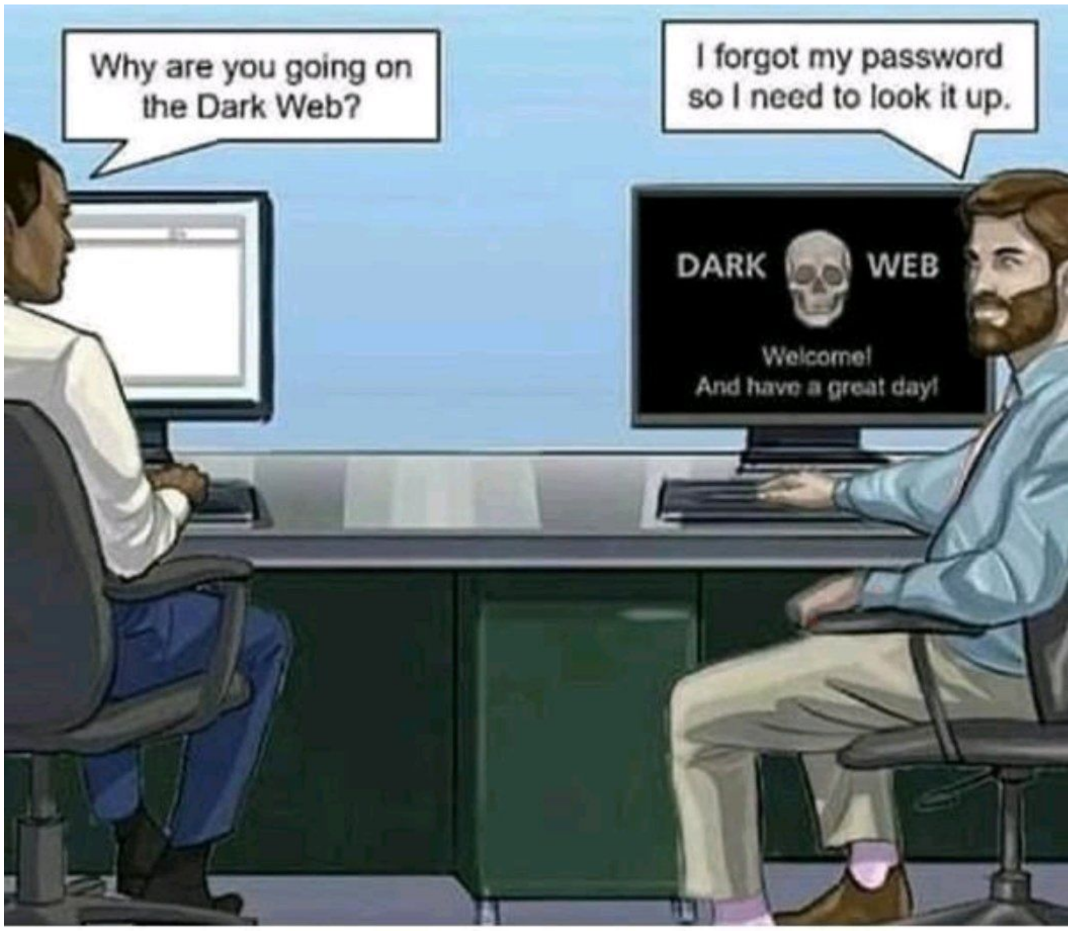
Security Now! #804 - 02-02-21
NAT Slipstreaming 2.0

This week on Security Now!

This week we examine another instance of a misbehaving certificate authority losing Chrome's trust. We cover a number of serious new vulnerabilities including an urgent update need for the just released Gnu Privacy Guard, another supply-chain attack against end users, a disastrous 10-year old flaw in Linux's SUDO command, and thanks to Google, some details of Apple's quietly redesigned sandboxing of iMessage in iOS 14. I'm going to share something that I think our listeners will find quite interesting about some recent architectural decisions for SpinRite, and then we'll conclude with a look at the inevitable improvement in NAT bypassing Slipstreaming.


Why are you going on the Dark Web?
I forgot my password so I need to look it up.
DARK WEB
Welcome!
And have a great day!

Browser News

Chrome rescinding another CA's root cert

As we know, our certificate-driven web server system of trust is based upon a chain-of-trust model where each chain is anchored by a signing authority's root certificate. That root cert contains their public key which matches their secret key used to sign the certificates which are presented by web servers. With that model, any web server presenting an unexpired identity-declaring certificate, that has been signed by any of the browser's trusted certificate authorities, will be trusted.

Several times in the past few years we've covered the interesting and often fraught news of signing authorities either deliberately abusing or inadvertently failing to properly authenticate the identities of those whose certificates they sign. The inevitable result is that they lose the privilege of having their signatures trusted by the industry's web browsers which renders their signatures useless and therefore worthless.

And today, we are there again. Google has just announced that they intend to ban and remove support from Chrome for certificates issued by the Spanish Certificate Authority “Camerfirma.” That revocation of trust will go into effect once Chrome 90 hits the releases channel in mid-April, two and half months from now, since it will no longer trust any Camerafirma signatures. This means that none of the otherwise-valid TLS certificates that have previously been signed by Camerfirma will be seen as valid by Chrome. So all of the web servers currently serving certificates with Camerfirma certificates will be invalid for all Chrome users.

Since untrusting any CA's root cert instantly ends that aspect of the CA's business — and also punishes their previous customers whose signed certificates no longer function — the decision to do this is always made only after the faulting party has been warned many times and only after it has become clear that their conduct is placing the greater Internet at risk.

In this instance, the final decision to drop trust for Camerfirma's certificates comes only after the company was given more than six weeks to explain a string of 26 incidents related to its certificate-issuance procedures. These incidents, detailed by Mozilla, date back to March 2017.

https://wiki.mozilla.org/CA:Camerfirma_Issues

The two most recent problems occurred last month, in January, even after Camerfirma was made aware it was under probationary investigation the month before, in December 2020. The incidents paint a clear and disturbing picture of a company that has failed to meet industry-agreed quality and security standards in return for the privilege of issuing TLS certificates for website operators, software makers, and enterprise system administrators. As we know, in one sense traditional Certificate Authorities are printing money. But in return for the ability to do so, they must adhere to strict quality control and conduct standards. Looking over the list of Camerfirma's 26 transgressions, it appears very clear that they have failed, over many years, to deserve the initial presumption of trust that they were given by the industry's web browsers. And so, that trust is being rescinded.

I've read over the detailed description of their many various failings through the years and I think it's useful and educational for us to get some sense for them. So, for example:

Issue R: Failure to disclose unconstrained sub-CA (DigitalSign) (2018)

In April 2018, Camerfirma failed to disclose an unconstrained sub-CA, despite Mozilla requiring in November 2017 that such disclosures be complete by 2018-04-15.

No explanation was provided by Camerfirma as to the cause of this omission nor were effective controls provided that would prevent such Mozilla policy violations in the future.

Issue T: Failure to disclose unconstrained sub-CA (MULTICERT) (2018 - 2020)

In the course of resolving Issue R, it was further discovered in July 2018 that Camerfirma had failed to disclose two additional sub-CA certificates, operated by MULTICERT. Mozilla Policy required that such sub-CAs be disclosed within one week of creation.

Camerfirma's explanation at the time was that they failed to consider that the person responsible for disclosing into CCADB would be unavailable, and that the backup for that person would be unavailable. They resolved this by adding a backup to the backup, in case the backup fails.

However, the disclosure in the CCADB turned out to be incorrect and misleading, as Camerfirma disclosed that they operated the sub-CA, when in fact it was externally operated. At the time, this was only detected because Microsoft had disclosed the CP/CPS they had for MULTICERT's root, which participated in Microsoft's root program. Camerfirma's explanation was that the person responsible for disclosing was overloaded, and that three people would be responsible for disclosures going forward.

In 2019, Camerfirma failed to provide correct audits for MULTICERT. Their explanation was that they only had one person responsible for communicating with their sub-CAs, and had failed to consider that the person responsible for communicating with sub-CAs and disclosing into CCADB would be unavailable. They stated that they intended to prevent such issues from recurring in the future by purporting to implement additional steps.

In 2020, Camerfirma again failed to properly disclose sub-CAs operated by MULTICERT, erroneously reporting them as covered by Camerfirma's CP/CPS. Their stated reason was because these new sub-CAs were not covered by a new audit from MULTICERT yet, although the expectations for how to disclose that had been previously communicated by Kathleen Wilson.

Issue X: MULTICERT Misissuance (2018 - 2019)

In August 2018, within weeks of having received a cross-signed sub-CA from Camerfirma (Issue T), MULTICERT misissued several certificates that violated the ASN.1 constraint for organizationName field length. Camerfirma's response was that both they, and MULTICERT, would regularly check crt.sh for certificate lint violations.

In October 2018, it was discovered that MULTICERT had misissued 174 certificates with an incorrect qcStatements extension. In response to the report that was provided, MULTICERT revoked the certificates, and then misissued new certificates with a validity period greater than 825 days to replace these the following month.

Further, after reportedly fixing the underlying issue, MULTICERT again misissued another certificate with a validity period greater than 825 days.

During the course of this investigation, MULTICERT also failed to revoke on the timeline required by the BRs, in order to give the customer more time to replace certificates.

In March 2019, it was discovered that MULTICERT's certificates also had insufficient entropy, containing only 63 bits of entropy, rather than the required 64.

Those are just three of the 26 issues that have been compiled since 2017. And, maybe having 63 bits of entropy rather than 64 is not a big deal. But the CA business — which, when performed properly, can be a money-printing goldmine — really is all about details and rule following. Every rule has been established for some good reason. And if a CA is sloppy about the amount of entropy in their certs, what else are they doing wrong?

Through the years we've discussed the walk of shame made by the StartCom subsidiary of WoSign, DigiNotar and even Symatec. In each case trust in their certs was revoked. DigiNotar filed for bankruptcy and Symantec sold its certificate authority business to DigiCert.

So far, the other browser makers have been silent on Camerfirma, but Google with Chrome has clearly made the right decision. So I'd expect to see similar announcements made by Apple, Microsoft, and Mozilla before long.

After Camerfirma is gone there will still be plenty of certificate authorities available for browsers and consumers both to trust. The best thing that comes from this sort of banishing, is the sobering reminder to all of the other CA's, who might be in danger of thinking that the privilege they enjoy to print money, is a right. That would be wrong.

Security News

An urgent update to the recently released GnuPG.

Version 1.9.0 of the famous and highly used Gnu Privacy Guard (GPG) was recently released on January 19. “Libgcrypt” is an open source crypto library and a module of GPG. Then, last Thursday, old old Tavis Ormandy, whom we haven't heard much from recently, of Google's Project Zero, publicly disclosed the existence of a “heap buffer overflow” in libgcrypt due to an incorrect assumption in the block buffer management code.

Tavis wrote: “Just decrypting some data can overflow a heap buffer with attacker-controlled data, no verification or signature is validated before the vulnerability occurs. I believe this is easily exploitable.”

Tavis forwarded his findings to the developers responsible for libgcrypt. And as soon as the report was received the team published an immediate notice for users:

“[Announce] [urgent] Stop using Libgcrypt 1.9.0!”

In the advisory, principal GnuPG developer Werner Koch asked users to stop using version 1.9.0, which, as a new release had begun to be adopted by projects including Fedora 34 and Gentoo. A new version of libgcrypt, version 1.9.1, was released in a matter of hours to address the severe vulnerability which was even too new to have a CVE number assigned.

In an analysis of the vulnerability, cryptographer Filippo Valsorda suggested that the bug was caused by memory safety issues in C and may be related to efforts to defend against timing side-channel attacks. So we have an instance of an attempt to mitigate one potential threat creating a new very real vulnerability.

As a consequence, users who had upgraded to libgcrypt 1.9.0 are being urged to download the patched version as quickly as possible. The GPG developers agreed with Tavis' assessment of the bug's severity. They said: “Exploiting this bug is simple and thus immediate action for 1.9.0 users is required.”

Thank goodness — or thank Google — for Travis. But you have to wonder what would have happened if Tavis had not been on the ball. How long would this newly introduced SERIOUS flaw have lingered within the GPG code?

It's clear that when a skilled developer examined the code without the inherent bias of the code's author, the problem was apparent. That suggests that malicious coders — whom we know are unfortunately every bit as talented as the world's best — might also be scouring the world's open source looking for exactly such flaws. To me this suggests two things:

First, we really need to appreciate how brittle our current coding and code is and try much harder to leave things alone that are already tried, tested and true. Stop messing with code that works. Stop trying to make things better that are just fine the way they are. Really.

The second thing is that we really need to reconsider our coding practices. We need implementation languages that inherently do not allow these sorts of mistakes to be make. Such languages exist but they're not macho and fun to use. Consequently a highly critical security-oriented cryptographic library for the widely used Gnu Privacy Guard is written in 'C' — the lowest level most macho language with the worst security track record. That could be owing to the fact that so much code is still being written in C. But it's also owing to the fact that the language provides exactly ZERO protection from doing anything the programmer wants. Which is why it's appealing to cowboys. (And yes ... I know ... I wrote SQRL in assembler, so who am I to speak about the need to use safe high level languages when security matters? I'm just saying... everybody **else** should do it! :) And that's, of course, the problem: Everybody thinks that everybody else should do it... with the foreseeable outcome that still, today, no one does.

So, good luck and god bless... and be sure to update your GPG code.

An interesting supply-chain attack
“BigNox” is a Hong Kong based company which publishes, among various products, an Android emulator for PCs and Macs called NoxPlayer. Their website claims that they have over 150 million users in more than 150 countries speaking 20 different languages. Though the BigNox follower base is predominantly Asian. The NoxPlayer is generally used by gamers to play mobile games on their PCs.

The discoverers of something fishy going on were our friends and recent TWiT network sponsor, ESET. They spotted a highly-targeted surveillance campaign involving the distribution of three different malware families via tailored malicious updates to selected victims based in Taiwan, Hong Kong, and Sri Lanka.

ESET spotted the first signs of something going on last September, and the compromise continued until “explicitly malicious activity” was uncovered last week, which prompted ESET to report the incident to BigNox.

ESET wrote:

> “Based on the compromised software in question and the delivered malware exhibiting surveillance capabilities, we believe this may indicate the intent of intelligence collection on targets involved in the gaming community.”

To carry out the attack, the NoxPlayer update mechanism served as the vector to deliver Trojanized versions of the software to users that, upon installation, delivered three different malicious payloads such as Gh0st RAT — used to spy on its victims, capture keystrokes, and gather sensitive information.

Separately, they found instances where additional malware like the PoisonIvy RAT was downloaded by the BigNox updater from remote servers controlled by the threat actor.

ESET said:

> “PoisonIvy RAT was only spotted in activity subsequent to the initial malicious updates and downloaded from attacker-controlled infrastructure.”

Unfortunately, BigNox was not very helpful when they were contacted by ESET who wrote: “We have contacted BigNox about the intrusion, and they denied being affected. We have also offered our support to help them past the disclosure in case they decide to conduct an internal investigation.”

So, if anyone hearing this is a users of BigNox's NoxPlayer, you probably need to take responsibility yourself for making sure you didn't receive malware. I have a link in the show notes to the ESET report which provides of IOCs — Indicators of Compromise — which anyone who's worried can check:
https://www.welivesecurity.com/2021/02/01/operation-nightscout-supply-chain-attack-online-gaming-asia/

The intrusions appear to be gaming-world centric and highly targeted. ESET wrote:

“In comparison to the overall number of active NoxPlayer users, there is a very small number of victims. According to ESET telemetry, more than 100,000 of our users have Noxplayer installed on their machines. Among them, only 5 users received a malicious update, showing that Operation NightScout is a highly targeted operation. The victims are based in Taiwan, Hong Kong and Sri Lanka.

We were unsuccessful finding correlations that would suggest any relationships among victims. However, based on the compromised software in question and the delivered malware exhibiting surveillance capabilities, we believe this may indicate the intent of collecting intelligence on targets somehow involved in the gaming community.”

In their posting, ESET details and diagrams the precise operation of the NoxPlayer update system, and concludes:

“We have sufficient evidence to state that the BigNox infrastructure (res06.bignox.com) was compromised to host malware, and also to suggest that their HTTP API infrastructure (api.bignox.com) could have been compromised. In some cases, additional payloads were downloaded by the BigNox updater from attacker-controlled servers. This suggests that the URL field, provided in the reply from the BigNox API, was tampered with by the attackers.”

So, here is a much smaller-scale version of the now-infamous SolarWinds intrusion which, one way or another, leveraged the software updating channel to quietly slide malware into a victim's machine.

Once upon a time, I believed that I could successfully take responsibility for not getting my system infected. We've talked about this through the years. Don't download anything that a website tells you that you need. That's never going to end well. Don't click on links in sketchy eMail. Be very wary of anything you download from a 3rd-party download repository. Try to only obtain things directly from their publishers... and so on.

But today, we all need to face the fact that we no longer have that control over our own machines’ destinies. Many of the tools, utilities and widgets that we use are being periodically updated. Notepad++ comes immediately to mind for me since it is constantly wanting to update itself. Not only is that annoying, since it seems to be working just fine for me, but it's also **inherently** dangerous. If their build process or update server gets compromised, in a very short time a huge number of Windows users will be infected. This makes them a high value target.

The solution for me is Windows Defender, which I no longer only recommend to others. I depend upon it as well. And since at the moment I'm sitting in front of a Win7 machine, I'm thankful to Microsoft for continuing to update Win7's Defender, even if they’ve otherwise abandoned it.

I was recently stating that any responsible company now needs to be performing continuous network intrusion detection surveillance because defenses have become too porous compared to the external pressure to get in. In an exact analogy on a personal level, today's end users must deploy their own local surveillance within their machines. Even someone who never clicks the wrong link or deliberately downloads a malicious Trojan can still be infected by an autonomous software update — and it’s gonna happen.

Apple quietly put iMessage in a sandbox in iOS 14.
The new code was discovered, reverse-engineered, described and documented by Google Project Zero's Samual Groß (pronounced "Gross" — thanks Elaine!).

https://googleprojectzero.blogspot.com/2021/01/a-look-at-imessage-in-ios-14.html

Samuel wrote:

On December 20, Citizenlab published "The Great iPwn", detailing how "Journalists [were] Hacked with Suspected NSO Group iMessage 'Zero-Click' Exploit". Of particular interest is the following note: "We do not believe that [the exploit] works against iOS 14 and above, which includes new security protections''. Given that it is also now almost exactly one year ago since we published the Remote iPhone Exploitation blog post series, in which we described how an iMessage 0-click exploit can work in practice and gave a number of suggestions on how similar attacks could be prevented in the future, now seemed like a great time to dig into the security improvements in iOS 14 in more detail and explore how Apple has hardened their platform against 0-click attacks.

The content of this blog post is the result of a roughly one-week reverse engineering project, mostly performed on a M1 Mac Mini running macOS 11.1, with the results, where possible, verified to also apply to iOS 14.3, running on an iPhone XS. Due to the nature of this project and the limited timeframe, it is possible that I have missed some relevant changes or made mistakes interpreting some results. Where possible, I've tried to describe the steps necessary to verify the presented results, and would appreciate any corrections or additions.

The blog post will start with an overview of the major changes Apple implemented in iOS 14 which affect the security of iMessage. Afterwards, and mostly for the readers interested in the technical details, each of the major improvements is described in more detail while also providing a walkthrough of how it was reverse engineered. At least for the technical details, it is recommended to briefly review the blog post series from last year for a basic introduction to iMessage and the exploitation techniques used to attack it.

Overview

Memory corruption based 0-click exploits typically require at least the following pieces:

- A memory corruption vulnerability, reachable without user interaction and ideally without triggering any user notifications

- A way to break ASLR remotely

- A way to turn the vulnerability into remote code execution

- (Likely) A way to break out of any sandbox, typically by exploiting a separate vulnerability in another operating system component (e.g. a userspace service or the kernel)

With iOS 14, Apple shipped a significant refactoring of iMessage processing, and made all four parts of the attack harder. This is mainly due to three central changes:

1. The BlastDoor Service

One of the major changes in iOS 14 is the introduction of a new, tightly sandboxed "BlastDoor" service which is now responsible for almost all parsing of untrusted data in iMessages (for example, NSKeyedArchiver payloads). Furthermore, this service is written in Swift, a (mostly) memory safe language which makes it significantly harder to introduce classic memory corruption vulnerabilities into the code base.

(What was I just saying about the need to change our implementation languages, especially where security is important? Apple has the maturity and discipline to do so.)

[Samuel then shows us a rough diagram of iMessage's message processing pipeline flow.]

He continues:

As can be seen, the majority of the processing of complex, untrusted data has been moved into the new BlastDoor service. Furthermore, this design with its 7+ involved services allows fine-grained sandboxing rules to be applied, for example, only the IMTransferAgent and apsd processes are required to perform network operations. As such, all services in this pipeline are now properly sandboxed (with the BlastDoor service arguably being sandboxed the strongest).

2. Re-randomization of the Dyld Shared Cache Region

Historically, ASLR on Apple's platforms had one architectural weakness: the shared cache region, containing most of the system libraries in a single prelinked blob, was only randomized per boot, and so would stay at the same address across all processes. This turned out to be especially critical in the context of 0-click attacks, as it allowed an attacker, able to remotely observe process crashes (e.g. through timing of automatic delivery receipts), to infer the base address of the shared cache and as such break ASLR, a prerequisite for subsequent exploitation steps.

However, with iOS 14, Apple has added logic to specifically detect this kind of attack, in which case the shared cache is re-randomized for the targeted service the next time it is started, thus rendering this technique useless. This should make bypassing ASLR in a 0-click attack context significantly harder or even impossible (apart from brute force) depending on the concrete vulnerability.

3. Exponential Throttling to Slow Down Brute Force Attacks

To limit an attacker's ability to retry exploits or brute force ASLR, the BlastDoor and imagent services are now subject to a newly introduced exponential throttling mechanism enforced by launchd, causing the interval between restarts after a crash to double with every subsequent crash (up to an apparent maximum of 20 minutes). With this change, an exploit that relied on repeatedly crashing the attacked service would now likely require in the order of multiple hours to roughly half a day to complete instead of a few minutes.

And then, for the remainder of his disclosure Samuel lays out the details of what he found.

In today's climate, it is this sort of anticipatory design-for-security that's needed. After the troubles that surfaced in iOS 13, Apple didn't simply patch those individual flaws. Apple fundamentally re-architected the entire iMessage processing system to preempt entire classes of next-generation attacks. One could argue that it should have always been that way. But what we've been witnessing is a slowly but surely constantly rising threat level and Apple has been in the middle of it all. Without question, lessons are being learned. Hopefully by the entire industry. Which is what makes Samuel's sharing of what Apple won't tell us so valuable. There's little doubt that future systems WILL be designed with these sorts of mitigations built-in from the start. But that can only happen if the bar for acceptable attack resistance is raised. Apple, thanks to Samuel's reverse engineering disclosure, has raised that bar again.

For the past 10 years, “SUDO” was only pseudo secure

Last week, a very serious bug came to light in the command-line parser of Linux's powerful SUDO command. As we know, SUDO allows non-root users to temporarily elevate their rights to obtain some root privileges, typically for performing maintenance and various system-level tasks.

Qualys, the discoverers of the flaw wrote:

The Qualys Research Team has discovered a heap overflow vulnerability in sudo, a near-ubiquitous utility available on major Unix-like operating systems. Any unprivileged user can gain root privileges on a vulnerable host using a default sudo configuration by exploiting this vulnerability.

Sudo is a powerful utility that’s included in most if not all Unix- and Linux-based OSes. It allows users to run programs with the security privileges of another user. The vulnerability itself has been hiding in plain sight for nearly 10 years. It was introduced in July 2011 (commit 8255ed69) and affects all legacy versions from 1.8.2 to 1.8.31p2 and all stable versions from 1.9.0 to 1.9.5p1 in their default configuration.

Successful exploitation of this vulnerability allows any unprivileged user to gain root privileges on the vulnerable host. Qualys security researchers have been able to independently verify the vulnerability and develop multiple variants of exploit and obtain full root privileges on Ubuntu 20.04 (Sudo 1.8.31), Debian 10 (Sudo 1.8.27), and Fedora 33 (Sudo 1.9.2). Other operating systems and distributions are also likely to be exploitable.

As soon as the Qualys research team confirmed the vulnerability, Qualys engaged in responsible vulnerability disclosure and coordinated with sudo’s author and open source distributions to announce the vulnerability.

The maintainers of SUDO follow-up last Tuesday, the 26th of January:

A serious heap-based buffer overflow has been discovered in sudo that is exploitable by any local user. It has been given the name Baron Samedit by its discoverer. The bug can be leveraged to elevate privileges to root, even if the user is not listed in the sudoers file. User authentication is not required to exploit the bug.

The concern is that the Internet is full of systems which quite naturally depend, for their security, upon the proven strength of the Unix / Linux account privilege model. And this, now well-known flaw, punches right through that. There are doubtless countless systems where an unprivileged user account is easily available with minimal authentication requirements. Such systems are utterly dependent for their security upon that unprivileged account remaining so. But any Linux system that went online after July 2011 will contain this flaw until and unless it is updated. And we know that Linux is embedded everywhere. And everything we've seen demonstrates that most of those systems will NEVER be updated.

To repurpose a now famous phrase: "Stand back and stand by." This new backdoor goldmine is likely to present a huge opportunity for both initial network intrusion as well as post-intrusion lateral movement within a network.

Sci-Fi

"The Expanse"

SpinRite

February 1st Progress Report

MUCH has transpired since my previous, January 3rd, announcement that work on SpinRite v6.1 had commenced.

To plan for what SpinRite v6.1 would, and would not, be, I needed to know how I was going to solve SpinRite's imperative need to boot on UEFI systems that no longer offer a traditional BIOS fall-back. So, while beginning to work on SpinRite, I also searched for any means to add BIOS support to a system that doesn't currently have any. The firmware which boots a system is intimately tied to its hardware. That's why systems need firmware in the first place. This means that it's not possible to simply run any BIOS on whatever hardware happens to be present. So I was hoping that someone somewhere might have created a UEFI-to-BIOS translation layer which would allow for BIOS calls to be translated into their UEFI equivalent. That would allow SpinRite to load such a translation layer, boot BIOS-dependent DOS, and then reuse everything that SpinRite already is on any UEFI system. But, unfortunately, nothing like that exists and no one has ever created such a thing because the need for something like that only exists due to SpinRite's unique and continuing dependence upon the legacy of 16-bit DOS.

This provided some needed clarity. The only way to move SpinRite forward would be to finally, after 35 years, end its dependence upon the BIOS and DOS. So, I began looking around for the optimal environment to host SpinRite's future.

I'll spare you the blow-by-blow. But, for what it's worth, everything was looked at. [Moving to Linux, using Windows PE, the ReactOS, or another of the hobby OSes. All of the many various embedded OSes, like VxWorks were examined.] In the end, I settled upon an extremely lightweight real time operating system for embedded applications called "OnTime RTOS32." It's a minimal OS that allows executable code to be written, tested and debugged under Windows (my preferred development environment) then the PE format executable is repackaged for operation under its own loader and environment. And it supports booting from either a BIOS or UEFI firmware.

But, what that meant was that in order to get to UEFI I would need to dramatically change the operating environment. Operating under DOS and in the Intel processor's real mode, means that two 16-bit registers are always being referenced, and it's a bit funky: The bits in a 16-bit segment register are shifted left four bits to create a 20-bit value. Then a 16-bit offset register is added to that 20-bit result. The 16-bit offset register means that you can access a 64K byte block at a time, and the 16-bit segment register determines where that 64k byte block is located with a granularity of 16 bytes. Together, this allows for addressing 1 megabyte of memory, and doing it with 16-bit registers which individually can only specify 64K. This is where the Intel 8080 and 8086's 1 megabyte memory limit comes from. A 20-bit address is kludged together from two 16-bit values with one shifted four bits to the left.

Intel later chips have 32-bit registers. So in a flat memory model the 32-bits directly specifies the address. And as we know, 32 bits can address 4 gigabytes.

Back when I wrote SpinRite I embraced segmentation — since that's all we had back then. So, for example, when you rotate through SpinRite's screens, each of those screens is referenced by a 16-bit segment with each screen starting at offset 0 within its own segment. On one hand it's messy. But it can also be incredibly efficient and economical. It is Intel's legacy.

The problem, moving forward, is that only the Intel 16-bit Real Mode, or 16-bit Virtual 86 Mode performs this segmentation math. So there is no future there. If the chip is not in one of the 16-bit modes, you don't have segmented memory. So… for the past three weeks or so, my plan had been to finish SpinRite v6.1 and that it would be the final release of SpinRite to run in 16-bit DOS. But any further investment in the 16-bit environment would not be portable to the future. So it makes no sense to invest more time and energy there. And speaking of time, we now know that booting on a UEFI-only system requires 32-bit code. So spending more time in 16-bit land would delay all SpinRite for UEFI systems.

So I was planning to finish SpinRite v6.1 and then switch to a 32-bit flat memory environment and start from scratch. In the spinrite.dev newsgroup I talked about probably initially just using a text console UI as a means of getting something running more quickly. But yesterday as I was writing that February 1st progress report another idea hit me that had never occurred to me before: Instead of scraping 16-bit SpinRite and starting over, why not convert the current 16-bit segmented memory SpinRite into a 32-bit flat mode SpinRite? It won't be like flipping a switch, since the assumption and use of Intel's wacky segmented memory and segmentation math is deeply intertwined throughout SpinRite. But today I'm certain it's the proper path.

Once v6.1 is launched, I will immediately begin the process of converting SpinRite from its 16-bit segmented-memory real mode form into a flat 32-bit memory model. This will create the first 32/64-bit SpinRite v7.0 capable of booting under either UEFI or BIOS firmware, and a SpinRite with a future. For the sake of speed, I plan to initially leave SpinRite's old and creaky user interface as is. v7.0 will offer the same advanced support as v6.1 but with UEFI. So it will run ATA/IDE and AHCI controllers at their maximum possible speed. But since there's no BIOS to fall back on, v7.0 will not initially be able to operate upon USB or NVMe devices at all. So v6.1 will remain the only solution for those until v7.1 and v7.2. After v7.0 is released, my plan would be to immediately add support for USB, creating v7.1 (and, of course, no charge for those who have upgraded to v7.0). And once v7.1 is released I would then immediately work to add NVMe support, creating v7.2.

At this point we'll have text-mode single-tasking SpinRite v7.2 running ALL current mass storage controller interfaces at their maximum possible performance. And all of that code will be reusable by SpinRite v8.

The plan for v8 is just a rough sketch at this point. But it will be a move to a mouse and GUI interface, the ability to simultaneously run on any or all of a system's drives, eventual file system awareness, file system structure recovery, structure-driven data recovery, file-based recovery and who knows what else.

As always, this is all subject to revision and refinement. But it's where we are as we start into February and it feels very right to me. It places the most functionality into SpinRite user's hands as quickly as possible, supports SpinRite's quickest migration from BIOS/DOS/16-bits -to-UEFI/BIOS 32/64 bits, while providing a clear and seamless path forward into SpinRite's future.

NAT Slipstreaming 2.0

Security Now! episode #792 which we recorded last November 10th was titled “NAT Firewall Bypass.” Perhaps we should have added a “1.0” to it in anticipation of the technique's inevitable evolution. Thus, today's podcast bears the title: “NAT Slipstreaming 2.0.”

I'll begin by reminding everyone where we were in November, and quickly place that 1st-generation exploit into context against today's 2nd generation exploit. The original NAT Slipstreaming attack relied on a victim within an internal network — tucked safely, they thought, behind their NAT firewall — who clicks on a link that leads to an attacker’s website which will trick the victim network’s NAT into opening an incoming path (either a TCP or UDP) from the Internet, to the victim's device. That was bad enough for all of our browsers to quickly move to block outbound access to a few additional remote port numbers.

Today's upgrade of NAT Slipstreaming extends this attack by allowing the remote attacker to trick the NAT into creating NAT traversal mappings to ANY device on the internal network, not only to the victim device which originally and unwittingly clicked the malicious link.

Many devices located on our internal LAN networks may be fine there, but not when exposed to the Internet. How many times have I talked about services that should NEVER, under any circumstances, be exposed to the Internet? (And I'm just asking rhetorically, please don't go count them — we all know that it's a common refrain of mine, here.) But this is precisely that version 2 NAT Slipstreaming allows. Many embedded devices have minimal local security — if any — because they're correctly presumed to only be accessible to local users. An office printer that can be controlled through its default printing protocol, or through its internal web server, is a perfect example. Or an industrial controller that uses an unauthenticated protocol for monitoring and controlling its functions. Or an IP camera that has an internal web server displaying its feed, which can often be accessed with default credentials, if any. The 2nd generation variant of the NAT Slipstreaming attack can access these types of interfaces from the Internet, resulting in attacks that range from nuisance to sophisticated ransomware threats.

In addition to network interfaces of devices that are unauthenticated by design, many unmanaged devices may also be vulnerable to publicly known but currently unpatched vulnerabilities. These could be exploited if an attacker is able to bypass the NAT firewall to initiate traffic that can trigger those known weaknesses. To gain some sense for this, in a recent study, Armis researchers found that 97% of industrial controllers vulnerable to URGENT/11 were left unpatched, more than a year after the critical vulnerabilities were first published. We talked about this back in 2019. Urgent/11 were a set of eleven 0-day vulnerabilities discovered in VxWorks, the most popular embedded OS. Again, today, 97% of VxWorks-based embedded systems remain vulnerable to well-known attacks since 2019. And 80% of Cisco devices which were vulnerable to CDPwn — still are nearly one year after those critical vulnerabilities were published.

So the point is, the only thing that's keeping a tremendous amount of havoc from reigning inside our LAN's is the fact that NAT translation normally makes for a terrific stateful firewall... and this new NAT Slipstreaming technique can bypass the security that we have come to take for granted.

Paraphrasing Samy Kamkar's summary of the attack which he discovered: “NAT Slipstreaming allows an attacker to remotely access any TCP/UDP service, bypassing the victim's NAT/firewall just by having the victim visit a website or causing their browser to run JavaScript contained within an online ad.”

Strict NAT traversal rules are straightforward, simple and elegant: They amount to “only accept incoming return traffic from remote IPs & Ports that recently received outbound traffic.” Period. But Samy's original inspiration was to observe that those simple and straightforward NAT traversal rules often needed to be bent in order to accommodate the needs of network protocols. We used Active FTP as an example, where the outbound control connection tells the server what port to connect to inbound. For that to operate transparently through NAT requires the NAT router to look inside the outbound FTP traffic for the outgoing port specification and then automagically reroute the server's new and technically unsolicited connection through to the proper expected port on the client machine on the LAN.

Collectively these are known as ALGs — Application Layer Gateways — because they require the router to be aware of the content of the packets passing through the router rather than just the fact of each packet. Under the NAT Passthrough tab of the WAN section of one of my Asus routers, it provides for PPTP, L2TP, IPSec, RTSP, H.323, SIP, PPPoE Relay and FTP passthrough. Conservative as I always am, when first setting up that network, I immediately disabled all of those ALG passthroughs, since I have no need for them. Consequently, I was quite puzzled when the Verizon LTE Network Extender refused to log onto Verizon after its installation. In digging through its troubleshooting FAQs, I noticed that it mentioned its use of IPSec ... and I simultaneously thought “Whoops!” and “Ah Hah!”. After re-enabling the router's IPSec ALG the

Verizon femtocell came right up and has been providing terrific five-bar service ever since. The lesson being... disable what you know you don't need, yes. But remember that you did so when something doesn't work right.

So what happened with the 1st and 2nd generations of NAT Slipstreaming? It turns out there's an even more insidious ALG present in our routers than Samy had originally exploited. Researchers at Armis were intrigued by what Samy originally found. And everyone knows what Bruce Schneier famously said: “Attacks never get less powerful. They only grow more powerful.”

They wrote:

Building upon Samy’s ideas, and combining them with some of our own, led us to the discovery of the new variant.

This new variant is comprised of the following newly disclosed primitives:

Unlike most other ALGs, the H.323 ALG, where supported, enables an attacker to create a pinhole in the NAT/firewall to any internal IP, rather than just the IP of the victim that clicks on the malicious link.

WebRTC TURN (Traversal Using Relay around NAT) connections can be established by browsers over TCP to any destination port. The browsers restricted-ports list was not consulted by this logic, and was therefore bypassed.

This allows the attacker to reach additional ALGs, such as the FTP and IRC ALGs (ports 21, 6667) that were previously unreachable due to the restricted-ports list. The FTP ALG is widely used in NATs/firewalls.

This also defeated the browser mitigations introduced shortly after Samy first published the NAT Slipstreaming attack, which added the SIP port (5060) to the restricted-ports list, but didn’t block the port from being reachable via a TURN connection.

H.323 is a VoIP protocol similar to SIP, which is also quite complex. For a network of VoIP phones to function properly, while having a NAT somewhere inside the topology, an H.323 ALG is required. The H.323 ALG needs to translate IP addresses that are contained within the application layer H.323 protocol packets, and open holes in the NAT in certain scenarios.

Most ALGs only need to worry about at most two addresses to translate within a session -- the IP addresses (and ports) of both sides of the TCP connection. However, H.323 is a telephony protocol, and supports call forwarding. Therefore, in this case, one party within the session can refer to a 3rd party IP address, belonging to the VoIP phone that the call should be forwarded to. Most H.323 ALGs support this feature.

The result of all this is that a single H.323 packet sent over TCP port 1720 that initiates call forwarding can open an external pinhole through the NAT to ANY TCP port of ANY internal IP on the network. Thus, NAT Slipstreaming just got a whole lot more powerful.

The Armis blog about this contains full details for anyone who wants more:

https://www.armis.com/resources/iot-security-blog/nat-slipstreaming-v2-0-new-attack-variant-can-expose-all-internal-network-devices-to-the-internet/

They tested many routers and found virtually all of them to be vulnerable to full port and IP exploitation. Since all of these attacks bounce scripts and network packets off of our web browsers, they then responsibly disclosed what they had found to all browser vendors:

- November 11, 2020 - A coordinated disclosure of the new variant was initiated with Chromium, Mozilla and Apple. The issue was acknowledged promptly by the vendors.
- January 6, 2021 - Chrome release v87.0.4280.141, contains a patch mitigating the new attack variant.
- January 7, 2021 - Microsoft's Edge released v87.0.664.75, based on Chromium v87.0.4280.141, therefore also contains a patch against the attack.
- January 14, 2021 - Safari releases v14.0.3 beta, that contains a patch against the attack. A stable version is expected to be released shortly.
- January 26, 2021 - Firefox releases v85.0 that contains a patch against the attack.

Recalling from our coverage of this last November, our browsers responded to the first Slipstreaming revelations by blocking HTTP and HTTPS access to TCP ports 5060 and 5061.

Now, they are also blocking HTTP, HTTPS, and FTP access to 69, 137, 161, 1719, 1720, 1723, and 6566 TCP ports. Google wrote:

"The NAT Slipstream 2.0 attack is a kind of cross-protocol request forgery which permits malicious internet servers to attack computers on a private network behind a NAT device. The attack depends on being able to send traffic on port 1720 (H.323). To prevent future attacks, this change also blocks several other ports which are known to be inspected by NAT devices and may be subject to similar exploitation."

The other lesson here is to always turn off any unneeded features of your router. My network was never vulnerable to any of this because its router has always had all of that extra and unused stuff disabled. If you don't know that you need some feature, turn it off. (And if that breaks something, then of course turn just that one feature back on. :)