Does Making The Kernel Harder Make Making The Kernel Harder?

Casey Schaufler
Intel Open Source Technology Center
Casey Schaufler

Kernel developer from the 1970’s

Supercomputers in the 1990’s

Smack Linux Security Module

Security module stacking

Photo Curtesy Ann Forrister
tl;dr

Yes
Why Don’t We Think The Kernel Is “Hard”?
It’s too easy to cause damage

- Buffer overflow
- Index underflow
- Stack stomping
People who want to do damage are too clever

- Buffer overflow attacks
- Invalid parameters
- Return oriented programming

But that’s not new, is it?
Old as the C compiler

+ The C language simplifies

+ Memory organization

+ Control flow

+ C is not strongly typed
Efficient and convenient

```
+struct ip_msfilter {
  +  . . .
  +  __u32  imsf_numssrc;
  +  __be32  imsf_slist[1];
+};

+u = ipm->imsf_slists[index];
```
Clever and precise

```c
union tcp_word_header {
    struct tcphdr hdr;
    __be32 words[5];
};

twh->words[3] = 0x8675
```
Why would I want to give that up?
You probably don’t

+ Strongly typed languages have their own issues
+ Object oriented programming adds overhead
+ The code base is really big

“Strong typing is for weak minds”
- Tom Van Vleck?
- James Gosling?
There are things we can do

- Use the typing that is available
- Fix what we know to be dangerous
- Prepare for failures
Typing? How does that help?
refcount_t

- Allocated object reference counts
- Should never be 0
- Detect use of freed object
What do we know is dangerous?
String functions

+ `strcpy(dest, src);`

+ `strncpy(dest, src, strlen(src));`
Automatic arrays

```c
int func(struct comp *p, int count)
{
    struct comp controls[count];
```
Casts

```c
struct cred *cred = (struct cred *cred) & i;

temp = (unsigned short)((int)(temp) + shift);
```
It’s not that they can’t be used safely

+ Checking may be expensive

+ Try to find all the callers
Stacks
Convenient for function parameters

- Push on call
- Pop on return
- Hardware accelerated
Convenient for mucking up

<table>
<thead>
<tr>
<th>previous function</th>
<th>arguments and stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>arguments and stuff</td>
</tr>
<tr>
<td>function that was called</td>
<td>arguments and stuff</td>
</tr>
</tbody>
</table>
Harder to get the wrong stack data

<table>
<thead>
<tr>
<th>previous function arguments and stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>gap</td>
</tr>
<tr>
<td>function arguments and stuff</td>
</tr>
<tr>
<td>gap</td>
</tr>
<tr>
<td>function that was called arguments and stuff</td>
</tr>
</tbody>
</table>
Erase what’s no longer needed

<table>
<thead>
<tr>
<th>previous function</th>
<th>arguments and stuff</th>
</tr>
</thead>
<tbody>
<tr>
<td>gap</td>
<td></td>
</tr>
<tr>
<td>function</td>
<td>arguments and stuff</td>
</tr>
<tr>
<td>gap</td>
<td></td>
</tr>
<tr>
<td>function that was called</td>
<td>arguments and stuff</td>
</tr>
</tbody>
</table>
A random thought
Attackers and developers hate randomization

+ For the same reasons

+ Real addresses are needed

+ Log are less useful

+ Debuggers get buggered
Structures

```
struct agamemnon {
    struct list_head *list;
    struct cred *cred;
    u64 flags;
    u32 banners;
    u32 bunting;
};

__randomize_layout

struct agamemnon {
    u32 banners;
    struct list_head *list;
    u32 bunting;
    struct cred *cred;
    u64 flags;
};

__no_randomize_layout
```
Stack pages are just pages

<table>
<thead>
<tr>
<th>function that was called arguments and stuff</th>
<th>previous function arguments and stuff</th>
<th>gap</th>
</tr>
</thead>
<tbody>
<tr>
<td>gap</td>
<td>gap</td>
<td>other stuff</td>
</tr>
<tr>
<td>other stuff</td>
<td>other stuff</td>
<td>function arguments and stuff</td>
</tr>
<tr>
<td>other stuff</td>
<td>gap</td>
<td>gap</td>
</tr>
<tr>
<td>gap</td>
<td>other stuff</td>
<td>other stuff</td>
</tr>
</tbody>
</table>
Functions can go in any order

- ssrbq_init
- ssrbq_reset
- ssrbq_rehash
- ssrbq_compute
- ssrbq_teardown

- ssrbq_compute
- ssrbq_teardown
- ssrbq_init
- ssrbq_reset
- ssrbq_rehash
Do I have To Worry About Performance?
Does the sun set in the west?
There is no measurable impact, can I check in?

I found one case with 2% impact, can I check in?

I fixed the performance, can I check in?

No, you have inadequate benchmarks.

No, you have demonstrated negative impact.

No, your benchmarks are not good enough.
Performance trumps security more often than not

- Performance is quantitative
- Easy to measure
Vulnerability is quantum

- Don’t know how it could possibly be vulnerable
- Hypothetically vulnerable
- Demonstrably vulnerable
- Exploited
Is It Worth The Bother?
Code Churn

✦ 180+ files with refcount_t
✦ 500+ instances
✦ Lots more to do
Runtime overhead

+ Hardened user copy

+ Checks in a lot of syscalls
Developer experience

- Simple as checkpatch
- Picky like %p
- Lots of compiler warnings
Harder Is Subjective
Yes, it is harder

- Community is buying in
- Working in the open is huge
- Amount of help has been awesome
- We’re all learning the bounds
Thank You