Security Now! #906 - 01-17-23

The Rule of Two

This week on Security Now!

This week we're back to answering some questions that you didn't even know were burning. First, is the LastPass iteration count problem much less severe than we thought because they are doing additional PBKDF2 rounds at their end? What sort of breach has Norton LifeLock protected its user's from — and have they really? What did Chrome just do which followed Microsoft and Firefox? And... is the Chromium beginning to Rust? Will Microsoft ever actually protect us from exploitation by old known vulnerable kernel drivers? What does it mean that real words almost never appear in random character strings? And what is Google's "Rule of Two" and why does our entire future depend upon it? The answers to those questions and more will be revealed during this next gripping episode of Security Now!

There appears to be a theme here…

Security News

A brief glimmer of hope

Since all LastPass users could use a bit of good news about now, I became excited when it appeared that things were not as bad as we thought last week. As we know, many users discovered that LastPass had never increased their client's local PBKDF2 iteration count from its earlier setting of 5000, or maybe 500, or in many cases as few as 1 iteration, thus resulting in trivial-to-decrypt LastPass backups.

One of our listeners with a sharp memory, Parker Stacey, wrote with an important quote from a page on LastPass' website. The page Parker quoted is titled "About Password Iterations" and most of this we already know. But there's an important line that was new to me. So in order for that important line to be understood in context, which is somewhat unclear, I'm going to share the short piece in its entirety:

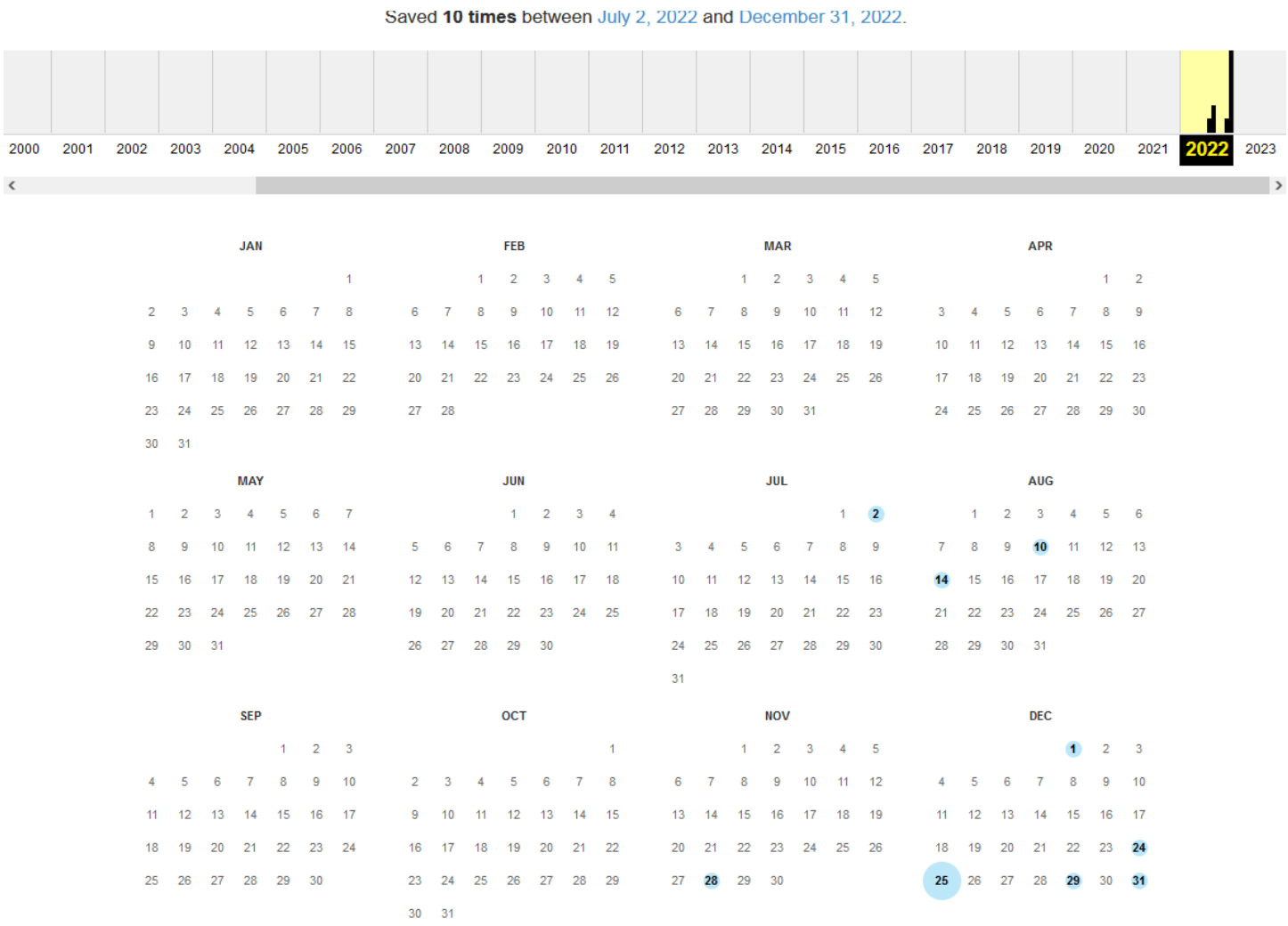
> *To increase the security of your master password, LastPass utilizes a stronger-than-typical version of Password-Based Key Derivation Function (PBKDF2). At its most basic, PBKDF2 is a "password-strengthening algorithm" that makes it difficult for a computer to check that any 1 password is the correct master password during a compromising attack.*
>
> *LastPass utilizes the PBKDF2 function implemented with SHA-256 to turn your master password into your encryption key. LastPass performs a customizable number of rounds of the function to create the encryption key, before a single additional round of PBKDF2 is done to create your login hash.*
>
> *The entire process is conducted client-side. The resulting login hash is what is communicated with LastPass. LastPass uses the hash to verify that you are entering the correct master password when logging in to your account.*
>
> *LastPass also performs a large number of rounds of PBKDF2 server-side. This implementation of PBKDF2 client-side and server-side ensures that the two pieces of your data – the part that's stored offline locally and the part that's stored online on LastPass servers – are thoroughly protected.*
>
> *By default, the number of password iterations that LastPass uses is 100,100 rounds.*
>
> *LastPass allows you to customize the number of rounds performed during the client-side encryption process in your Account Settings.*

https://support.lastpass.com/help/about-password-iterations-lp030027

So the key here is the news that this page at LastPass says that they are performing *"a large number of rounds of PBKDF2 server-side."*

Since this was the first I had heard of this – it wasn't anything that I recalled Joe Siegrist ever mentioning to me – and since it perfectly responded to the worries we talked about last week, and since LastPass has now been revealed to be, if nothing else, a bit klutzy, I was just a bit suspicious about exactly **when** this convenient page first appeared on LastPass' website. So I dipped into the Internet Archive's Wayback Machine to see when it had first indexed this page.

And what I found did little to assuage my suspicion:



The Internet Archive first indexed this page about a month and a half before the first August announcement of the breach. So while it did precede public disclosure of the trouble, it's not as if this page had been around for years. I shared my curiosity over this with Parker with the intention to do some more digging before today's podcast. But Parker's curiosity was also piqued and he tracked down a LastPass PDF document titled "LastPass Technical Whitepaper" which the Internet Archive had indexed and stored on December 18th of 2019. So about four years ago. And, looking for the same titled paper today it pops right up. None of the PDFs contains any edition dates, but both papers contain exactly the same paragraph which is more clear than the watered-down help page. It reads:

> *LastPass also performs 100,100 rounds of PBKDF2 server-side. This implementation of PBKDF2 client-side and server-side ensures that the two pieces of the user's data – the part that's stored offline locally and the part that's stored online on LastPass servers – are thoroughly protected.*

https://support.lastpass.com/download/lastpass-technical-whitepaper

I was curious about anything additional that the PDF's metadata might reveal. And I learned that the PDF was created using Microsoft Word 365 by Erin Styles on June 6th of 2022. As a quick sanity check, I noted that Erin's Twitter photo shows her proudly sporting a bright red LastPass sweatshirt.

So what would the flow of this system look like and what does it mean?

To their client, the user provides their eMail address which serves as salt, their LastPass master password, and their account's local client iteration count to an SHA-256 based PBKDF2 function. That produces a 256-bit encryption/decryption key that's never sent to LastPass. It is only ever used by the local client to encrypt and decrypt the master vault blob which LastPass stores for them and shares among the user's LastPass clients.

So how, then, does LastPass verify that they've logged on using their proper credentials? After running the local PBKDF2 iterations, one additional round of PBKDF2 is used to produce a 256-bit user logon verification token which LastPass calls the "Login Hash." That token **is** sent to LastPass to store and use to verify the user's proper logon credentials. LastPass uses the user's ability to provide that token to log them into their online vault and before sending their encrypted vault blob.

The problem that occurred to me was that only a single round of PBKDF2 was separating the user's secret vault decryption key from this "Login Hash" which the client provides to LastPass as its authentication. That meant that it could easily be brute-force reversed. But this apparently also occurred to LastPass. Although there's language about this in several places, the clearest description appears near the bottom of page 19 under the title of "Login Hash Storage":

> *LastPass receives the Login Hash from the user (following the default 100,100 iterations on the user's Master Password using PBKDF2-SHA256), the Login Hash is additionally salted with a random 256-bit salt, and an additional 100,000 rounds of PBKDF2-SHA256 are performed. That output is then hashed using scrypt to increase the memory requirements of brute-force attacks. The resulting hash stored by LastPass is the output of 200,101 rounds of SHA256 + script.*

I needed to pursue this all the way out to the end to understand whether, as I was hoping, LastPass might have used all of this additional much ballyhooed server-side PBKDF2'ing – even going so far as to deploy scrypt – to create a super-strength encryption key for use when saving their users' vault data. But there is no sign of that in their technical whitepaper.

What became clear was that they recognized the danger of only having that single round of PBKDF2 – essentially a single SHA-256 hash – separating their long-term storage of the user's Login Hash from the user's secret decryption key. If bad guys were ever to obtain those "Login Hashes" they would only need to reverse one SHA-256 hash to obtain the user's secret vault key. So, all of this extra server-side PBKDF2 they're talking about does **nothing** to more thoroughly protect the user's vault key beyond the iterations of the user's client. It only serves to much more strongly isolate the stored "Login Hash" from the user's secret vault key.

My first hopeful reading of LastPass' explanation had me hoping that they were taking that user's provided "Login Hash" and using it to strongly derive another encryption key, and then using that to re-encrypt the data provided by the user's client before storing it. That would have dramatically reduced the security impact of the many very low client-side iteration counts we know that many of our listeners discovered. But it's now clear that's not what they're doing. Under the heading of "User Data Storage" that document states: *"Sensitive vault data is encrypted client-side, then received and stored by LastPass as encrypted data."* There is no indication of or mention of any re-encryption, in fact, quite the opposite. And now we know that the server-side PBKDF2'ing is used to isolate the "Login Hash" from the user's key but not to better encrypt their user's data.

ECB or CBC?

The other question to be answered this week, thanks to the past week's worth of listener feedback from all of the people who used Rob Woodruff's PowerShell script to peer into their vaults was about the prevalence of the less desirable ECB encryption. One of our listeners provided a unique insight which simultaneously answered a question I had:

Mark Jones / @mjphd

> *You requested updates on LastPass. Regret as a loyal listener that I was stuck at 35000 iterations, not 100100. I found 28 ECB items, mostly secure notes. I now certainly regret putting so much in notes. A couple of comments. Incrementing iterations changed all to CBC. It appears you are correct in asserting any changes get rid of ECB.*

This explained the mystery of why my own vault had no instances of ECB. I know that my vault contained a number of very old account credentials from the beginning of my use of LastPass which would have never been updated and which therefore I expected to find still encrypted in the original ECB mode. But what Mark observed was that simply changing his vault's iteration count, as many of us did five years ago when we were covering this issue on the podcast, gave our client's the opportunity to re-encrypt all of those older ECB mode entries in CBC mode.

So another mystery solved and likely the end of our LastPass discussions until and unless something else happens.

Norton LifeLock Troubles

While we're on the topic of password manager troubles, we should touch on Norton LifeLock and more double-speak being produced by corporations that have grown too big to need to care. Before we get into this, we should note that what was formerly Symantec Corporation and NortonLifeLock are now renamed "Gen Digital" and are living at www.gendigital.com. Since Symantec had been acquiring companies over the past few years this new Gen Digital is now operating the brands Norton, Avast, LifeLock, Avira, AVG, ReputationDefender, and CCleaner.

Okay, what happened? Around 6,450 Norton LifeLock customers were recently notified that their LifeLock accounts had been "compromised" and by "compromised" we mean that unknown malicious parties have somehow arranged to log into those 6,450 accounts giving them full access to their users' password managers' stored data.

In a notice to customers, Gen Digital, the recently renamed parent of this collection of companies, said that the likely culprit was a credential stuffing attack as opposed to a compromise of its systems. This seems very odd since Gen Digital explained that by this they meant that previously exposed or breached credentials were used to break into accounts on different sites and services that share the same passwords. But that wouldn't explain why there was apparently what appears to be a quite successful targeted attack against 6,450 of their users. If these were username and password credentials leaked and/or somehow obtained from other unrelated site breaches, how is it that they just happened to be useful against 6,450 of LifeLock's account holders? Something doesn't smell right.

What I suspect actually happened is that LifeLock's web portal is, or was, lacking in brute-force password guessing protection. It's not okay for a website to allow a fleet of bots to pound away on its login page at high speed hoping to get lucky. BleepingComputer also covered this news and they posted a statement from a Gen Digital spokesperson saying: *"We have secured 925,000 inactive and active accounts that may have been targeted by credential-stuffing attacks."*

925,000 accounts. "We have secured them..." Does anyone know what that means? What does it mean "We have secured 925,000 inactive and active accounts that may have been targeted by credential-stuffing attacks." You "secured them." Well that's good. It's much better than not securing them, I suppose. Since we're not actually offered any information, just corporate PR speak, it's necessary to read between the lines. So my guess would be that Gen Digital currently has 925,000 Norton LifeLock customers, and due to completely absent web portal security and a lack of monitoring, for some length of time, once bad guys realized that there was no security to stop them, a fleet of bots was programmed to assault Norton LifeLock's login page guessing account credentials, at high speed, without limit. And, as a result of this lack of security, that fleet of bots was able to successfully login and compromise the accounts of 6,450 Norton LifeLock users.

Gen Digital did admit that it had discovered that these intruders had compromised LifeLock accounts beginning on December 1st, eleven days before its systems finally detected a "large volume" of failed logins to customer accounts. The red flag finally went up on December 12th when LifeLock became aware of it and presumably brought this attack to a halt. So I suppose that's what they meant when they said that they had "secured" those 925,000 active and inactive accounts. They basically halted an ongoing login attack after 6,450 successful logins and full account compromises of their customers.

Gen Digital is not saying how much earlier the attack was underway. So why 11 days from first successful compromise to first detection of an attack that had been ongoing for some time? If I had to guess I'd suggest that the bot fleet's attack was probably carefully throttled so as to not trip any alarms. And that after some successful undetected logins the fleet's operators may have started creeping its attack rate upward slowly to see how much faster they could go. Remember that before they were shut down they'd successfully scored against 6,450 accounts. So things had been going well for the attacking fleet.

And we don't even know that they actually were ever detected. We don't know that's what tripped the alarm and raised the red flag. Given that they had been stomping around within

6,450 of LifeLock's user accounts, I'd be surprised if some user didn't notice that something was amiss and contact Norton to report suspicious account activity. So it may have just as easily been that they were tipped off by a victim and that's how the attack was actually discovered.

Note that this highlights another good reason for choosing an iteration count that takes the web browser a few seconds to obtain a go/no-go decision. It also serves as good brute force protection against login attempts to your provider's portal.

Chrome follows Microsoft and Firefox:
Remember the certificate authority TrustCor? Last December we spent some time looking at the bizarre set of corporate shells and shenanigans being played by that shady certificate authority. And at the time it was difficult to understand why anyone would trust those clowns. Remember that one of the corporate users of their certificates was a deep packet forensics entity that was found to be selling their TLS middleboxes to agencies of the US federal government. And there was also that individual and private company whom no one had ever heard of before who had years before received a huge block of previously unused US Department of Defence IPv4 allocation. Something was very "off" about all this.

Finally, in response to a long dialog with a company representative that convinced no one, the reason were we talking about it then was that Microsoft and Mozilla both marked TrustCor's root certs as untrusted, thus immediately removing trust from any certificates signed by them. We're talking about this all again because Google and Chrome have now followed suit by removing TrustCor's CA certs from the Android and Chrome root stores. So, it's pretty much over for that group.

Chromium is beginning to Rust
Google's announcement and blog posting last Thursday was titled *"Supporting the Use of Rust in the Chromium Project"*. They wrote:

> *We are pleased to announce that moving forward, the Chromium project is going to support the use of third-party Rust libraries from C++ in Chromium. To do so, we are now actively pursuing adding a production Rust toolchain to our build system. This will enable us to include Rust code in the Chrome binary within the next year. We're starting slow and setting clear expectations on what libraries we will consider once we're ready.*
>
> *Our goal in bringing Rust into Chromium is to provide a simpler and safer way to satisfy **The Rule of Two**, in order to speed up development (less code to write, less design docs, less security review) and improve the security (increasing the number of lines of code without memory safety bugs, decreasing the bug density of code) of Chrome. And we believe that we can use third-party Rust libraries to work toward this goal.*
>
> *Rust was developed by Mozilla specifically for use in writing a browser, so it's very fitting that Chromium would finally begin to rely on this technology too. Thank you Mozilla for your huge contribution to the systems software industry. Rust has been an incredible proof that we should be able to expect a language to provide safety while also being performant.*

Everyone listening to this podcast has heard me lament that we're never going to get ahead of this beast of software flaws if we don't start doing things differently. What was that definition of insanity? Anyway, it’s great news that the Chromium project is taking this step. It will be a slow and very evolutionary move to have an increasing percentage of the Chromium codebase written in Rust, but this is the way that effort, and this eventuality, gets started.

And you may have noted that Google’s announcement mentioned “The Rule of 2” which we will take a close look at once we catch up with recent important news.

BYOVD and Windows Defender Failures

Another instance of BYOVD — Bring Your Own Vulnerable Driver — was just in the news last week. CrowdStrike documented their observation and interception of a eCrime adversary known variously as Scattered Spider, Roasted 0ktapus and UNC3944, leveraging a combination of credential phishing and social engineering attacks to capture one-time-password (OTP) codes or overwhelm targets using multifactor authentication (MFA) notification fatigue tactics. Once obtaining access, the adversary avoids using unique malware which might trip alarms. Instead, they favor the use of a wide range of legitimate remote management tools to maintain persistent access inside their victim's networks.

CrowdStrike's instrumentation detected a novel attempt by this adversary to deploy a malicious kernel driver through an old and very well known eight year old vulnerability dating from 2015 which exists in the Intel Ethernet diagnostics driver for Windows. The file is “iqvw64.sys.”

The distressing factor is that the technique of using known-vulnerable kernel drivers has been in use by adversaries for several years and has been enabled by a persistent gap in Windows security. Starting with the 64-bit edition of Windows Vista, and ever since, Windows does not allow unsigned kernel-mode drivers to run by default. That was easy to do and it shut down one path of exploitation. But clever attackers started bringing their own drivers with known and readily exploitable vulnerabilities... Such as this Intel Ethernet diagnostics driver.

What to do about that? In 2021, Microsoft stated that “Increasingly, adversaries are leveraging legitimate drivers in the ecosystem and their security vulnerabilities to run malware,” and that “drivers with confirmed security vulnerabilities will be blocked on Windows 10 devices in the ecosystem using Microsoft Defender for Endpoint attack surface reduction (ASR) and Microsoft Windows Defender Application Control (WDAC) technologies to protect devices against exploits involving vulnerable drivers to gain access to the kernel.”

So that was the plan. How did that work out? As we discussed some time ago, multiple security researchers over the past two years have repeatedly and loudly noted — pounding on Microsoft — that this was all apparently just feel-good nonsense being spewed by Microsoft and that the issue continues to persist as Microsoft continually fails to block vulnerable drivers by default.

The crux of the problem appears to be that any such proactive blocking requires pro-action from Microsoft. And any fair weighing of the evidence — many examples of which we've looked at during the past two or three years — would conclude that Microsoft has long since abandoned any commitment to true proactive security. They're no longer even finding most problems.

They increasingly rely upon the good graces of outside security researchers, and even then they drag their feet over implementing the required updates.

That said though, being fair to Microsoft, there is a flip side to this: We know that the last thing Microsoft wants to ever do is deliberately break anything. They have enough trouble with **inadvertently** breaking things, let alone doing so deliberately. So, proactively blacklisting anything, especially something like a network driver that would potentially cut a machine off from its network, is the last thing that Microsoft would choose to do, assuming that a choice was being made in the first place. But it occurs to me that this **is** something that a 3rd party could do.

I'm bringing this back up again because this was not supposed to happen. The last time we spoke about it last year, the presumption was — based upon clear statements made by Microsoft at the time — that this had all been a big oversight mistake for several years and that all was now going to be better. But this latest news is from last week and this has not been fixed.

So there's that. But the bigger point I want to make is that the need to be proactive; the need for Microsoft to be proactively blocking known-vulnerable drivers, seems like something we are never going to be able to get from Microsoft. For the foreseeable future at least, this appears to no longer be in their DNA. This is not who they are. But this, in turn, presents a huge and significant opportunity for a third party security company. Solve this problem. Be proactive in a critical area where Microsoft refuses to be... and make yourselves a lot of money by closing a persistent gaping hole in Windows security.

Closing the Loop

My Twitter feed was so overwhelmed with feedback from last week's call for feedback that I was unable to reply to Tweets as I usually try to. So please accept my thanks for the feedback and my apology that I probably didn't reply. I know that replying is not required, but courtesy and chivalry are not dead here and as many of my regular Twitter correspondents know, I often reply when I can.

One public tweet caught my eye for its cleverness which I wanted to share:

Someone tweeting as @mammalalien

> *@SGgrc sometimes it is hard to picture just how many more random strings there are than English words. So try this: How often have you EVER seen real words coincidentally appear in randomly generated passwords? I noticed it today for (I think) the second time ever.*

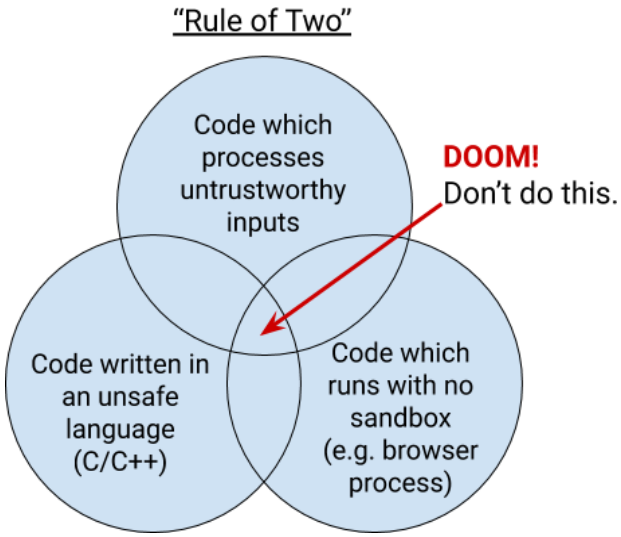
I think that's a very clever and worthwhile observation. It helps us to truly appreciate how much less entropy exists in non-random text, which we recognize as words in our language, such that it almost never appears by chance in truly random strings of characters.

The Rule of Two

In their posting about the adoption of Rust into Chrome, Google mentioned something they called "The Rule of Two." I didn't focus on it there because I planned to come back to it here.

In Google's official Chromium Docs they explain their so-called "Rule of 2" as follows:

> *When you write code to parse, evaluate, or otherwise handle untrustworthy inputs from the Internet — which is almost everything we do in a web browser! — we like to follow a simple rule to make sure it's safe enough to do so. The Rule Of 2 is: Pick no more than any 2 of:*
>
> - *Untrustworthy inputs;*
> - *Unsafe implementation language; and*
> - *High privilege.*



https://chromium.googlesource.com/chromium/src/+/master/docs/security/rule-of-2.md

This, of course, is reminiscent of the great old saying that in a project you can choose any two, but only two, of the following three outcomes: Good, Fast, or Cheap. You cannot get all three. You'll need to sacrifice the thing that's the least important to you from among them. You can have something good and fast, but then it won't be cheap. Or you can have something fast and cheap, but then it won't also be good. Or you can have something good and cheap, but then you won't be able to get it fast.

So for Google's similar "Rule of Two" they have:

- *Untrustworthy inputs;*
- *Unsafe implementation language; and*
- *High privilege.*

Let's let's take a closer look at these. Google explained...

When code that handles untrustworthy inputs at high privilege has bugs, the resulting vulnerabilities are typically of Critical or High severity. We'd love to reduce the severity of such bugs by reducing the amount of damage they can do (lowering their privilege), avoiding the various types of memory corruption bugs (using a safe language), or reducing the likelihood that the input is malicious (asserting the trustworthiness of the source).

For the purposes of this document, our main concern is reducing (and hopefully, ultimately eliminating) bugs that arise due to memory unsafety. A recent study by Matt Miller from Microsoft Security states that "~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues". A trip through Chromium's bug tracker will show many, many vulnerabilities whose root cause is memory unsafety. (As of March 2019, only about 5 of 130 public Critical-severity bugs are ***not*** *obviously due to memory corruption.)*

Security engineers in general, very much including the Chrome Security Team, would like to advance the state of engineering to where memory safety issues are much more rare. Then, we could focus more attention on the application-semantic vulnerabilities. 😊 That would be a big improvement.

So it's clear that the historic use of C and C++ has been the source of a great many past security vulnerabilities despite the coders of those languages doing the very best jobs they can. There is no date on this document, but it feels a few years old since they were citing Matt Miller's well known research, which we've cited before here, from March of 2019. From what was written above it's clearer that they were wishing and hoping for the move that Google announced last week with the formal incorporation of Rust as a 1st-class Chromium implementation language.

So let's examine and flesh out each of these three factors in the context of browser implementation:

First: Untrustworthy Inputs

Untrustworthy inputs are inputs that:

a) have non-trivial grammars; and/or
b) come from untrustworthy sources.

Google explains:

If there were an input type so simple that it were straightforward to write a memory-safe handler for it, we wouldn't need to worry much about where it came from for the purposes of memory safety, because we'd be sure we could handle it. We would still need to treat the input as untrustworthy after parsing, of course.

Unfortunately, it is very rare to find a grammar trivial enough that we can trust ourselves to parse it successfully or fail safely. Therefore, we do need to concern ourselves with the provenance of such inputs.

In other words, the stuff a browser confronts is both typically a highly complex language such as

HTML, CSS and JavaScript have unfortunately all been evolved into, and their source, such as a 3rd-party advertising server serving ads submitted by unknown entities, cannot be trusted. If the languages used to express these things was simple it would be easier to trust that our parsing of the language would protect us. But these are not simple languages. Google said:

> *Any arbitrary peer on the Internet is an untrustworthy source, unless we get some evidence of its trustworthiness (which includes at least a strong assertion of the source's identity). When we can know with certainty that an input is coming from the same source as the application itself (e.g. Google in the case of Chrome, or Mozilla in the case of Firefox), and that the transport is integrity-protected (such as with HTTPS), then it can be acceptable to parse even complex inputs from that source. It's still ideal, where feasible, to reduce our degree of trust in the source — such as by parsing the input in a sandbox.*

Okay. So that was Untrustworthy Inputs. What about the choice of Implementation Language?

Unsafe Implementation Languages

Google explains:

> *Unsafe implementation languages are languages that lack memory safety, including at least C, C++, and assembly language. Memory-safe languages include Go, Rust, Python, Java, JavaScript, Kotlin, and Swift. (Note that the safe subsets of these languages are safe by design, but of course implementation quality is a different story.)*

But what about **Unsafe Code in Safe Languages?** Google has this to say:

> *Some memory-safe languages provide a backdoor to unsafety, such as the unsafe keyword in Rust. This functions as a separate unsafe language subset inside the memory-safe one.*
>
> *The presence of unsafe code does not negate the memory-safety properties of the memory-safe language around it as a whole, but how unsafe code is used is critical. Poor use of an unsafe language subset is not meaningfully different from any other unsafe implementation language.*
>
> *In order for a library with unsafe code to be safe for the purposes of the Rule of 2, all unsafe usage must be able to be reviewed and verified by humans with simple local reasoning. To achieve this, we expect all unsafe usage to be:*
>
> - *Small: The minimal possible amount of code to perform the required task*
> - *Encapsulated: All access to the unsafe code is through a safe API*
> - *Documented: All preconditions of an unsafe block (e.g. a call to an unsafe function) are spelled out in comments, along with explanations of how they are satisfied.*

In other words, where a safe language such as Rust provides facilities for breaking out of safety in order to address some need, that region of unsafety must be small, contained, completely understood and well documented. Google continues...

Because unsafe code reaches outside the normal expectations of a memory-safe language, it must follow strict rules to avoid undefined behavior and memory-safety violations, and these are not always easy to verify. A careful review by one or more experts in the unsafe language subset is required.

It should be safe to use any code in a memory-safe language in a high-privilege context. As such, the requirements on a memory-safe language implementation are higher: All code in a memory-safe language must be capable of satisfying the Rule of 2 in a high-privilege context (including any unsafe code) in order to be used or admitted anywhere in the project.

Okay, so what about code privilege. That's the 3rd of the three from which we can only have two:

High Privilege

Google explains:

High privilege is a relative term. The very highest-privilege programs are the computer's firmware, the bootloader, the kernel, any hypervisor or virtual machine monitor, and so on. Below that are processes that run as an OS-level account representing a person; this includes the Chrome browser process. We consider such processes to have high privilege. (After all, they can do anything the person can do, with any and all of the person's valuable data and accounts.)

Processes with slightly reduced privilege include (as of March 2019) the GPU process and (hopefully soon) the network process. These are still pretty high-privilege processes. We are always looking for ways to reduce their privilege without breaking them.

Low-privilege processes include sandboxed utility processes and renderer processes with Site Isolation (very good) or origin isolation (which is even better).

Google then talks about two topics that we've discussed through the years as we've observed how difficult they appear to get right: Parsing and Deserializing. Remember that deserializing is essentially an interpretation job and interpretation is notoriously difficult to get correct because it appears to be nearly impossible for the coder of the interpreter, who inherently expects the input to be correct, to adequately handle inputs which are malicious.

Google says:

Turning a stream of bytes into a structured object is hard to do correctly and safely. For example, turning a stream of bytes into a sequence of Unicode code points, and from there into an HTML Document Object Model tree, with all its elements, attributes, and metadata, is very error-prone. The same is true of QUIC packets, video frames, and so on. Whenever the code branches on the byte values it's processing, the risk increases that an attacker can influence control flow and exploit bugs in the implementation.
Although we are all human and mistakes are always possible, a function that does not branch

on input values has a better chance of being free of vulnerabilities. (Consider an arithmetic function, such as SHA-256, for example.)

I thought that was a very interesting observation. We made SHA-256 branch-free so that differing code paths would not leak timing information and leave sniffable hints in our processor's branch prediction history. But a side effect of that also increased the algorithm's robustness against deliberate code path manipulation.

What surprised me a bit is that the Chromium security team is extremely literal about this rule of 2. They're not joking around and they actually apply this rule when evaluating new submissions. And they wrote some advice to those who would submit code to the Chromium project:

Chrome Security Team will generally not approve landing a new feature that involves all 3 of untrustworthy inputs, unsafe language, and high privilege. To solve this problem, you need to get rid of at least 1 of those 3 things. Here are some ways to do that.

Privilege Reduction *(which is obviously one of the three things)*

Also known as sandboxing, privilege reduction means running the code in a process that has had some or many of its privileges revoked. When appropriate, try to handle the inputs in a renderer process that is isolated to the same site as the inputs came from. Take care to validate the parsed (processed) inputs in the browser, since only the browser can trust itself to validate and act on the meaning of an object. Or, you can launch a sandboxed utility process to handle the data, and return a well-formed response back to the caller in an Inter-Process Communication message.

So these ideas are structural means for creating an arms-length client-sever relationship where a low privilege worker process does the unsafe work and simply returns the results to the higher privilege client. That way, if anything goes sideways, there's containment within the process that cannot do much with its malicious freedom.

As for verifying the trustworthiness of a source, they say that if the developer can be sure that the input comes from a trustworthy source – so not overtly attempting to be malicious – it can be okay to parse and evaluate it at high privilege in an unsafe language. (Yikes) In this instance, "a trustworthy source" means that Chromium can cryptographically prove that the data comes from a business entity that can be or is trusted. For example, in the case of Chrome, from an Alphabet company.

Google then talks about ways to make code safer to execute under the title of "Normalization". Writing to would-be Chromium coders, they explain:

You can 'defang' a potentially-malicious input by transforming it into a normal or minimal form, usually by first transforming it into a format with a simpler grammar. We say that all data, file, and wire formats are defined by a grammar, even if that grammar is implicit or only partially-specified (as is so often the case). For example, a data format with a particularly

simple grammar is SkPixmap. This 'grammar' is represented by the private data fields: a region of raw pixel data, the size of that region, and simple metadata which directs how to interpret the pixels. Unfortunately, it's rare to find such a simple grammar for input formats.

For example, consider the PNG image format, which is complex and whose C implementation has suffered from memory corruption bugs in the past. An attacker could craft a malicious PNG to trigger such a bug. But if you first transform the image into a format that doesn't have PNG's complexity – in a low-privilege process of course – the malicious nature of the PNG 'should' be eliminated and then safe for parsing at a higher privilege level. Even if the attacker manages to compromise the low-privilege process with a malicious PNG, the high-privilege process will only parse the compromised process' output with a simple, plausibly-safe parser. If that parse is successful, the higher-privilege process can then optionally further transform it into a normalized, minimal form (such as to save space). Otherwise, the parse can fail safely, without memory corruption.

For example, it should be safe enough to convert a PNG to an SkBitmap in a sandboxed process, and then send the SkBitmap to a higher-privileged process via IPC. Although there may be bugs in the IPC message deserialization code and/or in Skia's SkBitmap handling code, we consider this safe enough.

I think that the interesting message here for those of us who are not writing code for a browser is, first of all, to be thankful we're not, and secondly to more deeply appreciate just how truly hostile this territory is. It's a true battlefield. On this podcast I've often noted that the browser is that part of our systems that we blindly thrust out into the world and hope that it doesn't return encrusted with any plagues. When this environment is coupled with the insane complexity of today's browsers it's truly a miracle that they protect us as well as they do.

And all of this is up against the crushing backdrop of the imperative for performance. Google notes that: *"We have to accept the risk of memory safety bugs in deserialization because C++'s high performance is crucial in such a throughput- and latency-sensitive area. If we could change this code to be both in a safer language and still have such high performance, that'd be ideal. But that's unlikely to happen soon."* This was written several years ago. Rust was noted earlier, so it was at least on the radar. I wonder whether that might be the right compromise.

But for now, it's clear that the reason TaskManger shows us 30 processes spawned when Chrome or Edge launch is for containment. Low privilege processes are created and are given the more dangerous and time-critical performance-critical tasks to perform. They cannot be written in a safe but slow language. They need C++. So they are held at arm's length with their inter-process communications strictly limited to receiving a task to perform and returning the result of that work.

These notes for Chromium developers talks a bit more (wistfully) about the idea of using safe languages. Google writes:

Where possible, it's great to use a memory-safe language. Of the currently approved set of implementation languages in Chromium, the most likely candidates are Java (on Android only), Swift (on iOS only), and JavaScript or WebAssembly (although we don't currently use them in high-privilege processes like the browser) — meaning that JavaScript and WebAssembly are

currently only being used in browser web pages and extensions.

One can imagine Kotlin on Android, too, although it is not currently used in Chromium. (Some of us on the Security Team aspire to get more of Chromium in safer languages, and you may be able to help with our experiments.)

For an example of image processing, we have the pure-Java class BaseGifImage. On Android, where we can use Java and also face a particularly high cost for creating new processes (necessary for sandboxing), using Java to decode tricky formats can be a great approach. We do a similar thing with the pure-Java JsonSanitizer, to 'vet' incoming JSON in a memory-safe way before passing the input to the C++ JSON implementation.

So, no mention of Rust's adoption at this point several years ago, but we know that's changing.

This interesting discussion and guidance for would-be Chromium developers concludes by noting that all of this is aspirational and that, unfortunately, it doesn't reflect Chromium's current state. Under the final heading of "Existing Code That Violates The Rule" they write:

We still have code that violates this rule. For example, Chrome's Omnibox (the single URL and search box at the top of the UI) still parses JSON in the browser process. Additionally, the networking process on Windows is (at present) unsandboxed by default, though there is ongoing work to change that default.

So...

We're seeing an evolution across our industry. Web browsers have become so capable that they're now able to host full applications. That has enabled the relocation of those applications from our local desktop to the cloud, and a redefinition of the customer from an owner to a tenant. I, for one, hate this change. But those of us who feel this way are dying off and we're clearly irrelevant. But neither are my hands completely clean, since I'm composing these show notes in Google Docs, a stunning example of how well this new system can work.

But the gating requirement for any of this to work, for any of this future to unfold, is for our web browsers to survive on the front line of an astonishingly hostile Internet. No one who has been following this podcast for the past few years could have any doubt of the open hostility that today's web browsers face every time they suck down another page loaded with unknown code of unknown provenance containing pointers to other pages of code with the need to go get, load and run that code too.

I'm very glad that Google's Security Team is thinking about the problems they're facing, that they take mitigations such as this "Rule of Two" as seriously as they do, and that they are finally beginning to migrate to the use of safer languages, which, I'll note, was made possible by Mozilla' pioneering of Rust.