Linux kernel heap buffer overflow in fs_context.c since version 5.1 (seclists.org)

242 points by todsacerdoti 16 hours ago | hide | past | favorite | 97 comments

add comment

mjw1007 14 hours ago | next [–]

The Debian 11 release notes say:

« From Linux 5.10, all users are allowed to create user namespaces by default. This will allow programs such as web browsers and container managers to create more restricted sandboxes for untrusted or less-trusted code, without the need to run as root or to use a setuid-root helper.

The previous Debian default was to restrict this feature to processes running as root, because it exposed more security issues in the kernel. However, as the implementation of this feature has matured, we are now confident that the risk of enabling it is outweighed by the security benefits it provides.

If you prefer to keep this feature restricted, set the sysctl: user.max_user_namespaces = 0

Note that various desktop and container features will not work with this restriction in place, including web browsers, WebKitGTK, Flatpak and GNOME thumbnailing. »

Does anyone know a reason to keep this feature enabled on a server, other than Docker's rootless mode?

reply

prpl 14 hours ago | parent | next [–]

If you have a multi tenant server and don’t want to provide root access to users but want them to be able to run containers, otherwise it’s probably not necessary

reply

dathinab 14 hours ago | parent | prev | next [–]

some programs use it to sandbox themself without needing root. Through currently I can only think about desktop apps which do so.

reply

caaqil 14 hours ago | prev | next [–]

See it in action: https://twitter.com/ryaagard/status/1483592308352294917

reply

encryptluks2 14 hours ago | parent | next [–]

Arch shows some warnings about unprivileged user namespaces but it is enabled by default I believe which allows for rootless podman/docker. I didn't realize we'd actually see an exploit so soon

reply

stormbrew 11 hours ago | prev | next [–]

> An unprivileged user can use unshare(CLONE_NEWNS|CLONE_NEWUSER) to enter a namespace with the CAP_SYS_ADMIN permission, and then proceed with exploitation to root the system.

I'm confused by this, don't you need CAP_SYS_ADMIN to unshare(CLONE_NEWNS) to begin with?

From unshare(2):

> CLONE_NEWNS

> This flag has the same effect as the clone(2) CLONE_NEWNS flag. Unshare the mount namespace, so that the calling process has a private copy of its namespace which is not shared with any other process. Specifying this flag automatically

implies CLONE_FS as well. Use of CLONE_NEWNS requires the CAP_SYS_ADMIN capability. For further information, see mount_namespaces(7).

Edit: Oh does this work specifically *because* you're also unsharing into a new user namespace where you have that capability? This is kind of wild tbh.

reply

staticassertion 11 hours ago | parent | next [–]

Yeah, a lot of the Linux kernel code was reachable by root, and for a long time the attitude of a lot of kernel maintainers was that privesc from root didn't matter much.

But now any code can be root in its own namespace... so all of this code that's far less scrutinized is now reachable.

reply

boring_twenties 9 hours ago | root | parent | next [–]

Exactly the main argument that was being made against enabling this by default all those years ago.

reply

octoberfranklin 24 minutes ago | root | parent | next [–]

Indeed, this is exactly why containers are *not* a security feature.

Counterintuitively, using containers makes your system *less* secure because you have to expose this huge kernel attack surface to non-root users.

reply

alerighi 11 hours ago | parent | prev | next [–]

Some distributions disable user namespaces by default because they are considered a dangerous feature. And it probably is, in the end.

reply

brobdingnagians 8 hours ago | prev | next [–]

It seems like this `if (len > PAGE_SIZE - 2 - size);` is less natural than the correct code of `size + len + 2 > PAGE_SIZE`. Can anyone more familiar with C comment on why someone would write it the first way instead of the second way?

reply

pixelbeat__ 8 hours ago | parent | next [–]

Overflow https://www.pixelbeat.org/programming/gcc/integer_overflow.h...

reply

comex 24 minutes ago | parent | prev | next [–]

The problem is that *neither* option is correct if `size` and `len` can both be arbitrary unsigned integers. They can both overflow (counting underflow as a type of overflow). To be correct regardless of their values, you would have to write something like:

```
if (size + len < size ||
    size + len > PAGE_SIZE - 2)
```

But broadly speaking, adding explicit overflow checks everywhere makes code more verbose, and it isn't even a panacea, since it can be non-obvious whether any given overflow check is even correct (especially when C's arcane type conversion rules come into play). Plus, in theory every extra check makes the code larger and slower.

Therefore, it's common to see C code try to perform computations in a way that can't overflow in the first place. But that often requires putting them in a less natural form, as well as making assumptions about which things are bounded in which ranges. For example, even in the above example that does have an overflow check, I had to unnaturally put the 2 on the right hand side of the equation rather than the left, and assume that `PAGE_SIZE - 2` doesn't underflow. The fully natural computation would require two overflow checks:

```
if (size + len < size ||
    size + len + 2 < size + len ||
    size + len + 2 > PAGE_SIZE)
```

On the other hand, the buggy single comparison that was actually used:

```
if (len > PAGE_SIZE - 2 - size)
```

would be correct, and reasonably idiomatic, if `size` was known to be bounded below `PAGE_SIZE - 2`. Perhaps the author thought it was.

This is really an unfortunate limitation of C, though. It would be nice to be able to succinctly express "if size + len + 2 > PAGE_SIZE *or* the computation overflowed, then return error". From the hardware's perspective, that sort of check is really cheap; from the compiler's perspective, it's ripe for optimization. So if C had an ergonomic overflow-check

feature, you could standardize your codebase on using it for any calculation that's remotely related to buffer sizes, at very little performance cost.

With GCC extensions you can at least do this:

```
size_t tmp;
if (__builtin_add_overflow(size, len, &tmp) ||
    __builtin_add_overflow(tmp, 2, &tmp) ||
    tmp > PAGE_SIZE)
```

Using these builtins standardizes the form of overflow checks and avoids "is the overflow check correct" worries. But the ergonomics are arguably even worse, since you need to declare a temporary variable, and use long function names instead of normal mathematical operators.

reply

AceJohnny2 3 hours ago | parent | prev | next [–]

As a long-time embedded C coder: I agree that (with the added benefit on hindsight) the original code "smells".

I'm well-tuned to consider overflows, and adding numbers together instinctively makes me consider whether there's a risk... Similarly, subtracting numbers should make one consider underflows.

Perhaps the original author was used to working with signed values and forgot to consider the risk of underflow with these *unsigned* variables.

(I used to have C's type promotion rules on quickdraw, but I very soon learned that if I needed to refer to them, *I was probably doing something wrong*)

reply

pxeger1 13 hours ago | prev | next [–]

This is CVE-2022-0185 if you need to know it

reply

faisal_ksa 12 hours ago | prev | next [–]

I wander if rust (or any other memory safe system language in the future) could have avoided this exploit. If not, what could we do to avoid such exploits?

reply

gpm 12 hours ago | parent | next [–]

One method of forbidding the entire category of bugs is "bounds checks on integer arithmetic". Rust implements this in debug mode, but not by default in release mode, because it comes at a performance cost. To make this sort of solution ubiquitous you really want better hardware support to make bounds checking cheap.

Realistically I think it is unlikely you would have written the same exploit in rust even with integer overflow wrapping by default, because in idiomatic rust you end up using types with lengths attached to them, and memcpy methods that check that you didn't fuck up the lengths before copying. You absolutely could end up writing it in rust though (using unsafe code, but at some level unsafe code is inevitable for this sort of work), and you could if you really wanted to implement a similar set of safer buffer types in C that would provide a similar degree of prevention (though it would be more cumbersome to use than in rust).

reply

AceJohnny2 3 hours ago | root | parent | next [–]

> To make this sort of solution ubiquitous you really want better hardware support to make bounds checking cheap.

It's funny because it's *trivial* to implement underflow/overflow reporting in an ALU, but somehow that kind of event doesn't get reported to the offending program, at least at the naive C level.

reply

gpm 2 hours ago | root | parent | next [–]

My potentially incorrect understanding is that at the hardware level overflows basically always set flags indicating underflow/overflow, but

- Checking those flags and branching depending on extra instructions and comes at a performance cost, and that this could be instructions that trap on overflow instead of setting a flag. This can be solved, but needs instruction set level support.

- Requiring overflows are correctly handled at all comes at a performance cost at the optimization level (you can't turn (x + x - x) into a no-op), this is fundamental, but probably an acceptable cost if you solved the other issue.

C's arithmetic operators on unsigned operations also require the implementation doesn't return some sort of error on overflows, but for signed errors it would be a valid thing to do (since the behavior is undefined

by the spec), and you could use compiler supplied functions instead of the primitives for erroring on the unsigned operations as well (or a non-standards-compliant compiler flag).

Rust kept the option open in how they defined arithmetic. Currently it wraps in release mode, but it's explicitly backwards compatible to change that to a panic (rust's version of exceptions).

reply

ironhaven 6 hours ago | parent | prev | next [–]

I think this bug would not have been exploitable in rust. Rust would not catch the integer underflow, but the exploit was only possible because of memcpy.

You can line for line rewrite the code with unsafe rust and get the same exploit with a bad memcpy[1]. But this code would never have used unsafe rust.

The method that was exploited was "append user supplied strings to the end of a string". This can easily be written in safe rust, a thousand different ways. Here is one way it could have been written.

write!(&mut heapblockstring, ",{}={}\0", key, string).unwrap();

If this was the code in the kernel, you would have gotten an intentional crash and not privilege escalation

[1] https://doc.rust-lang.org/std/intrinsics/fn.copy_nonoverlapp...

reply

tialaramex 6 hours ago | parent | prev | next [–]

You can't have this bug in WUFFS. The WUFFS compiler will point out that it can't see any reason to believe the expression won't overflow, and any possibility of overflow is illegal in WUFFS, so your program is invalid and won't compile. So you need to write code which can't overflow.

Of course WUFFS is deliberately constrained to a simple world and thus utterly unsuitable for writing an operating system kernel. However, the principles do apply, the authors of this code didn't *want* overflow to be possible, they just didn't have a way to express that to their compiler.

In Rust it is at least easier to express that you don't want overflow:

```
let x = size.checked_add(len).unwrap().checked_add(2).unwrap();

if (x > PAGE_SIZE) { ...
```

The above code will panic (regardless of build parameters) if adding size + len + 2 overflows. If you don't want a panic, you can write some unwrap_or_else() code to pick some preferred value when overflow occurs, or you can handle the None result (which is what checked_add gives you for overflow) explicitly.

However, perhaps it should be possible to express *at compile time* that you want an expression which can't overflow at runtime, as in WUFFS.

reply

mustache_kimono 12 hours ago | parent | prev | next [–]

I'm not an expert, but I will say it may be easier to avoid an over/underflow with: https://doc.rust-lang.org/std/primitive.u32.html#method.satu...

And to check if one has occurred with: https://doc.rust-lang.org/std/primitive.u32.html#method.chec...

reply

menaerus 11 hours ago | parent | prev | next [–]

This got nothing to do with the memory but to the fact how CPU works with the integers. This means that (low-level) programming language fundamentally cannot solve this problem but only alleviate it either by:

1. Changing the semantics of integer arithmetic (e.g. saturate on overflow)

2. Keeping the semantics but babysit the computation during runtime so that the overflow/underflow can never happen (expensive)

reply

duped 11 hours ago | root | parent | next [–]

Modern CPUs will alert you to overflow and under flow. Rust actually panics on overflow or under flow conditions in debug builds by default.

It is not expensive to check for under flow at runtime in security critical code, and is actually mandatory for cases like this as it is UB in C.

reply

menaerus 10 hours ago | root | parent | next [–]

Sorry, but you're wrong in both of your claims.

First, unsigned integer underflow and overflow is _not_ UB. It is very well defined operation (wrap-around arithmetic) and the bug in question is not the result of undefined behavior and rust or whatever other bs I keep hearing around would have not solved it. It's the fundamental artifact of how CPUs work.

Secondly, CPUs have been "alerting" through their carry and overflow bits in registers since forever so this isn't some exclusive feature that only rust compiler writers were smart enough to take advantage of. The same code can be and is written where it matters in C and C++ code too.

It's not only the question if such extra checks are expensive (which they are given that integer arithmetic is such a fundamental operation and your favorite language disables it in release builds for the sakes of, I guess, nothing?) but it is also a question of all known _semantics_ of unsigned integer arithmetic. That's simply the way they work and I see no near future where the CPU hardware engineers would change that (they will not).

reply

▲ vlovich123 2 hours ago | root | parent | next [–]

What you actually want is to enable some kind automatic trapping behavior when a section of code is entered so that you can say “this set of math operations shouldn’t overflow”. That’s cheaper than what overflow bits get you although entering/exiting such a mode may be equally or more expensive.

The existence of the overflow bits and that overflow continues to remain a common security flaw indicates that there’s a disconnect between the mental model users have when writing this kind of arithmetic (ie they don’t think about it generally and C integer promotion rules don’t do any favors) and how CPU designers imagine you write code.

reply

▲ AceJohnny2 3 hours ago | root | parent | prev | next [–]

> First, unsigned integer underflow and overflow is _not_ UB. It is very well defined operation (wrap-around arithmetic) and the bug in question is not the result of undefined behavior

Shockingly true. Per the C Standard, "6.2.5 Types" paragraph 9:

A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.

reply

▲ im3w1l 10 hours ago | root | parent | prev | next [–]

You could imagine a version of the arithmetic instructions that traps on overflow. Or maybe a prefix for the normal instruction. Then it can be basically free in the happy path.

reply

▲ Diggsey 14 hours ago | prev | next [–]

I assume that `size + len + 2` can't *over*flow :)

reply

▲ snvzz 11 hours ago | prev | next [–]

There's likely many more of these.

As a reminder, Linux has millions of lines of code, and all of them run with supervisor privileges.

This is not a good architecture. Generally, you'd try to minimize the attack surface.

Multiserver, microkernel systems based on capabilities is where it's at.

seL4 is the better microkernel to build such a system on.

reply

▲ athrowaway3z 10 hours ago | parent | next [–]

There exist people who own their own hardware and are not providing an API to run arbitrary code.

If they get together and build an OS they are generally more interested in throughput than security models.

Both have pro's and con's but the fact is: one is more popular with the "just get something working" crowd for better and worse.

reply

▲ a-dub 10 hours ago | parent | prev | next [–]

i predict that tanenbaum will ultimately win the famous monolithic vs. microkernel design debate.

monolithic kernels are good for building features quickly and runtime performance, but the security design reminds me of 90s era computer security approaches, where firewalls were supposed to stop all threats and behind them security

was lax on internal networks. microkernels are much more similar to today's more effective defense in depth approaches.

what does it matter if your kernel is fast and featureful if you cannot trust it?

reply

elihu 4 hours ago | root | parent | next [–]

I think Tanenbaum will win the argument in the sense that the way high-security OS kernels will be written will look like programming a microkernel, but when you look at what the hardware actually executes it'll look like a monolithic kernel.

As an example, think of how Rust treats thread communication. The API looks like you're passing messages, but the whole data isn't being copied when you send a message, you're just sending a pointer. The compiler keeps track of making sure ownership rules are respected.

One of the nice things about using a message passing API internally even if the implementation is just single-address-space shared memory is that if you really want more security you can change the implementation to be one where actual message passing takes place, and threads are isolated into separate address spaces like you would do in a proper microkernel. Whether you turn that on or not depends on whether security or performance are more important for your chosen application. It would be much harder to convert something like the Linux kernel into a microkernel.

reply

VWWHFSfQ 8 hours ago | root | parent | prev | next [–]

> i predict that tanenbaum will ultimately win the famous monolithic vs. microkernel design debate.

Tanenbaum already lost the debate a long time ago. Minix is running in Intel CPUs but it's hardly exercising the benefits of a micro-kernel architecture. General-purpose PCs are running mono-kernels.

Maybe the micro-kernel will prove useful at some point in the future, but that's just theoretical. But _this_ war was lost. Maybe they'll win the next one!

reply

a-dub 7 hours ago | root | parent | next [–]

i predict that ultimately the linux kernel will prove too difficult to fully secure.

i predict that the successor will ultimately be a microkernel architecture with ipc that is built from the ground up to be hardened against hostile components and will be accelerated in hardware. i predict we'll see something like realtime cryptographic proofs of both the identity and correct execution of operating system component code and the code of user programs.

reply

throwaway984393 6 hours ago | root | parent | next [–]

I think the successor is *multi*kernel. Modern systems trip over themselves to provide isolated runtime environments within one kernel, but it's all a waste once you pop a hole in the one kernel. The answer to that is virtualization of entire kernels, ala Firecracker.

Rather than tying ourselves into knots trying to extend the one kernel, we can just run 50 kernels, and have abstractions to connect them into a distributed OS (think Plan9). This allows the kernel to be simpler, provides an easier path to standardized interfaces between components, and lets us infinitely scale a single system, with stronger isolation guarantees.

It's basically a microkernel but the components are farther apart. And because of that you can do things like have multiple different kernel versions interoperate over one stable abstraction/API/whatever. SSI clusters are another example of this, they just happen to be using the same kernel, but they don't have to.

reply

snvzz 9 hours ago | root | parent | prev | next [–]

>what does it matter if your kernel is fast and featureful if you cannot trust it?

And if the kernel is Linux, I'm not so sure about fast.

Relative to Linux, seL4 has:

- Order of magnitude faster context switch.

- Order of magnitude lower scheduler latency.

- Order of magnitude faster Inter-Process Communication.

reply

dundarious 8 hours ago | root | parent | next [–]

You can't really draw conclusions about systems built on Linux vs seL4 from that though. You can have far more IPC in order to do anything useful in microkernels. I'm not dismissing either OS or architecture, but just saying that extrapolating from micro-metrics is very hard when their use in the different architectures varies significantly.

reply

ylyn 2 hours ago | parent | prev | next [–]

You likely run only a small subset of those millions of lines, because the bulk of those are in drivers and arch-specific code..

reply

VWWHFSfQ 11 hours ago | parent | prev | next [–]

It's my understanding that the microkernel architecture is slow. Nearly unusably slow. And that's why nobody uses it. Am I off-base? I'm interested!

reply

roblabla 7 hours ago | root | parent | next [–]

The Nintendo Switch operating system, Horizon/NX, is a capability-based microkernel architecture. It's nowhere near unusably slow. In fact, its performance are pretty good. It does "cheat" a bit in that the whole OS runs on a single core, dedicating the remaining three cores to the game. But that may actually be the most efficient use of the hardware.

reply

snvzz 9 hours ago | root | parent | prev | next [–]

There's more myth than truth[0]. In the early days, they were slow. Mach, used in OSX, is a representative of those early days.

Liedtke's L4 proved that a performant microkernel is possible.

Later, SMP changed the scenario considerably, as all of a sudden the microkernel multiserver fits SMP like a globe, while monolithic kernels need the complexity of locks to handle it.

[0] https://news.ycombinator.com/item?id=10824382

reply

erichocean 8 hours ago | root | parent | next [–]

> Mach, used in OSX, is a representative of those early days.

Mach, the microkernel, is not using in OS X and never has been.

Mach, the API, is used (mostly due to legacy reasons).

OS X has a Linux-style monolithic kernel (with legacy Mach APIs in userspace).

reply

dippersauce 3 hours ago | root | parent | next [–]

>Mach, the microkernel, is not using in OS X and never has been.

The version of mach present in XNU is derived from OSFMK, which derives code from UoU's Mach 4 kernel, and from CMU's Mach 3 kernel. It contains improvements related to threading and contexts. It also definitely exists as more than an API. The code that makes up mach is present and identifiable, and thus is "in" XNU. [1]

>Mach, the API, is used (mostly due to legacy reasons).

See above. Abstractions are exposed, and traps are present, but this is not an API clone, it is mach, albeit modified from the original form. It was specifically chosen for its forward thinking benefits. Mach was not the basis for any major Apple OS prior to OS X. There were no "legacy reasons" to motivate its adoption.

>OS X has a Linux-style monolithic kernel (with legacy Mach APIs in userspace).

You do tend to see different opinions on this. Some argue macOS is a hybrid kernel, because it combines elements from a microkernel (mach), a monolithic kernel (BSD), and IOKit. A single address space is shared by the components though, which leads many to call it monolithic. [2]

1. https://github.com/apple/darwin-xnu/tree/main/osfmk

2. https://flylib.com/books/en/3.126.1.67/1/

reply

dralley 3 hours ago | root | parent | prev | next [–]

I thought Windows and OSX were closer to hybrid kernels, at least with respect to running a lot of their drivers in userspace.

reply

▲ dippersauce 3 hours ago | root | parent | next [–]

Almost every developer I have worked with considers NT and XNU to be hybrid at this point. Unfortunately, almost everything in the parent comment is incorrect.

reply

▲ 01100011 6 hours ago | root | parent | prev | next [–]

Generally yes. The microkernel architecture is theoretically better in many ways and this has led to many attempts over the years to switch to that architecture. Some attempts have been relatively successful. Over time, as hardware becomes more heterogeneous and more capable, we are likely going to see a move to microkernels.

Like many engineering decisions, the choice is not always clear cut. Other commenters have suggested microkernels are clearly superior and not choosing them is stupid, but there are many facets to that decision.

reply

▲ dijit 11 hours ago | root | parent | prev | next [–]

Yes, it’s going to be slower

All of those security checks between components and memory passing will cause it to be slower.

But that doesn’t mean it’s a worthy trade off.

People write software in python despite it being slower than C++.

reply

▲ hutrdvnj 10 hours ago | root | parent | next [–]

Except that anything that is actually performance critical is written as a C extension python module.

I think that low level filesystem operations are very performance critical.

reply

▲ jeffbee 10 hours ago | root | parent | next [–]

I doubt that mounting a filesystem is performance-critical. You could afford to fork an unprivileged user-space process written in Perl to parse these mount options and that would be fast enough for everyone.

reply

▲ YarickR2 9 hours ago | root | parent | next [–]

Tell that to dockerd mounting images layer by layer , with k8s doing all kinds of emptyDir/PVC mounts on top. Pod start up speeds are abysmal now, they would be positively glacial with userspafe permissions validations

reply

▲ stormbrew 7 hours ago | root | parent | next [–]

?? The layering unionfs (really aufs or overlayfs) mounts docker does are all in kernel. There's no userspace involvement in the actual file ops. I'm not at all sure what you think any of that has to do with the idea of usermode filesystem handling.

Anyways, on linux userland filesystem stuff (ie. fuse) is slow because the kernel manages the vfs layer and has to mediate. In a system designed for non-privileged vfs it doesn't have to be anywhere near so involved. And beyond that, there's really nothing in particular that a filesystem driver does that *needs* to be privileged to 'run fast'.

VFS is pretty much the posterchild case for how you can pull things out of a kernel and maintain performance, tbh. There's no reason at all it can't be fast except the kernel getting in the way rather than helping.

reply

▲ jeffbee 9 hours ago | root | parent | prev | next [–]

That's exactly my point. People who are doing lots of mounts are already demonstrably not performance-sensitive to a difference of a few milliseconds. They already waited 20 minutes for the stupid cluster autoscaler to provision a machine for them. They DGAF.

reply

▲ jeffbee 10 hours ago | root | parent | prev | next [–]

As more and more things move out of the kernel, the perceived performance problems of microkernels look less important. If you are doing your network protocols in user space, and your thread scheduling is in user space,

and you're not using a traditional filesystem much, then suddenly nobody cares how fast the kernel is.

reply

stormbrew 10 hours ago | root | parent | next [–]

Yeah this, really. For the most part even monolithic kernels have kind of reversed trend in the last decade or so and there's a lot of push to move critical code out of the kernel. A lot of new kernel apis are built to avoid context switching, and part of that usually involves moving to a more asynchronous kind of communication between process and kernel.

Often these APIs even look an awful lot like late generation microkernel shared memory buffer protocols. DRI and uring in linux for example.

A lot of the "microkernels are inherently slow" meme is built on how earlier port-based message passing kernels like mach worked.

reply

athrowaway3z 10 hours ago | root | parent | prev | next [–]

I'm really not understanding what you're saying here.

What the hell is a microkernel here? What kind of security are you talking about?

User space filesystem and network implementations still need access to the hardware. Multiplexing that access is a kernels job. The more you want to separate and hide that between clients the higher the cost.

As far as i understand your argument you are saying "If an application has a dedicated hard drive there will be little overhead"

reply

adwn 43 minutes ago | root | parent | next [–]

> *User space filesystem and network implementations still need access to the hardware.*

No, they don't. Filesystems don't need access to the hardware, that part is handled by the block device driver. Network stacks (e.g., TCP/IP) also don't need access to the hardware, that part is handled by the network interface driver.

reply

jeffbee 9 hours ago | root | parent | prev | next [–]

People think a monolithic kernel can be faster because of high-level abstractions that make many calls within the kernel, for example if I write to a TCP socket and everything else is handled for me, in the kernel, by function calls only. People believe this is faster than having an isolated network stack that has to pass messages to an isolated network driver and all that.

But increasingly people realize that the performance of writing to a TCP socket in a unikernel also pretty much sucks and you get a much better result as you move more and more of it into user space. For example you decide, correctly, that TCP is obsolete and you switch to QUIC. Now the existence of the Linux TCP stack is of no value to you. You furthermore discover that Linux firewall, traffic control, and routing also kinda sucks, so you start using raw networking. Then you discover that trying to get your frames processed on the right core at the right moment isn't great in Linux, so you just take over the whole net device with DPDK.

Now, *nothing* in the whole Linux network stack is of any use to you at all.

The same thing can happen with storage. Maybe you started with files on XFS but then eventually you were using disaggregated storage where a service takes over the whole device with SPDK, and all the storage users are talking to the service instead of the kernel.

reply

tialaramex 5 hours ago | root | parent | next [–]

Of course, although you may be able to go very much faster, you also now took on all this extra responsibility, and so complexity, and so likely bugs.

Chances are the weird corner case where the TCP implementation blows up was seen in a dozen other Linux applications and fixed, but the same case inside your entirely userspace QUIC implementation gets to explode while the boss is doing the live $100M pitch demo. Awkward.

Code that is competing with Linux for performance is likely to be highly concurrent (a big risk factor for bugs) and doing fancy gymnastics with memory (another risk factor) so we're not talking about a landscape where bugs are rare or one where they're very easy to track down.

I believe this sort of thing is one reason Linux is putting more emphasis on safety. As usual for HN this bug already has people saying both "Rust wouldn't catch this" and "actually Rust

would catch this", but even if you're quite sure Rust isn't the answer a safer Linux by whatever means makes this argument better for Linux than it is today.

reply

nwmcsween 10 hours ago | root | parent | prev | next [–]

This depends on the definition of the word microkernel, in the classical definition yes it will be much slower due to IPC, for something like an exokernel though it will be much faster than even a monolithic kernel.

reply

tremon 14 hours ago | prev | next [–]

What's the origin of the legacy_parse_param size parameter (from struct fs_context->fs_private->data_size)? Is this a mount option, a format-time fs configuration option, or does it require writing a specially-crafted inode to disk? The exploit says the user needs CAP_SYS_ADMIN, so I'm guessing it's the first one?

reply

shakna 14 hours ago | parent | next [–]

From the commit [0] that added it:

> Legacy filesystems are supported by the provision of a set of legacy fs_context operations that build up a list of mount options and then invoke fs_type->mount() from within the fs_context ->get_tree() operation. This allows all filesystems to be accessed using fs_context.

And then the description of the function itself:

> Add a parameter to a legacy config. We build up a comma-separated list of options.

It looks to be the second one.

[0] https://github.com/torvalds/linux/commit/3e1aeb00e6d132efc15...

reply

AshamedCaptain 11 hours ago | parent | prev | next [–]

I am surprised that mounting is now allowed inside containers. Doesn't this expose a load of new surface attack for the kernel? All these pesky academical filesystem code does not inspire a lot of confidence when parsing user data/disk images....

reply

carlhjerpe 12 hours ago | prev | next [–]

Somehow I first thought I was affected personally, but I'm on 5.15. Version numbers crossing 10 messes my head up more often than I'd be willing to admit in person.

reply

Eduard 12 hours ago | parent | next [–]

So you are affected...?

reply

carlhjerpe 11 hours ago | root | parent | next [–]

I need to stop posting HN on the subway, yes I'm affected. This might be a bad time to be on NixOS unstable. But it seems hydra has been green since 3 hours so I should get a patch soon.

reply

phendrenad2 13 hours ago | prev | next [–]

Does anyone offer security fix backports for Linux? If I'm stuck on Linux 5.1, is my only recourse to update or patch it myself?

reply

rwmj 12 hours ago | parent | next [–]

It's a one line patch in old code so it should apply easily. However if you encounter this kind of problem a lot I'd highly advise some kind of long-term supported Linux distribution. (I work at Red Hat on RHEL, and that's what people pay us for)

reply

adfsdsaf 13 hours ago | parent | prev | next [–]

If you're stuck on 5.1, you probably have a ton of other vulnerabilities too. 5.1 isn't even an LTS release, so support for it was dropped in 2019.

5.4 is the first LTS of 5.x, and is supported through 2025. You should try to find a way to get on an LTS kernel, or plan on managing a lot of kernel patches.

reply

singron 13 hours ago | parent | prev | next [–]

There are Linux stable branches that backport fixes. It looks like all the affected branches have the fix now: linux-5.16.y linux-5.15.y linux-5.10.y linux-5.4.y linux-rolling-lts linux-rolling-stable

EDIT: notably absent are linux-5.1.y and other non-lts releases. If you can't stay on the most recent stable release, you should use lts releases.

reply

whimsicalism 13 hours ago | prev | next [–]

The compiler can't warn about something like this? I guess unsigned integer underflow can be the intended behavior often.

reply

mustache_kimono 12 hours ago | parent | next [–]

It could, for other reasons, never underflow. C expects that you know what you're doing, and C expected you did a bounds check. But I agree. Cases like this should have a lint warn on them, saying -- "Wake up programmer!"

reply

menaerus 11 hours ago | root | parent | next [–]

Fundamentally this problem cannot be solved at the compile-time level because, well, code is dealing with the values which are only known during code execution runtime. So I don't think compiler can do much here other than providing you with a hint that you may rewrite your expression but only to reduce the risk of a potential error, e.g. `if (len + 2 + size > PAGE_SIZE)` still still remains feasible to unsigned integer overflow and to handle the problem fully one must either:

1. Write a lot of convoluted if..else logic such as https://wiki.sei.cmu.edu/confluence/display/c/INT30-C.+Ensur... and https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensur...

2. Or use compiler built-in intrinsics, e.g. https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins...

But almost nobody does that ... except probably where it really matters (not the OS kernel).

reply

kevin_thibedeau 5 hours ago | root | parent | next [–]

A dynamic_assert() function would be a more general way to check a predicate at runtime. It can be compiled out for release builds and report failures to a centralized log for cases where the assertion is ignored.

reply

mustache_kimono 9 hours ago | root | parent | prev | next [–]

What you say makes sense. I was obviously wishcasting. ;)

reply

whimsicalism 11 hours ago | root | parent | prev | next [–]

Not solved at compile-time, but warned at compile time?

reply

menaerus 10 hours ago | root | parent | next [–]

Warned about what exactly? Literally any operation on two unsigned integers can either underflow or overflow and any of those would still be correct and expected behavior.

reply

touisteur 11 hours ago | root | parent | prev | next [–]

Could be solved at compile time with proof of absence of runtime errors, which somehow forces you to handle all cases for any input.

reply

dahfizz 10 hours ago | parent | prev | next [–]

It's not obvious what the warning would be, unless you want a warning attached to every single arithmetic operation? The compiler can't know what `size` will be in this case.

reply

whimsicalism 10 hours ago | root | parent | next [–]

Comparisons in an if statement involving subtracting two unsigned variables from each other?

reply

kidd0 9 hours ago | prev | next [–]

Does it effect Amazon Linux2?

reply

jftuga 8 hours ago | parent | next [–]

Amazon Linux 2 AMI is now available with kernel 5.10[0]

Posted On: Nov 19, 2021

Amazon Linux 2 is now available with an updated Linux kernel (5.10) as an Amazon Machine Image (AMI). Kernel 5.10 brings a number of features and performance improvements, including optimizations for Intel Ice Lake processors and AWS Graviton2 processors powering the latest generation Amazon EC2 instances. Live patching for Kernel 5.10 is supported in Amazon Linux 2 for both x86 and ARM architectures.

[0]https://aws.amazon.com/about-aws/whats-new/2021/11/amazon-li...

reply

prototype 8 hours ago | prev | next [–]

did Linus talk about this exploit in a mailing list? if so, can someone link it here? its always interesting to read Linus' perspective

reply

jakeinspace 13 hours ago | prev [–]

Sorry to be somewhat off-topic, but I have a Linux kernel bug question. I found a very small kernel bug (no obvious security implication, only affecting 32-bit builds) at work a few weeks back while working on a custom kernel patch. I sent an email to the maintainers for that kernel subsystem, but didn't hear back. I'm not quite sure if I should keep pestering them until I get a response, or if I should be doing something else to get it addressed. Any suggestions from someone with experience?

reply

sweettea 13 hours ago | parent | next [–]

There are both maintainers and lists listed in MAINTAINERS (L: entries) -- did you Cc the mailing list? It might be good to bump the mailing list email if it's been several weeks, asking if there's more information you could provide.

reply

jakeinspace 11 hours ago | root | parent | next [–]

This is for timekeeping, which doesn’t look to have its own mailing list (just points me to the vger linux-kernel mail list). I didn’t Cc that mailing list, although I could.

reply

avar 8 hours ago | root | parent | next [–]

The LKML is still the right place to send patches that don't fall under one of the other subsystem mailing lists. When in doubt you can ask the LKML about the right place to send a patch.

reply

onphonenow 10 hours ago | parent | prev | next [–]

If you are filing a bug you will be ignored (or could be ignored) forever.

If you send in a patch - you are MUCH MUCH more likely to get a response. As it should be.

What is obvious to you as a patch may not be obvious to others, so if you can write and test your patch that would go a long way towards getting things to move forward. Bug reports are noise to many maintainers (they know there are lots, their focus is on code that fixes bugs).

reply

tych0 13 hours ago | parent | prev | next [–]

Standard advice is to wait two full weeks, then bump your thread (or rebase the patch and send a v2 if there's new conflicts with the maintainer's tree).

reply

jakeinspace 11 hours ago | root | parent | next [–]

It’s a 1-line patch, I didn’t send it in my initial email but maybe that was a faux pas.

reply

tych0 11 hours ago | root | parent | next [–]

I'd say sending the patch with git send-email --in-reply-to=<the header of your last email> is good. A patch is much easier to apply than write :)

reply

jakeinspace 11 hours ago | root | parent | next [–]

Thanks! Would you recommend I send to the maintainer(s), and Cc the mailing list?

reply

tych0 10 hours ago | root | parent | next [–]

Yeah, I'd just use whatever the output of ./scripts/get_maintainer.pl says. It will also suggest any recent committers to that area of the code, which I've found useful in the past. Usually I put the maintainers as To:, and everyone else as Cc:.

reply

charcircuit 13 hours ago | root | parent | prev | next [–]

It sounded like he just filed a bug.

reply

bonzini 12 hours ago | parent | prev [–]

What subsystem is it?

reply

jakeinspace 11 hours ago | root | parent [–]

Timekeeping

reply

worthless-trash 7 hours ago | root | parent [–]

You'd be surprised that some time_t bugs are actually security related.

reply