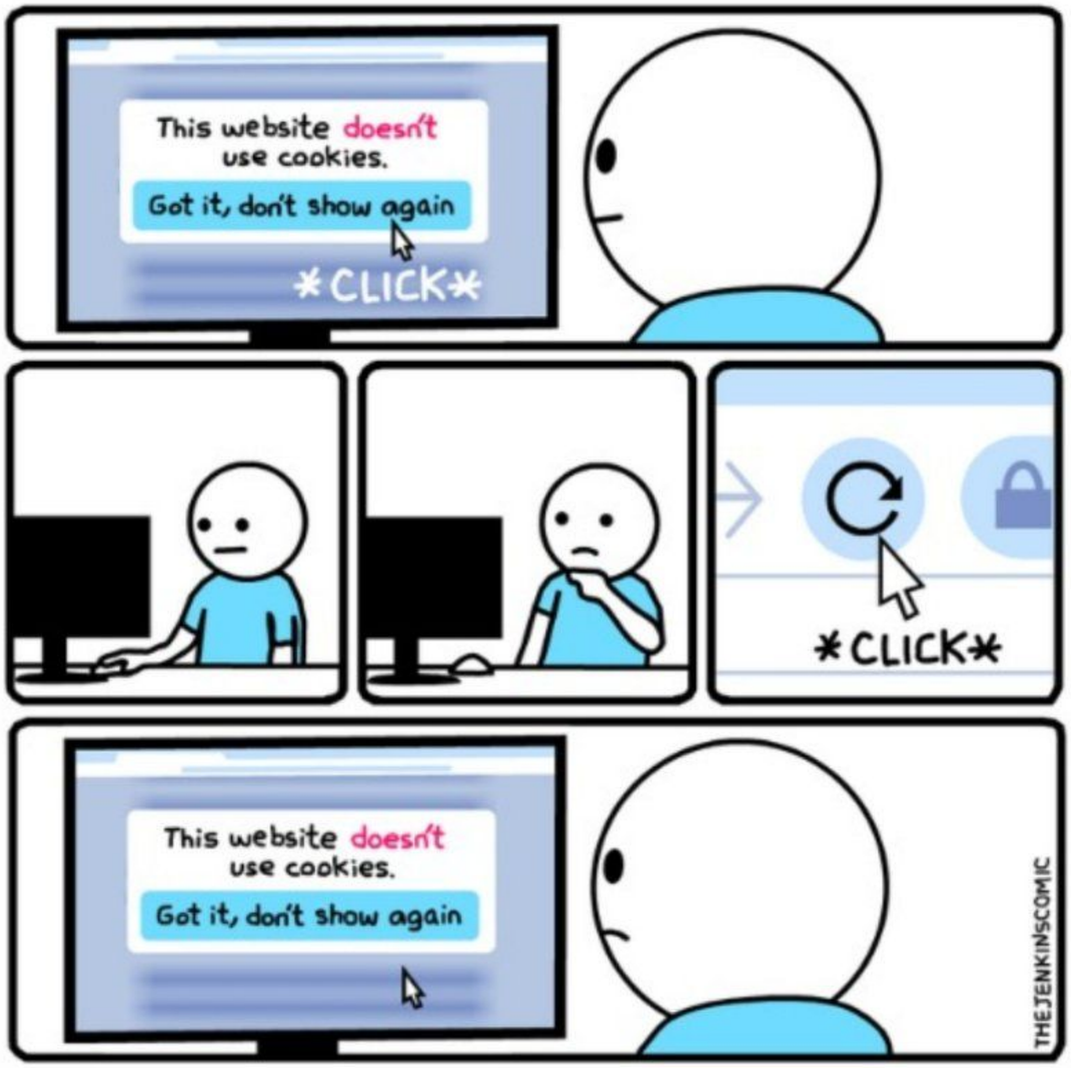
Security Now! #807 - 02-23-21

Dependency Confusion

(The Build Chain Hack)

This week on Security Now!

This week we'll follow-up on the Android SHAREit app sale. We look at a clever new means of web browser identification and tracking and at a little mistake the Brave browser made that had big effect. I want to remind our listeners about the ubiquitous presence of tracking and viewing beacons in virtually all commercial eMail today. We'll look at Microsoft's final SolarWinds Solorigate report and at another example of the growing trend of mobile apps being sold and then having their trust abused. I'll share a post from the weekend about a dramatic improvement in SSD performance after running SpinRite, but also why you may wish to hold off on doing so yourself. And then we're going to look at what everyone will agree was — and perhaps still is — a breathtaking oversight in the way today's complex software products are assembled which creates an inherent massive vulnerability across the entire software industry.



SHARTit Follow-up

Last week we covered the disclosures from Trend Micro about the quite buggy SHAREit Android app, not because it was a quite buggy Android app, but because this particular quite buggy Android app happened to have an active user base somewhere North of one billion — with a 'B' — users. We explained how Trend Micro's ZDI — Zero Day Initiative — folks had contacted SHAREit's publisher 90 earlier with a disclosure of the very many worrisome problems they had found. And how, after 90 days of no contact, they decided that they had done their due diligence and that the many problems were so bad, that now the responsible thing to do was to inform the public of the danger that they remained in.

I commented last week that perhaps after the public embarrassment of a widely respected security research firm disclosing the many design mistakes made by the app, its publisher would hopefully react. And we wondered what Google would be thinking about this being on their Play Store, too.

So, indeed, last Friday, the Singapore based SHAREit company issued a Press Release containing two bullet points:

- SHAREit app is a leading file sharing, content streaming and gaming platform. Since its inception, billions of users have entrusted SHAREit to quickly and securely share their files. The security of our app and our users' data is of utmost importance to us. We are fully committed to protecting user privacy and security and adapting our app to meet security threats.

- On February 15, 2021, we became aware of a report by Trend Micro about potential security vulnerabilities in our app. We worked quickly to investigate this report, and on February 19, 2021, we released a patch to address the alleged vulnerabilities.

I went looking for any update from Trend Micro, but so far, nothing. If they do decide to verify the fixes, it might take a while. But, also, their original disclosure was comprehensive enough to allow anyone who wished to, to verify them independently.

I also want over to the Google Play Store to see whether there might be an update history. I didn't find one. But I did notice that the app showed as being updated today, February 22nd. And the press release noted that the app was also updated on the day of the release, last Friday the 19th. So, perhaps their first attempt missed a few spots. Who knows? But if you're one of those 1 Billion+ SHAREit users, you might keep checking every few days until the app again settles down.

Browser News

This Week in Web Browser Tracking

Or: *“Where there's a will”* ... And there sure does appear to be a large amount of “will” behind browser tracking. It's been an often recurring topic since this podcast was born, and I invested a lot of my own time very early on in the cookie forensics work to examine how our browsers were handling 1st and 3rd party cookies. Today, those days seem as quaint as floppy disk viruses.

But browser tracking is back in the news because a group of researchers from the University of Illinois at Chicago will be presenting their paper on the use of Favicons for tracking during this week's virtual Network and Distributed Systems Security Symposium, NDSS 2021. Their 19-page paper is titled "Tales of FAVICONS and Caches: Persistent Tracking in Modern Browsers"

https://www.cs.uic.edu/~polakis/papers/solomos-ndss21.pdf

So, first of all, for those who are not aware, the FAVICON is the tiny icon that appears at the left end of browser tabs and, as its name suggests, when we save a website link as a "Favorite." GRC has its "Ruby G" (as we call it), TWiT has the cute little TWiT AND-gate icon standing on two feet, GRC's SQRL forums present the SQRL logo, and pretty much every site has a Favicon. (And, yes, I've heard it pronounced "fav-i-con." But just as a LIB is a library, a Favicon is a contraction of Favorite Icon. So, no, I'm not of the "fav-i-con" crowd.)

In any event, it turns out that all browser makers have assumed that since a favicon is sent from a site, that makes its flow unidirectional, so, unlike cookies or web beacons or signals used for fingerprinting, the favicon didn't need anti-tracking protection. And as a result of that, all browser makers cache their favicons in a separate cache which is never cleared when cookie are cleared or even when the browser's cache is cleared. And what's more, both the standard and incognito-mode interfaces of the browser share a single favicon cache. So, needless to say, if it turned out that there was some means of tracking users through favicons, not only would it be immune from the user's deliberate anti-tracking measures of flushing this or that cache, but it could also be used to breach the security boundary between the explicitly no-history-being-retained private browsing modes and the standard mode. Since there's only a single favicon cache, it would be possible to determine whether a browser had ever visited a site by seeing whether it requests a favicon from the site in question when the non-private browsing mode browser visits the site.

And, of course, we know where this is going. I was quite curious to learn how these guys had pulled this off, since a browser's retrieval of a favicon is a binary event — either it doesn't have one so it'll ask for it, or it already has one and thus won't need to ask again. The hack IS rather brute force, but I'll give them a "C" for clever. Here's how they described their work in their paper's Abstract:

> The privacy threats of online tracking have garnered considerable attention in recent years from researchers and practitioners. This has resulted in users becoming more privacy-cautious and browsers gradually adopting countermeasures to mitigate certain forms of cookie-based and cookie-less tracking. Nonetheless, the complexity and feature-rich nature of modern browsers often lead to the deployment of seemingly innocuous functionality that can be readily abused by adversaries. In this paper we introduce a novel tracking mechanism that misuses a simple yet ubiquitous browser feature: favicons. In more detail, a website can track users across browsing sessions by storing a tracking identifier as a set of entries in the browser's dedicated favicon cache, where each entry corresponds to a specific sub-domain. In subsequent user visits, the website can reconstruct the identifier by observing which favicons

are requested by the browser while the user is automatically and rapidly redirected through a series of subdomains. More importantly, the caching of favicons in modern browsers exhibits several unique characteristics that render this tracking vector particularly powerful, as it is persistent (not affected by users clearing their browser data), non-destructive (reconstructing the identifier in subsequent visits does not alter the existing combination of cached entries), and even crosses the isolation of the incognito mode. We experimentally evaluate several aspects of our attack, and present a series of optimization techniques that render our attack practical. We find that combining our favicon-based tracking technique with immutable browser-fingerprinting attributes that do not change over time allows a website to reconstruct a 32-bit tracking identifier in 2 seconds. Furthermore, our attack works in all major browsers that use a favicon cache, including Chrome and Safari. Due to the severity of our attack we propose changes to browsers' favicon caching behavior that can prevent this form of tracking, and have disclosed our findings to browser vendors who are currently exploring appropriate mitigation strategies.

This works, since sub-domains also each get their own favicons. So these guys set up an HTTP 302 redirection chain to take the user's browser on a 32-subdomain walk, where the primary domain indicates whether this browser has ever visited this site before. If so, if the browser doesn't need that main site's favicon, that's because this is not the browser's first visit. So the site sends the browser off across the chain of 32 subdomains, noting which of the sub-domains the browser asks for favicons from, and which not — but it never gives the browser any new favicons to store. That pattern of requests will be unique for that browser... because...

But, if the browser DOES ask for the favicon from the primary site, then the site knows that the browser has not yet been uniquely favicon tagged. So the site chooses a new 32-bit identifier for the browser and uses that identifier to judiciously and selectively dole out, or not, subdomain favicons to match the 32-bit binary identifier.

So, it's kinda brute force. But it's diabolical when you consider how none of our browsers have been treating their favicon caches with due respect. The attack is currently effective against Chrome, Safari, Edge (because it's Chromium), and it was effective against Brave (since Brave is also Chromium) until the researchers gave the Brave developers an early heads-up, which allowed Brave to beat the rest of the pack to an effective countermeasure. And, interestingly, Firefox would also have been vulnerable to the technique — they wanted very much to be — but a bug prevents the attack from working at the moment. The researchers explain:

As part of our experiments we also test Firefox. Interestingly, while the developer documentation and source code include functionality intended for favicon caching similar to the other browsers, we identify inconsistencies in its actual usage. In fact, while monitoring the browser during the attack's execution, we observe that it has a valid favicon cache which creates appropriate entries for every visited page with the corresponding favicons. However, it never actually uses the cache to fetch the entries. As a result, Firefox actually issues requests to re-fetch favicons that are already present in the cache. We have reported this bug to the Mozilla team, who verified and acknowledged it. At the time of submission, this remains an open issue. Nonetheless, we believe that once this bug is fixed our attack will work in Firefox, unless they also deploy countermeasures to mitigate our attack.

So this is cool for Firefox. They will doubtless fix the broken cache to speed things up, while simultaneously implementing a fix for this track attack.

It would be sad to lose the use of favicons, so I'm glad to note that every browser maker is on this and is working toward a fix. But FWIW, favicons CAN be disabled. I have three links in the show notes to DuckDuckGo queries about how to disable favicons in Chrome, Safari and Edge:

https://duckduckgo.com/?q=chrome+disable+favicon&atb=v110-1&ia=web
https://duckduckgo.com/?q=safari+disable+favicon&atb=v110-1&ia=web
https://duckduckgo.com/?q=edge+disable+favicon&atb=v110-1&ia=web

Brave's "Private Window with Tor" was not so private
The Brave browser has just fixed a privacy mistake that was sending DNS lookup queries for .onion domains out onto the public Internet rather than routing them through Tor nodes. This, of course, exposes users' visits to dark web sites. The mistake was fixed in hotfix release (V1.20.108) made available last Friday. So anyone who is depending upon Brave's nifty built-in "Private Window with Tor" feature for safer Tor-based anonymity will want to be sure to be running the v1.20.108 of later release.

The source of the bug is interesting. The trouble arose from Brave's internal ad blocking component. It uses DNS queries to discover sites that are attempting to bypass its ad-blocking capabilities. But the developers had forgotten to exclude .onion domains from these checks. Whoops.

The thing that's a bit disturbing is that the Brave folks have known about this for some time. It was reported to the HackerOne bug bounty site back on January 13th, after which the bug was fixed in a Brave nightly release in early February. Apparently, the fix was planned for a rollout in a larger "point release" v1.21.x. But the flaw was publicly disclosed on Ramble which pushed the Brave folks to advance their timetable and push the interim 1.20.108. I'm a bit uncomfortable with the idea that something that's as significantly privacy-critical as publicly resolving .onion domains without Tor's privacy protection wouldn't be regarded as an emergency with a patch pushed immediately. But... I suppose it's a brave new world.

Security News

Tracking with eMail Beacons
While we're on the subject of tracking, I'll just note that today a huge percentage of spam eMail contains embedded tracking beacons. These days the transition from text to HTML eMail is pretty much complete. Most people want to see pretty colors and fonts and logos in their eMail. And while it's possible for eMail to embed those graphics, it's also possible for the eMail to contain a serial-number encoded link back to the mothership. This essentially turns your eMail client into a web browser, so that the act of opening and viewing the lovely colorful eMail causes your eMail client to obtain an image from a remote server... which, because the URL is encoded with your identity, confirms your receipt and viewing of the note.

None of this is news. But I just wanted to refresh everyone's awareness of it. There was a recent report in the BCC News titled: “Spy pixels in emails have become endemic.” The report told the story of the fact that virtually ALL of today's commercial eMail embeds call-backs to confirm the viewer's reception and display of the eMail.

Many eMail clients have add-ons for blocking these spying beacons. It's also possible to configure clients not to display images, or to only display the eMail as an old school plain text document. No big takeaway here, but I just wanted to remind our security and privacy anti-tracking listeners that there's another vector of monitoring that often goes unobserved and under reported.

Microsoft's final “Solorigate” update

Last Thursday, Microsoft posted the final results of their research into the details of their own SolarWinds breach:

https://msrc-blog.microsoft.com/2021/02/18/microsoft-internal-solorigate-investigation-final-update/

I'll paraphrase from their posting...

We have now completed our internal investigation into the activity of the actor and want to share our findings, which confirm that we found no evidence of access to production services or customer data. The investigation also found no indications that our systems at Microsoft were used to attack others. Because of our defense-in-depth protections, the actor was also not able to gain access to privileged credentials or leverage the SAML techniques against our corporate domains.

We detected unusual activity in December and took action to secure our systems. Our analysis shows the first viewing of a file in a source repository was in late November and ended when we secured the affected accounts. We continued to see unsuccessful attempts at access by the actor into early January 2021, when the attempts stopped.

There was no case where all repositories related to any single product or service was accessed. [To me that sounds like “how can we put the best possible spin on this breach?”] There was no access to the vast majority of source code. For nearly all of code repositories accessed, only a few individual files were viewed as a result of a repository search.

For a small number of repositories, there was additional access, including in some cases, downloading component source code. These repositories contained code for:

- a small subset of Azure components (subsets of service, security, identity)
- a small subset of Intune components
- a small subset of Exchange components

The search terms used by the actor indicate the expected focus on attempting to find secrets. Our development policy prohibits secrets in code and we run automated tools to verify compliance. Because of the detected activity, we immediately initiated a verification process for current and historical branches of the repositories. We have confirmed that the repositories complied and did not contain any live, production credentials.

[I'll note that this sentence "Our development policy prohibits secrets in code and we run automated tools to verify compliance." should be very reassuring. How many times have we talked about vendors both small and large — can you say Cisco — introducing horrific remote vulnerabilities into their production products when "secret" accounts and passwords are embedded. We should all have learned long ago that secrets can never be kept — least of all in code.]

The cybersecurity industry has long been aware that sophisticated and well-funded actors were theoretically capable of advanced techniques, patience, and operating below the radar, but this incident has proven that it isn't just theoretical. For us, the attacks have reinforced two key learnings that we want to emphasize —embracing a Zero Trust mindset and protecting privileged credentials.

A Zero Trust, "assume breach" philosophy is a critical part of defense. Zero Trust is a transition from implicit trust—assuming that everything inside a corporate network is safe—to a model that assumes breach, and explicitly verifies the security status of identity, endpoint, network, and other resources based on all available signals and data. We've recently shared guidance for using Zero Trust principles to protect against sophisticated attacks like Solorigate.

Protecting credentials is essential. In deployments that connect on-premises infrastructure to the cloud, organizations tend to delegate trust to on-premises components. This creates an additional seam that organizations need to secure. A consequence of this decision is that if the on-premises environment is compromised, this creates opportunities for attackers to target cloud services.

"Good App goes Bad for Profit"

We have a second instance of a highly popular Android app being sold for the express purpose of exploiting the App's well-earned existing install base for AdWare profit.

For some time, Malwarebytes has been exploring how a previously trusted, useful barcode and QR code scanner app on Google Play, that accounted for over 10 million installs, suddenly became malware overnight.

After gaining a following and acting as just what it was — innocent software for years — more recently Android users began to complain that their mobile devices were suddenly full of unwanted advertisements.

The suddenly troublesome app, "Barcode Scanner" was finally identified as the source of this nuisance and is tracked as Android/Trojan.HiddenAds.AdQR. Malwarebytes determined that recent malicious updates of the app had resulted in aggressive advertisement pushing which had been added to the app's code. The app's analytics code was also modified and the updates were heavily obfuscated.

Malwarebytes concluded that the App's owner, Lavabird Ltd., was likely to blame since that was the ownership registration at the time of the update. And once reported, the software was pulled from the Google Play Store. What was equally suspicious was that, at the time, Lavabird did not respond to requests for comment. However, the vendor has since reached out to Malwarebytes with an explanation for the situation.

On February 12, Malwarebytes said that Lavabird blamed an account named "the space team" for the changes following a purchase agreement in which the app's ownership would change hands — is this beginning to sound familiar? Apparently, Lavabird was an escrow-like intermediary between the original author/seller and its purchaser. Lavabird purchased the Barcode Scanner on November 23, and the subsequent “space team” deal was agreed to two days later, on November 25.

Lavabird told the Malwarebytes folks that they develop, sell, and buy mobile applications. So monetary transactions of mobile applications has become a thing. Lavabird explained that “the space team” purchaser of Barcode Scanner was granted access to the app’s Google Play console for the purpose of verifying the software's key and password prior to purchase. And it was that provisional buyer who first pushed the malicious update out to Barcode Scanner users.

This made sense to Malwarebytes, who wrote: “Transferring of the app's signing key when transferring ownership of the app is a legitimate part of the process. Therefore, the request by “the space team” to verify that the private key works by uploading an update to Google Play seems plausible.”

After the update was performed, the app was transferred to the buyer's Google Play account on December 7. However, Malwarebytes said that at the time of the malware update, ownership was still tied to Lavabird. The first malicious update took place well before that on November 27 (two days after the original November 25th purchase), and subsequent updates obfuscated the malware's code... until January 5 when the app was yanked from Google Play.

This tactic was apparently more surprising to Malwarebytes than it would be to this podcast's listeners. Their researcher, Nathan Collier, wrote: “From our analysis, what appears to have happened is a clever social engineering feat in which malware developers purchased an already popular app and exploited it. In doing so, they were able to take an app with 10 million installs and turn it into malware. Even if [only] a fraction of those installs update the app, that is a lot of infections. And by being able to modify the app's code before full purchase and transfer, they were able to test whether their malware would go undetected by Google Play on another company's account.”

So, it seems that we need a new flag added to Android apps in the Google Play Store. An app's change of ownership need not be a malicious event, but it certainly can be. Enough so to raise justified suspicion. It would be nice to know, for example, when an update is available, whether it was coming from the App’s original and current owner. Though I’ll note that “the space team’s” sneaky “let us verify that we have access” approach would have initially defeated that intervention since this was done prior to the ownership change.

SpinRite

Subject: RS shows VERY obvious improvement after one pass of SR 6
Author: TheDukeofURL
Date: Saturday at 4:16 PM

https://forums.grc.com/threads/rs-shows-very-obvious-improvement-after-one-pass-of-sr-6.546/

Been trying out SR 6 / RS scanning and testing on a large bunch of drives I have. Spinning, SSD, laptop, SATA and PATA. What follows is a really good indication of what Spinrite can do with SSDs that are a bit on the old side and need a little TLC. Today's selection is a Crucial M4 2.5" 128GB SATA III MLC, which has been in service for close to 5 years.

```
BEFORE
Driv Size  Drive Identity     Location:    0      25%     50%     75%     100
---- ----- ---------------------------- ------- ------- ------- ------- -------
 81  128GB M4-CT128M4SSD2                258.3   283.4   309.1   296.0   296.2

AFTER (running Spinrite on Level 3)
Driv Size  Drive Identity     Location:    0      25%     50%     75%     100
---- ----- ---------------------------- ------- ------- ------- ------- -------
 81  128GB M4-CT128M4SSD2                533.5   533.5   533.5   533.5   533.6
```

SpinRite Level 3 — “Refresh the Surfaces” — So it reads and rewrites everything it finds.

As the Benchmark clearly demonstrates, the act of reading and rewriting the entire storage space of the SSD has DOUBLED its subsequent read performance as measured in one gigabyte reads from five locations across the SSD. And many have reported that this does, in fact, result in a noticeable improvement in the media’s perceived performance.

There has been a lot of discussion in the GRC Newsgroups about the source of this SSD slowdown over time. There are those who hold that the slowdown is caused by long-term fragmentation of the mapping between logical sector addresses and their physical location. The idea is that SSDs achieve their peak performance through parallel reading across many independent channels. And that a brand new SSD will have a uniform “map” that maximizes the read performance of that device. The theory goes that over time, the relocation of regions induces an unavoidable fragmentation of this map which causes “clumping” of the reads, thus reducing the channel parallelism.

It’s been noted that performing the SSD’s Secure Erase operation immediately restores the drives maximum ReadSpeed performance. One assumption is that this has effectively defragmented the SSD’s logical to physical mapping table, thus restoring full performance. But I believe that’s not what has happened. SSD’s “know” when and exactly where data has previously been stored. So, a Secure Erase function “unmaps” the logical-to-physical association so that, after the secure erase, a read instantly returns zeros for any read because the drive knows that nothing has been stored there yet since the secure erase.

We know that SSDs store data as electrostatic charges in a vast array of very slowly leaking capacitors. This is why we reported many years ago that storing SSDs offline in a hot environment would cause their data to be lost much more quickly than if offline SSDs were stored in a cold environment. It's well known that heating a semiconductor results in increased electron migration due to thermal agitation. An SSD's capacitor array WANTS to discharge. It's always trying to return to a state of maximum entropy. Heat helps that to happen more quickly.

SSDs rely upon surprisingly advanced built-in data recovery to deal with the reality of what Allyn Malventano, Intel storage guy, referred to as "charge drift." They will read and reread, applying differing thresholds to an SSD's bit value determinations in an attempt to obtain a read that can be read either with or without error correction, upon which we know SSDs rely upon heavily.

So, the only way to make sense of what "The Duke of URL's" results show us, is that in a way that **exactly matches** the idea of refreshing a spinning drive's magnetic surface to find and fix any bad spots before they become too bad to read and/or correct fully, rewriting that SSD could **not** have "defragmented" its logical-to-physical mapping. As far as we know, only a full drive erase will do that. What had to have happened is that many regions that were being slowed down by their need for on-the-fly error recovery and correction are now, once again, reading at full speed because the capacitors that store the SSD's data have been fully recharged. And not only did this double the read speed of the drive, it also had to have dramatically improved the drive's future long-term reliability.

I'm **not** suggesting that everyone should run out and run SpinRite on their SSDs today. Not yet. SpinRite needs to be optimized for this use. For one thing, not all regions are slow. As we've seen, ReadSpeed's detailed output shows wide variations in read performance and SpinRite is not yet smart enough to selectively re-write only an SSD's slow spots. That's what we want in order to reduce SSD media wear. Another issue is the page alignment of SpinRite's writes. SSDs erase and rewrite their data **only** in large page size chunks. SpinRite v6.1 will work in large chunks that are inherently page aligned. SpinRite 6.0, and earlier, have had a "track alignment" orientation of 63 sectors that no longer makes any sense for today's storage. And then there's TRIMming. We've touched upon this before. Rewriting the entire drive "de-trims" the entire drive — telling it that the entire surface is in use and containing valid data. But the drive's file system knows better. Windows can be instructed to re-trim a drive — that's what Optimizing the drive now does on SSDs. And Linux will periodically run a re-trim over its drives. But a future SpinRite will also be natively TRIM aware.

The Duke of URL's posting Saturday was a clear demonstration that SpinRite has at least as much to do in the future as it has in the past. I'm so excited to get us there!

Dependency Confusion

Two weeks ago, Alex Birsan put up a posting on Medium that drew a great deal of well deserved attention. It was so cool that I punted on talking about it last week so that I could make it our topic this week.

I mentioned last week that our picture of the week was closely related to today's topic because it showed the rickety house of cards — or it this case, a very rickety stacking of blocks with inherent dependencies upon blocks below. Which brings us to today's topic: “Dependency Confusion.” To set the stage, Alex explains a bit about today's package management by writing:

Ever since I started learning how to code, I have been fascinated by the level of trust we put in a simple command like this one:

```
pip install package_name
```

Some programming languages, like Python, come with an easy, more or less official method of installing dependencies for your projects. These installers are usually tied to public code repositories where anyone can freely upload code packages for others to use.

You have probably heard of these tools already — Node has npm and the npm registry, Python’s pip uses PyPI (Python Package Index), and Ruby’s gems can be found on… well, RubyGems.

When downloading and using a package from any of these sources, you are essentially trusting its publisher to run code on your machine. So can this blind trust be exploited by malicious actors?

Of course it can.

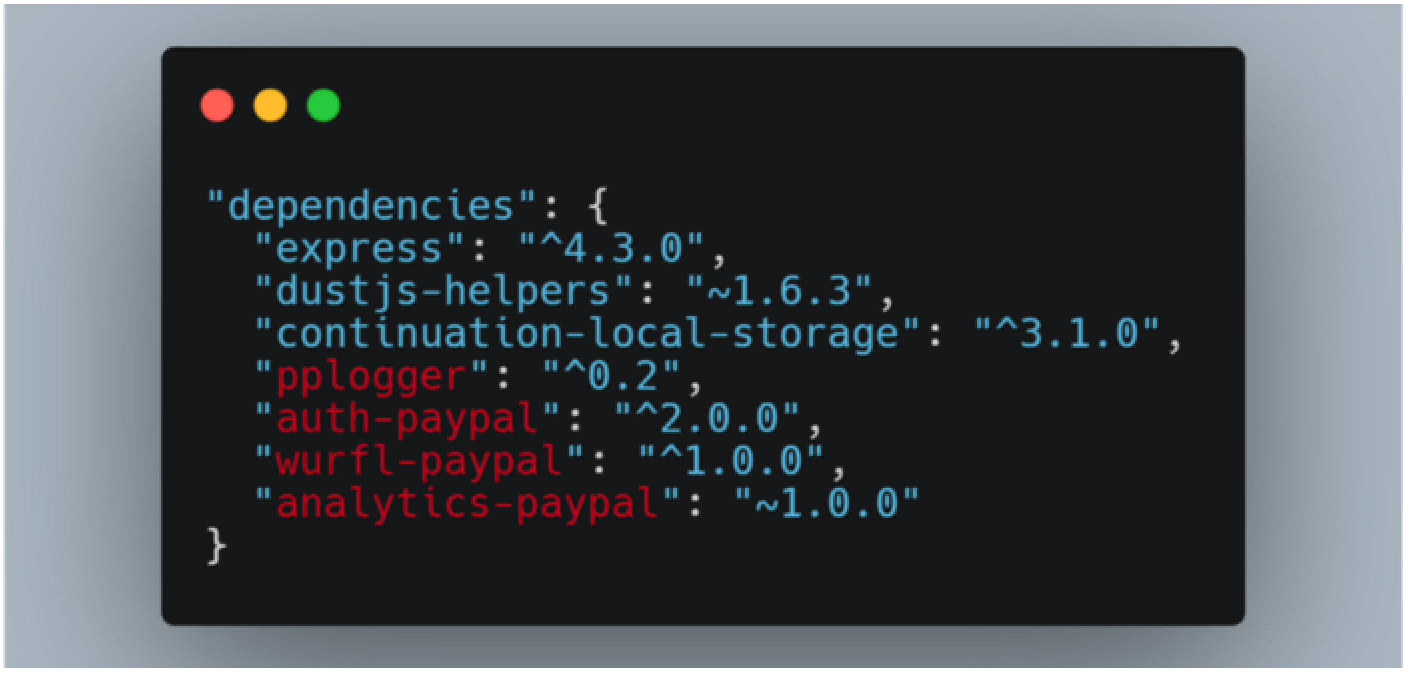
None of the package hosting services can ever guarantee that all the code its users upload is malware-free. Past research has shown that typosquatting — an attack leveraging typo’d versions of popular package names — can be incredibly effective in gaining access to random PCs across the world.

[And I'll note that we've talked about exactly that here in the past.]

Other well-known dependency chain attack paths include using various methods to compromise existing packages, or uploading malicious code under the names of dependencies that no longer exist.

While attempting to hack PayPal with me during the summer of 2020, Justin Gardner (@Rhynorater) shared an interesting bit of Node.js source code found on GitHub.

The code was meant for internal PayPal use, and, in its package.json file, appeared to contain a mix of public and private dependencies — public packages from npm, as well as non-public package names, most likely hosted internally by PayPal. These names did not exist on the public npm registry at the time.



```
"dependencies": {
  "express": "^4.3.0",
  "dustjs-helpers": "~1.6.3",
  "continuation-local-storage": "^3.1.0",
  "pplogger": "^0.2",
  "auth-paypal": "^2.0.0",
  "wurfl-paypal": "^1.0.0",
  "analytics-paypal": "~1.0.0"
}
```

With the logic dictating which package would be sourced from where being unclear here, a few questions arose:

- What happens if malicious code is uploaded to npm under these names? Is it possible that some of PayPal's internal projects will start defaulting to the new public packages instead of the private ones?
- Will developers, or even automated systems, start running the code inside the libraries?
- If this works, can we get a bug bounty out of it?
- Would this attack work against other companies too?

So I started working on a plan to answer these questions. The idea was to upload my own "malicious" Node packages to the npm registry under all the unclaimed names, which would "phone home" from each computer they were installed on. If any of the packages ended up being installed on PayPal-owned servers — or anywhere else, for that matter — the code inside them would immediately notify me.

At this point, I feel that it is important to make it clear that every single organization targeted during this research has provided permission to have its security tested, either through public bug bounty programs or through private agreements. Please do not attempt this kind of test without authorization.

Thankfully, npm allows arbitrary code to be executed automatically upon package installation, allowing me to easily create a Node package that collects some basic information about each machine it is installed on through its preinstall script.

To strike a balance between the ability to identify an organization based on the data, and the need to avoid collecting too much sensitive information, I settled on only logging the username, hostname, and current path of each unique installation. Along with the external IPs, this was just enough data to help security teams identify possibly vulnerable systems based on my reports, while avoiding having my testing be mistaken for an actual attack.

One thing left now — how do I get that data back to me?

Knowing that most of the possible targets would be deep inside well-protected corporate networks, I considered that DNS exfiltration was the way to go.

Sending the information to my server through the DNS protocol was not essential for the test itself to work, but it did ensure that the traffic would be less likely to be blocked or detected on the way out.

The data was hex-encoded and used as part of a DNS query, which reached my custom authoritative name server, either directly or through intermediate resolvers. The server was configured to log each received query, essentially keeping a record of every machine where the packages were downloaded.

[I'll note that this approach works beautifully. I use it for GRC's DNS Spoofability test and also as an ultra lightweight means of allowing GRC's more recent apps to check for updates without making explicit TCP connections. If, for example, you do an NSLOOKUP for "sqrl.ver.grc.com" you'll receive a virtual IPv4 address containing the current release number of GRC's SQRL client for Windows.]

With the basic plan for the attack in place, it was now time to uncover more possible targets.

The first strategy was looking into alternate ecosystems to attack. So I ported the code to both Python and Ruby, in order to be able to upload similar packages to PyPI (Python Package Index) and RubyGems respectively.

But arguably the most important part of this test was finding as many relevant dependency names as possible.

A few full days of searching for private package names belonging to some of the targeted companies revealed that many other names could be found on GitHub, as well as on the major package hosting services — inside internal packages which had been accidentally published — and even within posts on various internet forums.

However, by far the best place to find private package names turned out to be… inside javascript files.

Apparently, it is quite common for internal package.json files, which contain the names of a javascript project's dependencies, to become embedded into public script files during their build process, exposing internal package names. Similarly, leaked internal paths or require() calls within these files may also contain dependency names. Apple, Yelp, and Tesla are just a few examples of companies who had internal names exposed in this way.

During the second half of 2020, thanks to @streaak's help and his remarkable recon skills, we were able to automatically scan millions of domains belonging to the targeted companies and extract hundreds of additional javascript package names which had not yet been claimed on the npm registry.

I then uploaded my code to package hosting services under all the found names and waited for callbacks.

[Okay, so just to be clear: What Alex was exploring was the idea that there might be a fundamental flaw in the way package dependencies are currently being resolved within our industry, such that the building process for packages may first look to public repositories for needed software libraries BEFORE attempting to find and resolve dependencies which require non-public libraries.]

Alex writes: "The success rate was simply astonishing." — And I would add: Horrifying!

From one-off mistakes made by developers on their own machines, to misconfigured internal or cloud-based build servers, to systemically vulnerable development pipelines, one thing was clear: squatting valid internal package names was a nearly sure-fire method to get into the networks of some of the biggest tech companies out there, gaining remote code execution, and possibly allowing attackers to add backdoors during builds.

This type of vulnerability, which I have started calling dependency confusion, was detected inside more than 35 organizations to date, across all three tested programming languages. The vast majority of the affected companies fall into the 1000+ employees category, which most likely reflects the higher prevalence of internal library usage within larger organizations.

Due to javascript dependency names being easier to find, almost 75% of all the logged callbacks came from npm packages — but this does not necessarily mean that Python and Ruby are less susceptible to the attack. In fact, despite only being able to identify internal Ruby gem names belonging to eight organizations during my searches, four of these companies turned out to be vulnerable to dependency confusion through RubyGems.

One such company is the Canadian e-commerce giant Shopify, whose build system automatically installed a Ruby gem named shopify-cloud only a few hours after I had uploaded it, and then tried to run the code inside it. The Shopify team had a fix ready within a day, and awarded a $30,000 bug bounty for finding the issue.

Another $30,000 reward came from Apple, after the code in a Node package which I uploaded to npm in August of 2020 was executed on multiple machines inside its network. The affected projects appeared to be related to Apple's authentication system, externally known as Apple ID.

When I brought up the idea that this bug may have allowed a threat actor to inject backdoors into Apple ID, Apple did not consider that this level of impact accurately represented the issue and stated:

Achieving a backdoor in an operational service requires a more complex sequence of events, and is a very specific term that carries additional connotations.

However, Apple did confirm that remote code execution on Apple servers would have been achievable by using this npm package technique. Based on the flow of package installs, the issue was fixed within two weeks of my report, but the bug bounty was only awarded less than a day prior to publishing this post.

The same theme of npm packages being installed on both internal servers and individual developer's PCs could be observed across several other successful attacks against other companies, with some of the installs often taking place hours or even minutes after the packages had been uploaded.

Oh, and the PayPal names that started it all? Those worked too, resulting in yet another $30k bounty. Actually, the majority of awarded bug bounties were set at the maximum amount allowed by each program's policy, and sometimes even higher, confirming the generally high severity of dependency confusion bugs.

Other affected companies include Netflix, Yelp and Uber.

Despite the large number of dependency confusion findings, one detail was — and still is, to a certain extent — unclear: Why is this happening? What are the main root causes behind this type of vulnerability?

Most of the affected organizations were understandably reluctant to share further technical details about their root causes and mitigation strategies, but a few interesting details did emerge during my research and from my communication with security teams.

For instance, the main culprit of Python dependency confusion appears to be the incorrect usage of an "insecure by design" command line argument called --extra-index-url. When using this argument with pip install library to specify your own package index, you may find that it works as expected, but what pip is actually doing behind the scenes goes something like this:

- Checks whether library exists on the specified (internal) package index
- Checks whether library exists on the public package index (PyPI)
- Installs whichever version is found. If the package exists on both, it defaults to installing from the source with the higher version number.

Therefore, uploading a package named library 9000.0.0 to PyPI would result in the dependency being hijacked in the example above.

Although this behavior was already commonly known, simply searching GitHub for --extra-index-url was enough to find a few vulnerable scripts belonging to large organizations — including a bug affecting a component of Microsoft's .NET Core. The vulnerability, which may have allowed adding backdoors to .NET Core, was unfortunately found to be out of scope in the .NET bug bounty program.

Ruby's gem install --source also works in a similar way, but I was unable to confirm whether its usage was the root cause of any of my findings.

Sure, changing --extra-index-url to --index-url is a quick and straight-forward fix, but some other variants of dependency confusion were proven much harder to mitigate.

JFrog Artifactory, a piece of software widely used for hosting internal packages of all types, offers the possibility to mix internal and public libraries into the same "virtual" repository, greatly simplifying dependency management. However, multiple customers have stated that

Artifactory uses the exact same vulnerable algorithm described above to decide between serving an internal and an external package with the same name. At the time of writing, there is no way to change this default behavior.

JFrog is reportedly aware of the issue, but has been treating its possible fix as a "feature request" with no ETA in sight, while some of its customers have resorted to applying systemic policy changes to dependency management in order to mitigate dependency confusion in the meantime.

Microsoft also offers a similar package hosting service named Azure Artifacts. As a result of one of my reports, some minor improvements have been made to this service to ensure that it can provide a reliable workaround for dependency confusion vulnerabilities. Funnily enough, this issue was not discovered by testing Azure Artifacts itself, but rather by successfully attacking Microsoft's own cloud-based Office 365, with the report resulting in Azure's highest possible reward of $40,000.

For more in-depth information about root causes and prevention advice, you can check out Microsoft's white paper "3 Ways to Mitigate Risk When Using Private Package Feeds".

While many of the large tech companies have already been made aware of this type of vulnerability, and have either fixed it across their infrastructure, or are working to implement mitigations, I still get the feeling that there is more to discover.

Specifically, I believe that finding new and clever ways to leak internal package names will expose even more vulnerable systems, and looking into alternate programming languages and repositories to target will reveal some additional attack surface for dependency confusion bugs.

The power of this should be breathtaking. What if Alex was not a good guy? What if he was not the first to ever have this thought? What if a transient incursion was made into a massive corporate network, simply by having that corporation's build system pull malicious code from a public repository? Such code could be posted, exploited, then removed to eliminate the chance of its discovery. **Yikes.**