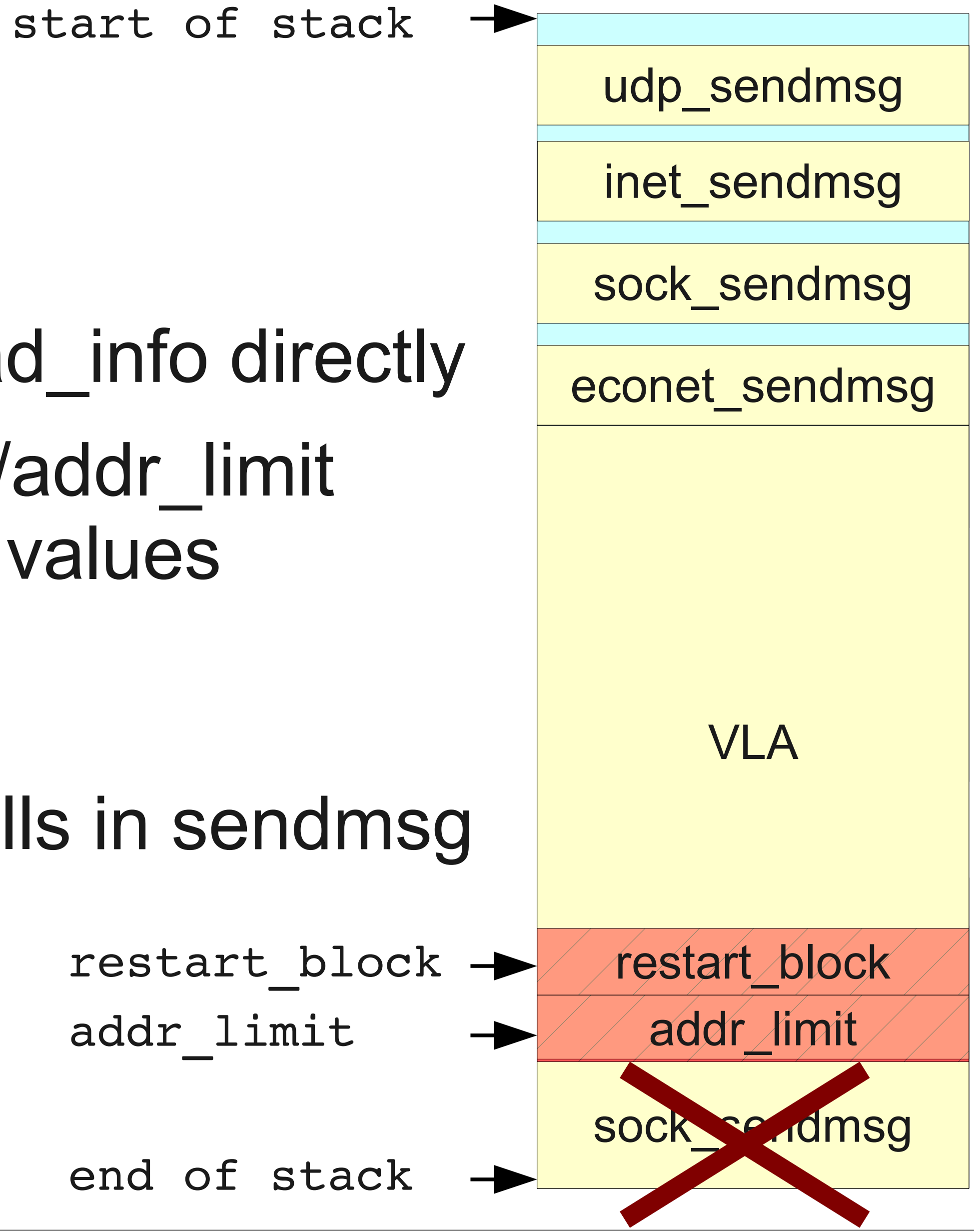
Oh snap! A VLA on the stack with an attacker controlled length!

```
        for (i = 0; i < msg->msg_iovlen; i++) {
        ...
            iov[i+1].iov_base = base;
            iov[i+1].iov_len = iov_len;
        ...
        }
```

Hey, we (mostly) control the contents too! Game over, eh?

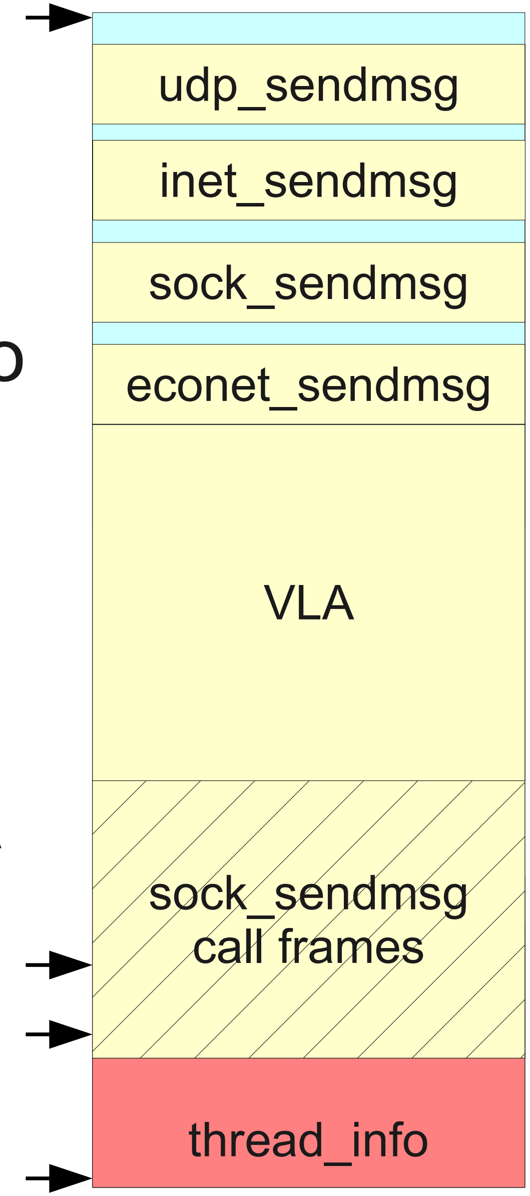
Attempt #1

- Attempt #1
  - Expand VLA to hit thread_info directly
  - Overwrite restart_block/addr_limit with attacker controlled values
- Thwarted!
  - Subsequent function calls in sendmsg will clobber sensitive thread_info members



Attempt #2

- Attempt #2
  - Expand VLA to just above thread_info
  - Overwrite using the stack frames of subsequent calls (sock_sendmsg)
- Semi-thwarted!
  - Overwrite value is uncontrolled and a kernel space value so restart_block is no good
  - What about addr_limit?



Attempt #2 continued

- We can hit addr_limit with a value that represents a high kernel space value
  - Overwrite of addr_limit occurs in sock_sendmsg call

```
oldfs = get_fs();
set_fs(KERNEL_DS);
err = sock_sendmsg(udpsock, &udpmsg, size);
set_fs(oldfs);
```

- You can't be serious...
  - addr_limit is being saved/restored before/after the sock_sendmsg call, nullifying our overwrite

Attempt #2 continued

- We could try other subsequent function calls besides sock_sendmsg
  - Cause error condition, return from econet_sendmsg early with a terminating mutex_unlock call. Eg:

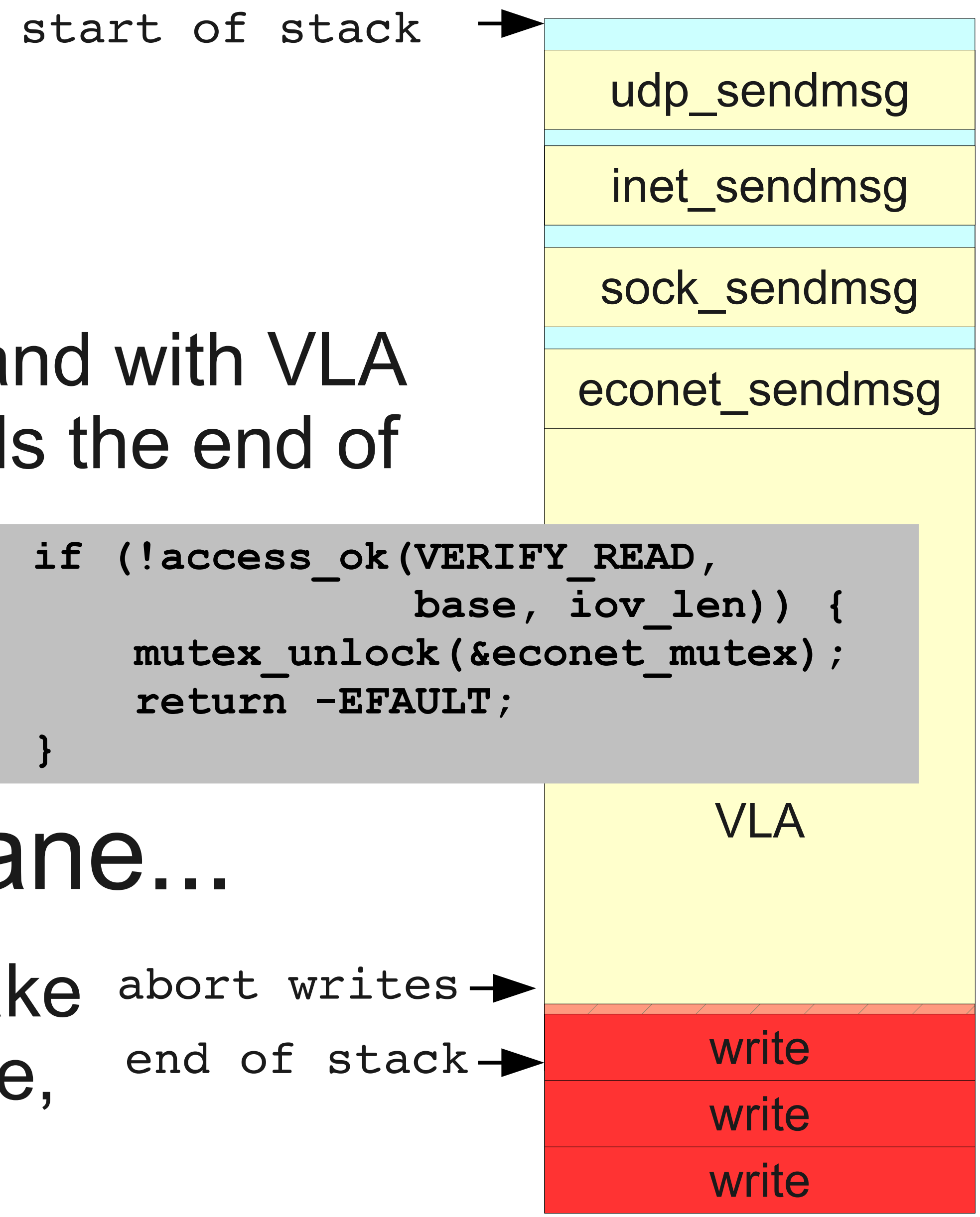
```
if (len + 15 > dev->mtu) {
    mutex_unlock(&econet_mutex);
    return -EMSGSIZE;
}
```

- Write offsets of the stack frame don't align
  - Pattern: chunks of two 8-byte writes w/kernel value
  - Hit restart_block with kernel value (useless) or hit both addr_limit (good) and preempt_count (crash)

Attempt #3

- Attempt #3
  - Blow past thread_info and with VLA and “write-back” towards the end of the kernel stack
  - Overwrite task_struct controlled address

```
if (!access_ok(VERIFY_READ,
                base, iov_len)) {
    mutex_unlock(&econet_mutex);
    return -EFAULT;
}
```



- Ok, this is just insane...
  - Yes, you can make a fake task_struct in userspace, but not in this century

Need a different approach

It's clear the thread_info technique is not going to work here due to extenuating circumstances

- If thread_info is out, what can we do?
- Nothing useful *on* the stack, but...
- Need some audience help here...

Trivia #2

Any ideas of what to do if the thread_info technique isn't going to work?

Android Phone

```
#################################################
##:::::::::::::::::::::::::::::::::::::::::::::##
##:'####:::::'#########:'##:::::::'########::##
##'##  ##:'##: ##.....:: ##:::::::'##.... ##::##
##..::. ####:: ##::::::: ##::::::: ##:::: ##::##
##:::::....::: #######::: ##::::::: ########::##
##:::::::::::: ##...:::: ##:::::::'##.... ##::##
##:::::::::::: ##::::::: ##::::::: ##:::: ##::##
##:el8.n3.net: #########: #########:. ########::##
##:::::::::::.........::.........::.........::::##
#################################################
```

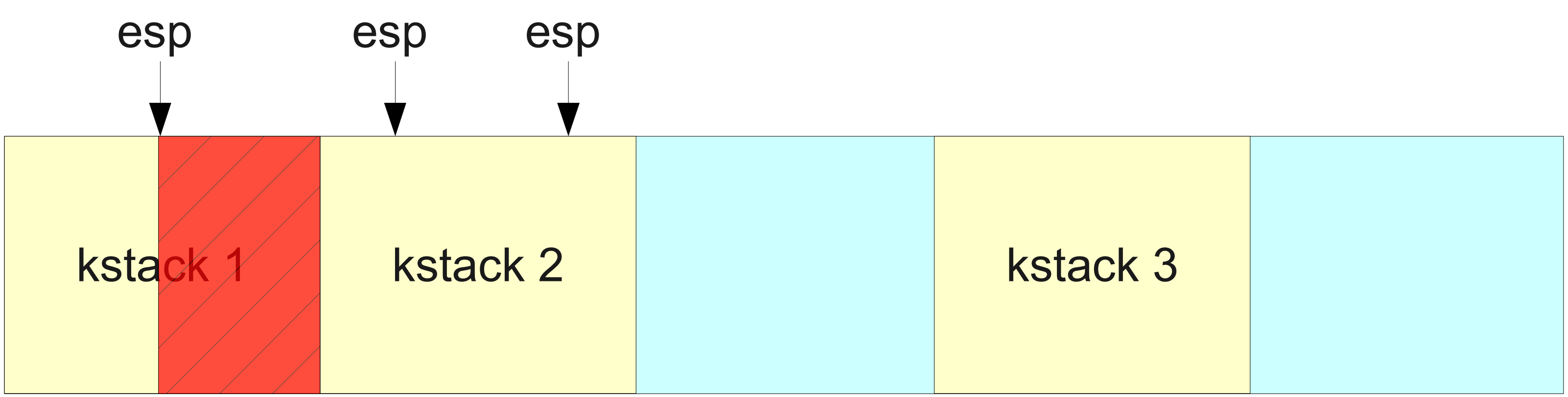
T-Shirt

1-855-FOR-0DAY

Phone Number

Beyond our stack

- A thread's kstack doesn't exist in a vacuum
- Each kstack allocated from the buddy allocator



- Screw *intra-stack* exploitation, let's talk *inter-stack* exploitation

Attacking adjacent kstacks

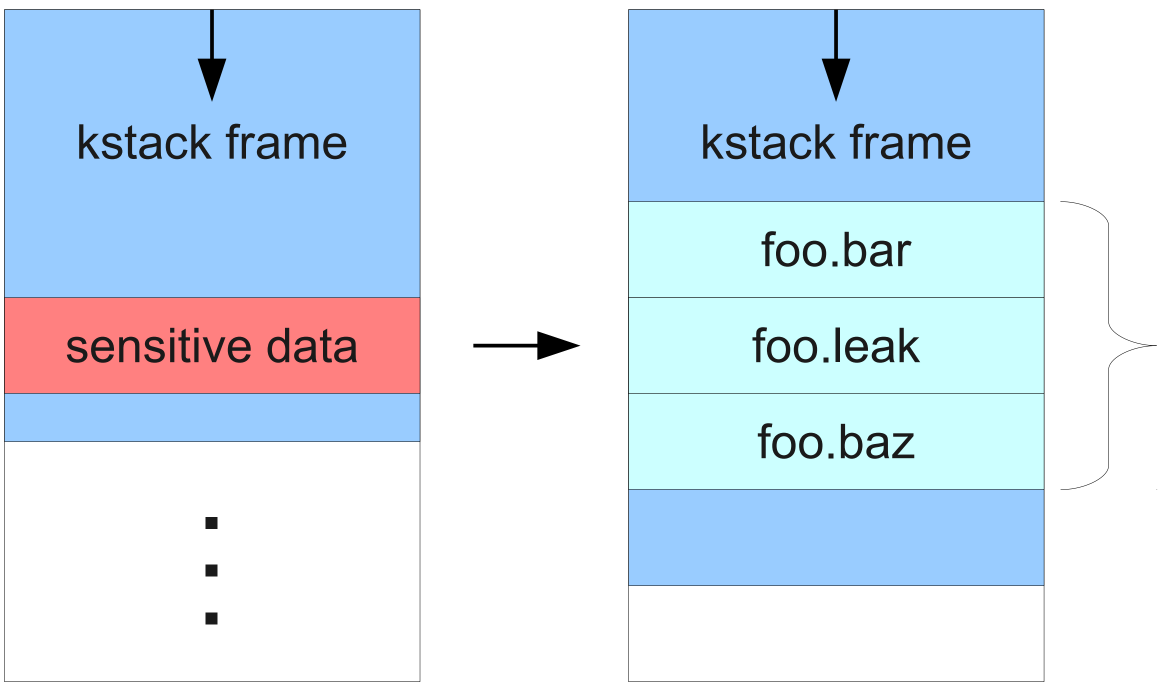
In an allocation-based overflow, we can blow past the end of our stack and into an adjacent stack!

- Two big questions:
  - How do we get two thread kernel stacks allocated adjacently?
  - How do we sanely modify another thread's stack to gain code exec?

We sort of did this with stackjacking self-discovery!

We sort of did this with stackjacking Obergrope!

Kernel stack 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

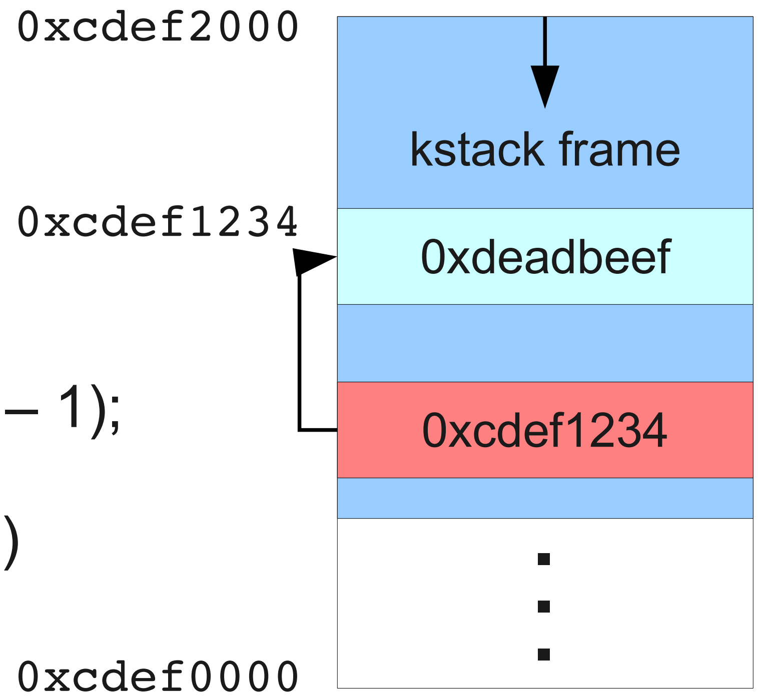
Kernel stack self-discovery

- 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)

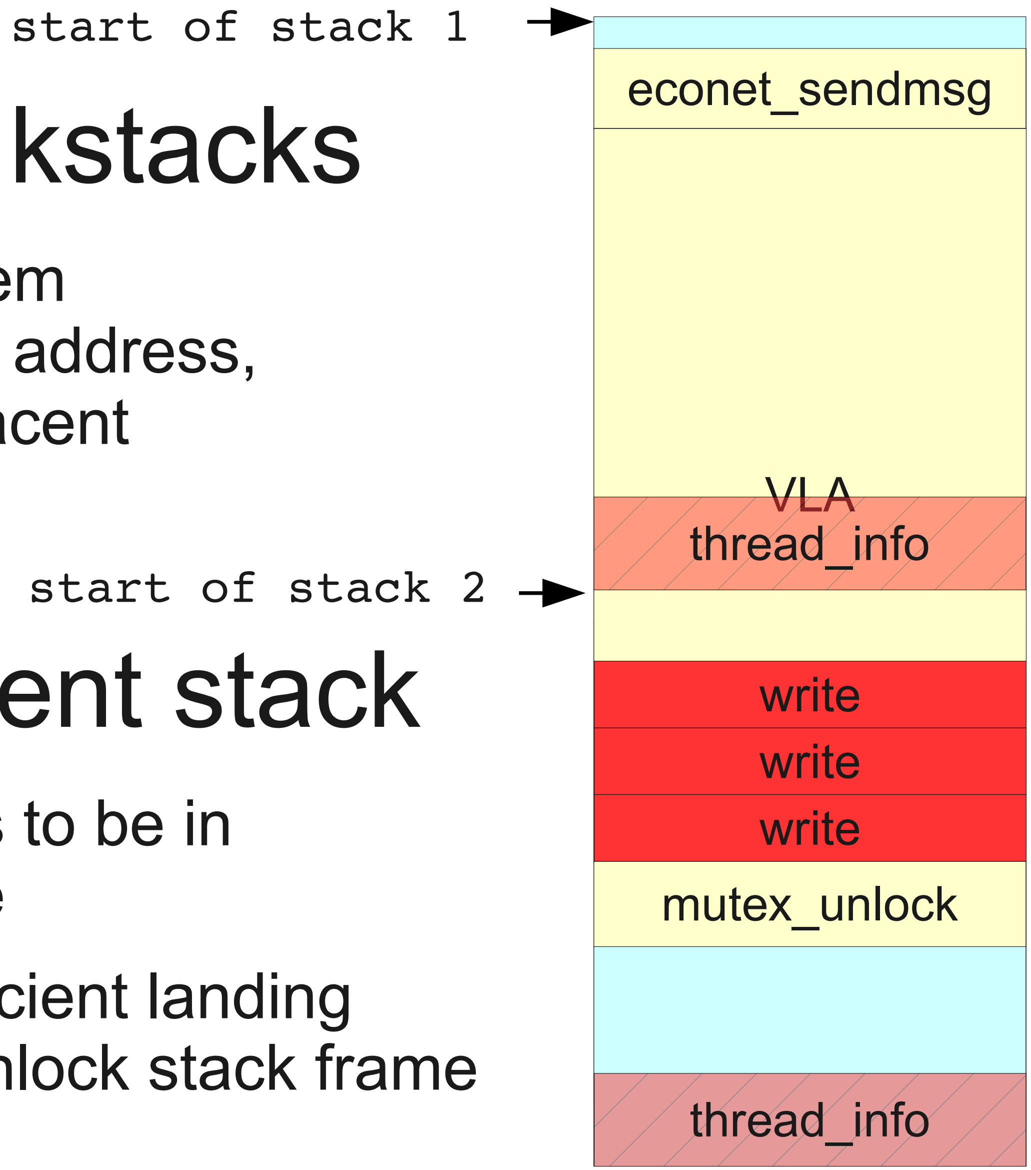
kstack_base = 0xcdef0000



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

Writing the adjacent kstack

- Getting adjacent kstacks
  - Spawn children, have them self-discover their kstack address, spin until we get two adjacent allocations
- Writing the adjacent stack
  - Process #2 kstack needs to be in a stable predictable state
  - Process #1 needs a sufficient landing zone to absorb mutex_unlock stack frame

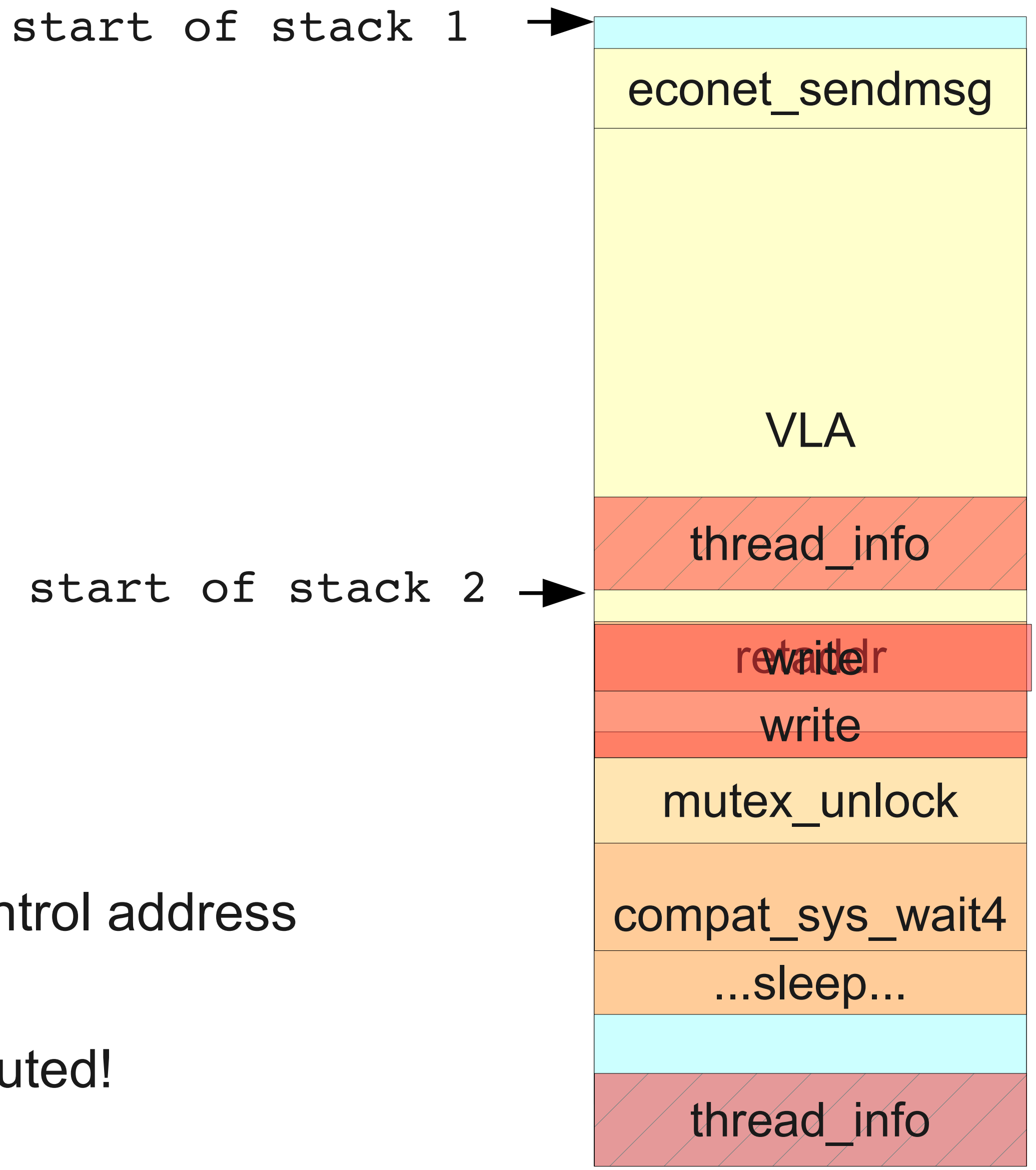


Sleepy syscalls are back

- Process #2 will enter a “sleepy syscall”
  - Arbitrary sleeping to avoid dangerous race conditions with the overflow write
  - While asleep, process #1 will overwrite a return address on process #2's kstack
- compat_sys_wait4 looks good
  - Hey, same function we used for stackjacking!
  - Large unused local stack vars to absorb the mutex_unlock stack frame

Final exploit flow

- Achieve adjacent kstacks
- Process #2 goes to sleep
- Stack overflow in process #1
- Overwrite return address on process #2 kernel stack
- Process #2 wakes up
- Process #2 returns to attacker control address
- Privilege escalation payload executed!



Demo

DEMO TIME?

http://jon.oberheide.org/files/half-nelson.c

The stack is back

- A brief history of stack overflows
- Stack overflows in the Linux kernel
- Exploiting exotic stack overflows
- **Discovering and mitigating stack overflows**

What is one way to discover potential stack overflow vulnerabilities?

Android Phone

```
################################################
##::::::::::::::::::::::::::::::::::::::::::::##
##:'####::::::'########:'##:::::::'#######:::##
##'##  ##:'##: ##.....:: ##:::::::'##.... ##::##
##..::. ####:: ##::::::: ##::::::: ##:::: ##::##
##::::.....::: #######::: ##:::::::: #######:::##
##:::::::::::: ##...:::: ##:::::::'##.... ##::##
##:::::::::::: ##::::::: ##::::::: ##:::: ##::##
##:el8.n3.net: ########: ########:. #######:::##
##::::::::::::.........::.........::........::::##
################################################
```

T-Shirt

1-855-FOR-0DAY

Phone Number

jono discovery method

Ghetto kstack overflow discovery mechanism:

Advanced l33t static analysis:

```
egrep -R "^[[:space:]]*(struct |char |(u)?int(8_t|16_t|32_t|64_t)? |void )[^=]+\[[a-z]+.*[\+\*]?.*\];" * | grep -v sizeof
```

Projected to win grugq's #grep2pwn 2012.

pipacs discovery method

The proper way to do it: gcc plugin

Artist's depiction of “theowl”

13:27 < pipacs> jono btw, i'm sorry to burst your infiltrate bubble but the next stackleak plugin will fix the alloca problems...

13:28 < pipacs> (and if you want to find all those bugs, the same plugin can tell you exactly where they occur ;)

pax_check_alloca verifies kstack sanity after alloca calls.

Inserted at compile time by stackleak_check_alloca into any functions that use __builtin_alloca.

See tools/gcc/stackleak_plugin.c in latest PaX patch

Exploiting hardened kernels

- On grsec/PaX kernels, thread_info is no longer stored at the base of the kernel stack
  - Mitigated the Rosengrope stackjacking method
  - So, the standard thread_info overwrite is ineffective

Can we use the adjacent process exploitation technique against hardened kernels?

- Yes...
  - But RANDKSTACK makes it hard and new STACKLEAK plugin makes it near infeasible

Mitigating exploitation

- Move thread_info off the stack!
  - Thwarts vanilla thread_info exploitation technique
  - Patches years ago to LKML, rejected by mainline
- Thwarting the adjacent process technique is a bit harder
  - Something like PaX's RANDKSTACK would make things harder

Wrap-up

- GIVE UP HEAPSTERS!
  - Win8 fixed everything, the heap is over
- Stack overflows *are* exploitable
  - At least in the Linux kernel
  - How about your favorite OS? Windows/BSD/etc?
- Don't shun “unexploitable” vuln classes
  - Other situations? Userspace via browser/js?

Greetz

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

Q&A

QUESTIONS?

Jon Oberheide
jon@oberheide.org
Duo Security