From “No Way” to 0-day: Weaponizing the Unweaponizable

“...you’re de0uing it wrong...”

Joshua Wise
Outline

• Intro
• Vulnerabilities: in general
  - What makes something easy to exploit?
• Vulnerabilities: a case study
  - Making something hard into something doable
• Briefly -- what went wrong?
  - How did this sort of thing happen?
• Q & A
Intro: Me

• Just some guy, you know?
• All-purpose embedded hacker
  - Got roped into Android at some point
• Recovering software guy
  - Now doing ASIC design
• Buzzword compliant
  - Working on IMB in ECE at CMU
Intro: You

• At least a little bit of kernel experience?
• Interested in security?
• Not a skript kiddie
  - No code for you to compile here
  - Enough description for a skilled programmer to repro this
Today’s vulnerability

- While looking for ways to root Android phones, came across...
  - CVE-2010-1084
- “CVE request: kernel: bluetooth: potential bad memory access with sysfs files”
  - “...allows attackers to cause a denial of service (memory corruption)”
- First showed up in 2.6.18, fixed in 2.6.33
  - ...ouch!
  - Raise your hand if you haven’t patched up to 2.6.33 yet
Mechanism of crash

• Classic vulnerability
  - for each Bluetooth socket, `sprintf()` onto the end of a string in a buffer
  - no check for end of buffer

• With a twist
  - gets the buffer from the frame allocator; scribbles into next frame (uncontrolled target)
  - contents not controlled
  - length only kind of controlled
Yesterday’s vulnerability

- Refresher: easy vulnerability
- Simple stack smash:

```c
void cs101_greeter() {
    // prof said it has 2 be setuid root 4 term axx
    char buf[1024];
    printf("What is your name?\n");
    gets(buf); // my prof said not to use gets(3)
    printf("Hello, %s!\n", buf); // so i used gets(buf), thats ok rite?
}
```

- Easily exploitable properties
  - Controlled target
  - Controlled length
  - Controlled contents (with a few limitations)
Watch that stack!

• What happens next?
  - “User” inputs something bad.

• Where does it go?
Watch that stack!

- What happens next?
  - "User" inputs something bad.
- Where does it go?

These addresses are for no machine in particular!
Watch that stack!

- What happens next?
  - “User” inputs something bad.
- Where does it go?

```
$ /afs/cs/course/15123-sfnoob/usr/aashat/bin/greeter
What is your name?
AAAAAAAAAAAA...∆/]θéåµf...
Hello, AAAAAAAA...
[+] pwned
```

These addresses are for no machine in particular!
Watch that stack!

- What happens next?
  - “User” inputs something bad.
- Where does it go?

$ /afs/cs/course/15123-sfnoob/usr/aashat/bin/greeter
What is your name?
AAAAAAAAAAAAA...∆∫∆∂ャュµ...
Hello, AAAAAAAAA...
[+] pwned
#
Why did that work so well?

• Remember the three *controls*:
  
  - Attacker-controlled *target*
    
    • Always blast the ret addr - same memory each time
  
  - Attacker-controlled *length*
    
    • We never blast off the end of the stack into segfaultland
  
  - Attacker-controlled *contents*
    
    • Write anything we want but 0x00 and ' \n'
From yesterday comes tomorrow

• Today’s exploit, at its core:
  - (for those of you following along at home, in l2cap_sysfs_show)
  - str = get_zeroed_page(GFP_KERNEL);
    ...
    for each l2cap_sk_list as sk:
      str += sprintf(str, "%s %s %d %d 0x%4.4x 0x%4.4x %d %d %d\n"
                      batosstr(&bt_sk(sk)->src),...,);

• What year is it? I seem to have forgotten
sprintf() out of control

• Issue is obvious, and crash is inevitable -- but what of our three controls?

• Controlled target
  - How is buf allocated?
    • sysfs buffer comes from frame allocator
  - What comes after?
    • Some other poor noob’s frame!
Frames are physical memory backings of pages.
- Don’t confuse with ‘stack frames’!

Pages are chunks of virtual memory.

Linux kernel has both mapped into A.S.!
- Needed for frame allocations (__GFP_KERNEL) -- more later
sprintf() out of control

• Issue is obvious, and crash is inevitable -- but what of our three controls?

• Controlled length
  - Writes take place through a sprintf() to a strange place
  - We can’t stop it before it smashes something else
sprintf() out of control

- Issue is obvious, and crash is inevitable -- but what of our three controls?
- Controlled contents
  - No data comes directly from us
  - All data comes formatted
sprintf() out of control

• Issue is obvious, and crash is inevitable -- but what of our three controls?

• **Zero for three!**

• Now would be a good time to start controlling our environment.
Target practice

• How can we control the target?
• Let’s use an old-fashioned heap spray.
  - ...but what?
  - First idea: kstack!
  • It worked so well in CS101, right?

“With Emarhavil, your target is our target.”
• Let’s assume:
  - kernel stack is the frame after the sysfs page
  - we know which pid the kstack belongs to
• Given that, what happens?
  - What does a kstack even look like?
Jenga

- Like other stacks, a kstack has stack frames.
- Unlike other stacks, a kstack has a TCB attached to it!
Jenga

• What happens when we write?

sprintf(str, "ownedownedownedowned"
        "ownedownedownedowned"
        ...");
• What happens when we write?

```c
sprintf(str, "ownedownedownedowned
        ownedownedownedowned
        ...");
```
Jenga

• What happens when we write?
  - TCB is clobbered!
  - Could be OK; this time not.

```c
sprintf(buf, "ownedownedownedownedownedownedownedownedownedownedownedownedownedowner"
...);
```
Getting physical

• What else goes in physical frames?
• Linux kernel has interesting mechanism called SLAB allocator
  - Creates uniform “caches” of specific objects
    • conveniently, frame-sized!
  - Localizes similar objects in memory
  - Avoids expensive variable-size allocation
  - Originally designed by the Sun guys
SLABs of memory

- **What’s in a SLAB?**

- **Where’s the list of SLABs available?**
  - SLAB metadata stored in... a SLAB!
SLABs of memory

• What’s in a SLAB?

• No per-SLAB header
  - Convenient...
SLABs of memory

- What’s in a SLAB?
  - No per-SLAB header
    - Convenient...
Who eats SLABs?

- Pretty much every kernel subsystem
  - `joshua@escape:~/linux$ find . | xargs grep kmem_cache_alloc | wc -l`
    - 305
  - `joshua@nyus:/proc$ cat slabinfo | wc -l`
    - 183

- Something in there has to be an easy target

- How about... file descriptors?
  - Stored in `struct file`, in SLABs
• What does a struct file look like?

```c
struct file {
  union {...} f_u; /* morally, two pointers */
  struct path f_path; /* morally, two pointers */
  struct file_operations *f_op;
  unsigned int f_count, f_flags, f_mode;
  ...
}
struct file_operations {
  struct module *owner;
  loff_t (*llseek)(...);
  ssize_t (*read)(...);
  ssize_t (*write)(...);
  ssize_t (*aio_read)(...);
```

Filed away for reference
Filed away for reference

- What does a struct file look like?
  - (best case!)

<table>
<thead>
<tr>
<th>f_u</th>
<th>f_u</th>
<th>f_path</th>
<th>f_path</th>
<th>f_op</th>
<th>f_count</th>
<th>f_flags</th>
<th>f_mode</th>
<th>...</th>
</tr>
</thead>
</table>

frame boundary

remember, this is at the start of a SLAB!

each block is one pointer size
What does a struct file look like?
- (really really best case!)

<table>
<thead>
<tr>
<th>str</th>
<th>f_u</th>
<th>f_u</th>
<th>f_path</th>
<th>f_path</th>
<th>f_op</th>
<th>f_count</th>
<th>f_flags</th>
<th>f_mode</th>
<th>...</th>
</tr>
</thead>
</table>

*frame boundary*

*get_zeroed_page comes from same pool as SLABs (more later)*

*remember, this is at the start of a SLAB!*

*Filed away for reference*
Filed away for reference

- What does a struct file look like?
  - Parts that the kernel can survive for a little while without darkened

```
str
(fhum de dum)
```

```
frame boundary
```

get_zeroed_page comes from same
pool as SLABs
(more later)

remember, this is
at the start of a
SLAB!
Great news!
Great news!

- In essence -- **struct file** can be paved over at will
  - ...just as long as we get a reasonable value into `f_op`.
One for three

• Remember the three controls:
  - Attacker-controlled *length*
  - Attacker-controlled *contents*
  - Attacker-controlled *target*

• Length is *no longer an issue*
  - We can go over by a little ways without causing an immediate crash
• It is *difficult* to write arbitrary content...
  
  - ...but easy to predict content.

  ```c
  str += sprintf(str, "\%s \%s \%d \%d \x%4.4x \x%4.4x \%d \%d \%d\n"
    batostro(&bt_sk(sk)->src),...);
  ```

• Usually looks like:

  ```
  "00:00:00:00:00:00 00:00:00:00:00:00 2 0 \x0000
  \x0000 672 0 1" repeated a bunch
  ```

  - well, as many times as we want...

• What does this mean for us?
• Data that looks like this *must* end up in the file structure.
  - "00:00:00:00:00:00 00:00:00:00:00:00 2 0 0x0000 0x0000 672 0 1"
  - Substring *must* end up in `f_op`!

• What, exactly, *can* go in `f_op`?
  - more importantly, *can this* go in `f_op`?
Addressability

• \( f_{op} \) is just a pointer into kernel’s A.S.!
  - Remember: kernel’s A.S. is superset of user’s A.S.
  - \( f_{op} \) can be pointer to user memory

• Game plan
  - Map all substrings
  - ASCII representations should be valid pointers to \( f_{op} \) target.
    • “00:0” \( \rightarrow \) 0x30303A30
    • “0:00” \( \rightarrow \) 0x303A3030
    • “0 0:” \( \rightarrow \) 0x3020303A
    • ...
Now what?

• We’re done, right?
Now what?

• Not so fast.
  - Real life, more likely:
Two for three

• Remember the three controls:
  - Attacker-controlled \textit{length}
  - Attacker-controlled \textit{contents}
  - Attacker-controlled \textit{target}

• Contents not controlled... but predicted.
  - We now have \textit{length} and \textit{contents} handled.
Let’s be buddies

• How do we *control* the relative placement of frames?
  - (i.e., the *target*)

• Physical frames allocated on Linux using “buddy allocator”
  - *Really* old best-fit allocator -- Markowitz, 1963
  - Works *really* well with fragmentation-reducing strategies like SLAB
  - `linux/mm/page_alloc.c`

• *Run in god-damn fear.*
Let’s be buddies

• Buddy allocator has important features
  - Injects determinism and predictability into otherwise unordered frame allocation
  - Localizes size-one frames when able

• Implementation details beyond scope of this talk
  - You gotta pick one, and I think SLAB is cooler
Localizer approach

• Plan:
  - Fill up memory
    • Cause frames that would result in discontinuities to be paged to disk
  - Free memory to generate contiguous chunks
  - Allocate chunks of memory for struct files
  - Allocate buffer page
    • Opening sysfs file does this. This is critical!
  - Allocate more chunks of memory for struct files
  - Fire!
Localizer approach

Initial configuration

<table>
<thead>
<tr>
<th>free</th>
<th>free</th>
<th>free</th>
<th>in use</th>
<th>free</th>
<th>in use</th>
<th>free</th>
<th>free</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>free</td>
<td>in use</td>
<td>free</td>
<td>free</td>
<td>free</td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>free</td>
<td>in use</td>
<td>free</td>
<td>free</td>
<td>free</td>
<td>in use</td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>free</td>
<td>in use</td>
<td>free</td>
<td>free</td>
<td>free</td>
<td>in use</td>
<td>free</td>
<td>free</td>
</tr>
<tr>
<td>free</td>
<td>free</td>
<td>in use</td>
<td>free</td>
<td>free</td>
<td>in use</td>
<td>free</td>
<td>free</td>
</tr>
</tbody>
</table>
Localizer approach

Allocate all memory for us

<table>
<thead>
<tr>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
</tbody>
</table>
Localizer approach

Free and allocate to get contiguous phys chunks

<table>
<thead>
<tr>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
</tbody>
</table>
Localizer approach

Release contiguous phys frames

<table>
<thead>
<tr>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
<th>ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>free</td>
<td>free</td>
<td>free</td>
<td>free</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
</tbody>
</table>
Localizer approach

Set up files, buffer, files

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>files</td>
<td>files</td>
<td>str</td>
<td>files</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
</tbody>
</table>
## Localizer approach

### Pwn

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td><strong>files</strong></td>
<td><strong>files</strong></td>
<td><strong>str</strong></td>
<td><strong>files</strong></td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
<tr>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
<td>ours</td>
</tr>
</tbody>
</table>
Three for three!

- Remember the three controls:
  - Attacker-controlled length
  - Attacker-controlled contents
  - Attacker-controlled target

- Target became controlled by deterministic memory permutation.

- Result: system owned.
So close, guys

/*
 * The code works fine with PAGE_SIZE return but it's likely to
 * indicate truncated result or overflow in normal use cases.
 */
if (count >= (ssize_t)PAGE_SIZE) {
    print_symbol("fill_read_buffer: \%s returned bad count\n",
             (unsigned long)ops->show);
    /* Try to struggle along */
    count = PAGE_SIZE - 1;
}
Demo
Conclusions

- Difficult-to-exploit bugs can be made easier by thinking about controlling your environment
  - Attacker-controlled length
  - Attacker-controlled contents
  - Attacker-controlled target

- Just because it’s not easy, that doesn’t mean that it’s impossible!
Conclusions

• Difficult-to-exploit bugs can be made easier by thinking about controlling your environment
  - Attacker-controlled \textit{length}
  - Attacker-controlled \textit{contents}
  - Attacker-controlled \textit{target}

• \textit{Just because it’s not easy, that doesn’t mean that it’s impossible!}

• Side conclusion:
  - Phone vendors: we will win. We have physical access; root on these phones will be ours. Please stop your crusade to keep me from using my own phone.
Questions?