SECURITY NOW!

Transcript of Episode #807

Dependency Confusion: The Build Chain Hack

Description: 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.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-807.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-807-lq.mp3

SHOW TEASE: It's time for Security Now!. Steve Gibson is here, and lots to talk about. We'll have more on that SHAREit vulnerability. Apparently it's been fixed now, but we'll tell you how it happened. We also have a lot more from Microsoft on the Solorigate story. And we'll talk about Dependency Confusion, a real security vulnerability that is causing a lot of consternation in businesses all over. So we'll talk about that and a lot more, coming up next on Security Now!. Stay tuned.

Leo Laporte: This is Security Now! with Steve Gibson, Episode 807, recorded Tuesday, February 23rd, 2021: Dependency Confusion.

It's time for Security Now!, the show where we protect your security, privacy, safety, knowledge base with Mr. Steve Gibson from GRC.com. You are knowledge base protection.

Steve Gibson: Just when you thought it was safe to leave the house.

Leo: No, no.

Steve: No, actually, the house is where a lot of these problems are happening.

Leo: No leaving the house. Yeah, it's true, isn't it, yeah. Hi, Steve.

Steve: Hello, my friend. This is 807, Security Now! 807.

Leo: Yes, yes.

Steve: Titled "Dependency Confusion." And I mentioned last week that if nothing earthshaking happened in the intervening week, this is what we would be talking about. And it's hard to imagine that anything could be earthshaking enough to push this one further away. I also mentioned that our Picture of the Week last week sort of was meant to tie in with this. But when I looked more closely at this, I thought, okay, this needs its own show. And no one is going to think this was the wrong decision. Just wait till everyone gets a load of the, I mean, the breathtaking mistake that the entire industry has been making and was caught out about. And arguably, it is so pervasive that only the major players have probably noticed this and fixed it. There will no doubt be lots of little backwaters that are going to be bitten. So this is huge. The subtitle is "The Build Chain Hack," and it's our main topic.

But there's a lot to talk about. We're going to follow up on the Android SHAREit app sale that we talked about last week. We're going to look at a clever new means of web browser identification and tracking, like yet another new way of doing this, which all the browsers at the time of this research are vulnerable to. Actually, the Brave browser got a little advance heads-up from these guys, and so they've already fixed it. But none of the other browsers have yet, so they will be. And also a little mistake, speaking of the Brave browser, that had a big effect.

I also want to remind our listeners about the ubiquitous presence and tracking of viewing beacons, which are now present in virtually all commercial email, just for those who might not be thinking about that. We'll also take a look at Microsoft's final SolarWinds Solorigate, as they named this thing, report, and another example of the growing trend of mobile apps being sold and then having their trust abused. This appears to be like a thing now.

I'm going to share a post from the weekend over in the GRC forums about a dramatic improvement in SSD performance that one of our testers found 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 I am sure will agree was and perhaps still is just a breathtaking oversight in the way today's complex software products are assembled which creates an inherent massive vulnerability across the entire software industry. I mean, this is not one vendor's widget that had a glaring mistake for a long time. This is, like, stunning. And Leo, because you understand how this stuff is assembled, assembling much of it yourself, I'm just glad you're seated for the podcast.

Leo: I'm ready. I am sitting on a ball, so I could fall off, if it's too shocking.

Steve: Oh, okay. Well, yes. Widen your stance, my friend.

Leo: I think I know where you're going with this, not having read the show notes yet. But I certainly know where there would be a supply chain issue with dependencies. So I can't wait to hear about this. Is it time for our Picture of the Week, Steve?

Steve: It is. And this one is a little, I mean, I guess our pictures are always a little geeky.

Leo: A little. A little.

Steve: That's the nature of this podcast. But this one is a little over - this is maybe over the top. So it shows this guy looking at a screen, and it says "This website doesn't [in red] use cookies." And then there's a button, "Got it, don't show again," which he clicks. And then he sits there for a minute, and he thinks, wait a minute, if it doesn't use cookies...

Leo: Don't do it, no.

Steve: Yes. So he clicks Refresh, and up comes the same screen: "This website doesn't use cookies. Got it, don't show again."

Leo: Proof. Proof.

Steve: Yes. Which of course is a problem because, as we know, something like cookies, some sort of tracking mechanism, is the only way of maintaining state from one browser request to another.

Leo: There you go.

Steve: So it kind of needs to have cookies or something if it's going to be able to remember that you clicked on, okay, yeah, fine, don't show it again. Which means - anyway, it's sort of self-contradictory. As I said, a little bit on the over-the-top geeky side.

Leo: Oh, but I love it because our audience gets it; right? Yup.

Steve: Yup, exactly. So 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...

Leo: Steve, it wasn't buggy. It did exactly what its creators wanted it to do.

Steve: Yes. Unfortunately. And so we explained how Trend Micro's ZDI, their Zero Day Initiative folks, had tried to contact SHAREit's publisher 90 days earlier with a disclosure of the very many worrisome problems they had found, and how after 90 days of no response they decided they'd done their due diligence and that at this point the many problems were so bad that now the responsible thing to do was to inform the public of the danger that those more than one billion users with a "b" remained in. So 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, maybe then its publisher would react. And we wondered also what Google would be thinking about this, still having that app on their Google Play Store.

And indeed last Friday the Singapore-based SHAREit company issued a press release containing two bullet points. First one says: "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." Okay.

Second bullet point: "On February 15th, 2021" - which was last Monday - "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 19th, 2021, we released a patch to address the alleged vulnerabilities." So looks like it took a public exposure and embarrassment. No doubt they heard from people, maybe Google, to get these guys to actually move because nothing, you know, three months of waiting and Trend Micro trying to say, hey, take a look at this, they were just blown off. Until, okay, now it's time.

I went looking for any update from Trend Micro, but so far there's nothing. If they do decide to verify the fixes, it might take a while. And this was only just last Friday that these SHAREit folks finally decided, ooh, okay, we'd better address this. But also Trend Micro's original disclosure from the Monday before was comprehensive enough to allow anyone who wished to, to verify these things independently; and, unfortunately, any bad guys who wished to, to take advantage of the more than one billion-plus downloads and installs of this SHAREit app.

So I also went over to the Google Play Store to see whether there might be an update history of any kind. I didn't find one. But I did notice that the app showed as being updated yesterday, meaning Monday, February 22nd, and they had in their press release said they had updated it on February 19th. So looks like it's now getting suddenly a flurry of updates. So maybe they're still fixing things that are being - maybe there's some dialog with the security community with Trend Micro that we're not aware of.

Again, as I said, Trend Micro as of yesterday didn't have anything. But it looks like this thing is finally going to get itself fixed, and hopefully Trend will stay on them to push this thing to a point where they now believe that it's safe. So we can hope. Anyway, an example of a security research firm being ignored, doing the right thing, being responsible, giving its publisher plenty of time, 90 days, and then saying, okay, we're lowering the boom. And only then did they fix it. And of course the downside with this is that it's going to probably take some length of time for some billion-plus users to update this. And until that happens, there are a billion-plus vulnerabilities.

And remember that this is not the app itself which is vulnerable, but the design of the app has permissions which allow its presence in the Android device to be abused by anything that the user installs that is aware of SHAREit's problem and the fact that it's probably - there stands a good chance of it being installed in the phone. So basically it's like the Android OS itself has been made far less secure by having SHAREit present. And so hopefully Google's aware of this. We'll see what happens.

This week in web browser tracking, or maybe where there's a will, dot dot dot. And there sure does appear to be a large amount of will behind browser tracking. I mean, it just must be that there's - it's just so valuable to know where we go on the Internet and to be able to track us. And we've been talking about this from the beginning of the podcast. From the day this podcast was born 15 years ago, tracking was an issue. And of course it's enabled by the pervasiveness of, well, entities like Google that have their hooks into us no matter where we go, and also large advertising networks that similarly are pushing their own content onto websites everywhere.

And it's the pervasiveness of one entity that arranges to get our browsers to make a query back to them, no matter where we go, that then wants to know, oh, who is that who has made that query? What do we know about them? And of course I invested a lot of my time in the early days where third-party cookies, third-party tracking, as they were called, cookies, and the forensics of those I investigated back with some code on GRC. Now the idea of using third-party cookies seems almost 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 this week their paper on the use of favicons for tracking during this weeks' virtual Network and Distributed System Security symposium, which is NDSS 2021. Their 19-page paper is titled "Tales of Favicons and Caches: Persistent Tracking in Modern Browsers." For anyone who wants more detail, although you're really not going to need it because I'm going to explain this all, I do have a link to their 19-page PDF in the show notes.

So for those who are not aware, the favicon is the tiny little icon that appears typically at the left of browser tabs and also, as its name suggests, when we save a website link as a favorite. So, for example, GRC has what I call the "Ruby G." TWiT has that cute little TWiT AND-gate icon with an eyeball, standing on its two feet. GRC's SQRL forums present the SQRL logo. And pretty much every site you go to will have its own favicon. And yes, I've heard it pronounced fav-i-con. But just as a lib is a library...

Leo: We know where you stand. We know where you stand.

Steve: Yes.

Leo: Yes.

Steve: A favicon is a contraction of favorite icon. So no, I am not of the fav-i-con crowd.

Leo: I am. I say fav-i-con and lib.

Steve: No kidding.

Leo: Oh, yeah.

Steve: All right.

Leo: I guess, you know, really the two positions are the pronunciation is determined by what it's an abbreviation for, your position. Mine is the pronunciation is determined by how it is used in its orthography, once the contraction is made. So if it weren't for that you know that it was a favorite, if you looked at favicon, you would say fav-i-con. But because you know it's a favorite, you say it's...

Steve: You're saying orthography versus morphography.

Leo: Perfect. Semantics versus syntax. Perfect.

Steve: Okay. And of course we have "jif" versus "gif," but let's not go there now.

Leo: Unh-unh. Unh-unh.

Steve: In any event, it turns out that, get this, all browser makers have assumed that since a whatever you want to call it, a fav-i-con or a fave-icon - actually, fav-i-con does kind of roll off the - anyway.

Leo: Uh-oh. Stick with morphology.

Steve: All browser makers have assumed that 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 cookies are cleared or even when the browser's other caches are cleared. And to add insult to injury, what's more, both the standard and the incognito mode interfaces of a browser share that 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 browser 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 browsing mode. Since there's only a single favicon cache, it would be possible to determine, for example, whether a browser had ever visited a site when it's in private browsing mode by seeing whether it requests a favicon from the site in question when not in private browsing mode.

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 yet, so it'll ask for it, or it always has one, and thus doesn't need to ask again. The hack is rather brute force, but I'll give them a "C" for clever. So here's how they describe their work in the abstract of their paper.

They said: "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." And of course we've talked about all manner of those sorts of things in the past.

They said: "In this paper we introduce a novel tracking mechanism that misuses a simple yet ubiquitous browser feature: favicons." Or fav-i-cons. They said: "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 subdomain. 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" - and we'll talk about the details of that in a second.

"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's 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." They said: "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" - so that's unique 4.3 billion different browsers - "in two seconds.

"Furthermore, our attack works in all major browsers that use a favicon cache" - in other words, all major browsers - "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."

Okay. So this works since subdomains 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. 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 a chain of 32 subdomains of that main site domain, noting which of the subdomains the browser asks for favicons from, and which it does not. But it never gives the browser any new favicons to store. That pattern of requests will be unique for that browser because, if the browser does ask for the favicon from the primary site, then the site knows that the browser has not yet been unique favicon tagged.

So now the site chooses a new 32-bit identifier, increments a master counter somewhere to get a unique identifier which it will assign to this browser. And it uses that identifier to judiciously and selectively dole out, or not, subdomain favicons as it takes the browser on its 32-subdomain walk to match the 32-bit identifier that it has chosen for the browser. So, yeah, it's kind of brute force. I think it's kind of clunky. But it's diabolical when you consider how none of our browsers have been treating their favicon cashes with the respect it is due.

The attack is currently effective a against Chrome, Safari, Edge - because of course it's Chromium. And it was effective against Brave also since Brave is Chromium, until the researchers gave the Brave developers an early heads-up, which allowed Brave to beat the rest of the pack to the deployment of an effective countermeasure.

And interestingly, Firefox, which I haven't yet mentioned, would also have been vulnerable to the technique, and in a sense they very much wanted to be. But it turns out that a bug in Firefox prevents the attack from working. The researchers explain this. They said: "As part of our experiments, we also tested Firefox. Interestingly, while the developer documentation and source code include functionality intended for favicon caching similar to other browsers," they said, "we identify inconsistencies in its actual usage. In fact, while monitoring the browser [meaning Firefox] 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've reported this bug to the Mozilla team, who verified and acknowledged it. At the time of submission, this remains an open issue. Nonetheless," they said, "we believe that, once this bug is fixed, our attack will work in Firefox, unless they also deploy countermeasures to mitigate the attack." Which of course they would when they fix it.

So anyway, it's kind of cool for Firefox. They had the cache. Turns out, due to a bug, they were never using it. So Firefox was always issuing requests for favicons for all the sites we've been visiting and so was never susceptible to this particular attack. On the other hand, when they fix this, they will speed up Firefox. That's good. And presumably they'll also deploy a fix for this, however they decide to arrange to do that.

So anyway, everybody's going to fix this. They're all on it. If anybody is really worried in the meantime, it turns out that all the various browsers provide some means for disabling their fetching of favicons. I don't know why anyone would. But maybe just to reduce your fingerprint. I can't imagine not having those cute little icons on all my browser tabs. I mean, they're really handy. But I do have three DuckDuckGo search links for Chrome disable Favicon, Safari disable Favicon, and Edge disable Favicon, if anyone just wants to click the links as opposed to putting it into their DuckDuckGo search engine.

So it can be done. It's going to get fixed. It was sort of a clever hack. And so props to those guys. Again, sort of brute force-y, you know, to store some combination of favicons in 32 subdomains off of the main domain. But it could get a unique ID in two seconds. Maybe you could do it in the background. While the main site was showing you its page, it would be sending some other resource off on that 32-website redirection chain. Basically the browser pulls up a subdomain. It receives a 302 redirect. And meanwhile, the browser has asked for the favicon for that subdomain. The browser then goes to the next domain in this chain, and that repeats 31 times. So kind of clever.

But I mentioned that the Brave browser has just fixed a privacy mistake. It was kind of bad. And I'm a little annoyed with them. So the privacy mistake was that the Brave browser was sending DNS lookup queries for .onion domains - in other words, domains that are supposed to be hidden, right, by Tor - out onto the public Internet, rather than routing them through Tor nodes. This, of course, exposes users' visits to dark websites. The mistake was fixed in a hot fix release of Brave 1.20.108 which was made available last Friday. So anyone who is depending upon Brave's otherwise nifty built-in what they call the "private window with Tor" feature for safer TOR-based anonymity will want to be sure to be running the 1.20.108 or later release.

And 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 DNS checks. Whoops. So the thing I mentioned that's a bit disturbing is that the Brave folks have known about this for some time. It was first reported to the HackerOne bug bounty site by the person who discovered it and thought it was bounty worthy back in the middle of January, on the 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, which would have been 1.21.something. But the flaw was publicly disclosed on Ramble, which pushed the Brave folks to advance their timetable and release the interim 1.20.108. So I'm a bit uncomfortable with the idea that something that's as significant, privacy critical as publicly resolving .onion domains without Tor's privacy protection wouldn't be regarded as an emergency with a patch pushed immediately by the Brave folks. I mean, it's one thing not to offer privacy. But to say that you're offering it and not deliver it, and know you're not delivering it, but still saying you're offering it, when you could fix it, that seems a little wrong to me. But, and I put this in the show notes, I suppose it's a brave new world. Yes.

And I will, before our next break, I will mention, just to sort of put this back on our listeners' radar, while we're on the subject of tracking, I'll note that today a huge percentage of email spam, like virtually all of it, contains embedded tracking beacons. And I know this is not news, but I just kind of wanted to remind everybody about it. These days the transition from text-only 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 an email to contain a serial number-encoded URL back to the mothership. This essentially turns your email client into a web browser so that the act of opening and viewing that 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.

And again, I'm not saying that any of this is news. But I just kind of wanted to refresh everyone's awareness of it. There was a recent report in the BBC news titled "Spy pixels in emails have become endemic." And the report told the story of the fact that today, virtually all commercial email embeds these callbacks to confirm the viewer's reception and display of the email. Many email clients offer add-ons for blocking these little spying beacons. Of course it's also possible to configure most clients not to display any images at all, or even go further and display the email as old-school plaintext. So there's no big takeaway for our listeners, but I just wanted to remind our security and privacy anti-tracking friends that there is another vector of monitoring that is going on, often unobserved and underreported. So just a heads up and reminder about that.

Leo: Yeah, that's why I use text-only email.

Steve: Yes.

Leo: HTML email is a bad idea. But people persist.

Steve: Yup.

Leo: It's actually worse than tracking pixels. You could embed malware in it, if it's a web page, and you don't have all the patches. Could be just as bad as that. All right.

Steve: And during the transition, remember that you used to be able to configure, what was it, Eudora and then Outlook, you could say "Use HTML," or "Only render as text." And that was our advice back then was oh, my god, you do not want IE having a presence in your email because...

Leo: I will only use email clients, MUAs, Mail User Agents that support text only. And I do, on the Mac, it's called MailMate, and it supports only text. There's TXT. RTF is probably more safe, I would bet.

Steve: Yes.

Leo: And then there's HTML, which is not. Now back to Steve.

Steve: So Microsoft's final, hopefully, report on what they named Solorigate. Last Thursday Microsoft posted the results of their research into the details of their own SolarWinds breach. And I will, I have for the show notes, paraphrased from their larger posting, which is a little self-serving and so forth.

They said: "We have now completed our internal investigation into the activity of the actor and want to share our findings, which confirm that we have 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." Okay, which that really wasn't a deal, but okay. "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."

They said: "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 of 2021, when the attempts stopped. There was no case where" - and this is interesting phraseology. I had to read this one a couple times. What? "There was no case where all repositories related to any single product or service was accessed." Which sounds like, how can we put the best possible spin on this? "There was no access where all repositories related to any single product or service were accessed." Okay.

Leo: Yeah. Parse that, yeah.

Steve: Yeah. They said: "There was no access to the vast majority of source code." Okay.

Leo: Vast majority. Vast majority.

Steve: That's right, uh-huh.

Leo: No access.

Steve: "For nearly all of code repositories accessed, only a few individual files" - you know, the juicy ones - "were viewed."

Leo: Just the good stuff, yeah.

Steve: Yeah. As a result of a repository search. In other words, the bad guys searched the repository and only viewed a few things.

Leo: Oh, good.

Steve: Yeah, gee. You think maybe they were the dumb files? Or maybe they were the good files because, you know, we know that they have a repository search. Okay, anyway. They said: "For a small number of repositories" - again, the ones the attackers wanted; right? - "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, and identity); a small subset of Intune components; and a small subset of Exchange components." Again, we know how good these guys are. They were probably worried about being, like any of this being caught. So they only got the things they wanted. Anyway, thank you, Microsoft.

"The search terms used by the actor indicate the expected focus on attempting to find secrets." Okay, now, this is interesting, and props to Microsoft. They said: "Our development policy prohibits secrets in code, and we run automated tools to verify compliance."

Leo: Hmm.

Steve: Yes. "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." And so I'll take note of this sentence. "Our development policy prohibits secrets in code, and we run automated tools to verify compliance." Yay.

Now, 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," and I have that in quotes, accounts and passwords are embedded? We should all have learned long ago that secrets can never be kept, least of all embedded in firmware. That's bad. And so, again, yay to Microsoft for recognizing code should never have secrets.

Anyway, they finish: "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" - unfortunately they said "learnings that we want to emphasize."

Leo: That's a Microsoft logism.

Steve: Oh, Leo.

Leo: Mary Jo Foley has mentioned it in the past, yeah. I don't like it.

Steve: Yes, we have some learnings to share.

Leo: Learnings.

Steve: Stand by. Anyway, and those learnings in this case are embracing a Zero Trust mindset and protecting privileged credentials. And then they expand on these two learnings: "A Zero Trust 'assume breach' philosophy is a critical part of defense. Zero

Trust is a transition from implicit trust, which is 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."

And you know, Leo, if they can't get the bugs out of Windows, I'm kind of glad they're at least spending all the money they have on their own internal security because that's better. I mean, at least, as far as we know, Windows doesn't have deliberate bugs that have been installed by bad guys. They're just bugs that Microsoft has made mistakes on. So that's better.

And they said: "We've recently shared guidance for using Zero Trust principles to protect against sophisticated attacks like Solorigate." And then they finish: "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."

So in other words, you don't want to automatically give your local network credentialed access to the cloud because, if bad guys get into your LAN, then they will leverage that automatic access to the cloud to move themselves into the cloud. And so don't do that. That's one of those "learnings" which Microsoft has had.

Leo: Learnings.

Steve: So anyway, a very good point about not having any embedded secrets in code. And it's sort of interesting that they were able to profile the repository queries that the bad guys were making to posit what it was they were probably hoping to find. And of course we also know that getting source code allows you to go over it from the standpoint of a bad guy perspective, finding things that the authors haven't found. And I would argue maybe they weren't so much looking for secrets as looking at, oh, look, here's a type confusion problem that we can leverage into a buffer overflow in Exchange Server. So buckle up. We'll see.

Leo: Wow. Wow.

Steve: 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. And one instance is an event. Two, I would say we're beginning to see a trend. And yes, perfectly placed sigh there, Leo.

Leo: Mm-hmm.

Steve: 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 had suddenly become 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 ads. This suddenly troublesome app was just called Barcode Scanner. Kind of generic. It was finally identified as the source of this nuisance and is now tracked as Android/Trojan.HiddenAds.AdQR.

Malwarebytes determined that recent malicious updates of that 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 to it 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. So there was a window of time during which the new owner had basically made the good app go bad, was generating revenue from this torrent of ads they were pushing, presumably realizing that their time in the sun was limited before Google would yank it. But they were going to make as much money during this window as they could.

So what was equally suspicious was that, at the time, Lavabird did not respond to requests for comment from Malwarebytes. However, the vendor has since reached out to Malwarebytes with an explanation of the situation. So that is to say, Lavabird said, okay, look, here's what's going on. So on the 12th of February, 12th of this month, which was, what, Friday, couple Fridays ago, 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.

Leo: Ah.

Steve: Yup. Apparently, Lavabird was an escrow-like intermediary between the original author and seller and its eventual purchaser. Lavabird purchased the Barcode Scanner on November 23rd. And the subsequent "space team" deal was agreed to two days later, on November 25th. Lavabird told Malwarebytes that they develop, sell, and buy mobile applications. So they're sounding a little bit fishy. But the monetary transactions for mobile applications, it turns out, 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 confirming the purchase. And it was that provisional buyer who first pushed the malicious update out to Barcode Scanner users. And 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 7th. 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, back on November 27th, which was two days after the original November 25th purchase. And subsequent updates obfuscated the malware's code. Until January 5th, when the app was yanked from Google Play.

So this tactic was apparently more surprising to Malwarebytes than it would be to our 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 updated the app, that's a lot of infections. And by being able to modify the app's code before full purchase and transfer, they were also able to test whether their malware would go undetected by Google Play on another company's account." That is, while it was still under another company's account.

So it seems we need a new flag added to Android apps in the Google Play Store. Or maybe like a flag added to the update process. An app's change of ownership need not be a malicious event, but it certainly can be. And as I said, we're now seeing instances where good Android apps are being purchased by somebody who wants to exploit them. And you could hardly blame the original author, who created an app. Maybe it wasn't generating any revenue. It was clean. It was therefore very popular. So he cashes out, you know, this 10-plus million user base for some money. That app is turned into malware. There's now a window of opportunity before Google gets wise to its misbehavior and removes it. But if during that length of time it can participate in bitcoin mining, or it can be showing ads that generate revenue, then unfortunately this is a viable profit model for taking free Android apps and despoiling them for some period of time before they go bad.

So the point is, though, the original app wasn't bad. It's an update which is bad. So it would seem useful when an update is presented to also note to the updating end user, oh, by the way, ownership of this has changed hands. And in fact maybe it's because of that, even what I'm suggesting, that the purchaser said, look, we're going to push an update before we confirm the purchase just so we can make sure we can. Of course that update could always be reverted if the purchase didn't go through. But still, it would avoid this approach.

But it seems to me with there being the gazillion free Android apps that there are, super popular because they're free, and also because they're behaving themselves, we're going to see this attempt to exploit the trust and credibility which has been built up for exactly that reason. And I don't know how we solve that problem. But we are now beginning to see the emergence of this as a new profit model.

What hasn't been a big profit model for the last, what is it, 17 years? I can never be accused of dinging people for constant update fees.

Leo: Grasping Steve, yeah.

Steve: So I do try to keep an eye on postings in the GRC forums. And I saw a recent posting titled "RS shows VERY [in all caps] obvious improvement after one pass of SR 6." And this was posted by the TheDukeofURL on this past Saturday at 4:14, or 4:16, rather. I've got a link in the show notes for anyone who's interested in seeing the whole thread. But TheDukeofURL posted last Saturday: "Been trying out SR 6 / RS scanning" - meaning ReadSpeed scanning, our benchmark - "and testing on a bunch of drives I have." He said: "Spinning, SSD, laptop, SATA, and PATA," you know, Parallel ATA. He says: "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, Multi-Level Cell, which has been in service for close to five years."

So he shows two ReadSpeed outputs, a before where he shows the 128GB M4 Crucial, and you know how ReadSpeed does benchmarks at 0, 25, 50, 75, and 100% points in the drive. So we have 258 mbps, 283, 309, 296, and 296. Then after running SpinRite on Level 3, same drive, 533.5, 533.5, 533.5, 533.5, and 533.6. So that's using a Level 3. Level 3 is the minimal writing, the "Refresh the Surfaces." So it basically just reads everything and writes everything back.

So what the benchmark clearly demonstrates is that the act of reading and rewriting the entire storage surface of the SSD has essentially doubled its subsequent read performance as measured in ReadSpeed's 1GB reads from five locations across the SSD. And I'll note that many users have reported that this does in fact result in a subsequent noticeable improvement in the actual perceived performance of the media. So it's not just some benchmarking attribute.

There's 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 parallelism, through the parallel reading across many, we could think of it as cores, which are independent channels in the SSD. And that a brand new SSD will have a uniform "map" that maximizes the read performance of that device.

So the theory goes that, over time, the relocation of regions introduces an unavoidable fragmentation of this map which causes clumping up of the reads, thus reducing the channel parallelism. And as we know, wear leveling is the idea that, as you're using this, if you're writing to what looks like the same logical location, you're actually writing to different physical locations, which the SSD is managing. And so the need to level out the wear so that your writing is not spotty, and so you're not burning out one spot on an SSD, that's causing this mapping fragmentation which reduces the ability of the drive to pull from its surface in parallel.

And it's been noted that performing the SSD's Secure Erase operation immediately restores the drive's maximum performance as seen under ReadSpeed. One assumption is that this has effectively defragmented the SSD's logical-to-physical mapping table, thus restoring its full performance. But I believe that's not what has happened. SSDs 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 zeroes for any read because the drive knows that nothing has been stored there yet since the Secure Erase.

And in fact you can see the same thing if you issue trims to the entire drive. If you issue trims, you're not securely erasing, but you are telling the drive there's nothing in these locations. You don't need to worry about them. That has the same effect of unmapping that logical-to-physical association. And suddenly the drive reads at full performance because it's lying. It knows nothing's been stored there, so it just sends back zeroes instantly.

Okay. So what does running SpinRite do? We know that SSDs store data as electrostatic charges in a vast array of very slowly leaking capacitors. It's unnerving, but it's the truth. And 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 turn to a state of maximum entropy. And heating helps that to happen more quickly.

As we know, SSDs rely upon surprisingly advanced built-in data recovery on the fly, it's happening all the time, to deal with the reality of what Allyn Malventano, who's Intel's 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 are relying heavily. So the only way to make sense of what TheDukeofURL's results show us is that in a way that exactly matches the idea of refreshing a spinning drive's magnetic surfaces 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 or trimming will do that. So what had to have happened is that many regions that were being slowed down by their need for extensive on-the-fly error recovery and correction are now once again reading at full speed because their 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 a dramatically improved effect on the drive's long-term reliability, at least for reading.

And I'm not suggesting that everyone should run out and run SpinRite 6 on their SSDs today. Not yet. SpinRite needs to be optimized for this application. For one thing, not all regions are slow. As we've seen, ReadSpeed's detailed output showed wide variations in read performance. And SpinRite is not yet smart enough to selectively rewrite only an SSD's slow spots, which is what we want in order to reduce SSD media wear. Another issue is the whole issue of page alignment of SpinRite's writes. SSDs erase and rewrite their data only in large page-size chunks. SpinRite 6.1 will work in large chunks that are inherently page aligned. SpinRite 6.0 and earlier versions have had a track alignment which was oriented to 63 sectors, which, I mean, that's a horrible number, 63. It's just like almost designed not to line up evenly with any binary sizes.

Leo: That's a prime number; isn't it?

Steve: It's horrible. It can't be prime because six plus three is nine. But it's definitely going to be bad. And then there's trimming; right? We touched on this before. Rewriting the entire drive detrims the whole physical surface, telling it that the entire drive is in use and contains valid data. But the drive's file system knows better. But SpinRite is not only rewriting valid files, it's rewriting the entire surface. Windows, when we talked about this before, can be instructed to retrim a drive.

That's what optimizing a drive now on SSDs does. And you can actually see it says there, if you run optimize, it says retrimming. And Linux we know will periodically run a retrim over its drives. But a future SpinRite will also be natively trim aware. So anyway, TheDukeofURL's posting Saturday was a clear demonstration that SpinRite has, I would argue, at least as much to do in the future as it has in the past. So I am very excited to get us there. And that's what's got all my focus right now.

Leo: Can you trim too often? Can you wear a drive out by trimming too much?

Steve: No. Trimming doesn't write anything to the surface.

Leo: Okay.

Steve: It only talks to the management layer in the controller, basically telling it that these regions have nothing of interest.

Leo: It's zero here, yeah.

Steve: It sort of puts it back to its original state.

Leo: Yeah, yeah.

Steve: Yeah, because the problem is, because of the fact that the SSD can only erase, and in order to write to an SSD, you have to write all zeroes to it, and then you selectively write ones where there were zeroes.

Leo: Oh, I didn't know that. Really.

Steve: Yeah.

Leo: That's interesting.

Steve: Yeah. You have to push the whole thing down. And these pages are huge. They might be like 4 or 8K in size, where a sector is still 512 bytes. So if you were to just rewrite one sector, and if the SSD thought that all of the regions around it were in use, it would have to read that whole thing, erase the whole large block, and then rewrite the whole block with that one sector changed. But if instead, if it had been trimmed to know that none of that was in use, then it wouldn't have to bother with that read/modify/write cycle. And if it knew that it had been previously erased, it could then push all of the bits. I think maybe erase writes ones, and then it just pushes zeroes where it needs them. But anyway, it's like surprising how much technology is in these things.

Leo: Yeah, yeah.

Steve: And it just, you know, we're just unaware of it.

Leo: Yeah. It's very cool, actually.

Steve: Very, very cool.

Leo: Nice. All right.

Steve: Our last break, and then oh my god, Leo. Just oh. Fasten, you know, buckle up.

Leo: I can't wait. We'll get to Dependencies in just a bit. All right. I am all ears. I want to hear about this dependency issue that you're covering here.

Steve: Oh, boy. This will not disappoint. Two weeks ago Alex Birsan put up a posting on Medium that drew a great deal of well-deserved attention. Unfortunately, it's not possible for it to have drawn enough attention. I mean, and we'll see why. And as I said, it was so cool that I punted on talking about it last week so I could make it our topic this week. And as I also mentioned, it relates to our Picture of the Week last week because that was a - in order to explain it to our listeners, we described it as a house of cards, although it was sort of a building of rickety blocks that had inherent dependencies upon the blocks below.

So Alex wrote: "Ever since I started learning how to code, I've been fascinated by the level of trust we put in a simple command like this one." And then he shows "pip install package_name." And for those who don't know, this is a simple means of asking your OS to go out and find from a repository some package and download it and install it into your system.

He says: "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've 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, not surprisingly, 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," he asks, "can this blind trust be exploited by malicious actors? Of course it can."

He says: "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 typoed 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 this problem here on the podcast in the past.

And he said: "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." But that's not what he did. He said: "While attempting to hack PayPal with me during the summer of 2020" - and this guy's an ethical hacker. He says: "Justin Gardner shared an interesting bit of Node.js source code found on GitHub." He said: "The code was meant for internal PayPal use, and in its package.json file appeared to contain a mix of public and private dependencies [which are to say] 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."

And Leo's got the picture of this on the screen. This is just some JSON showing a block named "dependencies." And inside we have one item, "express," showing a version 4.3.0. One is a "dustjs-helpers," v1.6.3. There's one "continuation-local-storage" with a version of 3.1. Then there's "pp" - probably PayPal - "logger" at 0.2 version. "Auth-paypal," 2.0.0. A "wurfl-paypal" with a version 1.0.0. And an "analytics-paypal," 1.0.0. So again, some public things and clearly some private things. But these are all dependencies for something that something is using.

So he says: "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 those libraries? If this works, can we get a bug bounty out of it?" In other words, can I make some money? "Would this attack work against other companies [meaning besides PayPal], too?"

He said: "So I started working on a plan to answer these questions. The idea was to upload my own" - and he has in quotes "malicious" because of course, again, they wouldn't hurt anybody, but thus quotes - "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," he said, "I feel it's important to make it clear that every single organization targeted" - now, I'll explain also that limited targeting was possible.

But, he says: "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." And he says: "Please do not attempt this kind of test without authorization." And of course we know why. We've seen unwitting hackers having agents knocking at their doors, and those agents tend to be very humorless.

So he says: "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." He said: "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," he said, "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."

And he says: "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." He says: "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." He says: "The server was configured to log each received query, essentially keeping a record of every machine where the packages were loaded."

And I'll pause here just to note that this approach does in fact work 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 were to use NSLOOKUP to look up sqrl.ver.grc.com, you'll receive a virtual IPv4 address containing the current release number of GRC's SQRL client. So it's a beautiful way of not raising any false alarms or upsetting anybody, yet still being able to make a lightweight query. And if I were logging, although I'm not, incoming queries, then essentially it is a way of identifying where SQRL clients are. This guy was explicitly logging all of the incoming DNS queries, which allowed him to tie those back to the networks and domains of machines that were inadvertently executing his code.

So he says: "With the basic plan of 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" - that's the Python Package Index - "and RubyGems. But arguably the most important part of this test was finding as many relevant dependency names as possible." That is to say, he wants to figure out what to name his, I don't want to call them trojan, but they kind of are, but his benign trojan packages, what to name them so that they will get pulled by mistake if they're going to be from public repositories.

So he says: "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." He said: "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 in public script files during their build process, thus 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." He says: "I then uploaded my code to package hosting services under all of 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 and use nonpublic internal libraries.

Alex wrote: "The success rate was simply astonishing." And I would add horrifying. He said: "They range 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 surefire 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," he wrote, "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 greater than a thousand employees category, which most likely reflects the higher percentage 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 Ruby gems.

"One such company is the Canadian ecommerce 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 Apple's 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" - you bet they didn't want to - "and they said 'Achieving a backdoor in an operational service requires a more complex sequence of events and is a very specific term that carries additional connotations.'" Whatever the hell that means. Anyway: "Apple however 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 report." Yet he got $20,000.

"The same theme from npm packages being installed on both internal servers and individual developer PCs could be observed across several other successful attacks against other companies, with some of the installs often taking place within hours or even minutes after the packages had been uploaded. Oh," he says, "and the PayPal names that started it all? Those worked, too, resulting in yet another $30,000 bounty." He said: "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 exactly how their root causes and mitigation strategies worked. But a few interesting details did emerge during my research and from my communication with security teams."

He said: "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." Okay. Get a load of this.

"First, checks whether library exists on the specified probably internal package index. Two, checks whether library exists on the public package index. In other words, PyPI. Three, installs whichever version is found. If the package exists on both, it defaults to installing from the source declaring 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. Any library that doesn't have a version higher than 9000 will be superseded by its presence, the spoofed presence on the public repository."

He says: "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." So he didn't make any money on that one. "Ruby's gem install source also works in a similar way," he says, "but I was unable to confirm whether its usage was the root cause of any of my findings." He said: "Sure, changing extra-index-url to index-url is a quick and straightforward 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 this writing, there is no way to change this default behavior." He says: "JFrog is reportedly aware of the issue, but has been treating this 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." He says: "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" this guy made. He says: "For more in-depth information about root causes and prevention advice, you can check out Microsoft's white paper 'Three Ways to Mitigate Risk When Using Private Package Feeds.'"

He said finally, as he's finishing: "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," he says, "I still get the feeling that there is more to discover. Specifically, I believe that finding new and clever ways to leak internal names will expose even more vulnerable systems, and looking into alternate programming languages and repositories to target will reveal some additional attack surfaces for dependency confusion bugs." And for what it's worth, he suggests to people who are reading that, and of course people listening to this podcast, that there may be more money to be made because these things are horrific, and they are generating maximum bounty payouts every single time they are found to infect or affect a major organization.

So hopefully the power of this is breathtakingly obvious. And you have to ask, I mean, think about this. What if Alex was not a good guy? What if he was not the first ever to have had this thought? What if a transient incursion was made into a massive corporate network simply by having the corporate build system or its servers pull malicious code from a public repository? Such code could be posted, exploited, then removed to eliminate the chance of discovery. I mean, it is horrifying.

Leo: Yeah.

Steve: So props to Alex for thinking of this. And, boy, what a glaring oversight on the part of the way these systems are being built.

Leo: I mean, any time you build software, you often will pull dependencies from sources like, if you're doing it in Linux, from the Linux repositories. These are public repositories.

Steve: Yup.

Leo: Is the problem here that these are private repositories?

Steve: It's that there's a hybrid. So they declare the dependencies. They were making the assumption that their private repositories would never exist on public repositories.

Leo: Got it, yeah.

Steve: And so the only match would be a local match. So it turns out...

Leo: So anything that's in public, they will check signatures and things like that. But they presume that anything on a private repository is safe.

Steve: Well, yes. And that only their private repository would have a name match.

Leo: Could be the source, right, yeah, yeah.

Steve: What this guy was assuming was what happens if we deliberately create a name collision, would the repository get pulled publicly first?

Leo: Right.

Steve: And in fact in one case the build system compares the version numbers and deliberately chooses...

Leo: The higher number, yeah.

Steve: ...what it thinks is the newer one.

Leo: Yeah, yeah, yeah.

Steve: In which case you could override with a bogus public posting on a repository. You override it just by declaring it a newer version than the public, like version 9000.

Leo: Right. Yeah, that's what these manifests often say is this version or better.

Steve: Yup.

Leo: And I guess the caret, that's what the caret signified in the repository you show.

Steve: Right.

Leo: Yeah. But, I mean, everything that builds software uses repositories. And most package managers on most systems will pull in code from a variety of public repositories.

Steve: Yeah.

Leo: And it's kind of on the user to validate those, really.

Steve: That's what makes this so breathtaking, Leo, is it's like, wait a minute. I mean, the system has been that fragile. In fact, it's so fragile it's hard to believe it's that fragile.

Leo: Yeah. Well, we all kind of pretend we...

Steve: Yeah, I know.

Leo: Yeah, just it'll work out. Just hope for the best.

Steve: We just go tiptoe through the tulips and hold our breath.

Leo: Yeah, just hope for the best. Wow, yeah. There's always that risk, you know, when you're pulling dependencies that you could be pulling a malicious dependency. There's a certain amount of trust that the repositories are maintained, and somebody's paying attention. I don't know if that's the case in a lot of situations, honestly.

Steve: Unfortunately, as we showed in our Picture of the Week last week, Wilbur in Nebraska...

Leo: That one guy, yeah.

Steve: ...is maintaining, you know, exactly. And in fact somebody tweeted me that there's one guy in Sweden maintaining cURL.

Leo: Yes, that's right.

Steve: That's the maintainer. And we are so dependent upon cURL working correctly. And it's like, let's hope it stays good.

Leo: Yeah. It's been a big deal for some time, actually. There's been a lot of talk about that. Daniel Stenberg's the benevolent dictator for life is what they call these guys.

Wow, what a story. Mr. Gibson, you've done it again, as always. This is a fascinating listen. I know everybody who listens or has heard this show knows that's the case. Let me tell you where you can get this show and previous shows and all future shows. Start at Steve's site, GRC.com. He has 16Kb audio, 64Kb audio, nice transcriptions written by a human being so you can read along or use them to search for anything in all 807 episodes. GRC.com.

While you're there, pick up a copy of SpinRite, currently 6.0. As you know, 6.1 is in process. That's going to be an amazing choice, not just for spinning drives, but for solid-state drives. And if you buy it now, you'll automatically get upgraded to 6.1. Plus you can participate in the development of 6.1. That's at GRC.com. Lots of free stuff there, too, including his feedback form. If you want to ask a question or leave a suggestion, GRC.com/feedback. Steve also takes suggestions on his Twitter handle, @SGgrc. His DMs are open, as the kids say. @SGgrc.

We have copies of the show at our website, TWiT.tv/sn for Security Now!. You can download audio or video there. You can also watch us do it live. We do the show live every Tuesday around about 1:30 Pacific, 4:30 Eastern, 21:30 UTC. The live stream's at TWiT.tv/live. People who are watching live, I encourage you to chat because the chatroom is also watching live at irc.twit.tv. The on-demand versions, as I mentioned, are at the website. You can also watch the YouTube channel. You can get them from Steve. But maybe the easiest thing to do is find a podcast client and subscribe. That way you can get it the minute it's available of a Tuesday afternoon so you can listen to it instantly. Steve, we have concluded the project for today. Thank you.

Steve: We'll be back next week with 808.

Leo: 808. See you then.

Steve: Bye.