Auditing Rust Crypto: The First Hours (kudelskisecurity.com)

149 points by lwhsiao on Feb 7, 2019 | hide | past | web | favorite | 86 comments

drexlspivey on Feb 7, 2019 [-]

The author JP Aumasson (known for inventing the BLAKE hash, SHA3 finalist) recently audited grin's underlying crypto library. Report here https://grin-tech.org/audits/jpa-audit-report.html

firethief on Feb 7, 2019 [-]

> Are sensitive values set to zero after being used? In Rust this may be done using the Drop trait rather than explicitly.

A zeroing Drop is theater unless the sensitive data is Pinned too. Rust likes to copy things around.

ncmncm on Feb 8, 2019 [-]

This isn't the half of it!

Optimizers consider code that zeroes a dying object to be dead code. (This is also a problem in C++ destructors.) It can be arbitrarily difficult to keep such code from being elided in various passes. Even inline asm instructions can be removed by a peephole pass.

Then, the CPU might decide not to flush all those zeroes from its writeback cache until some random time later when it needs that bit of cache for something else.

This is all wonderful for performance, usually, but it means you it takes a lot of dog-work to disable it.

If you have not verified that memory actually has zeroes in it, as seen by another core or DMA engine, it doesn't.

segfaultbuserr on Feb 8, 2019 [-]

Special primitives in programming languages should be provided to solve this problem, just like how Linux kernel uses a "barrier()" macro to mark non-reorderable code, and "wmb()" and "rmb()" to mark non-reorderable I/O.

OpenBSD provides

```
void explicit_bzero(void *s, size_t n);
```

In libc, which is used for this purpose. Recently, glibc added function, presumably because it's a good idea.

But it still has limitations due to lack of compiler support. I think the problem can only be solved at the compiler/OS level.

> The explicit_bzero() function addresses a problem that security-conscious applications may run into when using bzero(): if the compiler can deduce that the location to zeroed will never again be touched by a correct program, then it may remove the bzero() call altogether. This is a problem if the intent of the bzero() call was to erase sensitive data (e.g., passwords) to prevent the possibility that the data was leaked by an incorrect or compromised program. Calls to explicit_bzero() are never optimized away by the compiler.

> The explicit_bzero() function does not solve all problems associated with erasing sensitive data:

> 1. The explicit_bzero() function does not guarantee that sensitive data is completely erased from memory. (The same is true of bzero().) For example, there may be copies of the sensitive data in a register and in "scratch" stack areas. The explicit_bzero() function is not aware of these copies, and can't erase them.

> 2. In some circumstances, explicit_bzero() can decrease security. If the compiler determined that the variable containing the sensitive data could be optimized to be stored in a register (because it is small enough to fit in a register, and no operation other than the explicit_bzero() call would need to take the address of the variable), then the explicit_bzero() call will force the data to be copied from the register to a location in RAM that is then immediately erased (while the copy in the register remains unaffected). The problem here is that data in RAM is more likely to be exposed by a bug than data in a register, and thus the explicit_bzero() call creates a brief time window where the sensitive data is more vulnerable than it would otherwise have been if no attempt had been made to erase the data.

ncmncm on Feb 8, 2019 [-]

> I think the problem can only be solved at the compiler/OS level.

There's the problem. That is a lot of levels. There is support at certain levels; nowadays you can force writebacks from certain cache lines, and OSes let you keep pages from being swapped. The problem is that it needs coordinated support at all levels, but there is no money behind it, unlike features that may improve performance, and hardly anybody knows there is even a problem.

Spectre and Meltdown have raised awareness of security threats in infrastructure, but are so big they also absorb all the budget for it, and will continue indefinitely.

segfaultbuserr on Feb 9, 2019 [-]

Exactly. It's definitely possible, but something like this needs great amount of coordination and money.

I've once read something about Remote Direct Memory Access, it allows one to completely bypass the kernel, CPU, cache, and stream data directly to the network adapter of another computer - you break a lot of essential abstractions of modern computer system, meanwhile still be able to provide a consistent and usable infrastructure.

Its security equivalent (something allows low-level control of the protected data) is unlikely to happen, except for DRM, I guess.

ncmncm on Feb 8, 2019 [-]

It's not all bad news.

"Transactional Memory" seems to have flopped for lock-free synchronization, but it is a nice way to keep writes from ever going out to RAM or beyond. It reserves a bit of cache for temporary storage, and automatically wipes it if anything happens, like an interrupt -- even on a guest OS.

That is better than zeroing, and the toolchain won't fight you. But it might not be available everywhere you want it. Also, it wasn't designed for crypto, so there are probably vulnerabilities to the host OS.

monocasa on Feb 8, 2019 [-]

Transactional memory doesn't help you here. Everything will be cleared, including whatever you were trying to do with the secret.

ncmncm on Feb 8, 2019 [-]

It means you have to start over if it gets cleared. That adds latency spikes to processes that use it, which have to be planned for.

Everything around security is hard. It gets easier if you don't care whether it really is secure, which is common.

lilyball on Feb 8, 2019 [-]

I'm not particularly familiar with Pinned but the documentation on std::pin suggests it prevents moving data, not copying data (and is primarily designed around enabling self-referential structs).

kibwen on Feb 8, 2019 [-]

AFAIK the purpose of pinning is to statically ensure that the pinned data remains at the current address in memory, for the purpose of handing out references to the data that remain valid even if the precise lifetime of those references isn't statically known (the soundness and safety of Rust's async story relies on this property holding, so the devs are incentivized to uphold it). I won't claim to be able prove that it's impossible for a too-smart compiler to still somehow manage to have pinned data lingering somewhere else in memory, but it should be a good start.

legulere on Feb 8, 2019 [-]

Moves are exactly the problem, as the data is just copied, without drop running on the source, and they happen implicitly.

lilyball on Feb 8, 2019 [-]

Rust isn't going to just silently move the pointed-to data in something like Box<[u8; 32]>. It looks like the point of Pinned is to protect against operations like swap() that would move the pointed-to data, but the point here is to maintain the data's invariants (e.g. a self-referential struct cannot be moved or its self-references would be invalid) rather than to protect against silent moves (a call to swap() is definitely not silent).

masklinn on Feb 8, 2019 [-]

> Rust isn't going to just silently move the pointed-to data in something like Box<[u8; 32]>.

Of course not, but for efficiency you might have an [u8;32][0] instead and *that* will get memcpy'd when it's moved around.

[0] 32 bytes on the stack isn't much, a single Vec or String is 24

lilyball on Feb 8, 2019 [-]

A [u8;32] doesn't work with Pin. Pin only works when you wrap it around a pointer type. From the documentation on Unpin:

> Since Rust itself has no notion of immovable types, and will consider moves to always be safe, this trait cannot prevent types from moving by itself.

From the documentation on std::pin:

> In order to prevent objects from moving, they must be pinned by wrapping a pointer to the data in the Pin type. A pointer wrapped in a Pin is otherwise equivalent to its normal version, e.g. Pin<Box<T>> and Box<T> work the same way except that the first is pinning the value of T in place.

> First of all, these are pointer types because pinned data mustn't be passed around by value (that would change its location in memory). …

Looking at Pin itself, it appears that it works by only implementing DerefMut if its target is Unpin, thus preventing you from mutating (and therefore swapping/replacing) the data unless the data supports Unpin. But beyond that, it also only implements Deref and DerefMut if its wrapped type itself implements Deref/DerefMut, which means using Pin around a non-pointer basically prevents you from accessing the wrapped data.

tomjakubowski on Feb 8, 2019 [-]

It's not good enough to make the type non-Clone (and therefore, non-Copy)?

jake_the_third on Feb 8, 2019 [-]

Nope. Not at all.

Afaict, Rust memcopys moved objects if they're larger than some size when passed as function arguments. Also take into account implicit copies such as those left laying around when a vector outgrows it's heap segment.

ncmncm on Feb 8, 2019 [-]

Very, very far from good enough. I.e., the first step down a long and lonely road.

Crypto Implementation Is Hard. Especially when you need it to be secret. It is lots easier if you don't care.

tomjakubowski on Feb 8, 2019 [-]

Yeah. I guess that I just don't see what Pin provides that non-Copy/Clone does, when it comes to preventing the zeroed data from leaking elsewhere.

In other words, why is a zeroing Drop of a type more secure if it uses Pin too?

firethief on Feb 8, 2019 [-]

Copy/Clone/move is a typesystem distinction. At codegen time, they all can mean "memcpy this byte range". E.g. how else to return a struct by value?

ncmncm on Feb 8, 2019 [-]

The unpleasant answer is that both might do no good at all, by themselves.

masklinn on Feb 8, 2019 [-]

No. Both copies and moves are simple memcpy at the machine-level[0], the only difference is whether the compiler will let you keep using the source afterwards.

[0] though I guess the compiler might be smart enough to implicitly pass big-enough values by pointer instead

vardump on Feb 7, 2019 [-]

It's a bit surprising that the article didn't mention ensuring constant time algorithms are used. If latency or the emissions of the CPU (etc. externally observable) are data dependent on something secret, that secret might leak through the side channel.

One way to combat these side channels is to use constant time multiplication, loop conditions (including functions that might return early, like memcmp), etc.

Is it possible for Rust to have any new or surprising mechanisms that cause loss of constant time execution?

(Of course also other aspects can leak than the CPU instruction execution latency itself. For example cache effects might be measurable.)

nicoburns on Feb 8, 2019 [-]

> Is it possible for Rust to have any new or surprising mechanisms that cause loss of constant time execution?

I believe it's certainly possible. And the precise consequences of writing this code in Rust have yet to be fully explored. The `ring` (Rust port/fork of BoringSSL) library still links to constant-time primitives implemented in C.

jpfed on Feb 8, 2019 [-]

>Is it possible for Rust to have any new or surprising mechanisms that cause loss of constant time execution?

The ubiquitous assumption that all code should be executed as fast as possible isn't new or Rust-specific but it will be a continuing source of security surprises for the foreseeable future.

GordonS on Feb 7, 2019 [-]

I've only read about the relevance of constant-time comparisons in relation to comparisons, such as checking that a calculated hash is the same as a stored hash - could you elaborate on how it applied to symmetric encryption?

vardump on Feb 7, 2019 [-]

Some operations, such as integer multiplication and division have data dependent latency. Depends on CPU model.

See for example this page: https://bearssl.org/ctmul.html

> could you elaborate on how it applied to symmetric encryption?

Non-constant time multiplication is mostly (only?) an issue with asymmetric crypto.

fpgaminer on Feb 7, 2019 [-]

If an implementation isn't constant time it can potentially leak information about the key. i.e. the relative length of time it takes to encrypt may be dependent on properties of the key. So an attacker that can measure how long it takes you to encrypt something can begin to derive the key.

Similar attacks occur when your encryption uses lookup tables depending on key bits, uses different amounts of power, etc.

It's a surprisingly effective attack surface, and why the industry is moving away from older style schemes like AES and towards ARX based constructions like ChaCha.

saagarjha on Feb 7, 2019 [-]

> Look for any recursive function calls and see if they could be abused to overflow the stack.

This isn't an exploitable vulnerability in Rust, though? I thought the program would just abort if it overflowed the stack.

Hello71 on Feb 7, 2019 [-]

DoS is a vulnerability

saagarjha on Feb 7, 2019 [-]

Ok, point taken. It's generally much less severe than the other types of vulnerabilities, though.

muricula on Feb 8, 2019 [-]

If you can make a truly enormous stack array you can sometimes skip past the end of the stack onto the heap (or another section of memory). Then any writes to the array which the program thinks are going to the stack are actually going to the heap. This is sometimes known as stack clashing.

kam on Feb 8, 2019 [-]

At least on x86-64, Rust inserts stack probes to ensure that it can't skip over the guard pages at the end of the stack. Last I saw, it was waiting on LLVM support to add this for some other platforms, though.

saagarjha on Feb 8, 2019 [-]

Is this something you can do in Rust?

lilyball on Feb 8, 2019 [-]

I haven't tried, but maybe with something like

```
let huge_ary: [u8; 2097152 /* 2MiB */] = unsafe { ::std::mem::uninitialized() };
// hopefully huge_ary has bumped the stack pointer past any guard pages
// and any further stack values will overwrite the heap
```

saagarjha on Feb 8, 2019 [-]

Well, you're doing unsafe things there, so all bets are off.

lilyball on Feb 8, 2019 [-]

You might be able to hack it by saying something like

```
let huge_ary: [u8; 2097152];
if false_value_compiler_cant_prove_is_false {
  huge_ary = [0; 2097152];
}
// hopefully huge_ary has reserved space on the stack despite not being initialized
// and so any further stack values would overwrite the heap
```

estebank on Feb 8, 2019 [-]

You'll get

```
error[E0381]: borrow of possibly uninitialized variable: `huge_ary`
 --> src/main.rs:6:7
  |
6 |   bar(&huge_ary);
  |       ^^^^^^^^^ use of possibly uninitialized `huge_ary`
```

https://play.rust-lang.org/?version=nightly&mode=debug&editi...

lilyball on Feb 8, 2019 [-]

You're not supposed to *use* the array, it's just supposed to exist on the stack. I'm not sure how to ensure the stack space isn't reclaimed; maybe we can do something with declaring it up top, then putting the `if` that initializes it later on, and hope the compiler still reserves the stack space for huge_ary ahead of stack for other variables?

hyperman1 on Feb 8, 2019 [-]

Why do you need the if construction? The stack grows downards (at least on x86_64), and clearing with zeroes in general starts at the first adress (even if this isnt guaranteed). So the CPU sees a write 2097152 bytes below the last used stack location. Which might be in the heap if ther is no stack protection.

Then again, 2Mb is nothing on a 64 bit processor, so I don't think the chanse of hitting the heap is very big.

lilyball on Feb 8, 2019 [-]

Overwriting the heap is pointless if you immediately crash due to hitting the guard page(s). The point is to move the stack pointer without actually writing anything, so you can then write what you want into the heap without crashing.

The 2Mib was assuming you've already pushed the stack to its limit and are sitting just above the guard page(s). You could replace that with whatever size you think is appropriate.

CaliforniaKarl on Feb 7, 2019 [-]

Looking at the article, I like the Drop trait, as a way to ensure memory are cleaned before being released.

What is the preferred way in Rust to say "If at all possible, please do not allow this value/object/whatever to be paged out of RAM?"

monocasa on Feb 7, 2019 [-]

Those system calls don't really work like that; there isn't really a public way to say "don't save a copy of this page on disk" on Windows, OSX, or Linux.

For instance see Raymond Chen's explanation here

> Is VirtualLock safe to use for cryptographic purposes – that is, to avoid sensitive memory being written to disk?

> ...

> [There does not appear to be any guarantee that the memory won't be written to disk while locked. As you noted, the machine may be hibernated, or it may be running in a VM that gets snapshotted. -Raymond]

VirtualLock and the equivalents only guarantee that the canonical copy of that page stays in RAM, not that other copies won't make their way out to disk.

Edit: add link https://blogs.msdn.microsoft.com/oldnewthing/20140207-00/?p=...

Grollicus on Feb 7, 2019 [-]

With security mindset, "if at all possible do not allow" is the basically the same as "always allow".

As mlock / VirtualProtect are OS dependent I strongly suspect this is not part of Rust itself. Additionally, I suspect there would be some design problems for example putting this protected thing on the stack (protect the whole stack? Or just parts? Bookkeeping wether a stack lock can be released after a function exits?). How do you move values into / out of such an object?

So I don't think this exists, but I've been pleasantly surprised by Rust before.

My general solution for this case is to either mlockall the whole process if it is small / short-living or to have no swap in the machine, for example long running crypto servers. Who runs servers with swap anyways?

wyldfire on Feb 7, 2019 [-]

There's no target neutral way in Rust IIRC but there is a standard way among unix-style targets. It's page granularity, and requires superuser privileges though.

See mlock(2) for details [1].

[1] https://man.openbsd.org/mlock.2

the8472 on Feb 7, 2019 [-]

At least on linux unprivileged processes can lock a limited amount of memory.

> Since Linux 2.6.9, no limits are placed on the amount of memory that a privileged process can lock and the RLIMIT_MEMLOCK soft resource limit instead defines a limit on how much memory an unprivileged process may lock.

TheDong on Feb 7, 2019 [-]

It's pretty much impossible to zero out memory in all cases in any language. See [0].

In rust, there's the zeroize crate [1], which uses compiler fences and volitile memory to have a shot at it, but it's still not a sure thing.

That library doesn't help with the paging issue you mention, but what you can do for that is entirely platform dependent (up to the kernel's paging mechanism), and so rust isn't going to make it any easier or harder than just making the libc calls/syscalls yourself.

[0]: http://www.daemonology.net/blog/2014-09-06-zeroing-buffers-i...

[1]:https://github.com/iqlusioninc/crates/tree/zeroize/v0.5.2/ze...

kccqzy on Feb 7, 2019 [-]

Your first reference refers to the problem in C. The C standard has given C compilers too much leeway in managing memory (spilling things to the stack, copying values etc). That doesn't mean it's impossible to zero out memory in all cases in any language. Your rust crate similarly uses a combination of tricks to ensure zeroing buffers is not optimized away. The core problem here is compiler optimizations.

If you write in, say, assembler, you will have precise control in zeroing the buffers and not spilling sensitive register values to the stack, and also zero out XMM registers.

In fact that the very article you linked to outlined a solution in the form of a proposed language extension disagrees with your characterization that this problem cannot be solved in any languages. Clearly, the author thinks that it's still possible to zero buffers correctly in a language, but just that C is not such a language unless it gains his proposed extension.

pjmlp on Feb 8, 2019 [-]

> If you write in, say, assembler, you will have precise control in zeroing the buffers and not spilling sensitive register values to the stack, and also zero out XMM registers

Not really, because micro-coded CPUs have won and you don't have any control over what they do, and as last year has proven there are ways to exploit execution timing in micro-code.

ncmncm on Feb 8, 2019 [-]

This is an overoptimistic view. Yes, people are trying to create ways to zero things reliably, but all the machinery meant to improve performance is doing its level best to undermine that.

It has to be solved at all levels; each level gets a chance to sabotage it, and will.

TheDong on Feb 8, 2019 [-]

Rust is an llvm based language, which is to say the caveats that apply to optimizing c compilers almost all apply identically to rust since llvm at its core is more an optimizing c compiler than anything else.

vardump on Feb 7, 2019 [-]

> ... "If at all possible, please do not allow this value/object/whatever to be paged out of RAM?"

With the precondition you're running on *bare metal* (no virtualization), I think you'd need a kernel mode driver for that to cover all scenarios, including system hibernation and suspend.

You can reliably zero out the secret data before hibernation or even suspend by implementing power management calls – mlock or VirtualProtect aren't going to protect you from that leak.

blattimwind on Feb 7, 2019 [-]

You don't really need a driver for that. On Linux you could use a systemd unit with Before=sleep.target (or probably a udev rule). Windows has PBT_APMSUSPEND (which some applications use to quick-save before entering stand-by mode).

vardump on Feb 8, 2019 [-]

I think the systemd case is ok.

But PBT_APMSUSPEND is not necessarily. There's a 2 second timeout before Windows suspends or hibernates regardless. It's possible for the event handling thread not to run until *after* the timeout has passed, that is, after the system has returned from suspend or hibernate.

That's important considering the adversary might *induce* this kind of scenario to steal the secret.

Kernel mode driver does not have that limitation, because it can't be pre-empted.

codys on Feb 7, 2019 [-]

One thing to note is that it is not guaranteed that values are ever dropped. In rust, it is "safe" to leak memory (it wasn't possible to make it unsafe in rust).

zozbot123 on Feb 7, 2019 [-]

AIUI, this *should* only come up in rare cases, mostly involving RC cycles. The "should" proviso is there because the operation of "forgetting" that a value has to be dropped (thus de-facto leaking it) is in fact allowed in Safe Rust, and it's not immediately clear to me where *that* might come up in practice.

lilyball on Feb 7, 2019 [-]

std::mem::forget() is allowed in Safe Rust specifically because it's always possible to reconstruct it safely using Rc.

As for where it comes up, it's usually because you're handing ownership of the memory to something outside of Rust's normal memory management. In most cases you'll be using a higher-level API like Box::into_raw(), but if you look at the implementation of that you'll see it uses mem::forget() internally.

orf on Feb 7, 2019 [-]

How would that work at the operating system level, which is where page in and outs occur? You might be able to tell some OS's not to page out a specific page, but with variable length data that becomes impossible

charleslmunger on Feb 7, 2019 [-]

http://man7.org/linux/man-pages/man2/mlock.2.html at least for linux.

monocasa on Feb 7, 2019 [-]

mlock, like VirtualLock, doesn't guarantee that a copy won't go out to disk, only that a copy will always be in RAM.

charleslmunger on Feb 10, 2019 [-]

The notes section of the page I linked addresses this - it'll only be paged in a hibernate state where memory is powered down. It won't be paged without that.

monocasa on Feb 11, 2019 [-]

That's just one of many examples.

For instance if you're running in a VM, all bets are off, beacuse the host can and will page you out guest phys memory.

loeg on Feb 7, 2019 [-]

Have a separate allocation pool for such objects that uses the mlock syscall on memory ranges it allocates from the OS.

As you note, you can't just mlock memory handed out by the ordinary allocator, as you never know when it will be free to munlock.

pornel on Feb 8, 2019 [-]

There is no such method in the standard library. You'd get a pointer to the underlying memory (there's `Vec.as_ptr()` for example), and make whatever (C) system calls you want.

SilasX on Feb 7, 2019 [-]

Sorry if too off-topic, but I figured this was a good place to share my experience with trying to use the Rust `crypto` library, which I recounted in this comment [1]. It's not a security concern, but it does affect usability.

I had the simple task[2] of encrypting bytes with AES using the library. But (as detailed in the post), they didn't expose a simple, "give us plaintext bytes, give us key bytes, give us options, we'll spit back the ciphertext bytes". Instead, you have to delve into the irrelevant details of constructing mutable buffers (which require special care in Rust), so it can read from and write to them, and without even any good examples in the docs for how to construct those buffers.

It seems designed from the perspective of "I have extremely limited memory" -- which is fine, but why not have a simple wrapper on top of that for the common, non-embedded use case, and some example code?

[1] https://news.ycombinator.com/item?id=18943940

[2] for the Matasano/cryptopals challenge. Interestingly, I found that the hardest parts weren't figuring out the vulnerabilities, but in implementing a spec (every writeup was poor) or getting "off-the-shelf" software to behave as expected so I can have a utility function for it and forget about it from then on!

cetra3 on Feb 7, 2019 [-]

Are you talking about the crate `crypto` or `rust-crypto`? Both have not been updated in a couple of years.

The `ring` library is a bit more modern, but still requires a bit of a run around. Here's an example of encrypting a cookie: https://github.com/alexcrichton/cookie-rs/blob/master/src/se...

SilasX on Feb 7, 2019 [-]

Why does everyone leap to suggesting that 2016 was some kind of dark age of Rust when no one could be expected to do anything right? You're not the first [1].

I'm reminded of the Onion article about "Stating current year still the leading argument for reform" [2] (e.g. "It's 2019, people!")

I guess the corollary is, "That was 3 years ago" still leading excuse for shoddy work :-p

[1] https://news.ycombinator.com/item?id=18943056

[2] https://www.theonion.com/report-stating-current-year-still-l...

cetra3 on Feb 8, 2019 [-]

For a security library to have no updates in a couple of years, I would consider it to be unmaintained and a huge risk.

I don't doubt that 2016 had some great libraries in rust, of which a lot are still used today. But most of them are updated to use new language features like `?` instead of `try!`, new macros system, async libraries, etc..

Take a look at a library like `serde`, which has been around since 2015 (https://crates.io/crates/serde/versions) and is awesome.

fpgaminer on Feb 7, 2019 [-]

I don't believe that's what the comment you're replying to stated. The fact that `rust-crypto`, for example, has not been updated since 2016 doesn't imply that the code was necessarily bad in 2016, but that the project itself may not be maintained anymore. Static, perfect code is rare.

SilasX on Feb 7, 2019 [-]

I know they didn't state it. I said they *suggested* it. Which they, and the linked comment, did.

When your immediate reply is to dismiss the dominant rust crypto library in 2016 as being understandably bad because the development ended in 2016, you're *suggesting* that the year (and before) was some kind of dark age for rust.

>Static, perfect code is rare.

My criticism isn't "hey, there's a lingering bug that no one fixed and which broke my use case". My criticism is, "The design of the *exposed interface to the core library functionality* forces me to care about irrelevant implementation details and enable attributes (mutability) that my use case doesn't need."

When you say that a project developed in 2016 can be expected to fail on that front, what you're saying is, 2016 was too early to expect developers of a major project (in a hot language followed by a lot of smart people) to understand the concept of separation of concerns and abstraction of irrelevant details.

Do you see why I might be skeptical of the relevance of the year of development?

There's a similar point to be made about (the lack of) clear examples, a concept that didn't spring into existence in 2017.

kaoD on Feb 8, 2019 [-]

> When your immediate reply is to dismiss the dominant rust crypto library in 2016 as being understandably bad because the development ended in 2016, you're suggesting that the year (and before) was some kind of dark age for rust.

I was there and it actually was.

Rust 1.0 was just released and the ecosystem was mostly maturing at that point.

You're talking about version 0.2.36 of a library that had been in development for less than two years during a quite tempestuous time in Rust. I'm sure they were more concerned with making the code work, making it secure, etc.

s_tec on Feb 7, 2019 [-]

When somebody says "have not been updated in a couple of years," they are really just saying that the code is unmaintained. Unmaintained code is bad code, no matter what year it is from.

SilasX on Feb 8, 2019 [-]

Unmaintained code can explain edge-case bugs.

Lack of clear examples is not an edge case bug, it's failing to think about your user.

Breaking separation of concerns is not an edge-case bug.

I'm old, so I can assure you that, as of 2016, developers were well aware of the concepts of separation of concerns and abstraction. It being 2016 does not address the criticism I made. See also my longer reply.

https://news.ycombinator.com/item?id=19109660

oconnor663 on Feb 8, 2019 [-]

Rust 1.0 was in May 2015. No one's saying 2016 was a dark age, but the language has been moving quite fast since then. Even fundamental concepts like error handling have seen a major generation or two go by in that time (e.g. error-chain and failure). Many libraries from 2016 are still going strong, but a Rust library that hasn't gotten a commit in a couple years is much deader than the C++ equivalent.

jandrese on Feb 7, 2019 [-]

> I had the simple task[2] of encrypting bytes with AES using the library. But (as detailed in the post), they didn't expose a simple, "give us plaintext bytes, give us key bytes, give us options, we'll spit back the ciphertext bytes". Instead, you have to delve into the irrelevant details of constructing mutable buffers (which require special care in Rust), so it can read from and write to them, and without even any good examples in the docs for how to construct those buffers.

That is a surprisingly uncommon API for a crypto library. It's far more common that they make you do a lot of the setup that really should be under the hood and that if you get wrong your security will be ruined. Plus the docs won't explain what any of the values mean because obviously you wouldn't be using crypto unless you already took a college course in it.

It's far too rare that the docs even explain what an IV is, or what you should set it to, or if it needs to be kept secret, or even what the initials mean. I think it is on purpose to scare off developers that have not studied the field extensively, but in reality they have a job to do so they set it to 0 and then ship 84 million appliances where this code is baked into the firmware update system.

SilasX on Feb 7, 2019 [-]

I hear you. But, in fairness, tptacek mentioned a Go library in that thread that worked the way I was expecting, and provided good examples:

https://golang.org/pkg/crypto/cipher/#NewGCM

steveklabnik on Feb 8, 2019 [-]

We also had a discussion about it here https://news.ycombinator.com/item?id=18943056, if you recall.

SilasX on Feb 8, 2019 [-]

Yes, which is why I linked it in the thread, and your comment specifically. Did you see this comment? https://news.ycombinator.com/item?id=19109418

Why do you think rust developers weren't aware of separation of concerns or good examples in 2016?

tptacek on Feb 7, 2019 [-]

Which writeups were "poor"?

SilasX on Feb 7, 2019 [-]

Not the Cryptopal writeups -- I mean every explanation of the Mersenne Twister I found, how the values were scattered all over the place and it was difficult to get the expected result. Wikipedia's, example, is a mess to sort through:

https://en.wikipedia.org/wiki/Mersenne_Twister#Pseudocode

(I use the plural because I'm lumping that in with the difficulty of getting openssl to yield the expected results for AES encryption.)

richardwhiuk on Feb 7, 2019 [-]

> Review the dependencies listed in Cargo.toml: Will the latest version be used? (preferable but not always the right choice) Are these established, trustworthy packages?

The author seems to have forgotten about Cargo.lock....

lilyball on Feb 7, 2019 [-]

Cargo.lock is something the top-level program uses. If you're writing a library that does crypto, your Cargo.lock will be ignored by any upstream libraries/applications that depend on you (and because of this the recommendation is to not even check your library's Cargo.lock into version control).

richardwhiuk 12 months ago [-]

Ah right - I thought this was regarding security tool behaviour not an app - I'm aware Cargo.lock doesn't provide locking at an app level.

I consider a library insisting on locking it's second level dependencies for no good reason to be obnoxious behaviour. As producer of an app, I want the ability to upgrade your dependencies to patch security problems.

In my experience, excessive locking can make security problems worse not better.

Animats on Feb 7, 2019 [-]

If you have unsafe Rust code in your crypto code, you're doing it wrong.

oconnor663 on Feb 8, 2019 [-]

I'm not aware of any SIMD frameworks that are advanced enough to spare me from writing unsafe code yet, particularly if I want to ship multiple implementations and detect support dynamically.

Also I'm not sure you can get any constant time guarantees today without calling into C or asm.

adwn on Feb 8, 2019 [-]

Even *unsafe* Rust code is a bit safer than C. For example, in Rust unsafe code, using an uninitialized variable leads to a compiler error – you have to use something explicit like std::mem::uninitialized.

Applications are open for YC Summer 2020

Guidelines | FAQ | Support | API | Security | Lists | Bookmarklet | Legal | Apply to YC | Contact

Search: