Security Now! #937 - 08-29-23

The Man in the Middle

This week on Security Now!

This week we have a really wonderful picture of the week in the form of a techie "what we say" and "what we mean" counterpoint. So we're going to start off spending a bit of time with that. Then we're going to see whether updating to that latest WinRAR version might be more important than was clear last week. And while HTTPS is important for the public Internet, do we need it for our local networks? What about using our own portable domain for eMail? Does Google's new Topics system unfairly favor monopolies? If uBlock Origin blacks ads why does it also need to block Topics? Just how narrow (or wide) is Voyager 2's antenna beam and what does 2 degrees off-axis really mean? Do end users need to worry about that wacky Windows time setting mess? And what's the whole story about Unix time in TLS handshakes? What can be done about fake mass storage drives flooding the market? And finally, let's look at man-in-the-middle attacks. How practical are they and what's been their history?

Our picture of the week is so wonderful it needed an entire page to itself...

What we say	What we mean
Horrible hack	Horrible hack that I didn't write
Temporary workaraound	Horrible hack that I wrote
It's broken	There are bugs in your code
It has a few issues	There are bugs in my code
Obscure	Someone else's code doesn't have comments
Self-documenting	My code doesn't have comments
That's why it's an awesome language	It's my favorite language and it's really easy to do something in it
You're thinking in the wrong mindset	It's my favourite language and it's really hard to do something in it
I can read this Perl script	I wrote this Perl script
I can't read this Perl script	I didn't write this Perl script
Bad structure	Someone else's code is badly organised
Complex structure	My code is badly organised
Bug	The absence of a feature I like
Out of scope	The absence of a feature I don't like
Clean solution	It works and I understand it
We need to rewrite it	It works but I don't understand it
emacs is better than vi	It's too peaceful here, let's start a flame war
vi is better than emacs	It's too peaceful here, let's start a flame war
IMHO	You are wrong
Legacy code	It works, but no one knows how
^X^Cquit^\[ESC][ESC]^C	I don't know how to quit vi
Suboptimal implementation	The worst errors ever inflicted on humankind
These test environments are too brittle	Works on my machine. Have you tried restarting it?
Proof-of-Concept	What I wrote
Perfect solution	How Sales & Marketing are promoting it

Security News

WinRAR v6.23

Last week I mentioned the recent release of WinRAR v6.23 and I noted that it fixed a pair of critical vulnerabilities that could have been used to execute code on its user's systems. Then came the news that this pair of vulnerabilities had been discovered by bad guys and had been actively exploited at least as far back as last April. BleepingComputer's headline last Wednesday was *"WinRAR zero-day exploited since April to hack trading accounts"*. Since this really forms a cautionary tale which might catch any of us if we were to even briefly drop our guard, I want to share some of the details which BleepingComputer shared from Group-IB who made the original discovery. I've edited that BleepingComputer wrote for the podcast. So...

A WinRar zero-day vulnerability tracked as CVE-2023-38831 was actively exploited to install malware when clicking on harmless files in an archive, allowing the hackers to breach online cryptocurrency trading accounts. The vulnerability has been under active exploitation since April 2023 being used to distribute various malware families, including DarkMe, GuLoader, and the Remcos RAT.

The WinRAR zero-day vulnerability allowed the threat actors to create malicious .RAR and .ZIP archives containing innocuous files, such as JPG images, text files (.txt), or PDF (.pdf) docs. However, when a user opens a file, the flaw will cause a script to be executed that installs malware on the device. BleepingComputer tested a malicious archive shared by Group-IB, who discovered the campaign, and simply double-clicking on a PDF caused a CMD script to be executed to install malware.

The 0-day was fixed in WinRAR version 6.23, released on August 2, 2023, which also resolves several other security issues, including this flaw that can trigger command execution upon opening a specially crafted RAR file. In a report released last Wednesday, researchers from Group-IB said they discovered the WinRAR 0-day being used to target cryptocurrency and stock trading forums, where the hackers pretended to be other enthusiasts sharing their trading strategies.

*Their forum posts contained links to specially crafted WinRAR ZIP or RAR archives that pretended to include the shared trading strategy, consisting of PDFs, text files, and images. The fact that those archives target traders is demonstrated by the forum post titles, like "best Personal Strategy to trade with Bitcoin." The malicious archives were distributed on at least **eight** public trading forums, infecting at least a confirmed **130 traders'** devices. The total number of victims and financial losses resulting from this campaign are unknown.*

When the archives are opened, users will see what appears to be a harmless file, like a PDF, with a folder matching the same file name. However, when the user double-clicks on the PDF, the CVE-2023-38831 vulnerability will quietly launch a script in the folder to install malware on the device. At the same time, these scripts will also load the decoy document so as not to arouse suspicion.

The vulnerability is triggered by creating specially crafted archives with a slightly modified structure compared to safe files, which causes WinRAR's ShellExecute function to receive an

incorrect parameter when it attempts to open the decoy file. This results in the program skipping the harmless file and instead locating and executing a batch or CMD script, so while the user assumes they open a safe file, the program launches a different one.

The script executes to launch a self-extracting (SFX) CAB archive that infects the computer with various malware strains, such as the DarkMe, GuLoader, and Remcos RAT infections, providing remote access to an infected device. The Remcos RAT gives the attackers more powerful control over infected devices, including arbitrary command execution, keylogging, screen capturing, file management, and reverse proxy capabilities, so it could facilitate espionage operations too.

Okay. So think about this. WinRAR has been around forever. It's a trusted and robust archiving utility. I use it because it's highly configurable and when configured to use large compression block sizes and to generate a so-called "solid" archive it can achieve unmatched compression levels.

I'm trying to imagine what would happen if someone who appeared to be trustworthy were to post a link to a .RAR archive in a public forum. I guess my first thought would be to wonder why it wasn't a .ZIP file since that would be more common. But I might assume that the guy was more techie and about archiving since RAR does outperform ZIP. I would hope that I would have the presence of mind to drop anything like that into VirusTotal for its quick analysis and evaluation. (And I'll just note for everyone here that it's possible to avoid even downloading a suspicious file first, since VirusTotal can be given a link to download, obtain and analyze a file without us first needing to do so.) In any event, I can imagine that I might open the RAR file since I trust WinRAR. And that's really the crux of it, right? In this instance, my years-long trust of WinRAR would have been misplaced and if my guard was down that might have bitten me.

I'm not sure what lesson we can take from this. "Never trust nothin – and live in a cave" isn't a practical strategy. But being unfailingly skeptical of anything being offered over the Internet by someone who you don't actually know, probably **is** a practical strategy – in fact it's increasingly a survival strategy. And this points to the other mistake those victims made: They didn't really know the person who was offering the download. They couldn't have or they would have never downloaded the file being offered. The old adage of *"never taking candy from a stranger"* is as useful for adults as well as children.

So I suppose our takeaway is just to refresh our healthy skepticism of the Internet and to try to remember to never drop our guard. It only takes one mistake to ruin our day.

As has been the case recently, I've received so many great feedback questions from this podcast's listeners that we have another listener-driven podcast.

Closing the Loop

John May / @DeusXKing

Steve, I have been watching SN since I retired from my IT job in 2020. In this week's episode, 936, you talked about HTTP going the way of the dodo, if Google has their way. What about on private subnets that cannot be routed over the Internet. Why pop a message or hinder access if the traffic is staying local. I have many local devices I access via HTTP and don't want to add certificates. Hopefully these will be exempt. Glad 999 is not the end.

Yes. The reason we need the encryption and authentication that TLS provides to HTTPS on the Internet is that the flow of packet traffic between the endpoints is out of our control on public networks and potentially exposed to the whims of bad guys. But that's not the case within a closed private network where the network components and all packet transit are local, typically kept within a single residence. As our local devices such as routers, network attached storage, webcams, home assistants and other IoT gadgets have become more capable, control panels laced with lights, buttons and switches have been replaced by web pages published by a simple web server running in the device. And this is the point that John May is making. I'm sure many of us have had to fight with our web browsers when we wish to connect to a local router or other web interface over a simple HTTP connection. And as John May worries, Google appears poised to make this even more difficult in the future.

One solution would be to set policy based upon the distinction between publicly routable and private non-routable network IPs. To be clear, when we're talking about non-routable IPs we're talking about the three networks defined in the early days of the Internet which were specified by RFC 1918. The most common we all see when we're behind a consumer router is the address range that begins with 192.168.0.0, extending through 192.168.255.255. That's a group of 65,536 IPs, though for consumer use we generally keep the third octet fixed at 0 or 1 and use 256 IPs by changing the lower octet. In addition to the 192.168/16 network, RFC 1918 specifies a private network which is 16 times larger than that from 172.16 through 172.31 and the third and final is 16 times larger than that one consisting of all IPs beginning with 10 (dot).

It would be tempting to have browsers willing to simply trust all connections from IPs within those three private ranges and allow HTTP connections there. But doing so would mean that we trust every IP within that range. In a small residential setting that probably makes sense. But many large corporations also use these same private network ranges internally for their intranets. And in such settings, access to raw network traffic is probably not well protected. So eavesdropping becomes feasible and widely allowing HTTP across large corporate networks would probably be extending trust too far.

We typically don't have such security concerns within a small local network, but adopting a general browser policy of not requiring TLS connections for private networks might be too permissive since not all private networks are fully trusted networks. Fortunately, a middle ground that probably makes sense is the use of a self-signed certificate.

As we know, publicly-trusted certificates are signed by a certificate authority that the web browser already trusts. So the browser inherently trusts any certificates that any of its already-trusted certificate authorities have signed. Which brings us to the question, who has

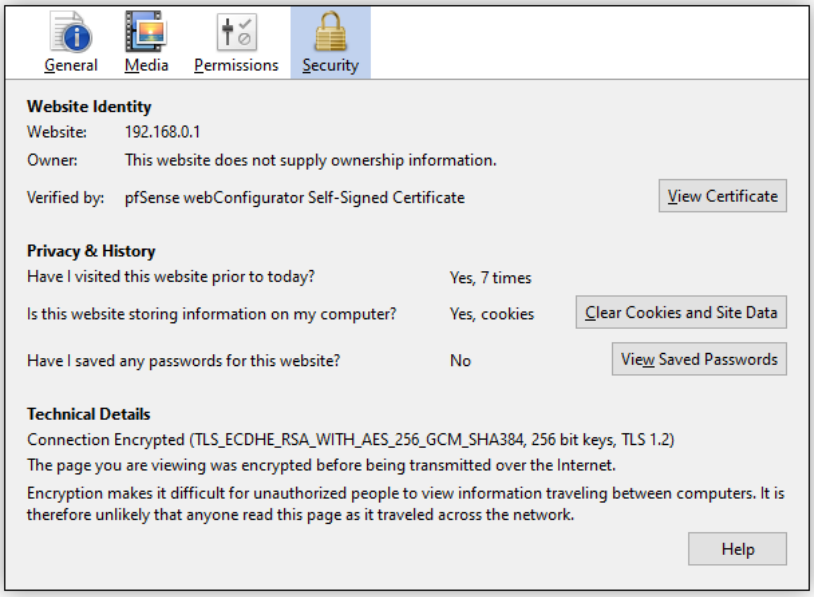
signed those certificate authority certificates? It turns out that in every case the certificate authority has signed its own certificate. Certificate authority root certificates are self-signed and they are trusted simply because they have been placed into the browser's root certificate store.

This means that nothing prevents an appliance like a router or a NAS from creating and signing its own certificate. When connecting over HTTPS with TLS, the first time a browser encounters a self-signed certificate it will balk, complain, and say to its user *"Hey! This guy is trying to pawn off a self-signed certificate which, naturally, wasn't signed by anyone I already trust. What should I do? Do you want me to trust it?"* To which the user can reply *"Yes, trust this certificate from now on."* The procedure varies by browser and version, but the self-signed certificate gets added to the browser's store of trusted root certificates and from that point on it will be possible to establish regular TLS/HTTPS connections with encryption and a limited level of authentication.

I say that it's a limited level of authentication since, as long as the device in question keeps the private key of the certificate it created to itself, no other device on the network or anywhere for that matter can impersonate it. The user's web browser will have stored and been told to trust the web serving device's matching public key. That's what's stored in the root store. So if the user's connection were to be intercepted by some other device, there's no way for it to reuse the trusted device's public key since it would never have access to its matching private key.

Many years ago I used OpenSSL to create a self-signed certificate with an expiration date 100 years in the future. I didn't want to be hassled by the need to keep updating my own self-signed certificates that had expired. It worked great at the time. I haven't tried to do that recently, but I suspect that today's nanny-browsers would complain that the certificate's expiration date is too far in the future. Or maybe self-signed certs are allowed to be exceptions.

In any event, for now we're able to tell our browsers to trust local HTTP connections. In Firefox I get a red slash across the padlock when I connect to my local ASUS router or my local Synology NAS. But when I connect to my local pfSense firewall router, the little padlock shows an exclamation point and if I drill down for more info I see that pfSense created a self-signed cert that I asked Firefox to trust:



The self-signed certificate is valid from Fri, 03 Jan 2020 through Wed, 25 Jun 2025. So that's a nice range. And that suggests that self-signed certs are allowed to have a longer life than current CA-signed certificates which are limited to 13 months or 397 days. Chrome was not happy with the pfSense self-signed certificate since I hadn't bothered to tell Chrome to trust it. But there's a simple process for adding the certificate to Chrome's root trust store.

I can't see any of the browsers removing the ability to add our own trust roots in the form of self-signed certificates. So even if Chrome should finally end all HTTP connections – which still seems unlikely – we should be able to survive by creating our own roots of trust.

Jack Skinner / @jackskinner83

Steve, I was listening to this week's podcast and had some feedback in regards to having 3rd parties host your email vs running your own. There is a 3rd option. Purchase your own domain name and use a 3rd party that allows its use. Then if the 3rd party host shuts down, you simply point your domain name to a new provider and you have access to setup and use those email addresses again.

I think that's a terrific idea! It moves a user's eMail away from the an end user's ISP who is probably hostile to eMail, while retaining full control over the eMail's domain. Bravo. And having a vanity eMail domain with freely created user accounts allows for maximum flexibility. A quick bit of Googling turned up a handful of such services.

Jack Skinner / @jackskinner83

Also, related to Google Topics. With the restriction that an advertiser will only get a topic if they have first seen that user on a different website related to that topic, do you think that this will give Google added power to be a monopoly in this space? It seems like sites will have the incentive to use a single advertiser.

That's an interesting point. The fact that an advertiser is only able to obtain topics for a visitor at a given site when that advertiser has previously seen them at a site that's associated with the topics that were going to be provided, favors larger advertisers with greater reach since the user's browser will have encountered that advertiser at many more websites, thus they will be more likely to qualify for receiving topic information about that user.

The answer to the question Jack asks is unclear to me. If websites are reimbursed by advertisers based upon how well targeted their visitors are – which is to say, how many topics are being returned to the advertisers on their site – then larger advertisers having greater reach would have superior targeting. If they were to reimburse more, then the site might choose to place more of their ads than an advertiser who is paying less. But that logic would appear to apply in today's advertising climate just as much as in tomorrow's climate with Topics. So it's unclear to me how Topics furthers a monopoly.

🇿🇦 Donn Edwards 💙 / @donnedwards

Hi Steve, if uBlock Origin is blocking all the ads on a website anyway, what difference does it make if Google Topics is on or off? As far as I can tell, a typical website doesn't use the topics API, only the sites that actually serve the ads. So if no ads are served, no topics are registered with the browser. Am I wrong?

That's a good point. But at least the way I use uBlock Origin is to allow "Acceptable Ads." I'm not against seeing ads on websites if it helps them to generate revenue. I'm against having things flashing and screaming at me and jumping up and down in a futile attempt to get my attention. The Acceptable Ads initiative identifies advertisers who agree NOT to do any of that. So it's not the case that as a uBlock Origin user I'm seeing no ads, I'm just not being driven into an epileptic seizure by the tame and considerate ads that I do see. https://acceptableads.com/

Philip Le Riche (also @pleriche@mastodon.org.uk)

Voyager2's antenna doesn't produce a pencil beam!! The 3.7 meter dish at S-band (~10cm wavelength) has a limiting resolution of around atan(0.1/3.7) or 1.5 degrees. So who knows, that might be 10dB down at 2 degrees off axis. So not so incredible that they were able to boost the power to shout and get a command through. To say its beam was off the earth by 5AU is therefore somewhat misleading.

I love having the math, Philip! Thanks! And that does explain why they even bothered to shout at Voyager 2. I'm sure they knew exactly how attenuated their signal would be.

Peter G. Chase / @PchaseG

Re: The Windows time problem. I use an ancient program called Dimension 4 which polls various time servers, in my case, every 30 minutes to keep my computers on time. After listening, I went in and turned OFF "Set Time Automatically" in Windows settings. Was this a relevant thing to do, or does this issue just apply to servers? I wasn't clear on that point. Thanks.

This appears to be relevant to Windows 10 and on and Server 2016 and on. I know that my own Win10 workstation had the "UtilizeSslTimeData" value in its registry. However, this is different from "Set Time Automatically" which should be left on. Windows normally references an Internet time server at "time.windows.com". So it makes sense to leave that feature enabled.

SKYNET / @fairlane32

Steve,
I'm curious, is the STS "feature" only in Domain Controllers? I have two servers running Server 2019 Standard and they've never exhibited different clocks. They are not DC's. That said; I do have a bunch of Dell Optiplex's (as you know ;)) that communicate with one of those servers because of our time and print management software for the public access computers. I have had in the past few weeks a few workstations with an error message during

boot up, that say's WARNING, clock not set! And giving me the option to hit F1 to try again or F2 to go into the bios setup to configure. I go to the bios and the clock is off. Not by 137 yrs mind you but it's still off. After resetting it and rebooting its fine. Sometimes I don't even bother, I just reboot the machine and windows boots fine and the clock shows the correct time. Is this behavior because of STS or is the local CMOS chip getting, shall we say, tired and sleepy. 😜 Thanks for any info.

Motherboards generally have a very long-life battery which keeps a very low power oscillator running to keep track of the time and date. But the batteries die over time and sometimes the clock is not read properly. I recently struggled with this with SpinRite since some motherboards disable hardware interrupts and cause time to be lost while SpinRite is running. So I needed to be reading the motherboard's real time clock on the fly, and many are surprisingly flaky. Part of the problem is that this is ancient technology and the real time clock uses what's known as a "ripple counter" instead of a synchronous counter. That means that each digit overflows into the next so when a digit wraps around from 9 back to 0, its carry ripples to the next digit to the left in the counter. So it's possible to sample the count while it's in the middle of updating and "rippling" and to get an incorrect result. So, for SpinRite, I designed another heuristic algorithm to figure out what time it was without being fooled by inaccurate values. And so far it appears that mine was designed somewhat more carefully than Microsoft's for Windows 10 and Server 2016.

Superevr / @superevr

Strangely, the TLS spec (going back to SSLv1) has a client hello that includes the Unix time stamp, but acknowledges that ***"Clocks are not required to be set correctly"*** *In 2013, someone drafted "Deprecating gmt_unix_time in TLS " https://ietf.org/archive/id/draft-mathewson-no-gmtunixtime-00.txt but it doesn't look like it ever left the draft phase. The acknowledgement of its futility raises the question of why is it there at all?*

So this brings us back to last week's discussion of heuristics. I left last week's podcast curious. So I did a bit of Googling. Here's what I learned:

We've spoken of cryptographic protocols which require a nonce. (Nonce standing for Number used once.) Typically, nonces do not need to be secret but they do need to be unique. If nonces are not unique then, in the case of TLS, this opens the opportunity for replay attacks. This brings us to the question of how to produce a nonce that's guaranteed to be unique.

One way, which was suggested in the earliest SSL documents, was to concatenate the UNIX time – a 32-bit binary count that changes once per second – with another 32 bits of random noise. Together, they would produce a 64 bit nonce. Each endpoint's 64-bit nonce would be concatenated with the other's to produce a final 128-bit nonce which is used for the TLS crypto.

When you think about it, if the requirement – and it's a firm requirement – is for an endpoint to never make the mistake of reusing a nonce, one way to do this would be to start with that 32-bit UNIX time which is going to change every second, then have a second 32-bit count which increments once for every new connection. The only way for such a system to issue duplicate

nonces would be if that second 32-bit counter were to wrap around and return to a previous count before the 32-bit UNIX time had changed, which it does once per second. Since a 32-bit count counts from 0 to 4.3 billion before it wraps back around to zero, it would be necessary for a single machine to initiate 4.3 billion TLS handshakes within the span of one second in order for a duplicate nonce to be issued. That's probably not going to happen.

Now, someone might suggest that using a simple counter to form the second half of the 64-bit nonce is a bad idea since it makes the system's nonce easily predictable. This is not a known problem for TLS, but there are simple and secure 32-bit ciphers that could be used to encrypt the 32-bit counter to produce an unpredictable sequence that would therefore never repeat in fewer than 4.3 billion uses.

Okay. But here's the point: The ONLY requirement for the 64-bit handshake nonce is that it never repeats. That's it. The use of UNIX time is merely a suggestion in the TLS specification. It's not a requirement. Which is why the OpenSSL folks decided not to do it.

Back in December of 2013, so nearly a decade ago, two guys, one with the TOR project and the other with Google, produced a TLS Working Group Internet-Draft standard with the title: *"Deprecating gmt_unix_time in TLS"* and the brief abstract reads:

> *"This memo deprecates the use of the gmt_unix_time field for sending the current time in all versions of the TLS protocol's handshake. A rationale is provided for this decision, and alternatives are discussed."*

https://www.ietf.org/archive/id/draft-mathewson-no-gmtunixtime-00.txt

I'm going to share the introduction from this note:

> *Current versions of the TLS protocol, dating back to the SSL 3.0, describe a gmt_unix_time field, sent in the clear, as part of the TLS handshake. While the exact format of this field is not strictly specified, typical implementations fill it with the time since the Unix epoch (Jan 1, 1970) in seconds. This practice is neither necessary nor safe.*
>
> *According to RFC 2246 ("The TLS Protocol Version 1.0"), gmt_unix_time holds "The current time and date in standard UNIX 32-bit format (seconds since the midnight starting Jan 1, 1970, GMT) according to the sender's internal clock. Clocks are not required to be set correctly by the basic TLS Protocol; higher level or application protocols may define additional requirements." This text is retained unchanged in RFC 4346 and in RFC 5246.*
>
> *The gmt_unix_time field was first introduced in SSL 3.0, the predecessor to TLS 1.0. The field was meant to preserve the protocol's robustness in the presence of unreliable random number generators that might generate the same random values more than once. If this happened, then SSL would be vulnerable to various attacks based on the non-uniqueness of the Random fields in the ClientHello and ServerHello messages. Using the time value in this way was meant to prevent the same Random value from being sent more than once, even in the presence of misbehaved random number generators.*

They go on to explain that, as I suspected and mentioned last week, one of the problems with

using UNIX time was that it could make clients identifiable by serving as a form of fingerprint. But mostly they note that the only reason for using UNIX time is that it was a convenient means for changing the bits in half of the 64-bit nonce every second to protect against a repetition in the other 32 bits from a poor pseudo random number generator.

And this brings us back to Microsoft: The bottom line is that Microsoft incorrectly assumed that every incoming TLS connection was **supposed** to contain the 32-bit UNIX time of the other machine. And now we know that's not true. It's at best a suggestion and a convenience. The original spec even explicitly states that connecting machines are not required to have the correct time. It's just meant to be an incrementing value. I'm certain that Microsoft built-in some sanity filtering of their own. That's why it mostly works. And they have that "confidence level" heuristic we talked about last week. But the underlying assumption that handshakes **will** contain GMT time – especially when the massively popular OpenSSL explicitly doesn't – is flawed and then, as we've seen, they are apparently shockingly stubborn about fixing this issue. It's hurting their users and they apparently refuse to care. I wanted to close the loop on the question of TLS handshakes and UNIX time. Now we know.

SpinRite

I mentioned last week that I had something else that I wanted to share which I thought everyone would find interesting. One of our very prolific testers, whose handle in our newsgroup and GitLab is "millQ", reported a month ago on July 27th, that one of his oldest 256MB flash memory devices was showing up RED in SpinRite. RED means that there's something that SpinRite doesn't like about it. None of his other USB flash drives had any trouble. Just that one.

One of the great things about GitLab for me is its issue tracking. I'm desperate not to let any problem fall through any cracks, but I cannot be everywhere all the time. As it happened, when millQ filed his issue, the gang was testing the latest SpinRite build while I was working on the Windows app which installs SpinRite onto USB drives, diskettes and CDs to make any of them bootable. So the DOS side wasn't where I was focused at the time. But once the Windows utility was ready for its first testing I turned the gang loose on it and switched back to SpinRite itself.

The error that SpinRite was reporting was that millQ's FLASH drive had failed SpinRite's transfer confidence testing. Although SpinRite would allow the user to retry the test or to push past the failure, SpinRite couldn't vouch for how it was working with the drive.

Remember that I mentioned last week that SpinRite now has an extremely flexible driver that learns about the environment it's in. In order to make absolutely positively certain that the driver has settled upon the right parameters for a drive and controller, it performs a series of confidence tests on each drive it encounters.

It first fills a 9-sector buffer with pseudo-random noise and copies it to a secondary buffer. It then goes to the end of the drive to read the drive's last nine sectors into the first buffer which should overwrite all nine sectors of pseudo-random data that was pre-loaded there. It then compares the two buffers to verify that none of the sectors match. That assures us that reading from the media into RAM is actually happening.

It then takes the 9 sectors that it read and copies them to the secondary buffer. It inverts the primary buffer and writes the 9 sectors of inverted data back to the drive. It then re-fills the buffer with new pseudo-random data, then re-reads the data, which will now be inverted if it was written, over the buffer. It re-inverts the buffer then compares all 9 sectors with the original drive data that was first read and saved. And it verifies that all 9 sectors match between the two buffers. And, finally, it rewrites the originally read data back to the drive, restoring it to its original contents.

SpinRite goes through those gymnastics to absolutely and positively confirm that it's able to properly both read and write to the drive across multiple sectors. Those tests and comparisons will fail unless that is all working. There are a number of other things that SpinRite does after that, but it was that confidence testing that millQ's FLASH drive was failing.

Several years ago, I might have doubted SpinRite more than the drive. But we now have 764 registered users in GitLab, so SpinRite has been quite well pounded on by now. And we don't see drives failing this. This was very unusual. I should note that read and write errors reported by the drive are **fine** at any time. If the drive itself says it's unable to successfully write or read, that's no problem since that's not what we're testing here. So SpinRite will simply move 9 sectors toward the front of the drive and try again. This entire process is tolerant of storage errors.

Here's the gotcha of millQ's drive: It was failing this test and was not reporting any errors. SpinRite told it to read 9 sectors and it said it did. And SpinRite confirmed that nine sectors had, indeed, been read. Then Spinrite told it to write 9 sectors and it said it did. But when SpinRite asked it to read those 9 sectors back, they were not the 9 sectors that had just been written. This drive was accepting write commands and was apparently ignoring them.

Since SpinRite now has the ability to check an entire drive this way, millQ started SpinRite at its new Level 5 which reads, inverts, writes, reads and verifies, then reinverts, writes then reads and verifies. What millQ discovered was that at exactly the halfway point, the drive **silently** stopped writing. It also sped up considerably, since writing to FLASH is much slower than reading. millQ's drive was labeled 256MB, and a query for the drive's size declared itself to be 256MB, but it was only storing 128MB.

And here's the real gotcha: No operating system would detect this. Operating systems assume that unless a drive reports an error writing data that data was properly written. And modern drives take on a lot of responsibility for making that statement true.

What was diabolical was that the drive appeared to be functioning perfectly. It was formatted with a FAT file system and millQ had used it to successfully store and retrieve a great deal of data through the years. But perhaps he had never stored more than 128MB of data.

When this was discussed in GRC's SpinRite.dev newsgroup, many people there, some who have a deep background in data recovery, were not at all surprised because it turns out that – and this is why I'm bringing this up – ***there is an epidemic of fake drives flooding the retail market.*** I suppose, in retrospect, we shouldn't be surprised. But I wanted to make sure all of our listeners here were aware.

Since the FAT file system that's the default on such drives places all of its file system metadata at the front of a drive – the famous FAT File Allocation Tables which give the file system its name – and the root directory, etc. – a 16MB compact flash card might have its firmware tweaked to lie about its size, then be relabelled and sold as a 512MB compact flash card. Such a card will format without any trouble and it will appear to be working perfectly. Then a wedding photographer sticks this brand new card into their camera and spends Saturday taking photos before, during and after a wedding, only to later discover that although a directory listing shows that all of the photos taken are there, the actual content of those files is not. And those precious memories are irretrievably lost forever.

This is a surprisingly common occurrence. Just Google "fake sdd" or "fake thumbdrive" and be prepared to get bored scrolling the screen. It's endless.

When this came up last week I jumped onto Amazon and purchased the cheapest 2TB thumb drive they had. It's silver and beautiful, with that blue USB tongue to indicate USB 3.0. And that 2 terabytes cost me $26.88. It arrived last Friday. So I plugged it into SpinRite and the screen immediately turned red. That drive immediately failed SpinRite's end-of-drive data storage test.

I was curious about what it was doing, so I stepped through the writing and reading and writing process at the end of the drive; and its behavior was bizarre. It seemed to sometimes write, but not reliably. And as I kept poking at it, it suddenly stopped failing and began working correctly – at least for those 9 sectors at the end. Were some bad spots remapped so that now they're good? Maybe. But remember that we're talking **16 TRILLION** bits of storage in a tiny sliver of metal-enclosed plastic. I'm skeptical that such a drive contains much wear leveling technology. I'm skeptical that it contains 16 TRILLION bits, but we'll get to that in a moment. What I know is that it was not initially working and after a bit of exercising it began to appear to work – at least at the end of the storage region. If other "unexercised" areas of the drive function similarly, relying upon this brand new drive from Amazon would be a disaster. And remember, writing to it and reading from it produce no errors. So there's no indication that it's not storing everything when, in fact, it may not be storing anything.

One possibility is that not all FLASH storage is created equal. Chips are manufactured in large wafers then cut into individual pieces and tested. So there's some after manufacturing testing that's done to qualify each chip for use. What happens to chips that fail to make the grade? Say that they don't get an F, but perhaps they get a D-. You have to imagine that there's a market for chips that fail to make the grade. Someone will buy them and package them for sale on Amazon or eBay. It looks like memory. It just isn't very good or reliable. And maybe it isn't memory at all.
But I think this is a relatively new phenomenon that has arisen in the last few years. I'm inclined to believe that millQ's drive, since it's so very old, may have just failed. It probably once was a 256MB drive and the chip that was providing the second half of its storage died. But there is absolutely no doubt that fake mass storage is a very real problem today.

This was sold as a high-end SSD. Nope.

So, the first question is "how can one tell if a solid state memory is fake?" Depending upon the cleverness of the memory controller, which is to say, to which lengths its perpetrator goes, there's actually only one way to know for **sure**, which is to simultaneously have every sector of the memory committed and in use and then verified. You'd fill the entire memory with deterministic pseudo-random data – this is done to prevent data compression which is apparently another trick that's employed by some cheaters. Once filled in this way, the entire memory is then read to verify that it still contains that deterministic pseudo-random data. This is the capability and the capacity that the memory claims it offers and it's what you've purchased. Anything less is a cheat, and there's no way to fake-out that test. If you really want to know for sure, it's necessary to do this because a much smaller memory could be arranged as an MRU – a Most Recently Used – cache so that moving through the memory reading and writing it piecemeal would be fooled, even though the entire memory could not be used at once because it didn't exist. Performing this true full test is beyond the scope of SpinRite 6.1. But now that I know this is happening out in the world, SpinRite 7 is going to incorporate this full pseudo-random fill and verify technology because this is the sort of ultimate assurance that SpinRite should be able to offer.

Having said that, in today's reality, it seems unlikely that scammers are going to go to the trouble of engineering an MRU cache to fake out lesser tests. As it is today, SpinRite 6.1 will instantly disqualify any fake storage that places its reduced memory at the front of the drive. SpinRite 6.1 will instantly detect that there's no storage at the end of the drive just as it did for millQ. And since this has been a recognized problem for a while, a handful of freeware utilities have been created to detect fake solid state storage. I haven't looked at this, so I can't vouch for their operation or behavior. But they exist and they're easy to find.

I don't know what I purchased from Amazon for $26.88. It looks all shiny and new, but no way can it be trusted to store data.

The Man in the Middle

Today's topic was inspired from another question from one of our listeners:

Jon David Schober / @jondavidschober

> *Hey Steve. Quick question regarding insecure HTTP traffic. Even if the site has no user interaction (logins), wouldn't it being HTTP still put the user at risk for malicious injection of stuff like JavaScript miners, or overriding downloads to include malware? Seems like something that HTTPS would secure even if the site were just read-only.*

That's certainly a great point and many of our listeners wrote to say the same thing. So I chose Jon's at random to represent all those who asked essentially the same question. And everyone is correct in their observation that the lack of authentication and encryption would mean that if someone could arrange to place themselves in a man-in-the-middle position to intercept traffic flow then an unsecured conversation could be altered.

So I wanted to take a moment to expand upon this. To first establish a bit of perspective and to also examine the practicality of man-in-the-middle attacks which are, I think, far more easily said than done. It's their practical difficulty that explains how we survived in a mostly unsecured Internet until Let's Encrypt came along.

We should remember that the lack of persistent encryption was the way most of the Internet worked until only very recently – and things mostly worked just fine without everything encrypted all of the time. As those of us who have been around for a while will recall, it was once the case that encryption was only employed on most websites when a login form was being delivered and its user's name and password contents were being submitted. The sense was that it was only during that brief interchange where eavesdropping really needed to be prevented. I'll be the first to say that was probably never really true. But at all other times, most websites were using more economical HTTP connections whose traffic was not safe from eavesdropping.

Our long-term listeners will recall "FireSheep", a Firefox add-on which provided a convenient user-interface for hijacking other people's logged-in session cookies, and thus their logged-in sessions. As I reported at the time, I tried it at my local Starbucks. I never impersonated anyone of course, but the FireSheep user-interface quickly populated with all of the logged-on accounts of the other patrons with whom I was sharing free WiFi. And all I would have had to do was click on one of those icons to be logged into the websites they were using, as them. It was a sobering experience.

Today, thanks to the pervasive use of TLS and HTTPS, we have communications privacy so that scenario cannot happen. So it's definitely a good thing that we have more security today than we did in the past. No question about it. But the use of HTTPS with TLS is also not an absolute guarantee. I'll remind our listeners of something that we covered at the time. A few years ago, the headline of a story in BleepingComputer read: *"14,766 Let's Encrypt SSL Certificates Issued to PayPal Phishing Sites."* 14,766.

Every one of those connections to those 14,766 fake PayPal sites was authenticated, encrypted,

super secure, immune to any man-in-the-middle interception... and malicious.

And, in fact, we know that the reason our web browsers have been gradually downplaying the increasingly pervasive use of HTTPS – which boasts its padlock – is to not give their users a false sense of security that just because there's a padlock all is well.

So how was it that we survived as long and as well as we did before most websites had their communications encrypted all of the time? The answer is, there's a world of practical difference between passive eavesdropping and active interception. Eavesdropping, as in the Firesheep example, is trivial, whereas active traffic interception is not something that's practically available to common malicious hackers. And this explains how the pre-encrypted Internet managed to survive.

We've often talked about practical traffic interception. Once upon a time, antiviral content scanners were installed at the network borders of corporations to examine and filter all of the traffic passing through, on the fly. Encrypted connections posed a problem. So the solution was to create "middle boxes" which would fully proxy such connections by intercepting and terminating TLS handshakes at the box, then establish a second connection to their user's browser within the organization. Since the traffic-intercepting middlebox needed to create certificates to stand-in for those remote websites it was proxying, it needed to have some means for browsers to trust the certificates it was creating and presenting. The controversial and underhanded way this was done was by obtaining a certificate from an already trusted certificate authority where that certificate itself had certificate signing authority. In that way it was possible for the middlebox to transparently create certificates as if it were a trusted authority. The industry quickly clamped down on that practice since allowing random hardware able to create fully trusted certificates was extremely dangerous and prone to abuse. So today, any TLS proxying middlebox that wishes to create trusted certificates on behalf of other websites can only do so by installing its own self-signed root certificate into every browser's trusted root store. This essentially makes it a trusted certificate authority, but only within the organization that's behind, and being protected by, the middlebox.

So both in the earlier unencrypted days and today, when nearly everything is encrypted, it's possible and practical for hardware to be installed into a network which allows for traffic interception, filtering and potentially, alteration. But, again, such interception hardware resides in a privileged position that's not available to hackers.

I was trying to think of an instance where traffic was actually being intercepted against the wishes and best interests of its users and I remembered that there was a time when some slimy ISPs were caught injecting their own crap into the unencrypted HTTP web pages of their customers. Now **there** is a perfect example of why we would want and would benefit from having end-to-end encryption of our web traffic. No one wants to have their ISP monitoring their activity, let alone injecting their own crap into content that our web browsers receive.

So, yes, there is an example, which was not theoretical, of an entity in a privileged position – our ISP whom we're paying to carry our data and connect us to the Internet – betraying our trust and not only spying on our activity but injecting its own traffic into our connections.
To actually pull off a man-in-the-middle attack requires that the attacker arrange to have a

privileged position where they're able to intercept, capture, buffer and analyze the traffic that's flowing past. It's easier said than done, but as we've always seen, where there's a will there's a way. And as those examples show, adding encryption to connections makes that job decidedly more difficult.

In doing a little more brainstorming, I came up with one way that Google's new "always try HTTPS first" approach would help. One of the other attacks we talked about in the past was the protocol downgrade. In such an attack, an attacker would arrange to convert all of a webpage's HTTPS URLs to HTTP. The browser wouldn't know any different and would make its queries over HTTP. But with Chrome's automatically try HTTPS, those rewritten HTTP's would be treated as HTTPS's, thus thwarting the attempt to downgrade connections for the sake of keeping a browser's communications unencrypted.

I think that what Google is doing is a good thing. Trying to upgrade an HTTP URL to HTTPS prevents any protocol downgrade horseplay, and provides more security for its users. And like many of our listeners who hope that we don't lose all use of good old HTTP – especially for local connections to our own appliances where encryption and authentication offer little, if any, value – I doubt that losing HTTP is in the cards anytime soon. After all, it took quite a while for FTP to finally be deprecated and removed from our browsers and the File Transfer Protocol was never as popular as HTTP remains today. I'm somewhat surprised by Google's stated 5 to 10% of sites visited by Chrome's users remain HTTP. That seems quite high today.

So... Does everything need to be HTTPS? I'm still not convinced. I understand the arguments and I agree with our listeners who pointed to the potential for man-in-the-middle attacks. We've just reviewed how difficult, but possible, such attacks actually are. They require an attacker to be in a privileged position with some serious hardware. HTTPS makes that more difficult but not impossible in every case.

It seems to me that the way things are today is probably right. If some random website, like the example I gave last week of ctyme.com, which publishes that wonderful list of PC and DOS interrupt APIs, doesn't care enough to encrypt their site even now that certificates to do so are freely available with minimal effort, then that ought to be up to them. Am I going to avoid using such a site because it's not encrypted? No. I'm put in much greater danger by downloading files over **HTTPS** from file archives. That's truly terrifying and I avoid doing so at all cost; but sometimes there's no choice.

As for HTTP-only sites, I think it ought to be the site's and their visitor's decision whether the site wants to remain unencrypted and whether their visitors want to visit. Overly demonizing the lack of authentication and encryption seems wrong, especially when Let's Encrypt is cranking out fraudulent TLS certificates at a phenomenal rate which are being used to spoof legitimate websites with full authentication and encryption.