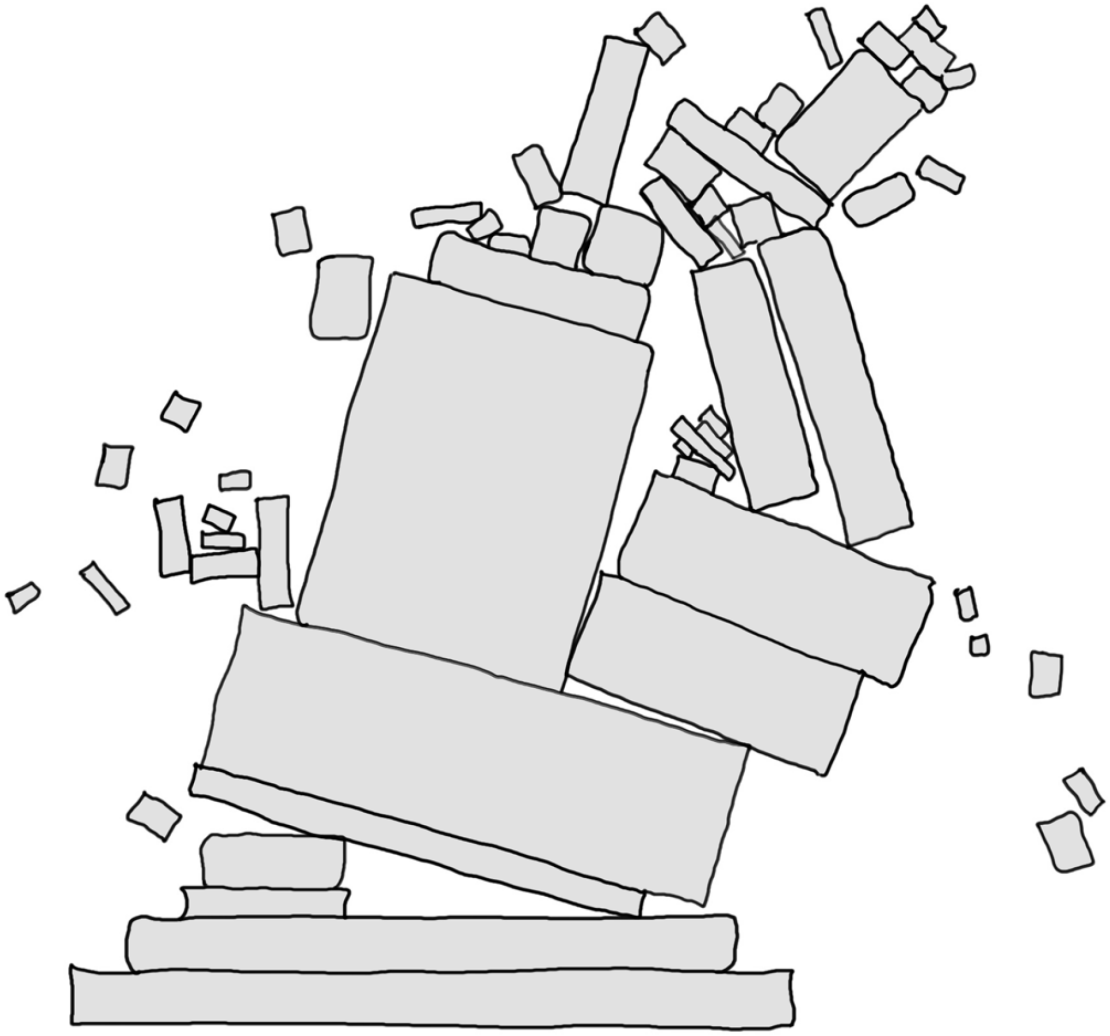
Security Now! #970 - 04-16-24

GhostRace

This week on Security Now!

What's the latest on that massive five year old AT&T data breach? Who just leaked more than 340,000 social security numbers, Medicare data and more, and what does that mean? Are websites honoring their cookie banner notification permissions? And why do we already know the answer to that question? What surprise has the GDPR's transparency requirements just revealed? And after sharing a bit of feedback from our listeners, we're going to go deeper into raw fundamental computer science technology than we have in a long time... and it may be inadvisable to operate any heavy equipment while listening to that part.

This wonderful concept suffers from being a bit of an inside joke, but for those who understand it, well... it's quite wonderful:



(If you need a hint, see: https://xkcd.com/2347/)

With thanks to Simon Zerafa for bringing this to my attention.

Security News

An update on the AT&T data breach

I wanted to follow-up on our report from two weeks ago about the massive and significant AT&T data breach. Reading BleepingComputer's update, I was put in mind of the practice known as "rolling disclosure" where the disclosing party successively denies the truth, only admitting to what is already known when evidence of that is presented. The trouble here, of course, is that AT&T is a massive publicly owned communication's carrier that holds deep details about their U.S. consumer users... and an examination of this incident suggests that they should be acting far more responsibly and, moreover, that by not having done so for the past five years the damage inflicted upon their own customers has likely been far worse than it needed to be.

BleepingComputer's reporting was headlined: *"AT&T now says data breach impacted 51 million customers"* and the annoyance in this piece's author is readily apparent. I've edited it just a bit to be more clear for this podcast, but BleepingComputer posted:

> *AT&T is notifying 51 million former and current customers, warning them of a data breach that exposed their personal information on a hacking forum. However, the company has still not disclosed how the data was obtained. These notifications are related to the recent leak of a massive amount of AT&T customer data on the Breach hacking forums that was [first] offered for sale for $1 million in 2021.*
>
> *When threat actor **ShinyHunters** first listed the AT&T data for sale in 2021, AT&T told BleepingComputer that the collection did **not** belong to them and that their systems had not been breached. Last month, when another threat actor known as **MajorNelson** leaked the **entire** dataset on the hacking forum, AT&T once again told BleepingComputer that the data did not originate from them and that their systems were not breached.*
>
> *After BleepingComputer confirmed that the data did belong to AT&T and DirectTV accounts, and TechCrunch reported that AT&T's logon passcodes were in the data dump, AT&T finally confirmed that the data **did** belong to them.*

Like I said... "rolling disclosure" —> "Oh! **THAT** data!" Uh huh. That data.

> *While the leak contained information for more than 70 million people, AT&T is now saying that it impacted a total of 51,226,382 customers. AT&T's most recent notification states:*
>
> *"The [exposed] information varied by individual and account, but **may** have included full name, email address, mailing address, phone number, social security number, date of birth, AT&T account number and AT&T passcode. To the best of our knowledge, personal financial information and call history were not included. Based on our investigation to date, the data appears to be from June 2019 or earlier."*
>
> *BleepingComputer contacted AT&T asking why there is such a large difference in impacted customers: "We are sending a communication to each person whose sensitive personal information was included. Some people had more than one account in the dataset, and others did not have sensitive personal information."*

The company has still not disclosed how the data was stolen and why it took them almost five years to confirm that it belonged to them and to alert customers.

I can understand that today they may not know how it was stolen. It's at least believable that if they didn't ever look, they never would have found. But the denial of the evidence they were shown many years ago is quite difficult to excuse. And it appears that it is not going to be excused.

Furthermore, the company told the Maine Attorney General's Office that they first learned of the breach on March 26, 2024, yet BleepingComputer first contacted AT&T about it on March 17th and the information was for sale first in 2021.

While it is likely too late, as the data has been privately circulating for years, AT&T is offering one year of identity theft protection and credit monitoring services through Experian, with instructions enclosed in the notices. The enrollment deadline was set to August 30, 2024, but exposed people should move much faster to protect themselves.

Recipients are urged to stay vigilant, monitor their accounts and credit reports for suspicious activity, and treat unsolicited communications with elevated caution. For the admitted security lapse and the massive delay in verifying the data breach claims and informing affected customers accordingly, AT&T is facing multiple class-action lawsuits in the U.S.

Considering that the data was stolen in 2021, cybercriminals have had ample opportunity to exploit the dataset and launch targeted attacks against exposed AT&T customers. However, the dataset has now been leaked to the broader cybercrime community, exponentially increasing the risk for former and current AT&T customers.

Just so we're clear here, and so that all of our listeners new and old understand the nature of the risk this potentially presents to AT&T's customers: Armed with the personal data that has been confirmed and admitted to have been disclosed – specifically social security numbers, dates of birth and physical addresses – that's all that's needed to empower bad guys to apply for and establish new credit accounts in the names of credit worthy individuals. This is one of the most severe consequences of what is commonly known as "identity theft" and it's a true nightmare.

The way this is done is that an individual's private information is used to impersonate them when a bad guy applies for credit in their name using what amounts to their identity. That bad guy then drains that freshly established credit account, and disappears, leaving the individual on the hook for the debt that has just been incurred. Since anyone might do this themselves then claim that it wasn't really them and that *"it must have been identity theft your Honor"*, proving this wasn't really them running a scam can be nearly impossible and among other things it can mess up someone's credit forever.

There's only one way to stop this, which is to prevent anyone, *including ourselves*, from applying for and receiving any new credit by preemptively **freezing** our credit reports at each of the three major credit reporting agencies. I know we've covered this before, at the beginning of the year, in fact. But it bears repeating and we may have some new listeners who haven't already heard this or existing listeners who intended to take this action but never got around to it.

So here's my point: In this day and age of inadvertent online information disclosure, it is no longer safe or practical for our personal credit reporting to ever be left unfrozen by default. Rather than freezing our credit **if** we receive notification of a breach that might affect us, as 51,226,382 of AT&T's current and former customers are now being informed five years after the fact, everyone should have their credit always frozen by default and then only briefly and selectively unfrozen when a credit report does actually need to be made available.

The last time we talked about this I created a GRC shortcut link for that podcast, which was #956. But I want to make it even easier to get to that page. So you can get all the details about freezing your credit reports by going to: https://grc.sc/credit. That will bounce your browser over to a terrific article at Investopedia.

340,000 social security numbers leaked

And just to put an exclamation mark at the end of that news, another disclosure was just made and reported under the headline: *"Hackers Siphon 340,000 Social Security Numbers From U.S. Consulting Firm"*. Cysecurity news wrote:

> *Greylock McKinnon Associates (GMA) has discovered a data breach in which hackers gained access to 341,650 Social Security numbers. The data breach was disclosed last week on Friday on Maine's government website, where the state issues data breach notifications. In its data breach warning mailed to impacted individuals, GMA stated that it was targeted by an undisclosed cyberattack in May 2023 and "promptly took steps to mitigate the incident."*
>
> *GMA provides economic and litigation support to companies and government agencies in the United States, including the Department of Justice, that are involved in civil action. According to their data breach notification, GMA informed affected individuals that their personal information "was obtained by the U.S. Department of Justice ("DOJ") as part of a civil litigation matter" supported by GMA.*
>
> *The purpose and target of the DOJ's civil litigation are unknown. A Justice Department representative did not return a request for comment.*
>
> *GMA stated that individuals that were notified of the data breach are "not the subject of this investigation or the associated litigation matters," adding that the cyberattack "does not impact your current Medicare benefits or coverage. We consulted with third-party cybersecurity specialists to assist with our response to the incident, and we notified law enforcement and the DOJ. We received confirmation of which individuals' information was affected and obtained their contact addresses on February 7, 2024."*
>
> *GMA notified victims that "Your private and Medicare data was likely affected in this incident" – ya gotta love the choice of the word "affected" – "Oh!, you mean as in 'was obtained by malicious hackers?'". And they said, "which included names, dates of birth, home addresses, some medical and health insurance information, and Medicare claim numbers, including Social Security numbers.*
>
> *It remains unknown why GMA took nine months to discover the scope of the incident and notify victims. GMA and its outside legal counsel, Linn Freedman of Robinson & Cole LLP, did not immediately respond to a request for comment.*

So as I noted above, this is now happening all the time. It is no longer safe to leave one's credit reporting unfrozen, and freezing it everywhere will only take a few minutes. Since you won't want to be locked out of your own credit reporting afterwards, be sure to securely record the details you'll need when it comes time to briefly and selectively unfreeze your credit in the future.

Cookie Notice Compliance

The following very interesting research was originally slated to be this week's major discussion topic. But after I spent some time looking into the recently revealed GhostRace problem I wanted to talk about that since it brings in some very cool fundamental computer science about the problem of "race conditions" which, in our 20 years of this podcast we have never touched upon.

But what this team of five guys from ETH Zurich discovered was very interesting too. So here's that, now. They began by asking themselves: "When we go to a website that presents us with what has now become the rather generic and GDPR-required Cookie Permission pop-up, do sites where permission is **not** explicitly granted and cookie use is explicitly denied actually honor that denial?" And with that open question on the table, the follow-on question was: "Can we arrange to create an automated process to obtain demographic information about the cookie permission handling of the top one hundred thousand websites? Well we already know that the answer to the second question is "yes" we can automate this data collection. And the results of their research will be presented during the 33rd USENIX Security Symposium which will take place this coming August 14th through the 16th in Philadelphia. Their paper is titled *"Automated Large-Scale Analysis of Cookie Notice Compliance"*
https://www.usenix.org/system/files/sec23winter-prepub-107-bouhoula.pdf

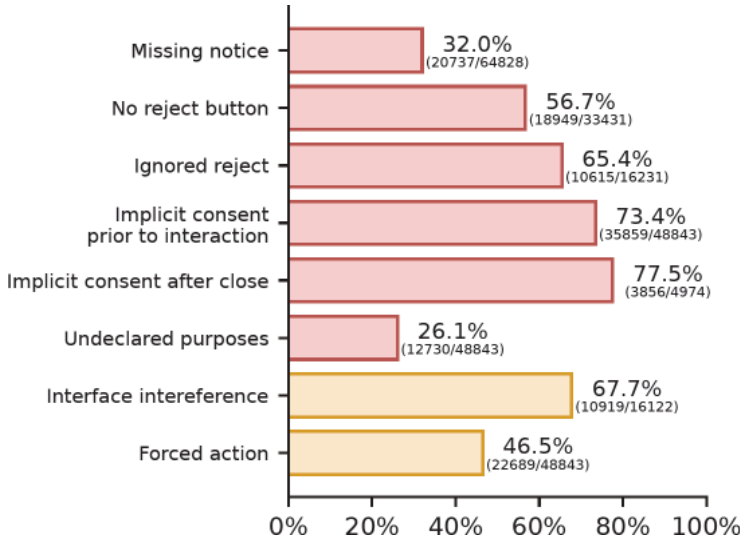
Perhaps it won't come as any shock that when they found was somewhat disappointing. Did they find that 5% of the 97,090 web sites they surveyed did not obey the site's visitors' explicit denial of permission to store privacy invading cookies? No. Was the number 10%? Nope, not that either. How about 15%? Nope, still too low. Believe it or not, 65.4% of all websites tested, so just shy of fully two thirds of all websites tested do not obey their visitors' explicit privacy requests. They wrote: *"we find that 65.4% of websites do not respect users' negative consent and that top-ranked websites are more likely to ignore users' choices despite having seemingly more compliant cookie notices."* The Abstract of their paper says:

> *Privacy regulations such as the General Data Protection Regulation (GDPR) require websites to inform EU-based users about non-essential data collection and to request their consent to this practice. Previous studies have documented widespread violations of these regulations. However, these studies provide a limited view of the general compliance picture: they are either restricted to a subset of notice types, detect only simple violations using prescribed patterns, or analyze notices manually. Thus, they are restricted both in their scope and in their ability to analyze violations at scale.*
>
> *We present the first general, automated, large-scale analysis of cookie notice compliance. Our method interacts with cookie notices, e.g., by navigating through their settings. It observes declared processing purposes and available consent options using Natural Language Processing and compares them to the actual use of cookies. By virtue of the generality and scale of our*

analysis, we correct for the selection bias present in previous studies focusing on specific Consent Management Platforms (CMP). We also provide a more general view of the overall compliance picture using a set of 97k websites popular in the EU. We report, in particular, that 65.4% of websites offering a cookie rejection option likely collect user data despite explicit negative consent.

I suppose we should not be surprised to learn that what's hiding behind the curtain is not what we would hope. 2/3rds of companies, and in general, the larger they are, the worse is their behavior, are simply blowing off their visitors' explicit requests for privacy. So not only are we now all being hassled by the presence of these cookie permission banners, but two out of every three required "do nothing clicks" have no actual effect.

The researchers discovered additional unwanted behavior which they termed "dark patterns." Specifically...

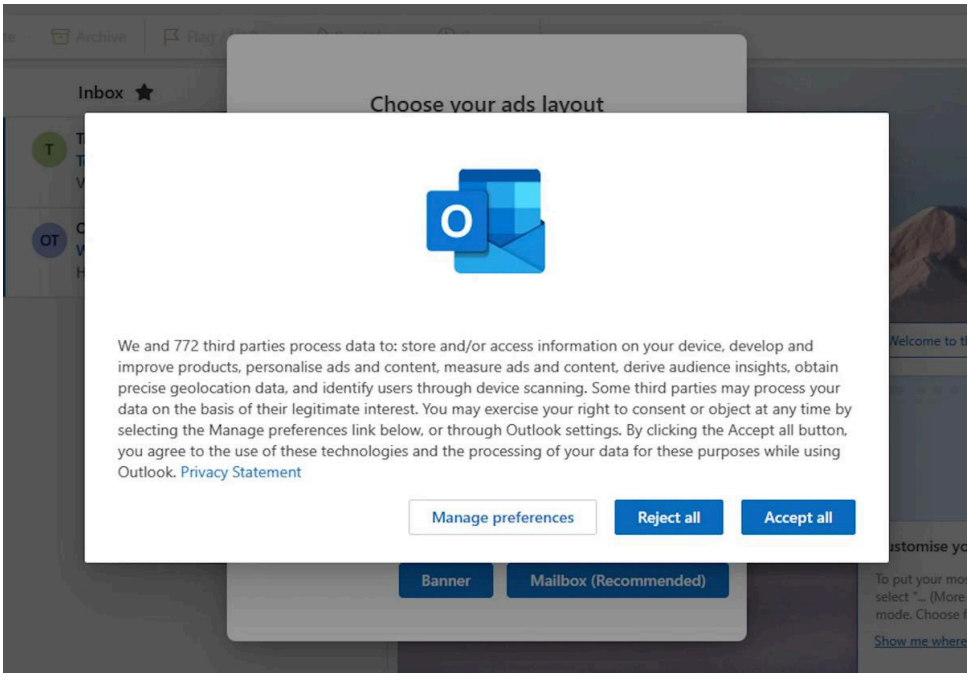


What this means is that the true enforcement of our privacy cannot be left to lawmakers and their legislation, nor to websites. New technology needs to be brought to bear to take this out of the hands of all third parties. And I know it seems insane for me to be saying this, but this really is what Google's Privacy Sandbox has been designed to do. And as with all change, advertisers and websites are going to be kicking and screaming and they're not going to go down without a fight. But the advertisers are also, reluctantly, currently in the process of upgrading their ad delivery architectures for the Privacy Sandbox because they know that with Chrome commanding 2/3rds market share, they're going to have to work within this brave new world that's coming soon, as in "real soon" ... later this year. Google's not messing around this time. They've given everyone years of notice. Change is coming.

The GDPR does enforce some transparency

Speaking of the GDPR which created this accursed cookie pop-up legislation which currently plagues the Internet, holding to the "there's two sides to every coin" rule, the requirements for disclosure that's also part of the GDPR legislation occasionally produces some startling revelations when conduct that parties would have doubtless much preferred to remain off the radar are required to be seen.

In this case we have the new Outlook app for Windows. Thanks to the GDPR, users in Europe who download the Outlook for Windows app, will be greeted with a modal pop-up dialog that displays a user agreement and requires its users to consent. The breathtaking aspect of this is that the dialog starts right out stating: "We and **772 third parties** process data to: store and/or access information on your device, develop and improve products, personalize ads and content, measure ads and content, derive audience insights, obtain precise geolocation data, and identify users through device scanning."



That was not a typo. It says "We and 772 third parties..." Wow. I don't know what to think about that. It's really mind boggling. For one thing, it's disappointing that only those in the EU get to see this. U.S. regulators are not forcing the same transparency requirements on Microsoft, and Microsoft certainly doesn't want to show any more of this than they're forced to.

Earlier we were talking about the confidentiality of our data and the challenge that presents. Having Microsoft profiting from the sale of the contents of its user's eMail seems annoying enough. At the same time, that's gMail, too. But when there are, by Microsoft's own forced admission, 772 such third parties who all, presumably, receive paid access to this data, doesn't it feel as though Microsoft should be paying us for the privilege of access to our private communications? Instead, we get free eMail. Whoopee!

Closing The Loop

ndom91 / @ndom91

> *@SGgrc: been hearing you talk about the physical input requirement for security setting modification, just wanted to mention Fritz and FritzBox, a common German home router manufacturer who's been doing this for a while.* ***Annoying, yes. Effective, also yes*** 😅

When we've talked about this previously I haven't mentioned that many SoHo routers have that dedicated WPS hardware button that is a physical button for essentially the same reason. It allows a network device to be persistently connected to the router without the need to provide the device with the router's WiFi SSID name and password. The button being physical requires someone to prove their presence by pressing the button. So the notion of doing something similar to similarly protect configuration changes makes a lot of sense.

But then another listener suggested...

Rob Woodruff / @fulori

> *I'm listening to the part of SN 969 where you're talking about the five security best practices and it occurs to me that, regarding the requirement for a physical button press, it won't be long before we start to see cheap, Chinese-made, WiFi-enabled IoT button pressers on Amazon. What could possibly go wrong?!*

Yes indeed. The downside of the physical button press requirement is its inherent inconvenience. But that's also the source of its security. It would be nice if we could have the security without the inconvenience, but it's unclear how we would go about doing that. But to Rob's point, even if a user chose to employ a 3rd-party remote control button presser, that would still provide greater security than having no button at all. For one thing, it would be necessary to hack two very different systems, like having multifactor authentication. But the most interesting observation, I think, is that generic remote attacks against an entire class of devices that were known to be "button protected" would never even be launched in the first place because it would be known that they could not succeed. So even a router that technically breaks the rules with an automated button presser gains the benefit of what is essentially herd immunity.

Rami Vaspami / @vaspami

> *@SGgrc I don't see the benefit if netsecfish kept quiet. This would've been quietly exploited in perpetuity. Instead, now, everyone knows to deprecate those devices and the issue has an end in sight. Better to know.*

If, indeed, current D-Link NAS owners could have been notified as a consequence of netsecfish's disclosure on GitHub then, okay. Maybe. But they still would have had to act quickly since the attacks against these D-Link NAS boxes required less than two weeks to begin. But the problem is that **nothing** about netsecfish's disclosure on GitHab suggests that even a single user of these older but still in use D-Link NAS devices would have received notification. Notification from whom? How would any random user know what's been posted in someone's GitHub account? D-Link said "sorry, but those are old devices for which we no longer bear any responsibility."

But we know who does know what's been posted to someone's GitHub account, since it took less than two weeks for the attacks to begin. So I would argue that it is not "better to know" when it's by far predominantly the bad guys who will be paying attention and who will know.

And finally, John Moriarty wins the "feedback of the week award". He tweeted:
John Moriarty / @JoMozComplains

> *Catching up on last week's #securitynow and hearing @SGgrc lament the fans spinning up and resources being swamped by the Google's Chrome bloat while it was "doing nothing" - perhaps it was busy running and completing ad auctions?*

Yes indeed. It's quite true that our browsers are going to be much busier once responsibility for ad selection is turned over to them.

SpinRite

I don't have any big SpinRite news this week. Everything continues to go well and I'm working to update GRC's SpinRite pages just enough to hold us while I get eMail running and then back to work on SpinRite's documentation. I'm seeing more reports of SSD original speed recovery. While catching up in Twitter I did run across a good question about SpinRite:

Paul / @masonpaulak

> *Hi Steve. Long time podcast listener and Spinrite owner, thanks for the many gems of info over the years. Over the weekend I saw the ram test in SR61. Please would you give some insight into how long I should let it run to test 1GB, and am I wearing it out as writing to an SSD would? Many thanks, Paul*

The best answer to this is really quite frustrating because the best answer is: The longer the better. That said, I can clarify a couple of points. The total amount of RAM in a system has no bearing on this, since SpinRite is only testing the specific 54 megabytes it will be using once the testing is over. While the testing is underway, a test counter is spinning upwards on the screen and a total elapsed testing time is shown. For each test it fills 54 megabytes with a random pattern then comes back to reread and verify that memory. So the longer the testing runs the greater the likelihood that it will find a pattern that causes trouble.

The confusion, and Paul's question, comes from the fact that the new startup RAM test can be interrupted at any time by its user to move into SpinRite. So when should that be? Best practice, if it's feasible, for any new machine where SpinRite v6.1 has never been run before, would be to start it up and allow it to just run overnight doing nothing but testing it RAM. In the morning the screen will almost certainly still be happy and still showing zero errors. But the screen might have turned RED indicating that one or more verification passes failed. After I asked that first guy who was having weird problems that made no sense to try running MemTest86, it reported errors and I decided I needed to build that into SpinRite just to be safe. Users do not need to use it at all, but it's there. And several other of SpinRite's early testers were surprised to have SpinRite detect previously unsuspected memory errors on some of their machines. So SpinRite's built-in RAM memory test is sort of useful all by itself. But once it's run for a few hours on any given machine, it can just be bypassed quickly from then on.

GhostRace

The first question to answer is "What's a Race Condition?" A race condition is any situation, which might occur in hardware or software systems, where outcome uncertainty or an outright bug is created when the sequence of closely timed events, in other words a race among events, determines the effects of those events. For example, in digital electronics, it is often the case that the state of a set of data lines is captured when a clock signal occurs. This is typically known as "latching" the data lines on a clock signal. But for this to work reliably, the data lines need to be stable and settled when the clocking occurs. If the data lines are still changing state when the clock occurs, or if the clock occurs too soon, a race condition can occur where the data that's latched is incorrect.

Here's how Wikipedia defines the term:

> *A race condition or race hazard is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events, leading to unexpected or inconsistent results. It becomes a bug when one or more of the possible behaviors is undesirable. The term "race condition" was already in use by 1954, for example in David A. Huffman's doctoral thesis "The synthesis of sequential switching circuits". Race conditions can occur especially in logic circuits, multithreaded, or distributed software programs.*

Race conditions are also a huge issue with today's modern software systems because our machines now have so many things going on at once where the independent actions of individual threads of execution must be synchronized. Any failure in that synchronization can result in bugs or even serious security vulnerabilities.

To understand this fully we need to switch into computer science mode. I often receive feedback from our listeners wishing that I did more of this, but I don't often encounter such a perfect opportunity as we have today. Back when we were talking about how processors work during one of our early sets of podcasts on that topic, we talked about the concepts of multithreading and multitasking.

In a preemptive multithreaded environment the concept of a thread of execution is an abstraction. The examples I'm going to draw assume a single core system, but they apply equally to multi-core environments. When a processor is busily executing code, performing some task, we call that a thread of execution since the processor is threading its way through the system's code, executing one instruction after another. After a certain amount of time, a hardware interrupt event occurs when a system clock ticks. This interrupts that thread of execution causing it to be suspended and control to be returned to the operating system. If the operating system sees that this thread has been running long enough and there are other waiting threads, the OS decides to give some other thread a turn. So it does something known as a "context switch". A thread's "context" is the entire state of execution at any moment in time. So it's all of the registers the thread is working with and any other thread-specific volatile information. The operating system saves all of that thread context and reloads another thread's context – which is to say everything previously saved from another thread. The OS then jumps to the point in the code where that other thread was rudely interrupted the last time it was

preempted, and that other thread resumes running just as though nothing had ever happened.

And here's a critical point. From the viewpoint of the many threads all running in the system, the threads themselves are completely oblivious to all of this going on. As far as they are concerned, they're just running and running and running without interruption as if each of them were all alone in the system. The fact that there's an overlord frantically switching among them is unseen by the threads. I just opened Task Manager on my Windows 10 machine. It shows that 2836 individual threads have been created and are in some state of execution. That's a busy system.

So now let's dive deeper into very cool and very important coding technology. The reason for this dive will immediately become clear once we look at the vulnerabilities researchers from VU Amsterdam and IBM have uncovered in ALL contemporary processor architectures – Intel, AMD, ARM, RISC-V ... any architecture that incorporates – get ready for it – speculative execution technology as all modern processor architectures do.

Okay. Imagine that two different threads running in the same system share access to some object. Anyone who has been around computing for long will have heard the term "object oriented programming." An "object" is another abstraction that can mean different things in different contexts. But what's most important is that it's a way of thinking about and conceptually organizing complex systems that often have many moving parts. So when I use the term "object" I'm just referring to something as a unit. It might be a single word in memory, or it might be a large allocation of memory, or it could be a network interface. In the examples we'll be using, our object of interest is a word of RAM memory that stores a count of events. So this object is a simple counter.

And in our system, various threads might, among many other things, have the task of incrementing this counter from time to time. So let's closely examine two of those threads. Imagine that the first thread decides that it wishes to increment the count. So it reads the counter's current value from memory into a register. It might examine it to make sure that it hasn't hit some maximum count value. Then it increments the value in the register and stores it back out into the counter memory. No problem. The shared value in memory was incremented. But remember that in our hypothetical system there are at least two separate threads that might be able to do this.

So now imagine a different scenario. Just like the first time, the first thread reads the current counter value from main memory into one of its registers. It examines its value and sees that it's okay to increment the count. But just at that instant, the system's thread interrupt occurs and that first thread's operation is paused to give other threads the opportunity to run. And as it happens, that other thread that may also decide to increment that counter decides that it wishes to do so. So it reads that shared value from memory, examines it, increments its register and saves it back out to memory. And as luck would have it, for whatever reason, while that second thread is still running, it does that four more times. So the counter in memory was counted up a total of five times. Then after a while, that second thread's time is up and the operating system suspends it to give other threads a chance to run.

Eventually, the operating system resumes our first thread from the exact point where it left off. Remember that before it was paused by the operating system it had already read the counter's

value from memory, examined it and decided that it should be incremented. So it increments the value in its register and stores it back out into memory – just as it did the first time.

But look what just happened! Those five counter increments that were performed by the second thread which shares access to the counter with the first thread were all lost! The first thread had read the counter's value into a register. Then while it was suspended by the operating system, the second thread incremented that shared storage five times. But when the first thread was resumed it stored the incremented counter value that it had previously saved back out, thus overwriting the changes made by the second thread.

This is a classic and clear example of a race condition in software. It occurred because of the one in a million chance that the first thread would be preempted at the exact instant, between instructions, when it had a copy of some shared data that another thread might also modify and it hadn't yet written it back to their shared memory location. The picture I painted was deliberately clear because I wanted to highlight the problem. But in the real world it doesn't happen that way and these sorts of race condition problems are **so** easy to miss. Imagine that two different programmers on the same project were each responsible for their piece of the project and each of them increments that counter. If they didn't realize that the other had code that might also choose to increment that counter, this would introduce an incredibly subtle and difficult to ever find bug. Normally, everything would work perfectly. But then every once in a blue moon the code, which looks perfect, would not do what it was supposed to. And the nature of the bug is so subtle that coders could stare at their code endlessly without ever realizing what was going on.

Last Tuesday was Patch Tuesday and Microsoft delighted us by repairing another 149 bugs which included two that created vulnerabilities which were under active exploitation. Microsoft's software does not appear to be in any danger of running out of bugs to fix, and race conditions are one of the major reasons for that. How many times have we talked about so-called "Use After Free" bugs where malicious code retains a pointer to some memory that was released back to the system and is then able to use that pointer for malicious purposes? More often than not those are deliberately leveraged race condition bugs.

Okay. So now the question is, as computer science people, how do we solve this race condition problem? The answer is both tricky and gratifyingly elegant. We first introduce the concept of **ownership** where an object can only be **owned** by one thread at a time, and only the current owner of an object has permission to modify its value.

In the example I've painted, the race condition collision occurred because two threads were both attempting to alter the same object, the shared counter, at the same time. If either thread were forced to obtain exclusive ownership of the counter until it was finished with it, the competing thread might need to wait until the other thread had released its ownership, but this would completely eliminate the race condition bug. So at the code level, how do we handle ownership?

We could observe that modern operating systems typically provide a group of operations known as synchronization primitives. They're called this because they can be used to synchronize the execution of multiple threads that are competing for access to shared resources. The primitive that would be used to solve a problem like this is known as a "Mutex" which is short for "mutual

exclusion", meaning that threads are mutually excluded from having simultaneous access to a shared object. So, by using an operating system's built-in Mutex function, a thread would ask the operating system to give it exclusive ownership of an object and be willing to wait for it to become available. If the object were not currently owned, then it would become owned by that first requesting thread and that owning thread could do whatever it wished with the object. Once it was finished it would release its ownership, thus making the object available to another thread that might have been waiting for it.

But I said that what actually happens is both tricky and gratifyingly elegant and we haven't seen that part yet. How is a Mutex actually implemented? Our threads are living in an environment where anything and everything that they are doing might be preempted at any time without any warning – this literally happens in between instructions, between one instruction and the next. And in between any two instructions anything else might have transpired. If you think about that for a moment you'll realize that this means that acquiring ownership of an object cannot require the use of multiple instructions that might be interrupted by the operating system at any point. What we need is a single indivisible instruction that allows our threads to attempt to acquire ownership of a shared object while also determining whether they succeeded – and that has to happen all at once.

It is so cool that an instruction as simple as "exchange" can provide this entire service. The exchange instruction does what its name suggests: It exchanges the contents of two "things". These might be two registers whose contents are exchanged, or it could be a location in memory whose contents are exchanged with a register.

We already have a counter, the ownership of which is the entire issue. So we create another variable in memory to manage that counter's ownership, to represent whether or not the counter is currently owned by any thread. We don't need to care which thread owns the counter because if any thread does, no other thread can touch it. All they can do is wait for the object's ownership to be released. So we just need something binary – a '1' or a '0'. If the value of this counter ownership variable is '0' then the counter is not owned by any thread. And if the counter ownership variable is not zero, it's '1', then some thread currently owns the counter. Okay, so how do we work with this and the exchange instruction?

When a thread wants to take ownership of the counter it places a '1' in a register and **exchanges** that register's value with the current value of the ownership variable in memory. If, after the exchange, its register which received the previous value of the ownership variable is a '0' then that means that until the exchange was made the counter was not owned by any thread. And since this exchange placed a '1' into the ownership variable at the same time – with the **same** instruction – our thread is now the exclusive owner of the counter and it is free to do anything it wishes with it without fear of collision.

If another thread comes along and wants to take ownership of the counter, it also places a '1' in a register and exchanges that with the counter's ownership variable. But since the counter is still owned by the first thread, this second thread will get a '1' back in its exchanged register. That tells it that the counter was already owned by some other thread and that it's going to need to wait and try again later. And notice that since what it swapped into the ownership variable was also a '1' this exchange instruction did not change the ownership of the counter. It was owned by

someone else before and it still is. What the exchange instruction did in this instance was to allow a thread to query the ownership of the counter. If it was not previously owned it now would be by that query. But if it was previously owned by some other thread, it still would be.

And, finally, once the thread that had first obtained exclusive ownership of the counter is finished with the counter and is ready to release its ownership, it simply stores a zero into the counter's ownership variable in memory, thus marking the counter as currently un-owned. This means that the next thread to come along and swaps a '1' into the ownership variable will get a '0' back and know that **it** now owns the counter.

We call the exchange instruction when it's used like this an "atomic" operation because like an atom, it is indivisible. With that single instruction we both acquire ownership if the object was free while also obtaining the object's previous ownership status. I suppose I'm weird but this is why I love coding so much.

The holistic way of viewing this is to appreciate that ONLY atomic operations are safe in a preemptively multithreaded environment. In our example, if the counter in question **only** needed to be incremented by multiple threads, and that increment could be performed by a single indivisible instruction, then that would have been "thread safe" (which is the term used). And there are instructions that increment memory in a single instruction. But in our example, the threads needed to examine the current value of the counter to make a decision about whether it had hit its maximum value, and only then increment it. That required multiple instructions and was not thread safe. So what do we do? We protect a collection of non-thread safe operations with a thread-safe operation. We create the abstraction of ownership and use an exchange instruction, which being one instruction is thread safe, to protect a collection of operations that are not thread safe.

And this brings us to "GhostRace", the discovery made by a team of researchers and the question: what would it mean if it were discovered that these critical, supposedly atomic and indivisible synchronization operations were not, after all, guaranteed to do what we thought?

For their paper's Abstract the researchers wrote:

> *Race conditions arise when multiple threads attempt to access a shared resource without proper synchronization, often leading to vulnerabilities such as concurrent use-after-free. To mitigate their occurrence, operating systems rely on synchronization primitives such as mutexes, spinlocks, etc.*
>
> *In this paper, we present GhostRace, the first security analysis of these primitives on speculatively executed code paths. Our key finding is that **all** of the common synchronization primitives can be microarchitecturally bypassed on speculative paths, turning **all** architecturally race-free critical regions into Speculative Race Conditions or SRCs.*
>
> *To study the severity of SRCs (Speculative Race Conditions), we focus on Speculative Concurrent Use-After-Free (SCUAF) and uncover 1,283 potentially exploitable instances in the Linux kernel. Moreover, we demonstrate that SCUAF information disclosure attacks against the kernel are not only practical, but that their reliability can closely match that of traditional Spectre attacks, with our proof of concept leaking kernel memory at 12 KB/s.*

Crucially, we develop a new technique to create an unbounded race window, accommodating an arbitrary number of SCUAF invocations required by an end-to-end attack in a single race window. To address the new attack surface, we also propose a generic SRC mitigation to harden all the affected synchronization primitives on Linux. Our mitigation requires minimal kernel changes and incurs only ≈5% geometric mean performance overhead on LMBench.

And then they finish their Abstract by quoting Linus Torvalds on the topic of Speculative Race Conditions saying: *"There's security, and then there's just being ridiculous."* Right. Maybe not so ridiculous, after all.

What this comes down to is a mistake that's been made in the implementation of current operating system synchronization primitives in an environment where processors are speculating about their future. Somewhere in their implementations is a conditional branch that's responsible for making the synchronization decision and in speculatively executing processors it's possible to deliberately mistrain the speculation's branch predictor to effectively neuter the synchronization primitive. And if that can be done it's possible to wreak havoc.

Here's what the researcher's explained:

Since the discovery of Spectre, security researchers have been scrambling to locate all the exploitable snippets or gadgets in victim software. Particularly insidious is the first Spectre variant (exploiting conditional branch misprediction), since any victim code path guarded by a source if statement may result in a gadget. To identify practical Spectre-v1 gadgets, previous research has focused on speculative memory safety vulnerabilities, use-after-free, and type confusion. However, much less attention has been devoted to other classes of (normally architectural) software bugs, such as concurrency bugs.

To avoid (or at least reduce) concurrency bugs, modern operating systems allow threads to safely access shared memory by means of synchronization primitives, such as mutexes and spinlocks. In the absence of such primitives, e.g., due to a software bug, critical regions would not be properly guarded to enforce mutual exclusion and race conditions would arise. While much prior work has focused on characterizing and facilitating the architectural exploitation of race conditions, very little is known about their prevalence on transiently executed code paths. To shed light on the matter, in this paper we ask the following research questions:

"How do synchronization primitives behave during speculative execution? And what are the security implications for modern operating systems?"

To answer these questions, we analyze the implementation of common synchronization primitives in the Linux kernel. Our key finding is that all the common primitives lack explicit serialization and guard the critical region with a conditional branch. As a result, in an adversarial speculative execution environment, i.e., with a Spectre attacker mistraining the conditional branch, these primitives essentially behave like a no-op (no-op is short for "no operation" means that the synchronization primitives do not synchronize.) The security implications are significant, as an attacker can speculatively execute all the critical regions in victim software with no synchronization.

Building on this finding, we present GhostRace, the first systematic analysis of Speculative Race Conditions (SRCs), a new class of speculative execution vulnerabilities affecting all

common synchronization primitives. SRCs are pervasive, as an attacker can turn arbitrary (architecturally) race-free code into race conditions exploitable on a speculative path—in fact, one originating from the synchronization primitives' conditional branch itself. While the effects of SRCs are not visible at the architectural level (e.g., no crashes or deadlocks), due to the transient nature of speculative execution, a Spectre attacker can still observe their microarchitectural effects via side channels. As result, any SRC breaking security invariants can ultimately lead to Spectre gadgets disclosing victim data to the attacker.

So, essentially, the specter of Spectre strikes again. Another significant exploitation of speculative execution has arisen. And need we echo once again Bruce Schneier's famous observation about attacks never getting worse and only ever getting better?

Under the paper's mitigation section they write:

To mitigate the Speculative Race Condition class of vulnerabilities, we implement and evaluate the simplest, most robust, and generic one: introducing a serializing instruction into every affected synchronization primitive before it grants access to the guarded critical region, thus terminating the speculative path. This provides a baseline to evaluate any future mitigations, and, as mentioned, mitigates not only the Speculative Condition Use After Free vulnerabilities presented in the paper, but all other potential Speculative Race Condition vulnerabilities.

So essentially what they're saying is that, as with all anti-Spectre mitigations, the solution is to deliberately shut down speculation where it's causing trouble. In this instance, this requires the insertion of an additional so-called "serializing" instruction into the code at the point just after the synchronization instruction and before the result of the synchronization instruction is used to inform a branch instruction. This prevents any speculative execution down either path following the branch instruction. The problem with doing that is that speculation exists specifically because it's such a tremendous performance booster. And that means that even just pausing speculation will hurt performance.

To determine how much this would hurt, for example, Linux's performance, these guys tweaked Linux's source code to do exactly that, to prevent the processor from speculatively executing down both paths following a synchronization primitive's conditional branch. What they found was that the effect can be significant. This introduces a 10.82% overhead when forking a Linux process, 7.53% when executing a new process, 12.35% when deleting an empty 0K file and 11.37% when deleting an 10K file. Those are particularly bad. But overall, as they wrote in their abstract, they measured an 5.13% performance cost from their mitigation. That's huge, and it suggests just how dependent upon synchronization primitives today's modern operating systems have become. For a small change in one very specific piece of code to have that much impact means that it's being used constantly.

And finally, as for the disclosure of their discovery, they wrote:

We disclosed Speculative Race Conditions to the major hardware vendors (Intel, AMD, ARM, IBM) and the Linux kernel in late 2023. Hardware vendors have further notified other affected software (OS / hypervisors) vendors and all parties have acknowledged the reported issue (which has been given CVE-2024-2193).

Specifically, AMD responded with an explicit impact statement which was that existing Spectre-v1 mitigations apply, pointing to the attacks relying on conditional branch mis-speculation like Spectre-v1.

The Linux kernel developers have no immediate plans to implement serialization of synchronization primitives due to performance concerns.

It's unclear what the ultimate solution is, here... but it's clear that we don't have it yet.

We have a processor performance addicted industry, that several years ago awoke to the unhappy news that the extremely complex, tricky and sophisticated way our processors had been arranging to squeeze out every last possible modicum of performance was also inherently exploitable, because it opened the processor to side-channel attacks.

It turned out that performance was being enhanced by allowing the processor to dynamically learn from the execution of past code and data. Not only could this stored knowledge reveal secrets about the data that had recently been processed, but it was possible for any software sharing the same hardware to probe for and obtain those secrets.

And that's still where we are today. No one wants to relinquish the performance we've come to need, but the way we're achieving that performance inherently comes at the cost of security and privacy.

https://download.vusec.net/papers/ghostrace_sec24.pdf