SECURITY NOW!

Transcript of Episode #311

Anatomy of a Security Mistake

Description: This week, after catching up with a collection of interesting security events, Steve and Leo take a close look at a recently discovered security coding error, examining exactly how and why it occurred, to understand how easily these kinds of mistakes can be made - and how difficult it can be to EVER find them all.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-311.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-311-lq.mp3

Leo Laporte: It's time for Security Now!, the show that covers your security and your privacy online. Leo Laporte here in a little bit of an unusual situation. This is what we call our "living room set" in the new TWiT Brick House studio. Steve Gibson is here. Steve, first of all, thank you for coming all the way up to do TWiT on Sunday. It was really fun having you.

Steve Gibson: It was a lot of fun, Leo, to be there, to see the first podcast, well, to be part of the first podcast from the new studio.

Leo: Yeah. It was a great show, and it was all my old friends, and so that was kind of special. And normally we'll be doing this show - in fact, if somebody tunes in next week and didn't know that we were at a new studio, they'd probably just think nothing had changed. But we will be back in my office next week. And the office is built to simulate the old TWiT Cottage.

Steve: Oh, neat. I think that's great for continuity and, yeah.

Leo: Yeah, it'll be fun. But I kind of like this, too. I mean, sitting here comfortably with you and having you on the monitor and all that, that works pretty well, too. So today we're going to talk about...

Steve: Sort of a fireside chat.

Leo: ...the Anatomy of a Security Mistake. What is that?

Steve: I am so excited about this episode because it began with someone tweeted me something that they saw and the details of a new flaw that had just been discovered after 13 years in open source code. It's in the Openwall, in SUSE, in some flavors of PHP, a problem with the implementation of password hashing. And when I looked at what the problem was, the reason I got excited was it's a perfect example of the thing we've been talking about ever since the beginning of the podcast, of why security is so difficult; how programmers could look at this code as much as they want to, and it would look perfect, but there's a subtle bug in what the code does.

And the reason this captured my imagination for the podcast is that during the, I mean, this is going to be a propeller-head podcast. This is going to be a "sit down and focus." But all of our listeners, if they follow with me, I'm going to explain some things about binary representation in computers, how signed and unsigned integers are represented. We need to have a little bit of that background. Then I could explain what this one line of code does and why it's wrong and the obvious security implications. So it's just - it's such a great example, I mean, in the real world, not just generalities, as we're normally sort of forced to use, but an exact explicit example of why security is so difficult. And so Anatomy of a Security Mistake. This is going to be a great podcast.

Leo: I'm the one who always says, oh, use open source software because so many eyes are looking at it, it's so much more reliable, so much more secure. And this has escaped people for years.

Steve: Well, and the way it finally got found was somebody was messing around with password cracking. And I think it was "John the Ripper" was the program being used. And his results didn't match what he expected. And so literally it forced him to drill down into the code, looking for was it his mistake, or was it a mistake somewhere else? And it turns out it was a mistake in a 13-year-old, it's like since 1998, original Blowfish hashing code where - and I've often said how it's necessary to have a debugger sometimes just hit you in the face with the problem because otherwise you could just stare at it all day, and it just looks fine.

Leo: A little insight into the world of programming and how easy it is to make mistakes and how hard it is to find them. All right, Steve Gibson. I have my notes in front of me. Let's start with our security updates, shall we?

Steve: Well, yeah. Anyone who's using iOS may have seen that a couple days ago Apple changed the most recent update we got just a week ago. We talked about 4.3.4, and very quickly on its tail they updated us to 4.3.5. There's a different update, I think they're running different version numbers on the CDMA version with Verizon. But all the AT&T and traditional iOSes and iPads have just been brought up to 4.3.5. The nature of the problem is really interesting. From Apple's site, they said under "Impact," they said: "An attacker with a privileged network position may capture or modify data in sessions protected by SSL or TLS." So it's like, whoa, okay, that, you know, "privileged network position," we know that that's code for man-in-the-middle-style attack, meaning that there was some way to intercept SSL sessions. So I thought, okay, well, what's going on? Their description says - still doesn't really tell us.

Apple said: "A certificate chain validation issue existed in the handling of X.509 certificates." X.509 is just a standard formatting for SSL certificates that's industry wide. They said: "An attacker with a privileged network position may capture or modify data in sessions protected by SSL/TLS." That's basically repeating themselves. "Other attacks involving X.509 certificate validation may also be possible. This issue is addressed through improved validation of X.509 certificate chains." Well, yeah, improved. Listen to the details from the people who discovered the problem, an outfit called Recurity.

The Recurity Lab blog wrote, just yesterday on July 26:, "Recurity Labs recently conducted a project for the German Federal Office for Information Security" - whose acronym is BSI - "which, amongst others, also concerned the iOS platform. During the analysis, a severe vulnerability" - and they're not kidding - "a severe vulnerability in the iOS X.509 implementation was identified. When validating certificate chains, iOS fails to properly check the X.509v3 extensions of the certificates in the chain. In particular, the 'Basic Constraints' section of the supplied certificates are not verified by iOS. In the Basic Constraints section, the issuer of a certificate encodes whether or not that issued certificate is a CA certificate, i.e., whether or not it may be used to sign other certificates. Not checking the CA bit of a certificate basically means that every end-entity certificate can be used to sign further certificates.

"To provide an example, when a CA issues a certificate to a website, the CA will usually set the CA bit of this certificate to "false." Assume the attacker has such a certificate, issued by a trusted CA to attacker.com. The attacker can now use their private key to sign another certificate which may contain arbitrary data. This could for instance be a certificate for bank.com" - that is to say anything.com - "or even worse, it could be a wildcard certificate containing multiple common names such as *.*, *.*.*, and so forth. iOS has been failing to check whether the attacker's certificate was actually allowed to sign subsequent certificates and considers the so-created universal certificate as valid. The attacker could now use this certificate to intercept all SSL/TLS traffic originating from an iOS device. However, SSL/TLS is not the only use for X.509. Every application that makes use of the iOS crypto framework to validate certificate chains is vulnerable."

Okay. So what's very cool here is that, for any of our listeners who haven't already updated themselves, there's something cool they can do. If you go to https://issl.recurity.com, these guys have set up a test. If you use the Safari browser on any iOS device, and you don't get a notification of there being a problem, then you've got this problem. I tried it on my own iPads this morning, and it's neat. Well, I mean, it demonstrates the problem. You get taken to a page that says, if you didn't just see a security popup, you're vulnerable to this.

Leo: Now, this red screen with the invalid server certificate, that's what I'm supposed to see.

Steve: Correct. That means...

Leo: I'm doing this on a desktop, which is why I see it. It's not on iOS.

Steve: Right. And, for example, I tried it under Firefox 5, and I immediately got Firefox's "This connection is untrusted" popup.

Leo: And that's what you want.

Steve: That's, yes, that's what you want to see. And then the popup said, "You have asked Firefox to connect securely to issl.recurity.com. But we can't confirm that your connection is secure." Okay, now, the reason is that the certificate chain is not signed by a valid certificate authority because Firefox is correctly processing the v3 extensions of the certificate, which iOS, until just, I mean, just now - this apparently has been a problem forever. So Apple has just - it was just now found, and it has just now been fixed. So, for example, anybody who's got an iOS version not 4.3.5, who didn't just update in the last day or two, will be able to go with their version of Safari to issl.recurity.com and see a page with no problems. And what that has done is essentially it means that bad guys can sign their own certificates and produce certificates for any websites they want to. So this is a really bad problem which has just now been fixed and has apparently been a problem since the first iPhone happened.

Leo: Now, if you got the new iOS update, you shouldn't see that error.

Steve: And I did, I updated. The iPad where I had no problem going to that page, when I tried it again, I just hit refresh on Safari after updating, I got a popup which warned me that iOS was unable to validate the security of this site. I could look for more details, or I could say ignore it or cancel going there. So you ought to absolutely see that there's a problem with this when you go to issl.recurity.com after the patch. But if you have the chance to check it out before the patch, it's kind of fun to see that, whoops, this slipped through. And it certainly doesn't anywhere else because everything else is checking that flag, saying this certificate cannot be used to sign others, and Apple's not checking that.

Leo: Archanic (sp) has a good point in the chatroom. He says you can only update a 3GS or later. So if you have an older iPhone, this bug is going to persist.

Steve: Wow. That is a problem. I wonder if Apple will fix that because this is bad.

Leo: Yeah, I mean, I bet they don't. Really, honestly, they want you to buy a new phone.

Steve: Well, while we're in Macs, a new tool has been created. And I love this little news blurb because this is something we've talked about, again sort of in a theoretical vein, and it's happened in the real world, as so many of the things that we talk about end up having happened. This is an expensive tool from a company called Passware. By "expensive," I mean $995. And, well, they're getting that much money. The hackers can do the same thing for free.

So this was an article, I think it was in Wired.com. It said: "A new tool can steal your Mac's passwords in minutes, even if the machine is locked, sleeping, or encrypted." And it turns out this is also true for PCs, for a reason we'll see in a second. They said: "Passware can cajole your computer into revealing all its secrets, including login passwords and the contents of its Keychain app, in just minutes. All someone needs to do is plug in the USB stick with the app in another machine, tap through a few menus, plug a" - and here it is - "Firewire cable into the target Mac or PC and catch the magic. And it doesn't matter whether you have encrypted your data using Apple's FileVault app or another tool such as TrueCrypt. The vulnerability still exists." Now, the moment I saw this I thought, well, yeah, we've talked about this.

Leo: Right. You have to store these unencrypted passwords in memory so you can use them.

Steve: But more importantly, Firewire is the vulnerability, remember, because Firewire is a DMA interface.

Leo: Ah, we did talk about this before.

Steve: Yes. So a Firewire device has direct memory access, which is what DMA stands for, into a system's memory. It's essentially - it's a master on the bus. And it's able to simply read and write. Now, as I'm sure that I remember, although I didn't just refresh my memory for the story, there is a means by which the Firewire interface can be restricted in the ranges of memory that any IO devices have access to. But that's apparently still not being done. So as you said, Leo, when a machine is in use, even when it's sleeping or suspended, the keys necessary for the machine to use itself have to be available in memory.

[Talking simultaneously]

Leo: DoubleDJ (sp) in our chatroom says, well, couldn't they be hashed in memory? But I think you'd have to have them in the clear at some point; right?

Steve: Correct. You could even, if you didn't store the version in the clear, you're still storing the actual in-use code, which is there actually being transacted in memory. So we've seen code which is able to use the expanded version of the password in order to reconstruct what the password is. So this is definitely a vulnerability. But what I would do, certainly on the PC I know it's possible to disable your Firewire interfaces. And if you're not someone who uses Firewire, absolutely arrange to disable it completely if you can because it's a big, glaring hole.

Leo: I presume they're going to fix it. And the other point to be made is somebody has to have - and we talked about this before with the freezing the memory thing - somebody has to have access to your physical hardware. And if somebody has access to your computer and the hard drive, you've got a whole passel of other problems anyway.

Steve: That's really true.

Leo: They're got your stuff.

Steve: Although you might assume that locking your machine or sleeping it or putting it in standby would protect you. And in this case it doesn't because it does have access even in those modes.

Leo: Right.

Steve: Okay. So a bunch of people tweeted me about this Apple battery pack hack. What's interesting is it was discovered...

Leo: I love this story.

Steve: It's a wacky story. It's by a really good guy, or a true hacker who's not a hack. Charlie Miller we've spoken of before because he's a repeat winner of the Pwn2Own annual hacking competition.

Leo: They call him "Safari Charlie" because he always wins by hacking Safari. That's his nickname.

Steve: Yup. So he will be revealing all the details and showing this at the forthcoming Black Hat conference on August 4th. What he discovered was, he was poking around in the Mac, as always, and something caused him to do some research into the Apple firmware management for their battery. Some years ago Apple updated the firmware on their battery controller for the MacBook Pro. Charlie went back, found the patch, reverse-engineered the patch code which is able somehow to talk to and unlock the controller, over which there's not much documentation. And he discovered that every Mac ever made has the same password used to unlock the microcontroller which controls the lithium polymer prismatic battery used in all MacBook Pros.

Leo: Now, what does "unlocking a battery" mean? I don't even understand what that means.

Steve: So, well, it's a funny coincidence because I'm super current on this because I've been researching battery management for this portable device that I'm in the process of getting ready to build.

Leo: The dog annoyer. The crow destroyer.

Steve: Exactly. Yeah, it turns out that, in order to generate 150 watts of continuous RMS power, which is what this thing will be able to do, I mean, this is overkill in every sense of the word, this little portable device needs to generate 110 volts, peak to peak, at 12 amps, from a battery system. So I need to have, because I'm using what's called a "full wave bridge" amplifier for this 4-ohm super tweeter, I need to have 55 volts of DC. You get that by putting 15 cells of lithium polymer batteries in a chain. Each cell is...

Leo: Couldn't you just use a car battery? Wouldn't that be more efficient?

Steve: No, no. And in fact the way audio amplifiers in cars, in autos work, they only have access to 12 volts. Of course, they have access to lots of amperage. So in order to drive, in their case, probably 8-ohm subwoofers and 8-ohm speakers, 12 volts will not do the job. You can't just switch 12 volts into 8 ohms and get the kind of power which we know deafens teenagers' eardrums who are driving around in these cars. So what they have is they have a switching power upverter which takes the 12 volts at about 100 KHz. They switch it back and forth. They use 100 KHz because that allows them to use a much smaller core on a power transformer to step up the 12 volts, up to something like 50 volts. Then they rectify it, filter it, and then amplify that. So in my case I don't need to do that because I can just use three 18.5 battery packs. It turns out that the RC model industry has a great need for high-power DC for their high-end, high-performance helicopters and planes and cars.

Leo: Right. And they need to be light. You need a lot of juice in a small, lightweight package.

Steve: Exactly. So I use three five-cell packs, giving me a total of 15 cells. So but I thought it would be fun to learn about managing these batteries. Well, it turns out that all of our laptops do that. And if anyone's ever looked at the battery connectors we now have on all of our laptops, you'll notice that it doesn't have just like a big plus and a minus connection.

Leo: No, no, no.

Steve: It's got a comb. It's got a series of connections. What those are, those are taps into every cell of the battery. And the word "battery" itself means a bunch connected together. It's a battery of cells that our laptop is using because the laptops need also more voltage and more current. So our laptops essentially feed out of every connection between the cells and the battery. They feed that out. In the laptop then is a battery management system which deals with charging and discharging the batteries.

The reason you need to look at each cell is that cells don't all have, just coming from the factory, they don't all have exactly the same power handling. And so there will be one or two cells that will be just characteristically weaker than the others. So over-cycling these up and down and up and down and up and down, just through daily use, one or more of the cells will end up with a lower voltage across it than alternative cells in the pack. So it's called "cell desynchronization." And this ends up shortening the total lifetime of the battery pack.

So what these battery management systems do is they look at the individual charge across each cell. And they will bleed the cells that have a higher charge a little extra down to match the cell with the lowest charge. Or during charging they will shut some of the charging current around the higher charge cells, again in order to sort of automatically rebalance the charge in each cell of the pack. So there's much more going on here than typical users are aware. And there's even freeware that is available that for matching controllers you're able to look inside this and look at the voltage of each cell in your pack, see how many cycles it's had, the current charge state and all this other information.

So what Charlie learned was that by deliberately and maliciously reprogramming the firmware in his battery microcontroller, he was able to brick a number of batteries. He basically killed six batteries. And these are $130 MacBook Pro batteries. One of the things that every battery has is a management chip which prevents the battery from ever being used if it drops below about 2 volts, and ever allowing a charge of more than 4.2 volts. So if anyone's ever had the experience of a lithium ion battery appearing to just be completely dead, what actually happened was that one or more cells in the battery dropped below 2 volts. And at that point it is essentially taken offline, and you are never able to use the pack again. So what Charlie was able to do was to prevent the technology that limits the discharge to 3 volts, allowing it to go lower and pass under that 2-volt threshold, and then you are never able to recover from that. So at this point nothing has been done, that we know of, maliciously.

Leo: Could you make it blow up??

Steve: But everyone knows that this is now possible. And it's going to cause some serious problems if Apple doesn't get on the stick here and randomize the password on all of their machines to prevent people from being able to go into the microcode. The problem is, once they do that, they will never be able to patch the microcode of their own microcontrollers again. What Apple can patch, bad guys can, too. So this is an interesting little hack.

Leo: Yeah. Can they make it blow up? I mean, could it be explosive as a result?

Steve: He has posited that it's possible, although he's never made it happen. I'm suspicious of that. I can see how you could brick them by allowing them to over-discharge. But there should be per cell overcharge protection. And in fact it's in battery packs that don't have that that we have seen fire problems.

Leo: So you couldn't override it in the firmware.

Steve: I don't think so.

Leo: No, that wouldn't make sense.

Steve: Because there's firmware management that's outside the pack that lives on the motherboard. But then inside the pack itself there are per cell chips which prevent individual cells from ever being overcharged. They will just refuse to allow that cell to take an overcharge. And they also monitor the cell temperature and will cut off if the cell gets overly hot. So I'm almost sure you cannot make anything explode. But once you drive a cell down below 2 volts, it's over. That entire pack is an expensive $130 paperweight. And that you can do by misprogramming the firmware.

Leo: And again, you'd have to have physical access. Oh, maybe not.

Steve: No, no. This is...

Leo: You could do this in software.

Steve: Yes, this is a 100 percent software glitch. So he did notify Apple several weeks ago of what he found, and that he would be talking about this on August 4th. Here we are toward the end of July, and everyone knows about it now. So you can imagine there are some bad guys rubbing their hands together, thinking this would be a fun piece of malware to deliver when people surf to malicious websites.

Leo: Heh, heh.

Steve: Yeah. Okay. Under Miscellany, I upgraded to v5 of Firefox. And I'm not sure whether it was doing that or whether it was a new version of Certificate Patrol, but my Certificate Patrol, which I love and have talked about often, suddenly got very noisy. Essentially it seemed to have forgotten all the sites I had ever been to, and it was popping up a dialogue, like, all the time. Anytime I established an SSL connection that I hadn't established recently, I got a popup.

Well, that's old behavior. The new behavior is also - it was graying out the Okay button, not allowing me to click it to dismiss the popup and allow the page to display, which was really annoying. They've added a new feature which I don't need, and I don't know why it's there because it's just useless. So the good news is, and I wanted to share this with any other listeners who have had similar problems, but even if you don't upgrade to 2.0.8, Certificate Patrol is customizable in a way that I never bothered to before, but now I wish I had, and I have now.

So if you go under Firefox's Tools menu and then choose Add-ons, and then from the Add-ons page choose Extensions, and then under Certificate Patrol they've got an Options dialogue. That pops up a dialogue that now has four options. The first one, that was enabled by default, says, "Newly accepted certificates are always shown in a popup by default." I have now turned that off, vehemently. Because I really don't care as new certificates are being shown. What I care about is any changes to those. That's the significant value that Certificate Patrol offers.

Well that's the second checkmark that says, "All certificate changes, even harmless ones, are always shown in a popup by default." And so I left that on, which I think is a good thing. And then there's also, "Show already accepted wildcard certificates again when they match in a new hostname." Well, I was glad to see that there was an option for that. I turned that off because Google, for example, uses a *.google.com, and I was getting gmail.google.com, secure.google.com, googleplus.google.com, it was like all these *.google.com certificates. And it's like, that was just boring. So I'm happy to turn that off.

And then I turned on, which wasn't the default, and users can choose to or not, the last option is "Store certificates even when in Private Browsing Mode." Well, I'm not a big user of private browsing mode due to the nature of my environment here. But if I am in private browsing mode, I think I'd like to have certificates stored. The concern is that this might be a bit of a privacy breach because, if someone looked at the storage certificates, they would know where you had been, even if you were in private browsing mode. So you might want to leave that off, just so that private browsing stays absolutely private. In my case, I don't really use private browsing mode, but I would like to have it store those even when I am in private browsing mode. So that's what I said.

So that's just a tip for people using Certificate Patrol. Mine has just gone completely quiet now, and I'm much happier with it. I know that it will let me know if something wacky happens, and that's really why I have it in the first place.

Leo: Yeah. You don't need to know about harmless certificate changes. Thanks for letting me know, but I'll pass.

Steve: Yeah. Well, I know that a lot of listeners will just find it interesting to know that a certificate has been changed, it's like expired. And I think Twitter has had a lot of recent changes because I got some tweets from people saying, hey, Steve, have you noticed that Certificate Patrol is telling you about those? And it's like...

Leo: Yeah, I saw somebody in the chatroom saying that today, yeah.

Steve: Right. I am wrapping up my reading of "Daemon." I wanted just to give another shout-out to it. Many of our listeners have started, and I've had a flood of positive feedback...

Leo: Great book, yeah.

Steve: ...from the recommendation. And I will be shortly on to "Freedom," the sequel to it.

Leo: "FreedomTM." Remember the TM in the word.

Steve: Ah, okay.

Leo: It's important because there's a lot of books named "Freedom," including a big bestseller this year called "Freedom" by Jonathan Franzen. So it's "FreedomTM."

Steve: Okay. And there is a - someone asked me where to find it, and there is a "Daemon" website.

Leo: Oh, yeah.

Steve: So you can just put in, like, "Daemon the book" into Google, and it will take you to the "Daemon" website where you can find this and links to Amazon for downloading it for Kindle and so forth.

Leo: Daniel Suarez is the author. That will help you find it, too. He's great. We're going to have him on the show soon because his new book is due out soon. But he's just done such a great job.

Steve: A third book?

Leo: Yeah.

Steve: Oh, cool.

Leo: He's been working on it for over a year now.

Steve: Cool.

Leo: Yeah, it's supposed to be out soon.

Steve: Someone tweeted me, when I was talking about coming back to the animal annoyer product that I don't - or project - that I don't have a good name for yet.

Leo: That's a euphemism.

Steve: Someone tweeted me a recommended schematic capture and PC board fabricator. And I just can't find it back in my Twitter feed. So I wanted to give a shout-out to anyone who has a recommendation. That person, again, if you could tweet it to me again, I'd appreciate it. And I'll make sure I don't lose it this time.

And I did have a fun and always different SpinRite story to share from Mike DeLucia, who wrote, "SpinRite Story: Happy but Confused." He said, "Steve, I'm a Security Now! listener and registered SpinRite owner of many years, though I just had my first 'opportunity,'" in quotes, "to use it. The experience left me happy but confused. Let me explain. The other night we had a lightning storm. And though my system was turned off and unplugged, I believe a very close lightning strike may have caused a problem." Yikes.

Leo: Oh, sure. Oh, sure. Having it turned off does nothing. Unplugged is not good.

Steve: Exactly. And if he was, like, still connected to his Ethernet, then that's the way in.

Leo: Right.

Steve: He said, "When I booted my system, it booted okay but immediately started acting strangely. Responses to my commands caused it to slow to a crawl. I'd launch a program, and it would literally take a minute or so to respond, all the while showing the busy circle thingy." He says, "I rebooted, same thing. So I booted SpinRite, ran it at Level 2. It ran for about an hour and stopped, displaying a red 'division overflow error' dialogue box." Now, that's actually SpinRite monitoring its own math in order to see and catch if there's a problem.

So he said, "The dialogue displayed a bunch of hex info like 'error occurred at bo4eeax blah blah blah blah blah,' and asked me to write down all this data and reboot SpinRite. It also said, I believe, that SpinRite would ask for this data to aid in its repair process." And actually that's not the case. What it said was that, if the problem persisted, that you could use that, send that to GRC support, and we'd be able to figure out what happened and fix it if it was a problem at our end.

Anyway, he says, "I rebooted but could never see anything different than the normal SpinRite process. I thought it was going to ask me for the data I copied. Confused, I restarted Level 2, figuring I'd get the same error and reread the directions. To my amazement, I never got that error message. Instead, SpinRite stopped about 14 percent through and began analyzing a certain portion of the disk. It stayed there for over two hours until I went to bed with it happily analyzing away. I awoke this morning to a green dialogue box stating that SpinRite had completed all of its work. And the best news is that all is well with my system now. As I wrote, 'happy but confused.' What happened? Did I miss something in the red error message that asked me to save and reenter data when asked to do so? I was never asked."

Leo: I'm happy but confused.

Steve: "Anyhow, I thought I'd share the story. Thank you for all you do for the community. Peace, Mike DeLucia."

Leo: That's great. All right, Steve.

Steve: Okay.

Leo: Let's talk about it, the Anatomy of a Security Mistake.

Steve: Okay. So I want to give Walid credit. He's @walid. It looks like Walid Damouny tweeted some time ago, a few weeks ago. He said, "This is a month old, but I ran into it only today: a hole in crypt_blowfish." And he gave me a link to an article.

So this is a password-hashing library where Blowfish encryption, which is a well-known standardized encryption library, is being used to hash passwords in PHP, some popular Linux distros, Openwall Linux - that's OWL - SUSE, ASP Linux, ALT Linux, and Annvix, which all use this crypt_blowfish for hashing their password databases. And this was discovered by someone using a "John the Ripper" password cracking program because he wasn't getting the results that he expected from his code. And by drilling down into the system - and that's an advantage of open source. Even though this problem did exist for 13 years, since 1997 or '98, an advantage of it is that he was able - he had access to the source, so he was able to figure out what the problem was. So I love this because I can, in the course of this podcast, in an audio-only channel, I can explain this. And it's just it's a toe-curlingly cool mistake.

Okay. So we need to first understand about how numbers are represented by computers. So we need a little bit of a foundation paving in some fundamental technology of binary numbering and computers. And this is relevant, not just to this mistake, but to everything. This is the way computers work. We know that computers store numbers in binary. And, for example, zero, the number zero, is however long the word is. It might be an 8-bit byte. It might be a 2-byte, 16-bit value or, for example, a 4-byte, 32-bit value. The value zero is just universally agreed is just all the bits are set to zero. If we want to store a one, we set the first bit to one. To do a two, we set the second bit on and the first bit off. And to do a three, the first bit and the second bit are both set on, and so on, in sort of binary counting mode.

In the same way, another way to think about this is everyone's familiar with decimal, where the units, like the first position is the units position that can have a value of zero through nine, then the next position is the tens position that can have a value of zero through nine, and the next position is the hundreds position. So there, each digit in decimal has 10 times the value by, like, the value weight of the prior one. In binary, since we only have two values, each position has two times the value of the prior one. So one, two, four, eight, 16, 32, 64, 128, 256, 512, 1,024, 2,048 and so forth, numbers which low-level geeks learn very early in their programming career.

So the question is, given that binary - for example, say we had an 8-bit byte. If all bits off is zero, then if we have an unsigned value, all bits on, we've often talked about, is 255 because that's 128, which is the amount represented by the eighth bit in the byte, so it's 128 plus 64 plus 32 plus 16 plus 8 plus 4 plus 2 plus 1, which is the sum of all those values, gives us 256 as the maximum unsigned value. But what about if we want to store a signed value, meaning it could have negative values? If we start with zero and go upwards, towards in the positive direction, we know that we, essentially counting in binary, we have the binary one, then two, then three, then four, then five and six and so forth. If we subtract one from zero, what happens is, with binary subtraction, we go to all ones.

So whereas that all ones note, that all ones value, if the word were unsigned, is equal to 255, if we treat this byte as a signed value, then negative one is all ones. And then all ones except the first bit being on, if the first bit's off, that's negative two. Then all ones except the second bit being off and the first bit on is negative three. And negative four is all ones but the first two bits off, and so on. So you sort of - you subtract backwards from all ones in order to get negative values. And the same thing applies if we had two bytes, then that would function, or three bytes or four bytes or eight or however many. And this is called "two's complement arithmetic."

There's a simple way of essentially negating a value with this kind of representation, which is you invert the number - you invert all the bits and add one. So, for example, say that we had zero, the value zero. If we invert them all, that turns all the bits to ones. And if we add one to that, that overflows in binary, bringing it back to zero. Which means that zero and negative zero are the same value, which makes sense. That's what you'd expect. If we had a one, and we wanted to make it a negative one, well, so we had - if we had a one in the register, which would be the lowest bit is one, and everything else is zero, we invert all the bits, so now they're all ones except the first one, which is zero, and add one, making them all ones. And as we know, that's the value for negative one, if we have a signed representation. So, okay, so there's our little tutorial on the signed representation of binary values in computers.

Leo: This bites people all the time, by the way.

Steve: I'm sorry?

Leo: Signed versus unsigned. It's always biting people. You see these bugs a lot.

Steve: Yes. Well, it bit this guy. It bit this guy. So with that understanding, we have a hashing algorithm which takes in input characters. And as we know, characters in, for example, ASCII are a byte size. And that is to say eight bits. But the encryption system works in 32-bit chunks. So we needed an algorithm which could take in eight-bit characters one at a time and essentially sort of stack them up into a 32-bit value, so it would take four eight-bit characters and sort of fill a 32-bit register and then do something else with it. So it would accept them sequentially and fill this register.

So in terms of the algorithm that's used, and this is what's in the source code for this system, this 32-bit register is set to zero at the beginning of one of these four-byte shifting loops. So it's set to all zeroes. Then whatever happens to be there is shifted to the left by eight bits, meaning that everything in the first, in the lowest byte is moved over to the next byte. Everything in the second byte is moved over to the third. Anything in the third byte is moved over to the fourth. And nothing in the fourth byte ever falls off the end, which would otherwise happen, because this is only done four times, just enough for the value going in to move, like the first value that goes in to move all the way over to the fourth byte. And so each time this is done, whatever is in this 32-bit register where we're assembling these characters, whatever is there is moved over by a byte. And then this new character is ORed in.

Now, we know from talking about logic the way logic works. If you have a register which is all zeroes, and you OR something into it, what you just get is the result will be what you ORed in because zero ORed with anything is whatever you ORed. So the act of shifting this 32-bit register eight bits to the left always results in the lower eight bits being zeroed because that's the way the shift operation functions. And then we OR this new character in, which essentially puts it into the lowest byte of this 32-bit register. And that's done four times in order to accept four characters, four of these byte-size characters, one at a time. And we end up with a 32-bit value.

And so looking at the code, you can see that that's what this does. And, I mean, it looks just fine except that the character was defined just as a char, c-h-a-r, in the C language that this was written in. And a char is a signed value by default in C. I don't know why that's the default because it doesn't make any sense for char to have a signed value. But typical values in C are signed, unless you override them by declaring them as an unsigned char or an unsigned short integer, depending upon the language convention. So this char value by default is a signed value.

Well, what C, the language C, observes when the programmer is ORing this character value into a 32-bit value, it sees the sizes are different. You're trying to OR an eight-bit value with a 32-bit quantity. And we know that's what the programmer wants to do. But the compiler says, wait a minute, we need to sort of pad out that eight-bit value to 32-bits in order to OR it in. Except that, because it is a signed value, the compiler does something called "sign extension." And what a sign extension does is it duplicates the highest bit in the value - remember that that last bit is set to a one. As soon as that value goes negative, we end up with, like, 1111111, all the way out, so that the highest bit in the binary quantity is considered to be the sign bit. If it's set, that means that the rest of the bits determine the amplitude, essentially, with that high bit determining the sign, positive or negative.

So the compiler sees that the programmer is ORing an eight-bit signed value into a 32-bit register; and, wanting to do the right thing, it sign extends. Which means, if the eight-bit value had its high bit set, all the other bits get set to one, and then that 32-bit quantity is ORed into this register. Well, ORing ones into the register sets all the bits to one. Which means, since this is being shifted over three times, as soon as you attempt to merge a character that is negative, that is, whose sign bit is set, all the prior characters, that is, up to three previous characters that may have been ORed in and shifted previously, are wiped out. All of their bits are set to one because this character we're ORing in had its high bit set. The C compiler said, oh, well, we're ORing in a signed value. So we need to preserve the sign of this eight-bit byte when we convert it into a 32-bit value. Doing that sets all of the other 24 bits to ones. Then that 32-bit composite is ORed in to the register, setting all of its 24 bits to one, so essentially overwriting, over-ORing whatever was there before.

So the consequence of this tiny mistake, where the compiler did technically what it was told to do but not what the author certainly intended because they never intended for the sign extension to be applied and wipe out the three previous things that had been ORed in and shifted over, is that only every fourth character in the user's password is significant. So...

Leo: [Laughing]

Steve: Isn't that a cool mistake?

Leo: Wow. And, you know, it's the kind of thing, when you're reading the code - because you have to really intimately understand what a compiler does. So when you're reading the code, you may not realize that that's how the compiler's going to react. You see the programmers' intent instead of seeing what the result is going to be after compilation.

Steve: Yes. And so - exactly. And I've often talked about how seductive a process it is to read someone's code. You get into their mindset. You're looking at what they're doing. And you're going, okay, now we set the temp to zero. Yeah.

Leo: I get it. I get it.

Steve: We shift the temp over by eight. Yeah. And then we OR this in. Yeah. I mean, and so it takes, literally, it takes truly watching the code execute. And then it's like, oh, my god. And then it is so obvious, once you see it, what the mistake is. And I have to say, this is one of the benefits, one of the beauties of coding in assembler because nothing is done that you don't explicitly ask for.

Leo: There's no compiler in between saying, oh, this is what you want to do. You're telling it.

Steve: Exactly. There is...

Leo: There's no s-e-x happening unless you say so.

Steve: Correct, exactly. There is a MOVZX, which is move zero extend, and MOVSX, which is move sign extend. So those are the lowest level operations in assembly language that correspond to those exact instructions on the machine. Now, C was invoking move sign extend without it being obvious. But when you're coding in assembler, if I were writing this code, I'd say move zero extend because I would have to explicitly make that conversion from a byte to a 32-bit object. And knowing that I'm about to OR that thing in, I would never sign extend.

And so, again, the compiler, sure, it's easier to use. But a tiny mistake like this ends up being a problem. And in the updated source code now there's what's called a "cast" in front where it says in parentheses "(unsigned char)." So it's overwriting the default char declaration of that byte, saying, "C compiler, treat this as unsigned." So that's all that had to be added to fix this bug. Which caused, if you were typing in a password, you'd type in the first character, it would go into the first byte. You'd type in the second character, they would move the first one over and put the second one in. You'd type in the third one, it would move the first two over, put in the third one. You'd type in the fourth one, it'd move the first three over, put in the fourth one. But if that fourth character had its high bit set, it would wipe out the previous three. So the bug was that passwords are hugely reduced in strength so that only every fourth character is significant.

Leo: So if you had an eight-character password, you've only got a couple of letters in, yeah. It's very...

Steve: Only two characters matter.

Leo: Yeah, very easy to crack. Wow.

Steve: So in concluding, in the real world consequences for this from the pages where this was discussed, the guys who went through this said:

"The safest solution for administrators with potentially bad password hashes" - because what this means is that all of the passwords which had ever been hashed by this are weak, you know, they're all bad. They were hashed only using every fourth character of what the user put in. So they said, they wrote:

"The safest solution for administrators with potentially bad password hashes, which could include those running Openwall Linux (OWL), SUSE, and ALT Linux [and so forth], which can ... use [the] crypt_blowfish for hashing the password database, is to invalidate all passwords and have their users input new ones. That could prove to be a logistical nightmare, however, depending upon how easily and securely users can set new passwords without authenticating with their existing password. Requiring that users change their existing passwords after logging in is an alternative, but one that might give attackers a window in which to operate. It also leaves passwords unchanged for any users that do not log in.

"The risks of attackers using this flaw to log into a site are likely to be fairly small, but they do exist. If a site's login process is susceptible to brute-force attacks, this bug makes it somewhat easier to do so, at least for specific password types. On the other hand, if the password database has been exposed, and some passwords were not crackable, at least for attackers using cracking programs other than JtR, this information [that has now been released] would give them the means to crack those passwords [much more easily]. In the end analysis, it is an exploitable hole, but not the kind to send administrators into panic mode."

Leo: And you can't just fix the library because then the passwords stop working; right?

Steve: Correct. And in fact the patch now that brings us to 1.1, it actually has an "if" statement where they formally say "signed char," and the "if" is controlled by a new parameter to the Blowfish set key function which is called "sign extension bug." So you can call this asking for the sign extension bug in order to allow people to perform both the old-style hash and the new-style hash, if you want to do this kind of sort of seamless migration from the old to the new. And then they said:

"It is somewhat amazing that this bug has existed for 13+ years in a fairly widely used library. Up until now, no one has tested it in this way, at least publicly, which should serve as something of a warning to those who are using well-established software, both free and proprietary. With free software - [for example] crypt_blowfish has been placed into the public domain, which may be a bit legally murky but is in keeping with free software ideals - there are more and easier opportunities to examine and test the code, but that's only effective if the testing is actually done."

Leo: Yeah. That's what I was saying at the beginning. It's open source, which means people can look at it. But they've got to look at it. And they've got to look at it with a certain kind of mindset.

Steve: Actually, I would argue you can't look at it. You have to test it. And that's...

Leo: Yeah, you have to look at a debugger and watch it happen.

Steve: Yeah, well...

Leo: I mean, I guess if you're looking for unsigned problems with unsigned chars or signed chars, you might be smart enough now to look at code and say, oh, they're assuming that it's going to be treated as unsigned, but it won't be.

Steve: Well, and they said: "There are, without doubt, bugs still lurking in various security-oriented libraries that we depend upon every day, so testing those libraries, as well as code that is not being used for security purposes, regularly and systematically can only help find them. While it took more than a decade for this bug to come to light, it's worth pointing out that it certainly could have been [observed] by others in the interim. Attackers clearly do their own testing and are generally not inclined to release their results. That may not be the case here, but one should always recognize that public disclosure of a vulnerability is not necessarily tied to its discovery."

Leo: It could have been discovered a long time ago.

Steve: Exactly.

Leo: Seems like the kind of thing that a lint program, or a program that - automated program that goes through code might give you a warning saying, hey, you know, this is an unsigned char, but you should be aware it might be treated as signed, or something like that.

Steve: That's a very good point. Now, I mean, it's hard to see why you would ever want a signed extension ORing different-size things. That is to say, because that's always going to wipe out the rest of the bits of the thing you're ORing it into. It will always do that. So you're right, Leo, a code analyzer that would say, well, you know, this is questionable, did you mean this, that would bring it to a programmer's attention; and he'd go, oh, crap, and immediately say "unsigned char," casting that particular instance. Or just declaring it in the original declaration as an unsigned char would also prevent the problem. So just a cool mistake that I would imagine our listeners who closed their eyes and followed along with me understand now.

Leo: It's fascinating. It really is. I mean, it's one of those things where, you know, I was reading an article the other day about Google, and they have a thing called "code review" where your peers review - all code before it's put into production at Google is reviewed by your peers. And they even talk about this in a code review situation. It's very frequently that, as you do a code review, you kind of want the code to work, and you kind of understand the mindset of the programmer. And this is not what should be happening in a case like this. You should probably be running it through a debugger and watching it, watching the data change.

Steve: Well, and if you think about it, too, a malicious hacker wants to find problems. They're inherently looking at every single line, saying, okay, can I exploit that? Can I exploit that one? Can I exploit that? You're right, Leo, it's a different mindset. And I don't know that somebody on your team can develop that same kind of truly adversarial mindset.

Leo: Right, yeah. You've got to have the hacker's brain. Steve, always a great pleasure to do this show. And this is a fun one. This is the kind of thing that I think people really love the show for because it's an insight into how code is written, mistakes that are made, and how hackers work, as well. Really great stuff.

Steve: And it really did happen.

Leo: Yeah. Steve's got GRC.com, the Gibson Research Corporation. That's where you can find SpinRite, the world's finest hard drive and maintenance utility, GRC.com. You can also find this show and all the previous shows, all 310 previous shows, including 16KB versions for the bandwidth impaired, transcripts, and all the show notes, as well. GRC.com. Next week we're going to do a Q&A, so that means, Steve, people should go to GRC.com/feedback and leave their questions about this or any topic we talk about or anything that's on your mind, GRC.com/feedback.

And when you get there, by the way, lots of free stuff Steve puts out like Wizmo and DCOMbobulator, the Password Haystacks. Just browse around the site. There's a ton of good stuff at GRC. Thanks, Steve. It was so great seeing you on Sunday. Thanks for coming up for the show.

Steve: Glad to do it.

Leo: Really appreciate it. Thanks for putting up with our new studio and the little bits and pieces we're getting working. I think we're getting it down, though, I have to say. This show next week will be in my studio, so we'll have to start all over again...

Steve: Okay.

Leo: ...debugging. We've got our own debugging to do. Steve Gibson, take care. We'll see you next week on Security Now!. Bye bye.

Steve: Thanks, Leo.