Security Now! #875 - 06-14-22
The PACMAN Attack

This week on Security Now!

This week will, I expect, be the last time we talk about passkeys for awhile. But out listeners are still buzzing about it, and some widespread confusion about what Apple presented during their WWDC developer's session needs a bit of clarification. While doing that, I realized and will share how to best characterize what FIDO is, which we're going to get, with respect to SQRL, which we're not. I also want to turn our listeners onto a free streaming penetration testing security course which begins Wednesday after next. Then we have a TON of listener feedback which I've wrapped in additional news. And one listener's question, in particular, was so intriguing that I'm going to repeat it but not answer it yet, so that all of our listeners can have a week to contemplate its correct answer. And although I wasn't looking for it, I also stumbled upon a surprising demonstration proof that we are, indeed, living in a simulation. When I share it, I think you'll be as convinced as I am. And finally, as suggested by this podcast's title, we're going to take a very deep dive into the past week's headline-capturing news that Apple's famous M1 ARM chips all contain a critical bug that cannot be fixed. Just how bad is it?

Security News

Apple’s Passkeys presentation at WWDC 2022: https://grc.sc/875
(https://developer.apple.com/videos/play/wwdc2022/10092/)

I heard from many different listeners that during the WWDC 2022 developer presentation on Passkeys, Apple talked about synchronizing keys. So I listened carefully to the entire presentation. For anyone who's interested, the 34-minute presentation video is this week's GRC shortcut of the week. So it's: https://grc.sc/875 I believe that what these people thought they heard, was Apple addressing the need for the type of synchronization we talked about last week — synching across non-Apple ecosystems such as Android and Windows. But I found no mention of that anywhere in the presentation, nor is it anywhere in the developer docs, which I've also linked-to in the show notes:

https://developer.apple.com/documentation/authenticationservices/public-private_key_authentication/supporting_passkeys

The types of passkey sharing Apple supports is, first and foremost, using the iCloud Keychain to dynamically synchronize keys across an individual user's Apple ecosystem. So, as we know, all Apple devices will remain synchronized. The other form of sharing Apple described uses AirDrop to pass logon credentials to another user. And AirDrop can also be used to permanently pass a passkey to someone else for some site or service, permanently adding **it** to their Keychain to use from then on. But so far, from everything I've seen, Apple has in **no** way suggested that they will ever be synchronizing passkeys with external non-Apple platforms. Nothing has been said, so far, either way.

But Apple’s example solution, of using Airdrop to send a passkey to another person’s iDevice for their subsequent use, highlighted something that’s important to understand: SQRL is a complete solution for secure remote login, whereas FIDO2, with its passkeys, is a replacement for usernames and passwords. The two are not the same. For example, take the case of giving someone else access to a site. If you give them your passkey, which is Apple's solution, then they are now “you” on that site in every meaningful way. When **they** authenticate, it’s **you** authenticating because they’re using your passkey. It's the exact equivalent of you giving them your username and password. And since they are you, they can see your settings, your private details, everything that you can see and do when you login using that same passkey. And since they are you, they’re presumably also able to change the passkey to lock you out. And they can presumably pass it along to others, unless Apple has realized that secondary passkey sharing should be blocked, which would be possible. I don’t know either way, but I sure hope they considered that. In any event, when you voluntarily give your passkey to someone else, your site access has now escaped from your control.

We solved this with SQRL. If you want to allow someone to share some access to an account as a guest — for example, sharing a Netflix account — you obtain a one-time invitation token from the site, and provide it to them. When they attempt to login to the site with **their** own SQRL ID, the site doesn't know them, so it prompts them to either create a new account or to use an outstanding invitation if they have one. Their use of the invitation identifies them to the site as a guest of yours, enabling them to subsequently login to **your** account using **their** SQRL ID. Since they are using their SQRL ID, and guests are unable to request invitations for others, you, as the account owner, retain control. And, you're able to rescind their guest access status at any time. And this all scales seamlessly to enterprise use when hundreds of users might need to share access to common resources. It's called "Managed Shared Access", and it's part of the SQRL solution. It's already there. We have an online demo with the entire solution working and its operation is fully worked out and specified. And there's so much more.

As it stands, the FIDO2 passkeys system is certainly more secure than usernames & passwords. No doubt about it. It's definitely superior. But the FIDO designers were crypto people working to solve one small part of the much larger problem of practical real-world user authentication. They didn't think the whole problem through because that was never their charter. They could credibly say that wasn't their job. It wasn't. But even in a FIDO2 passkeys world, that job still needs to be done. SQRL does it, but unfortunately, FIDO and passkeys does not do it.

Unfortunately, this means that instead of being as revolutionary as it could have been, we get another half-baked solution. It's better than what came before, but it missed the opportunity which comes along so rarely, to address the practical needs of, and really solve, the network authentication problem. Rather than the true breakthrough that SQRL's adoption would have meant, we're going to get incremental progress. It's definitely progress. But because it wasn't really thought through as an entire solution — FIDO is basically a crypto hack — it also brings a whole new set of problems. If FIDO is to be our solution, we really do need some form of centralized passkey storage and synchronization, not only within a vendor, but also across vendors.

Last week, someone calling themselves Captain Jack Xno (@minderwiesen) mentioned @SGgrc in a tweet to someone else, so it appeared in my Twitter feed. Captain Jack wrote:

You may be excited about #passkeys, but #SQRL was carefully developed over 7 years by @SGgrc and solves problems you may not even realize you'd have (potentially cross-platform portability)

And yeah, as we know, SQRL does that, but it does so much more, also.

𝔇𝔯. 𝔑𝔞𝔱𝔥𝔞𝔫 𝔓. 𝔊𝔦𝔟𝔰𝔬𝔫 / @drnathanpgibson

Hi Steve, loved your detailed breakdown of Passkey. You mentioned waiting for password managers to start providing sync services for these FIDO2 private keys. I see that LastPass seems to be promising something "later this year." blog.lastpass.com/2022/06/passwo… Do you know anything more about when this syncing might be coming to a password manager near me?

So, I saw LastPass' blog posting last Monday the 6th. It was a bit confusing because they were also promoting the use of no more passwords immediately, apparently by the use of their own LastPass authenticator on any handset with biometrics. Thus, you could unlock your LastPass vault without using a master password.

Separate from that immediate announcement was, indeed, a forward looking statement of their intention to support FIDO2 passkeys. So that's not today, but at least one major password manager is taking aim at this problem. And if one does, they'll all need to. So I suspect that the biggest effect of Apple's, Google's and Microsoft's support may be to induce websites to bring up their own support for WebAuthn.

WebAuthn
Let's talk a bit about WebAuthn. The heavy lift that FIDO **will** face and the SQRL would have faced, was the need for backend web server support. As we know, it's surprising how slowly things change. No question about it, it's going to take years. It's going to take all of the major web server platforms to build it in, then for all of the existing web servers to be upgraded to support it. And since it's not clear that there's a huge benefit to them, since things are sort of working as-is, it's going to take a while. Think about how long it took for the "Logon with Facebook" and "Logon with Google" Oauth kludge to finally become popular. And it's still far from universal. It's around, you encounter it, but you certainly can't use it everywhere.

What you **can** use everywhere is the original fill-out-the-form username and password.

The reason password managers were an overnight hit was that they provided an entirely client-side solution which did not require any backend change to web sites. Web servers didn't know or care how a user was arranging to provide their username and password. And they didn't need to. But make no mistake, automated form-fill of username and password is and has always been a horrific kludge. The fact that kludges mostly work doesn't make them any less kludgy.

WebAuthn finally, and at long last, changes that. The adoption of WebAuthn, which was approved and formalized by the W3C consortium three years ago, back in 2019, represents a massive and long needed update to username and password authentication. As I mentioned last week, FIDO and SQRL work essentially the same: They both give a web server a public key. The web server generates a random nonce which it sends to the browser. The browser, holding the matching private key, signs and returns that nonce to the web server, and the web server uses the public key it has on file to verify the returned signature. What WebAuthn is and does is provide all of the mechanics, definitions, protocols, specifications and implementations of that new form of interchange between a web server and a web client.

We're likely going to be facing a chicken and egg situation for some time. It will be good that Apple and Google and Microsoft will all be supporting passkeys. But when iOS 16 arrives, with its built-in passkey support, you'll probably only be able to login to Apple.com, Google.com, and maybe Microsoft.com, due to the heavy lift of change that will be required on the back end.

But WebAuthn is the key. It provides a complete replacement for the insecure mess of usernames and passwords. And, interestingly, WebAuthn optionally supports SQRL's chosen 25519 elliptic curve, with its special properties that allow for non-random deterministic private key synthesis. So it might be possible, someday in the future, to transparently run a modified SQRL solution to use SQRL-style deterministic passkeys on the server infrastructure that FIDO built.

Perhaps, indeed, the only kind of progress we **can** practically make... is incremental.

FREE Penetration Testing course with Kali Linux
Last Wednesday, Offensive Security, the people behind Kali Linux, announced that they would be live-streaming their ‘Penetration Testing with Kali Linux (PEN-200/PWK)’ course sessions on Twitch later this month. I mention this because a subset of the course's material — all of the streamed content — will be open to the public on Twitch at no charge.

For those who don't know, Kali Linux is a Debian-based Linux distro which is targeted toward facilitating information security tasks such as Penetration Testing, Security Research, Computer Forensics and Reverse Engineering.

The course in question, PEN-200 is a paid course which helps students prepare for the Offensive Security Certified Professional (OSCP) certification exam. Before the pandemic it was only available in person. But during the COVID response live training was suspended and Offensive Security moved their courseware online for remote instruction. So, today a 26-part, 13-week, twice weekly hour long online course will be offered to prepare students for the OSCP certification. And non enrolled viewers are welcome to audit the course on Twitch.

There's an FAQ in the show notes which details everything you'll need to get going. I grabbed two most relevant Q&A's from that page:
https://help.offensive-security.com/hc/en-us/articles/6702904332564

Where and when will it be happening?

The OffSec Live streaming sessions are currently planned to start June 22nd, 2022 **[Wednesday after next]** *at 12 PM - 1 PM Eastern U.S. and run through December 7, 2022 12 PM - 1 PM Eastern U.S.*

The OffSec Live: PEN-200 streaming sessions will consist of twice-weekly, hour-long interactive presentations. We will also be streaming a weekly office hour where we will address any questions students may have regarding the PEN-200 course and prior streaming sessions.

How much will this cost?

The OffSec Live: PEN-200 Twitch streaming sessions will be free and open to the public.

And now Surfshark
Surfshark announced that they will be following ExpressVPN in pulling out of India.

Miscellany

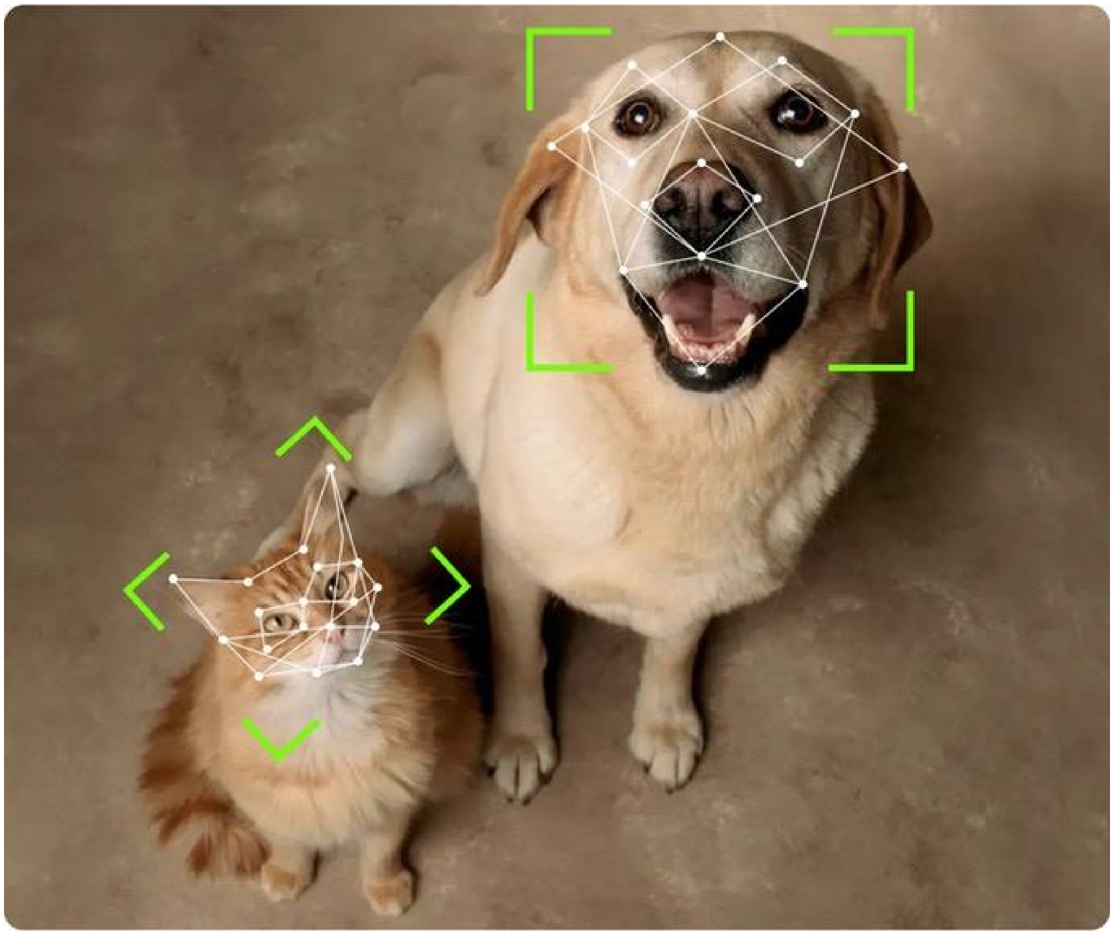
Proof of Simulation:
Leo, I’ve stumbled upon a proof that we are, indeed, living within a simulation. My proof that we are in a simulation is that I have identified a clear and incontrovertible bug that exists within the simulation itself. It’s a bug because it defies reality, which is afterall, what the simulation is intended to simulate. And what makes this so compelling is that **all** of us are aware of this bug, but all just shrug it off without it ever occurring to us that that is what it is. So, here’s the bug:

It’s a clear failure in the rules of probability. If the rules of probability were being properly simulated, when you attempt to plug-in one of those original USB-A style rectangular plugs, the plug’s orientation would be wrong only HALF of the time. But we all know that it is not 50/50! Everyone’s experience is that the first time we attempt to plug-in one of those, it is almost always wrong. It is wrong far more often than 50/50. Think about that. Everyone knows this, and we’re all been blithely ignoring it. It should be 50/50. It must be 50/50. But it’s not.

A valid use for facial recognition: The Smart Pet Door!
https://www.kickstarter.com/projects/petvation/petvation-the-smart-automatic-pet-door-powered-with-ai

Also, the other day I encountered an absolutely valid and very clever application for facial recognition which I doubt that anyone would have a problem with. The bad news is that it’s called “Petvation” — which has just got to be about the worst name that anyone has ever come up with. “Petvation”, my god. I’m sure there must be a wonderful name for such a product.

So what is it? Though you’d never know it from the name, it’s a little automated doggie door, or cat door, which employs facial recognition of the household canine or feline, so that they approach the door their identities are verified and only if they are known to it, will it raise the barrier and allow them to pass. It’s brilliant.



https://www.kickstarter.com/projects/petvation/petvation-the-smart-automatic-pet-door-powered-with-ai

The text which accompanies this horribly named but otherwise wonderful looking innovation notes that: *"It can also recognize the most common unwanted guests — raccoon, wolf/coyote, bear, deer, squirrel, rabbit, duck, boar, skunk, fox, rats, boar, snake, chicken, and more — and deny them entry to your home."*

It's unclear to me why explicit recognition of unwanted pests is important if it's truly being discriminating about the identity of the family's pets — and what if you do have a pet duck? But, in any event, perhaps that's just to make sure that nothing unwanted can get in, even if it's some other clever animal wearing a dog mask to spoof the identity of your pooch.

And, of course, it's smartphone controlled, supports time-locks, remote lock and unlock, different schedules, entry and exit rules, and everything you could want. The mechanism's design appears to be quite solid, and if my household were pet-equipped I'd be quite tempted.

I should note that I stumbled upon this over on Kickstarter when I was following-up on a recent communication from Ed Cano — Ed's 48th communication. Ed is the originator of the SandSara, which may actually finally ship. Apparently many **have** shipped and they're working their way through customs at the moment. The SandSara, for those who don't recall, is a nifty looking tabletop bed of fine grained sand through which a steel ball bearing is rolled by a magnet located underneath the bed. The magnet is moved by a pair of stepper motors using a quite ingenious mechanism. This podcast discovered this Kickstarter project back in March of 2020, just as COVID was descending upon the world. Ed's original timeline anticipated fulfillment during August that same year, so it appears that it'll finally be happening just shy of two years late.

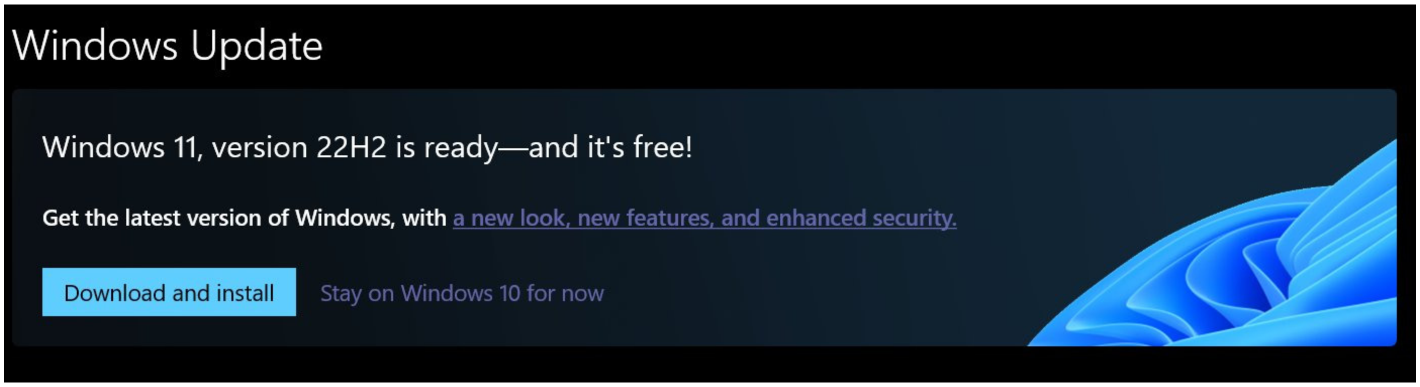
But I have no complaint with Ed's approach. His communication has been good and consistent, and he's proven himself to be an absolute perfectionist. Though we'll be receiving this amazing-looking gadget nearly two years late, it appears that what we'll be receiving will have been well worth the wait: https://www.kickstarter.com/projects/edcano/sandsara/description

His project has just shy of 2,000 backers, and I know that many of them are among our listeners since Ed told me so. When we discovered this, we got quite excited at the time.
So our collective wait may finally be nearing an end. :)

Closing The Loop

Robert Wicks / @bertwicks

You called it Steve. This just popped up on my ancient i7-4770 system that has ***no TPM.***



Windows Update

Windows 11, version 22H2 is ready—and it's free!

Get the latest version of Windows, with a new look, new features, and enhanced security.

Peter G. Chase / @PchaseG

Ha! Ha! My only slightly ineligible-for-WIN-11-upgrade PC (processor) is now magically eligible, and Windows is pestering me to do it. Nothing changed on my end. Must be them... You said they might miraculously change the requirements. But I still don't want Win 11.

Okay, so what happened? Under the category of "You just can't make this stuff up" we have the explanation for what Robert and Peter and many others experienced last week. The Verge covered the story last Thursday under the unbelievable headline (if it weren't Microsoft): *"Microsoft has accidentally released Windows 11 for unsupported PCs"* (And it runs just fine!) I added that last bit, but it does run just fine. Paraphrasing from what The Verge wrote:

Microsoft released the final version of its next big Windows 11 update (22H2) to Release Preview testers on Tuesday, and ***accidentally*** *made it available to PCs that are not officially supported. Oops.*

Twitter and Reddit users were quick to spot the mistake, with hundreds of Windows Insiders able to upgrade their Windows 10 machines on older CPUs. Microsoft has strict minimum hardware requirements for Windows 11, leaving millions of PCs behind, so the mistake will once again highlight the company's controversial upgrade policy. [Which is to say, it runs just fine everywhere even though we don't want to give it to you.]

Windows 11 officially requires Intel 8th Gen Coffee Lake or Zen 2 CPUs and up, with very few exceptions. While there are easy ways to install Windows 11 on unsupported CPUs, Microsoft doesn't even let its Windows Insiders officially install beta builds of the operating system on unsupported PCs, so this mistaken release is rather unusual.

Microsoft is aware of the mistake and says it's investigating. [Oh, well that's comforting.] *"It's a bug and the right team is investigating it,"* [As opposed to having the wrong team investigate it which would just add insult to injury.] *says the official Windows Insider Twitter account. If you managed to install Windows 11 on an unsupported PC and were expecting only Release Preview updates for Windows 10, you should be able to rollback the unexpected upgrade in the settings section of Windows 11.*

And speaking of large numbers…

Jeff Parrish / @kb9gxk

You can go to math.tools/calculator/num… to convert that number. It's large, just the first couple of numbers: One sexseptuagintillion five hundred fifty-two quinseptuagintillion

=OR=

one vigintillion,
five hundred fifty-two novemdecillion,
five hundred eighteen octodecillion,
ninety-two septendecillion,
three hundred sexdecillion,
seven hundred eight quindecillion,
nine hundred thirty-five quattuordecillion,

one hundred forty-eight tredecillion,
nine hundred seventy-nine duodecillion,
four hundred eighty-eight undecillion,
four hundred sixty-two decillion,
five hundred two nonillion,
five hundred fifty-five octillion,
two hundred fifty-six septillion,
eight hundred eighty-six sextillion,
seventeen quintillion,
one hundred sixteen quadrillion,
six hundred ninety-six trillion,
six hundred eleven billion,
one hundred thirty-nine million,
fifty-two thousand,
thirty-eight

Roy Ben-Yosef / @WizardLizardRoy

Hi Steve! Just heard your piece on NSW drivers license. And maybe I'm missing something, but isn't it all client side? They can do all sorts of fancy crypto and whatnot, can't I just write my own app that looks the same and shows whatever I want? Is there no (or shouldn't be) server side variation somehow? Long time listener since 2011, you got me into the cyber security business (since 2012). Thank you!

Ryan (via eMail)

Hello Steve,

Regarding the New South Wales digital drivers license. I work as my company's PKI expert, and I liked your solution to the DDL problem. I wanted to let you know that your proposed PKI based solution can also prevent spoofing of the drivers license QR code. Dancing flowers are all well and good, but could be spoofed if you were motivated enough. The way to prevent copying the QR code and passing it off as your own lies in exactly the type of certificate that you issue. If you issue a signed DDL certificate like you suggested, with the Key Usage attribute of digitalSignature (and the Enhanced Key Usage attribute of emailProtection) then your DDL can sign arbitrary messages. This is how encrypted S/MIME email works. In our case, the DDL could sign your name and age and include the signature in the QR code.

But, couldn't this QR code be copied? Well, the next step is to have the person verifying the ID check the timestamp to see exactly when the QR code was signed. You could have the user push a button (or enter a PIN) in the app and the generated certificate could be trusted for a limited amount of time, say 30 seconds. This also lets you customize your trust. A police officer's scanning device could trust a QR code for 30 seconds, a bouncer's device could trust a QR code for 1-2 minutes, so you can pull up your QR code while you're in line and keep the line moving. The chances of being able to predict the exact 30, 60, or 120 second window that you'll need to produce the QR code means that setting the clock ahead on your friend's phone and pregenerating a QR code are vanishingly small. Live long and prosper, Ryan

Tom Davies / @daviest11

*It always grates a bit when you dunk on GDPR for "ruining browsing". Please have a look here... https://www.coventry.ac.uk/ As you can see, it's perfectly possible to be GDPR compliant only forcing a single interaction on the user. What has "ruined browsing" is not GDPR but web hosts reliance on tracking users to the point where they would rather ruin your browsing experience than allow you to opt out. I've not dug into it that much but I'm sure we've all seen examples where it's clear that sites are deliberately making it *extremely* difficult to do anything other than accept the default "please track me" option. It really seems that if you don't want to be tracked, these sites do not want you to visit.*

I was curious to see what Tom was referring to. So I went over to https://www.coventry.ac.uk/. What Tom means is that it's possible for a website to request to set a cookie in order to no longer need to notify about setting cookies. Okay. And just for the record, since Tom says this is grating on him, we've previously made it clear that the strict privacy regulations had been in place long before the GDPR, but that everyone was happily ignoring them — thus no annoying cookie warnings — until the GDPR was enacted to give those regulations some teeth. I certainly don't disagree with the bias toward tracking that Tom notes, but it really was the enacting of the GDPR that made our lives miserable.

Philip Le Riche / @pleriche

SN874: The only essential diff betw Passkeys and SQRL is that SQRL uses computed priv keys. So could SQRL be made to register its priv key with a FIDO2 website and respond to a FIDO2 auth challenge? Then SQRL would work on any FIDO2 site! (But please, finish Spinrite first!!)

Philip anticipated that I talked about at the top of this episode. The use of deterministic private keys may be possible under WebAuthn. If so, then yes, SQRL's single-key approach could transparently replace the big-bucket-of-keys approach used by FIDO without sacrificing any security.

As for SpinRite, everyone should know that I'm done with SQRL. I've handed it off and over to a **very** competent group of developers who hang out over at https://sqrl.grc.com. So no one need worry for a nanosecond that I will again become distracted by SQRL. That's not going to happen.

lyntux / @lyntux

I wanted to remind you, @SGgrc, that a Yubikey can only hold up to 25 credentials each. That's not a problem, since there aren't that many FIDO2 websites out there, but that could change. That's another reason why SQRL is better.

Yes. And the fact that Yubikeys have a 25 credential limit also beautifully demonstrates that the FIDO project's work was never aimed at mass use and adoption. They were originally focused upon using super-secure hardware dongles for authentication to just a few crucial sites. That system was never able to scale. So when it failed to get off the ground, they relaxed their requirements and produced FIDO2.

Wesley Fick @cataclysmza

Hi Steve. I'm a long-time Security Now listener, and I hope that this makes its way into a future episode. With the way things are going with FIDO2, what's the value of services offered by Google and others to sign into websites with your Google account? My Google account cannot be logged into by anyone - they need the username, the full password, and a 2FA challenge from my phone. And I can use that to log into sites securely. The experience is similar to that of SSO. Let's say that Google adopts FIDO2 the way you suggested that Apple will. Is there any benefit to signing into websites with your Google account?

The trouble with the "Logon with Google" and "Logon with Facebook" is that they use the OAuth system which inherently places a heavy tracking agency — Google or Facebook — right smack dab in the middle of the logon flow. When any of the "Logon with..." systems are used, cookies are exchanged and the provider of that logon service knows who you are and where you went. This is why those companies are offering those services. It's not out of the goodness of their hearts. So one of the things that WebAuthn got exactly right is that it's a strictly 2-party authentication solution.

Juho Viitasalo @juhov

Hi! I tried the latest SpinRite exe from your dev directory but could not run it in Win7. I was hoping to burn an ISO that way. I have a failing SSD I need to massage. Can you tell how to use these executables you have on your server. Thanks!

I'm sure that Juho is referring to the reference I recently made to: http://grc.com/dev/spinrite. That directory contains a gazillion incremental SpinRite development releases. But they are DOS executables, not Windows EXE's. So they definitely will not run under Windows. But moreover, even if you did run them under DOS, they do not yet move the user past the system inventory stage where SpinRite is figuring out what you've got and how it can best work with it. That's where all of our collective work has been focused so far.

Once I do have something that's operational, all existing SpinRite licensees will be able to obtain their own copies from GRC's server. But we're not quite at that stage yet. So, I apologize for the confusion.

###

Okay... If anyone listening to this podcast has only been half listening. **Now** is the time to bring your entire focus to bear on this next question from a listener because it wins the award for the best question, maybe ever. And I'm going to leave it hanging and unanswered until next week. So everyone will have the opportunity this next week to ponder the question and its answer. And Leo and everyone in the Discord and chat rooms, behaved yourselves. Please resist the temptation to blurt out any spoilers for everyone else who wants to ponder this very intriguing question. So here it is:

Erik Osterholm / @erikosterholm

Hi Steve,

Long time listener. I'm a little confused on the key length discussion. I always thought that encrypting twice merely doubled the effective strength. Here's my reasoning:

Imagine you have an algorithm and a key that you can fully brute force in one day. If you add one bit to the key, you double the key space, and therefore it takes twice as long to brute force (or two days.)

If you instead encrypt twice, then it takes one day to decrypt the outer ciphertext, at which point you get back the first output of ciphertext. Then in one more day, you can brute force the first/inner ciphertext. Like in the first example, this takes two days.

It seems to me that these are equivalent. Is there something I'm missing?

And that's all we're going to say about it now. Think about it for the next week. And Leo, let's take our last break and then we're going to delve into the so-called "PACMAN Attack" — which leverages an unpatchable and unfixable flaw which exists across Apple's custom M1 chipset.

The PACMAN Attack

https://pacmanattack.com/
https://pacmanattack.com/paper.pdf

By far the biggest headline grabbing news this past week was the teaser that Apple's much-vaunted M1 chips contain a critical bug, or bugs, that cannot be patched or repaired.

First of all, only part of that is true. The reason Apple is being singled out, is that their M1 ARM architecture chips are the first to use a cool new feature of the ARMv8.3 architecture known as Pointer Authentication. Everyone else plans to use it too, they just haven't gotten their newest chips out yet.

The heading of the 14-page research paper published by four researchers at the MIT Computer Science & Artificial Intelligence Laboratory reads: *"PACMAN Attacking ARM Pointer Authentication with Speculative Execution."*

So, we have the spectre (pun intended) of speculative execution raising its ever ugly head, this time in a new context, a new feature of the ARMv8.3 architecture, initially represented by Apple's proprietary M1 chip.

Okay. So what is it?

PACMAN is a novel hardware attack that can bypass the benefits of an emerging feature known as Pointer Authentication (PAC. So that's the PAC of PACMAN). The authors of this work present the following three contributions:

- A new way of thinking about compounding threat models in the Spectre age.
- Reverse engineered details of the M1 memory hierarchy.
- A hardware attack to forge kernel PACs from userspace on M1.

They wrote: *"PACMAN is what you get when you mix a hardware mitigation for software attacks with microarchitectural side channels. We believe the core idea of PACMAN will be applicable to much more than just PAC."* So these guys believe that they've uncovered another entire class of exploitable microarchitectural flaws which are introduced by attempts to mitigate software attacks. We'll talk about Pointer Authentication in a second, but here's what the researchers wrote for the summary Abstract of their paper:

This paper studies the synergies between memory corruption vulnerabilities and speculative execution vulnerabilities. We leverage speculative execution attacks to bypass an important memory protection mechanism, ARM Pointer Authentication, a security feature that is used to enforce pointer integrity. We present PACMAN, a novel attack methodology that speculatively leaks PAC verification results via micro-architectural side channels without causing any crashes. Our attack removes the primary barrier to conducting control-flow hijacking attacks on a platform protected using Pointer Authentication.

We demonstrate multiple proof-of-concept attacks of PACMAN on the Apple M1 SoC, the first desktop processor that supports ARM Pointer Authentication. We reverse engineer the TLB hierarchy on the Apple M1 SoC and expand micro-architectural side-channel attacks to Apple processors. Moreover, we show that the PACMAN attack works across privilege levels, meaning that we can attack the operating system kernel as an unprivileged user in userspace.

I loved how they set this up and framed this work in their paper's short introduction, so I want to share it, too:

Modern systems are becoming increasingly complex, exposing a large attack surface with vulnerabilities in both software and hardware. In the software layer, memory corruption vulnerabilities (such as buffer overflows) can be exploited by attackers to alter the behavior or take full control of a victim program. In the hardware layer, micro-architectural side channel vulnerabilities (such as Spectre and Meltdown) can be exploited to leak arbitrary data within the victim program's address space.

Today, it is common for security researchers to explore software and hardware vulnerabilities separately, considering the two vulnerabilities in two disjoint threat models. In this paper, we study the synergies between memory corruption vulnerabilities and micro-architectural side-channel vulnerabilities. We show how a hardware attack can be used to assist a software attack to bypass a strong security defense mechanism. Specifically, we demonstrate that by leveraging speculative execution attacks, an attacker can bypass an important software security primitive called ARM Pointer Authentication to conduct a control-flow hijacking attack.

Okay, so what is ARM Pointer Authentication?

We've talked about related attack mitigation measures in the past. Remember "stack cookies"? They're a form of a canary. The idea with Stack Cookies is that the system places little bits of unpredictable crypto hashes on the stack as a means for detecting a buffer overrun. Before executing a return from a subroutine, the bit of code performing the return first checks the stack for the proper cookie and only executes the return if the value found at the location matches what's expected. The idea is that attackers don't have nearly sufficient visibility into the system to know what the cookie should be, so their attempt to overrun a buffer on the stack is inherently somewhat brute force. The cookie allows their overwriting to be proactively detected before the contents of the stack is trusted and acted upon.

What the ARMv8.3 architecture introduces is a clever means of adding a tiny authentication hash into ARM pointers as a means for detecting any unauthorized changes. As we know, the ARM architecture is 64-bits side. That means that it will have 64-bit pointers. But 64-bits is 18,446,744,073,709,551,616. Said another way, that's 18.446 billion gigabytes of pointer space. Since no one has nearly that much RAM, that means that a large number of the high-order bits of 64-bit ARM memory pointers are always going to be zero. So the clever ARM engineers realized that they could set a boundary point in their pointers. To the right of the boundary is a valid pointer. And to the left of the boundary are a bunch of otherwise unused bits which can now be very cleverly used to validate the value of the bits on the right. Here's how the MIT guys describe this:

Memory corruption vulnerabilities pose a significant security threat to modern systems. [Right. We're talking about this here more or less continually.] *These vulnerabilities are caused by software bugs which allow an attacker to corrupt the content of a memory location. The corrupted memory content, containing important data structures such as code and data pointers, can then be used by the attacker to hijack the control flow of the victim program. Well-studied control-flow hijacking techniques include return-oriented programming (ROP) and jump-oriented programming (JOP). In 2017, ARM introduced Pointer Authentication (PA for short) in ARMv8.3 as a security feature to protect pointer integrity. Since 2018, Pointer Authentication has been supported in Apple processors, including multiple generations of mobile processors and the recent M1, M1 Pro, and M1 Max chips. Multiple chip manufacturers, including ARM, Qualcomm, and Samsung, have either announced or are expected to ship new processors supporting Pointer Authentication. In a nutshell, Pointer Authentication is currently being used to protect many systems, and is projected to be even more widely adopted in the upcoming years. Pointer Authentication makes it significantly more difficult for an attacker to modify protected pointers in memory without being detected. Pointer Authentication protects a pointer with a cryptographic hash. This hash verifies that the pointer has not been modified, and is called a Pointer Authentication Code, or PAC for short.*

So, this is really clever. It essentially makes 64-bit pointers self-authenticating such that any change to any of the pointer’s 64 bits will trigger a protection fault and cause the operating system to immediately terminate the offending program.

The Achilles heel of this approach is that there are not a large number of authentication bits available. In the case of macOS v12.2.1, which extravagantly uses 48 of the total 64 bits for valid pointers, only 16 bits remain available for authentication. As we know, 16 bits is a famous number — it’s 64K. That’s the total number of possible combinations that can be taken by any 16-bit value. Again, here’s what the researcher’s describe:

Considering that the actual address space in a 64-bit architecture is usually less than 64 bits, e.g., 48 bits on macOS 12.2.1 on M1, Pointer Authentication stores the Pointer Authentication Code together with the pointer in these unused bits. Whenever a protected pointer is used, the integrity of the pointer is verified by validating the PAC using the pointer value. Use of a pointer with an incorrect PAC will cause the program to crash. With Pointer Authentication in place, an attacker who wants to modify a pointer must correctly guess or infer the matching PAC of the pointer after modification. Depending on the system configuration, the size of the PAC, which ranges from 11 to 31 bits, may be small enough to be bruteforced. However, simple bruteforcing approaches cannot break Pointer Authentication. The reason is that every time an incorrect PAC is used, the event results in a victim program crash. Restarting a program after a crash results in changed PACs, as the PACs are computed from renewed secret keys. Moreover, frequent crashes can be easily captured by anomaly detection tools.

The breakthrough that this guys achieved was finding a way to successfully brute force probe for the proper authentication code given a modified pointer. By doing that, they’re able to make up their own PAC for whatever pointer value they need. So, the final piece of description I’ll share from their paper is:

***The PACMAN Attack.** In this paper, we propose the PACMAN attack, which extends speculative execution attacks to bypass Pointer Authentication by constructing a PAC oracle.*

Given a pointer in a victim execution context, a PAC oracle can be used to precisely distinguish between a correct PAC and an incorrect one without causing any crashes. [In other words, exactly what's needed for brute-forcing.] *We further show that with such a PAC oracle, the attacker can brute-force the correct PAC value while suppressing crashes and construct a control-flow hijacking attack on a Pointer Authentication-enabled victim program or operating system.*

The key insight of our PACMAN attack is to use speculative execution to stealthily leak PAC verification results via microarchitectural side channels. Our attack works relying on PACMAN gadgets. A PACMAN gadget consists of two operations: 1) a pointer verification operation that speculatively verifies the correctness of a guessed PAC, and 2) a transmission operation that speculatively transmits the verification result via a micro-architectural side channel.

The pointer verification operation is performed by an authentication instruction (one of the new instructions in ARMv8.3), which outputs a valid pointer if the verification succeeds and an invalid pointer otherwise. The transmission operation can be performed by a memory load/store instruction or a branch instruction taking the output pointer as an address. If a correct PAC is guessed, the transmission operation will speculatively access a valid pointer, resulting in observable micro-architectural side effects. Otherwise, the transmission step will cause a speculative exception due to accessing an invalid pointer. Note that we execute both operations on a mis-speculated path. Thus, the two operations will not trigger architecture-visible events, avoiding the issue where invalid guesses result in crashes.

So, now we bottom-line it. What does all this actually mean for those relying upon the security of Apple's M1-based devices? In their own FAQ, they provide some reality-check answers:

Does this attack require physical access?
Nope! We actually did all our experiments over the network on a machine in another room. PACMAN works just fine remotely if you have unprivileged code execution.

Should I be worried?
As long as you keep your software up to date, no. PACMAN is an exploitation technique. On its own it cannot compromise your system. While the hardware mechanisms used by PACMAN cannot be patched with software features, memory corruption bugs can be.

Can I tell if someone is using PACMAN against me?
Much like the Spectre attack our work is based on, PACMAN executes entirely in the speculative regime and leaves no logs. So, "probably not."

Why do you need a software bug to do the PACMAN attack?
PACMAN takes an existing software bug (memory read/write) and turns it into a more powerful primitive (pointer authentication bypass). For our demo, we add our own bug to the kernel using our own kernel module. In a real world attack, you would just find an existing kernel bug.

Our demo only uses the kernel extension for adding a software bug. The entire attack runs in userspace. So, an attacker with an existing bug could do the same attack without the extension.

Is PACMAN being used in the wild?
To our knowledge, no.

Apple's response didn't offer much. They officially said:

> *"We want to thank the researchers for their collaboration as this proof-of-concept advances our understanding of these techniques. Based on our analysis, as well as the details shared with us by the researchers, we have concluded this issue does not pose an immediate risk to our users and is insufficient to bypass device protections on its own."*

Okay. That was carefully worded to be exactly, factually, correct. And it is. Generally speaking, the common M1 chip user does not have anything more to worry about after this than they did before this. But the authors never suggested that this could be used in any form of stand-alone attack. What this **does** mean is that successful attacks against current and future ARM chips, equipped with Pointer Authentication, will need to be more complex in order to also defeat this new extra layer of protection. As is, Apple's M1 chips based upon the ARMv8.3 architecture are thus significantly more difficult to successfully attack than other systems which lack this additional, though imperfect, layer of protection.

And I'll close with this observation: Apple's allocation of 48-bits for their pointers, which affords access to more than **a quarter million gigabytes of RAM**, is obviously excessive in the extreme. Before this paper was published Apple's engineers probably thought: "What the hell, we have 64-bits of pointer space, so we'll use the lower 48 for pointing and the upper 16 for pointer authentication. Assuming that it was not possible to brute-force probe for the correct value of those 16 bits, Apple's engineers likely thought that there would only be one chance in 65,536 of an attacker correctly guessing the proper hash for a modified pointer and a wrong guess would cause the victim program to be immediately terminated.

But now we know that 16 bits is not sufficient. So Apple COULD choose to significantly strengthen their Pointer Authentication by tightening down on the active pointer space, maybe even setting it up dynamically based upon the amount of RAM a system contains. A large system with 64 gigabytes of RAM requires pointers to consume only 36 bits. That leaves 28 bits for authentication, which is **4,096 times** more difficult and time consuming to brute force, with no other overhead or cost to the system.

In any event, the sky is once again not falling and all is well.