SECURITY NOW!

Transcript of Episode #846

HTTP REQUEST SMUGGLING

Description: We're going to start off this week by taking a careful look at a shocking proposal being made by the Internet's Engineering Task Force, the IETF. They're proposing a change to a fundamental and long-standing aspect of the Internet's routing which I think must be doomed to fail. So we'll spend a bit of time on this in case it might actually happen. Then Microsoft reveals some results from their network of honeypots, and we update on the progress, or lack of, toward more secure passwords. GoDaddy suffers another major intrusion, and just about every Netgear router really does now need to receive a critical update for the fifth time this year. This one is very worrisome.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-846.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-846-lq.mp3

Then we're going to finish by winding up our propeller beanie caps to explore the emerging threat represented by HTTP Request Smuggling, which everyone will understand by the time we're done. It's a sneaky way of slipping sensitive and damaging web requests past perimeter defenses. Oh, and on this November 23rd, a special Happy Birthday wish to Evan Katz, a long-time friend and follower of this podcast.

SHOW TEASE: It's time for Security Now!. Steve Gibson is here. Wow, a very bad plan from the IETF. It's just a proposal right now, but Steve wants to nip it in the bud. And you know what, you're going to agree. We'll take a look at Netgear routers again. Another zero-day, a mess. And then look at the status of brute-forcing passwords. What are the worst passwords, the most brute forced? The answer will not surprise you. It's all coming up next on Security Now!.

Leo Laporte: This is Security Now! with Steve Gibson, Episode 846, recorded Tuesday, November 23rd, 2021: HTTP Request Smuggling.

It's time for Security Now!. Yes, you've been waiting all week long, you're a very patient person, for this guy right here, Steve Gibson. Hello, Steve. Happy Thanksgiving week.

Steve Gibson: Yo, happy, yes. Indeed. And I missed your birthday. Was it actually on Sunday?

Leo: No, it's Friday.

Steve: Oh, last Friday?

Leo: This Friday. You didn't miss it.

Steve: Oh. Oh, well, in that case...

Leo: Send gifts care of TWiT, 1351B Redwood Way. No, I don't like gifts, as you know. Once you get to a certain age, you don't even want to think about it.

Steve: Well, I was put in mind because this is Evan Katz's birthday today.

Leo: Oh, dear friend, yeah.

Steve: And of course you and I both know Evan. He's a prolific communicator. And he...

Leo: That's a nice way to put it.

Steve: And he tweeted, the reason I know it's his birthday, I don't track his birthday, but he said, he tweeted that it was 10 years ago today that I surprised him at the Ritz Carlton in Dana Point. His wife set this up. And his wife's name is Ruth. And so Ruth contacted me, and she said it would just blow Evan's mind if he walked into the parlor at the Ritz Carlton and had you there to hang out with him. So I said, oh, what the heck.

Leo: Aw, that's very sweet of her.

Steve: Let's meet.

Leo: How nice of you, too.

Steve: It was very cool, yeah. And it turns out he's a big chess buff.

Leo: I know, I know.

Steve: It was from Evan that I learned that humans just - it's over for us.

Leo: It's over. Not even close. Not even close. But the human world championship is starting Friday. So that's always fun.

Steve: On your birthday.

Leo: On my birthday. They did it just for me.

Steve: Oh, there you go.

Leo: Magnus Carlsen, the Norwegian...

Steve: Current champion?

Leo: Yeah, current Norwegian world champion will face Nepomniachtchi. Nepo, they call him, because his name is ridiculous.

Steve: Well, I was just recommending "Queen's Gambit" to someone who has never seen it.

Leo: Oh, is that good. Oh, that's good TV.

Steve: I just can't wait to - oh, oh, it's so good.

Leo: Really good show.

Steve: So we have an interesting episode, sort of weighted at each end as opposed to just at the end. We're going to start off this week by taking a careful look. Oh, I should mention that this Security Now! Episode 846 for, what is this, three days before your birthday, November 23rd?

Leo: Well, if you want to - yeah. Actually my birthday, I am wrong. I lied. My birthday's the 29th. I don't know what I was thinking. Michael's birthday is Friday. Mine is a week, is coming up. A week, next week.

Steve: Yeah, next Monday.

Leo: Next Monday.

Steve: All right. Anyway, there's a lot of birthdays happening.

Leo: Yeah, I got confused.

Steve: We're going to be talking about HTTP Request Smuggling, which was going to be last week's topic until it got bumped for something sort of more timely. But I did mention it because it was part of an attack chain which I thought was going to be tied into the topic of the week. But now it's tied into this topic, this week's topic. Anyway, HTTP Request Smuggling a very tricky way of smuggling HTTP Requests across the border, literally. So that one's going to be kind of tricky, but I think really interesting for our technically inclined listeners.

We're going to start off, though, by taking a careful look at a shocking proposal being made by the Internet's Engineering Task Force, the IETF. They're proposing a change to a fundamental and longstanding aspect of the Internet's routing, which I think is doomed to fail. So we're going to spend a bit of time on this in case it might actually happen, which will surprise me. But so that's first. Then Microsoft reveals some results from their network of honeypots, and we update on the progress or, sadly, lack of, toward more secure passwords.

GoDaddy suffers a major, another, I think it's like the fifth, major intrusion in recent years. And just about every Netgear router really does now, as of now, need to receive a critical update. This is the fifth bad problem they've had this year. This one's very worrisome because it's so easy to exploit. So anybody with a Netgear router pay attention. We're going to finish, as I said, by winding up our propeller beanie caps to explore the emerging threat represented by HTTP Request Smuggling, which I'm pretty sure, although it's a little tricky, I think everyone's going to understand this by the time we're done. And even if you kind of get kind of a feel for it, it's still, it's very cool the way it works. And unfortunately, it's of importance because it allows sensitive and damaging web requests to sneak or be smuggled past perimeter defenses. So I think a really good podcast for everybody. And you and I were talking about our Picture of the Week. We've got a...

Leo: It's a funny one.

Steve: This is a great one, yeah.

Leo: Yeah, and I think HTTP Request Smuggling will be a great topic of conversation this year at Thanksgiving dinner tables all over the country.

Steve: Yeah, cocktail party, like what's new?

Leo: Yeah, great thing you can bring up and show your brains. Then don't forget to tell people you heard it on Security Now!. That's important.

Steve: So our Picture of the Week.

Leo: I love it. I love it.

Steve: Shows, I'm not sure, maybe there's drugs behind there because there's sort of a medical theme to this. It's going to be difficult for me to describe. I would recommend that people get the show notes if you don't - if you're curious. Now, are those forceps or sutures, those...

Leo: I think they're forceps, yeah. They're like scissors with a point that you can clamp.

Steve: Correct. Well, and then down at the back where your finger loops are is a ratchet, so you're able to close them and then, like, lock them closed. So, okay. So people have seen like surgical scissors, right, forceps. The problem that this enterprising individual solved was how to lock a cabinet that had two handles where there was, like, an air gap underneath the handle. Like so in order to open this cabinet you'd slip your fingers under the handle and then pull it towards you. Well, this cabinet is not locking, yet clearly they need to lock it. Well, they have a padlock, but padlock is the standard size padlock with a hasp of the normal size. It won't straddle these two handles which are on opposite sides or on opposite doors to this cabinet.

So the person opened these forceps, slid one of the handles through the middle of the forceps between the finger loops, then locked the forceps, which brought the loops, the two loops of the forceps close enough to the other handle that they were able to open the padlock, run it through both loops of the forceps, thus keeping them closed and locked, and through the other handle, and then locked the padlock. So basically the forceps sort of form an extension of this padlock. Anyway, it's extremely clever. And, you know, you can imagine sort of like an intellectual puzzle, if someone said here's what you've got to work with; you know? You've got a crayon, and you've got forceps, and you've got a padlock. And your job, your goal here is to lock this cabinet. It's like, okay. How many people are going to figure that out? I'm not sure. Anyway.

Leo: Thanks, by the way, to Encrypted Beard in our IRC, who says he sent that to you.

Steve: Oh, yes, indeed. Thank you.

Leo: Yeah, yeah.

Steve: Okay. So this effort or this idea, this scheme from the IETF, well, I titled this "An Idea Whose Time Has Passed." Okay, now, I've got to create some background here for everyone who isn't, like, really tuned up on Internet IP addresses and protocols and things. As we know, there are several ranges of IPv4 addresses that are formally designated as unroutable. Three ranges of addresses were originally specified and set aside by RFC 1918, and that's almost when that one was published. Oh, actually no, it was in '96, 1996. The smallest of these three unroutable networks or address ranges is the one which most consumer routers use for their default local LAN. And I'm sure most people are familiar with that, right, 192.168, and then often dot 0 or dot 1. Consequently, any IPv4 address beginning with 192.168 is considered to be part of a local private network, and no public routers will forward packets addressed to those IPs anywhere.

Okay. In the case of 192.168 dot whatever dot whatever, this creates a nice block of 64,000 IPs because you know, each of those numbers is a byte. So we've got two bytes. So that's 16 bits, and so that's 64K IPs.

Leo: And 10-dot's unroutable, as well; right?

Steve: Well, correct. So that's the small one. The middle size one, which RFC 1918 reserved, was 172.16 dot anything dot anything through 172.31 dot anything dot anything. Now, so that provides a network having 20 bits. The 20 least significant bits of those IPs are reserved for specifying the machine within that network. So that's one million machines. And the third and largest is the one you mentioned, Leo, is 10-dot. So that's anything beginning with 10 dot anything anything anything. So that gives us 24 bits for specifying a specific machine, and so 24 bits is 16 million machines on the local network. So those are super ample for non-routable IP ranges. And as we know, for quite some time there were so many IPv4 IPs that some were never allocated.

Back when we first were using and talking about and fans of the Hamachi virtual network system, it was cleverly using the 5-dot network since no one ever had. 5-dot had never been allocated. So it wasn't like reserved. It just no one ever got around to needing it. And so the guy who did it was super clever. He said, hey, you know, I can't use the RFC 1918 networks like 10-dot because people might want to use Hamachi on a 10-dot network. And so the network we use for Hamachi has to be disjoint. It has to be completely separate. But it can't be any public routable IP or things wouldn't ever be able to get to Hamachi anyway. So clever that he thought to use 5-dot. And of course it got allocated some time ago because, as we know, with time, IPv4 space has become depleted. And, you know, someone looked over and said, hey, there's 16 million IPs that begin with 5 that no one's ever used. Let's start handing them out. And so they're all gone now.

Okay. So another set-aside block of IPv4 space are all those addresses beginning with 127. Anyone who's spent much time poking around with IP networking on any IP-enabled operating system will have encountered the concept of the local loopback IP, or really the loopback network of which one IP is most often used. By universal convention and explicit specification, 127.0.0.1 always refers to the local machine, like the machine you're at. It has an IP of 127.0.0.1 and whatever other IP like from the LAN or WAN or whatever. So you can kind of think of 127.0.0.1 as an alias for the local machine's IP. If you ping 127.0.0.1 on any machine that you're at, that you're in front of, while any portion of your local network is running, that pinging is guaranteed to succeed if the local net stack is up because it causes the machine to essentially ping itself. No packets go anywhere. And this is used for all kinds of purposes.

Many developers running a local web or some other server on a machine will bind that server's IP to some port on 127.0.0.1, like 127.0.0.1:80, for example, for web services. And that makes that bound server's services available to any client running on that machine. So, for example, you could put into your web browser 127.0.0.1. And if there was something bound on port 80 or 443, you would get the page that that web server was displaying. So that's the story.

Now, although only 127.0.0.1 is typically used, the entire 127-dot network of 16 million IPs has been set aside as a local loopback network. And in fact, if you open a Windows command prompt and enter the command "route print," or, you know, Linux, you can print the routing table in Linux or Unix or macOS, any Internet-based operating system has its own local routing table. And in the show notes I've got the result on my typing "route print." And you have a couple entries for the local machine's actual IP, but in the table I see 127.0.0.0 with a netmask of 255.0.0.0, meaning that that entire network is recognized and will be routed to the interface 127.0.0.1. In other words, this routing table in the machine I'm sitting in front of has allocated the entire 127-dot network for its own local use. And that's 16 million IPs; right? Because it's got all of the lower 24 bits are available underneath the 127.

So on page 31 of the RFC that set this up - it's RFC 1122, dated way back even before the 1918, this one is from 1989, RFC 1122 - page 31 of that RFC very clearly states 127 and then any, you know, any bits, and it says "internal host loopback address. Addresses of this form MUST NOT" - and, you know, this is in the formal RFC language, so all capitals "MUST," all capitals "NOT" - "appear outside a host." And that explains why I did a serious double-take, and I really did check the date to be certain it didn't say April 1st when last week I encountered an official IETF Standards Track proposal titled "Unicast Use of the Formerly Reserved 127/8."

Okay. So Unicast is the formal name just for standard packets, for example, as opposed to broadcast or multicast. You know, Unicast is what everything is mostly except for when they're not. You know, so all the packets we talk about going places, those are Unicast packets. They're sent to a specific address. So this thing says Unicast use of the formerly reserved - formerly reserved - 127/8 network is what it's saying. So this pending IETF proposal is suggesting that the definition of the 127-dot class A network should be changed to allow most of its 127-dot space to be publicly routable in order to provide nearly 16 million more IPv4 addresses.

So, okay, the abstract for this insanity, it's very short. It says: "This document redefines the IPv4 local loopback network as consisting only of the 64K" - actually it says 65,536 - "addresses 127.0.0.0 to 127.0.255.255." In other words, in networking notation, you know, netmask notation, 127.0.0.0/16. "It asks implementers" - which, you know, is everyone - "to make addresses in the prior loopback range 127.1.0.0 through 127.255.255.255 fully usable for Unicast use on the Internet." Okay, now, in other words, the routing table in everyone's computer is now wrong. It won't work. It's broken. It won't send any packets beginning with 127 anywhere. But the IETF, and this not April Fools, they're saying, yeah, we changed our mind. And we'd like to have those addresses back, please.

Okay. So since I think this is so interesting...

Leo: Oh, my god. This is a nightmare.

Steve: ...and I'm sure - Leo. It's, oh, okay. Well, okay. Since I'm sure that any of this podcast's listeners who are aware of Internet engineering, as you obviously are, Leo, are currently picking themselves up off the floor, I'm going to share a few more of the good bits from the proposal. So, like, what?

So introduction. They say: "With ever-increasing pressure to conserve IP space on the Internet" - ahem, IPv6 - "it makes sense to consider where relatively minor changes" - okay. First of all, relatively minor, okay - "changes can be made can be made to fielded practice to improve numbering efficiency. One such change proposed by this document is to allow the Unicast use of more than 16 million historically" - okay, and it is more because it's like 16 million, 700 and something or other, so yeah, more than 16 million - "historically reserved addresses in the middle of the IPv4 address space. This document provides history and rationale to reduce the size of the IPv4 local loopback network" - and they have in parens ("localnet") - "from a /8 to a /16, freeing up over 16 million IPv4 addresses for other possible uses."

They said: "When all of 127.0.0.0/8 was reserved for loopback addressing, IPv4 addresses were not yet recognized as scarce. Today, there is no justification" - and I agree with him on this point - "there is no justification for allowing 1/256th of all IPv4 addresses for this purpose, when only one of these addresses [127.0.0.1] is commonly used, and only a handful are regularly used at all. Unreserving the majority of these addresses provides a large number of additional IPv4 host addresses for possible use, alleviating some of the pressure of IPv4 address exhaustion."

So because I think this is so interesting, they said: "Background. The IPv4 network 127/8 was first reserved by Jon Postel in 1981 under RFC 0776. Postel's policy was to reserve the first and last network of each class, and it does not appear that he had a specific plan for how to use 127/8. Apparently, the first operating systems to support a loopback interface as we understand it today were experimental Berkeley Unix releases by Bill Joy and Sam Leffler at the University of California Berkeley.

The choice of 127.0.0.1 as loopback address was made in 1983 by Joy and Leffler in the code base that was eventually released as 4.2 BSD. Their earliest experimental code bases used 254.0.0.0 and 127.0.0.0 as loopback addresses. Three years later, Postel and Joyce Reynolds documented the loopback function in November of 1986 in RFC 0990, and it was codified as a requirement for all Internet hosts three years after that, in RFC, the one I mentioned first, 1122.

"The substantive interpretation of these addresses has remained unchanged since RFC 0990 indicated that the network number 127" - that is, the entire network 127 - "is assigned the loopback function, that is, a datagram sent by a higher level protocol to a network 127 address" - meaning any IP beginning with 127-dot - "should loop back inside the host. No datagram 'sent'" - and they had that in quotes because - "to a network 127 address should ever appear on any network anywhere. Many decisions about IPv4 addressing contemporaneous with this one underscore the lack of concern about address scarcity. It was common in the early 1980s to allocate an entire /8 to an individual university, a company, a government agency, or even a research project." After all, no one's using this crazy Internet. And we have 256 of those /8 networks. So, you know, let's just give them out.

"By contrast," they write, "IPv6, despite its vastly larger pool of available address space, allocates only a single local loopback address (::1)," and that's defined in RFC 4291. "This appears," they write, "to be an architectural vote of confidence in the idea that Internet protocols ultimately do not require millions of distinct loopback addresses."

Leo: No, one would do, yeah.

Steve: Yeah. And that's, like, what most people use; right? 127.0.0.1.

Leo: This is localhost. This is home sweet home.

Steve: That's it.

Leo: Yeah.

Steve: That's right. They said: "Most applications use only the single loopback address 127.0.0.1 (localhost) for IPv4 loopback purposes, although there are exceptions. For example, the systemd-resolved service on Linux provides a stub DNS resolver at 127.0.0.53."

Leo: Oh, I didn't know that.

Steve: Which is kind of cute because port 53 is DNS's port. So it's like, let's give it 127.0.0.53, probably bound to port 53. They finish this section: "In theory, having multiple local loopback addresses might be useful for increasing the number of distinct IPv4 sockets that can be used for inter-process communication within a host. The local loopback /16 network retained by this document" - that is, remember, so 127.0, we get to keep that, so dot 0 dot anything dot anything, it's 127.1 all the way up to 127.255. That's what they're proposing removing. So you get to keep a .16, which means there would be 64K, you know, 65,536 IPs, each that could have a port. Ports are 16 bits. So basically you have 32 bits of local connectivity, and that's as we know 4.3 billion. So that ought to be enough because, again, mostly we're using 127.0.0.1.

And they finish with Section 3 titled Change in Status of Addresses Within 127/8: "The purpose of this document is to reduce the size of the special-case reservation of 127/8, so that only 127.0/16 is reserved as the local loopback network. Other IPv4 addresses whose first octet is 127," which is to say 127.1 through 127.255, "are no longer reserved and are now available for general Internet Unicast use, treated identically to other IPv4 addresses and subject to potential future allocation.

"All host and router software SHOULD [in all caps] treat 127.1 through 127.255 as a global Unicast address range. Clients for auto-configuration mechanisms such as DHCP SHOULD [all caps] accept a lease or assignment of addresses" - because, you know, like, they won't now. They go, what? Anyway, "SHOULD accept a lease or assignment of addresses within 127.1 through 127.255 whenever the underlying operating system is capable of accepting it. Servers for these mechanisms SHOULD assign this address when so configured."

Okay. Now, in this case I'm not even going to rhetorically ask what could possibly go wrong because the question doesn't need to be asked. The question is what have these people been smoking?

Leo: They probably are using one of those old forceps to do it, whatever it was.

Steve: Oh, yes. That way you would smoke it right down to the bitter end. Okay. Just think, just think of all the many millions of existing embedded TCP/IP stacks in all the many millions of existing IoT devices that sold and shipped with what has, since the dawn of the Internet, been a default local routing table that will never forward any packet starting with 127. Just like they said in Section 2. It's like, yeah, 127 should never appear on any network. And now they're saying, well, could we change our mind? No. You can't change your mind.

You know, every Internet device, every appliance there is has a routing table like Windows in front of me, like everybody has. Whatever, even your phone, you know, well, that's probably IPv6 from carriers. But as we know, none of these existing devices will ever be updated or changed. And what of all the many millions of small office/home office, you know, SOHO routers running embedded Linuxes that will also never be updated and will similarly never forward any packets starting with 127? You could send one into it, and it would die there because its stack won't forward it. It won't send it anywhere. Its stack says anything beginning with 127 goes to the local interface. That is, it dies when it hits a stack like that.

So what's interesting to me is from a political standpoint the only way to read this is as an extreme desperation move which is really interesting for its own sake, and an appreciation of just how badly no one wants to move to IPv6. You know, we've had it, what, 20 years? No. It's going to be interesting to see what happens to this proposal. Those of us who see the folly of this are not alone. I grabbed a sampling. Here's a sampling of nine tweets from just one thread on Twitter when someone mentioned this upon finding it.

Ben Aveling tweeted: "Was it a mistake to allocate an entire A class to the loopback range? Yes, yes it was. Is it too late to fix that mistake? Yes, yes it is."

Leo: Yes, yes it is.

Steve: Rich Mirch tweeted: "Real products use it. For example, F5's BIG-IP has several subnets under 127/8. Loopback is configured as a /24." And I have a screen shot in the show notes. It's very dim. But you can see a 127.2.0.2/24, a 127.1.1.254/24, a 127.20.0.254/16. The point is, because these have been unusable, people have used them, you know, within a controlled environment like F5's Big 5 products. They're able to say, hey, you know, we're going to - since it's absolutely impossible to route these, we can use them safely.

Andy90 tweeted: "TLS 1.2 was around for so long that most middlebox vendors didn't handle handshakes appropriately, thus TLS 1.3 has to masquerade as 1.2." He said: "I dread to think how many network tools simply are hardcoded and untested for anything like this and would break routing on the spot."

Jim Kladis tweeted: "I've seen 127 addresses in backbone hops." That's sort of what F5 is doing. He says: "There's much better waste to go after. IBM," then he says, "/8. HPE," of course HP Enterprise, right, "/8. Xerox /8." He says: "Then the tech universities and anything the DoD still has." And of course we've talked about this before; right? And he's right. There are ridiculously large current reservations that are absolutely not in use.

Luca Francesca said: "If only we had an alternative like, I don't know, IPv6? I know, I know, bleeding-edge stuff. It's only 20 years old."

One Matt Among Many tweeted: "Thinking of the number of firewall rules I've seen, by default, every one of them, which have 'Allow 127/8.'" He says: "Please just use IPv6 already."

Pascal Ernster tweeted: "Before even *considering*" - and he has that in asterisks for emphasis - "embarking on this journey of nonsense, they should reassign a dozen of the 14 /8s that the DoD has. And when those have been reassigned, continue with the /8s that are currently assigned to Apple, Ford, Prudential, and the United States Postal Service." He says: "Heck, I'd even go as far as saying that all /8s should be assigned to regional Internet registries."

Jonathan Katz tweeted: "Other than security, there is the practicality of this. I'm sure there is lots of software, legacy and otherwise, out there with 127/8 hard-coded in it which would break badly." And that includes every operating system we're using today.

And finally, David tweeted: "Pretty much all operating systems boot up with a 127.0.0.0/8 loopback by default. Changing that would require updates to software and firmware, or changes to startup scripts. These newly routable addresses would be inaccessible to many devices. Who would want them?" And that really is...

Leo: Well, that's a good point. You're freeing up stuff nobody would want anyway.

Steve: Yes. Exactly.

Leo: That's a really good point.

Steve: And nobody, you just couldn't ever know that you were like, you know, you get all set up, and you hook yourself onto the Internet, and why does it go so quiet? I'm not getting - why am I not getting any - you know, I've got 127.2.0.0, and nothing's coming in.

Leo: Nobody's going to use it.

Steve: And, you know, the points that were made about all the other low-hanging...

Leo: Right.

Steve: .../8 networks I think were really interesting and right on point. It must be that companies see their legacy allocations as corporate assets.

Leo: Yup. Yup.

Steve: They must know that they'll never ever have any need for nearly that much space as they currently own. And what's - I guess it must just be, I mean, for the IETF to be considering this craziness, it just must be that it's like, what is it, impolite to say, you know, Xerox? Really? Do you need all that? 16 million IPs? Really? Because, like, the world wants them. And as a matter of fact, IPv4 addresses are currently trading at $36 apiece. And a 16 million /8 allocation is currently valued at more than half a billion dollars.

Leo: Wow. Wow.

Steve: So that price has been rising steadily through the years as IPv4 desperation has been increasing. So you can imagine that corporations are sitting on these things like, hey, you know, this is appreciating faster than bitcoin.

Leo: Yeah.

Steve: We're just going to sit here and wait until this levels off. At some point IPv6 will start happening. And when we see the price stop increasing, then we'll start letting some of our /8 allocation up for sale. But, you know, it's difficult to defend having the non-profit U.S. Department of Defense squatting on so many /8 blocks. I don't know. Somebody, one of our techie senators, we do have a few, thank goodness, they ought to take a look at that and go, you know, DOD, you don't need all that.

Leo: Wow.

Steve: Anyway, I thought it was really interesting. Of course the counterpoint is IPv6. It's there. It's ready to use. But, boy, there is really some pressure against doing so.

Leo: Back to Steve, more Security Now!.

Steve: Yeah. So Microsoft actually has a job position known as - I'm not kidding, Leo - "Head of Deception."

Leo: Now, that's a job I want. I love that.

Steve: I figured he was leading the Windows 11 team, but no.

Leo: That's their marketing department. Oh, no, no.

Steve: There actually is a position, Head of Deception. The slot is currently occupied by a security researcher named Ross Bevington. Ross is the guy who puts pots of tasty honey out onto the Internet, thus the deception, and collects all of the attempts that are made to get in. He spent 30 days collecting more than 25 million brute-force attacks against a tantalizing SSH server, and he came away with a few interesting insights. The graph that is here in the show notes shows one of the things which he came up with. This is a distribution of SSH passwords seen in brute-force attacks by length of the password brute forced. And this shows a big peak at six, meaning six-character...

Leo: What?

Steve: I know. Six-character passwords. 30% of all passwords used in brute-force attempts are six characters in length.

Leo: Wow. Wow.

Steve: Crazy. So 25 million brute-force attacks, and here's what he found. He found that 77%, just over three quarters of all attempts, guessed a password between one and seven characters. So again, the brute forcers are brute forcing what they have learned works. Three quarters, 77%, are guessing between one and seven characters. A password over 10 characters was only seen in 6% of brute-force attempts. Only 7% of the brute-force attempts analyzed in that sample data included a special character. Let me say that again. Only 7% of brute-force attempts included a special character. 39% had at least one number. And none, not one of the brute-force attempts used passwords that included white space. So...

Leo: Oh, that's interesting. I never even thought of it including white space. Huh.

Steve: Uh-huh. Apparently no one has, and so no one brute forces. So there's a little takeaway.

Leo: Oh, good to know, yeah.

Steve: For our password designers. Ross's conclusion is that the attackers don't bother attempting to brute force long passwords. They just - they know that the password space is too large for them to try. And so as this graph shows, this peaks at six, drops down to about 14% that were seven characters in length, 10% that were eight characters in length. For some reason there's a little increase at nine characters. But then it just has a tail that falls off to 5% that were 10 characters long, looks like maybe about 2% were 11 characters long, and 1% were 12 characters long. And so the good news is long passwords are your friend. They are not going to get brute forced, based on the data that Microsoft has collected.

He said that based upon data from more than 14 billion brute-force attacks attempted against Microsoft's network of honeypot servers through September of this year, attacks on Remote Desktop Protocol servers had tripled compared to 2020, which tracks everything that we've been seeing that we've been talking about, they saw a rise of 325%. Network printing services saw an increase of 178%, as well as Docker and Kubernetes systems, which saw a relatively smaller increase of 110%. So attacks are on the rise. You know, you really don't want to expose RDP to the Internet without lots of additional protection. And as I've said, the thing to do is to put it behind a VPN so you can use a VPN's security, then get to the RDP server and log into that.

Tied into that was a report just issued from NordPass. They published their annual analysis of password use across 50 countries. The top 10 most common passwords, like still, in 2021, currently are, the absolute most-used password, and this also explains, by the way, Leo, why there's a peak on six-character-long passwords, the thing that is holding the pole under that tent that is that chart is the password 123456.

Leo: Oh, lord. For SSH. I mean, you might as well just, you know, use Telnet. You know, if you're going to take the trouble of using something secure, make a password.

Steve: Yes.

Leo: For crying out loud.

Steve: 103,170,552 hits on the "password," and I put that in quotes, 123456. Not easy to forget. Pretty easy to brute force. And everybody tries it. And what was interesting was that was more than twice the number of hits of number two. Number two, guess, 123456789. Oh, Leo 789. That explains the other piece.

Leo: There's the other tent pole, yes.

Steve: The other little tent pole under that chart, yup.

Leo: Yup.

Steve: That was at 46,027,530 hits. Then we have for the lazier people, 12345. Yeah, just can't really be troubled to hit that sixth key. And then, believe it or not, at 22 million is qwerty, Q-W-E-R-T-Y. Because, you know, it's right there in front of you. Also, almost equal to QWERTY, coming in with eight characters, is just shy of 21 million, and the password is itself, P-A-S-S-W-O-R-D. And for someone who thought, I'm not going to go all the way out to 123456789, I'm going to be tricky and stop at 8 because, you know, they're not going to check that. It's like, okay.

Also in the six-character password category is 111111. And then we also - and then that was at 13.3 million. And we have 123123 because, you know, that's even trickier than 123456. That's at 10.2 million.

Leo: It's also easier to enter.

Steve: It is, yes.

Leo: Ba da dump, ba da dump.

Steve: Brump brump, you're right. And then number nine is 1234567890, that's just shy of 10 million at 9.646 million. And 1234567. So basically nobody cares. You know? And remember the studies have been done. You know, if I've got a candy bar here, would you give me your password for a candy bar? And most people, yeah, you know, I'm hungry. That looks like a good candy bar. Is that a Milky Way?

Leo: Yeah.

Steve: So among their other findings, the researchers found that, in their words, a "stunning number" of people use their own name - that's tricky - as their password.

Leo: Oh, good.

Steve: Charlie appeared as the ninth most popular password in the U.K.

Leo: Charlie bit my finger.

Steve: Onedirection was a popular music-related password option.

Leo: Oh, yeah, yeah, the group.

Steve: And the number of times Liverpool appears could indicate how popular the football team is. But in Canada hockey was unsurprisingly the top sports-related option in active use. Swear words are also commonly employed.

Leo: Yes, yes.

Steve: And when it comes to the...

Leo: Not in the dictionary.

Steve: You know? And when it comes to animal themes, dolphin was the most popular choice internationally. So Leo, it looks like monkey is out.

Leo: Dolphin has replaced monkey.

Steve: Unfortunately, dolphin has replaced monkey.

Leo: Just a plug for using ssh-keygen. Generate an ECC private and public key pair.

Steve: A certificate.

Leo: Yup.

Steve: Yes.

Leo: I do this on all, you know, GitHub, everywhere. And at first I was using the same private-public key everywhere. And I said, you know what, it's easy, simple. So I have a different one for every machine, makes it easy to revoke a machine.

Steve: And if you have a good SSH client, it will keep them all straight for you and manage them.

Leo: Absolutely, yeah.

Steve: So all you do is select who you want to log onto, and it does the job.

Leo: It's much better.

Steve: I use Bitvise for Windows. I'm really happy with Bitvise.

Leo: Cool, that's a good tip. I'm not familiar with that. That's good.

Steve: Anyway, I'm somewhat surprised, since those are purely numeric, without any letters or special characters, most contemporary websites would never allow them to be used. But, you know, I was thinking, if they were in place before specific requirements began to be added to sites, you know, I can see as legacy passwords they could still be valid today.

Leo: Well, and also SSH usually you're controlling the server. I mean, the server is probably not configured to be that secure.

Steve: Right, that's a very good point, yeah.

Leo: And is allowing that, obviously.

Steve: Yeah.

Leo: So, yeah.

Steve: Wow. Unbelievable. And if this is enterprise login, because I mean the presumption is that these are enterprise SSH servers. How could they not enforce a policy that makes more sense? That's just mindboggling. Well, yesterday the well-known Internet domain registrar and more recently cloud hosting provider GoDaddy said that a hacker gained access to the personal information of more than 1.2 million customers of its WordPress hosting service using a compromised password which gave the attacker access to the provisioning system in their legacy codebase for their Managed WordPress. So reading between the lines it sounds like they had, like, a legacy codebase. They had some, like a WordPress hosting management system that they'd stopped using; right? Like they were no longer using it, but they didn't turn it off because, hey, if it's not broke, wait till a hacker finds it.

Being one of the world's largest domain registrars and a web hosting company providing services to more than 20 million customers worldwide, as well as being publicly traded, they needed to promptly inform the U.S. Securities and Exchange Commission. In the SEC documents which were filed yesterday, GoDaddy stated that it discovered the breach last Wednesday after noticing what they called "suspicious activity" on their Managed WordPress hosting environment. Subsequent investigation revealed that a hacker had unfettered access to GoDaddy's servers for more than two months.

Leo: Oh.

Steve: Yeah, ouch, since at least September 6. And based upon the available evidence, the hacker had gained access to up to 1.2 million active and inactive managed WordPress customers who had their email addresses and customer numbers exposed; the original WordPress admin password that GoDaddy had issued to those customers when a site was first created; for active customers, their secure FTP login and database usernames and passwords were exposed; for a subset of active customers the SSL private key was exposed. Ouch.

GoDaddy had already reset the secure FTP and database passwords exposed in the hack as well as the admin account password for customers who were still using the default one that GoDaddy had originally issued when their sites were created. GoDaddy is currently in the process of issuing and replacing new SSL certificates for affected customers. And they've notified law enforcement and are working with an IT forensics firm to further investigate the incident. Customers were also notified of all this, or like a sanitized version of this actually, yesterday.

And for those who are counting, this is not GoDaddy's first trouble with breaches. Last May GoDaddy alerted some of its customers that an unauthorized party used their web hosting account credentials in the previous October of 2020 to connect to their hosting account via SSH. In that instance GoDaddy's security team discovered the breach after spotting an altered SSH file in GoDaddy's hosting environment and suspicious activity on a subset of GoDaddy's servers. In other words, they saw this guy doing stuff.

And back in 2019 scammers used hundreds of compromised GoDaddy accounts to create 15,000 subdomains, attempting to impersonate popular websites and redirect potential victims to spam pages offering bogus products. And before that, earlier in 2019, GoDaddy was found to be injecting JavaScript into U.S. customers' sites without their knowledge, thus potentially rendering them inoperable or impacting their overall performance. So there's been some problems.

I registered GRC.com in December of 1991, a few months after Microsoft registered Microsoft.com. And I chose Network Solutions since they were the original Internet registrar. But our longtime listeners will know that I could finally no longer tolerate Network Solutions' slimy upselling tactics. It was necessary to say "no" to various offers over and over again and, more annoyingly, to carefully read and uncheck various enabled-by-default additional cost services which other registrars were by then offering for free.

So I started to feel as though my loyalty had become misplaced, and I decided that I had to move. As we know, I chose Hover, also a sponsor of the TWiT Network, and I've never looked back. When I was making the switch, I considered GoDaddy. Mark Thompson uses them and recommended them. But they were too brightly colored and sort of hyper-commercial-looking.

Leo: Yeah. Talk about upsell. They're the worst.

Steve: Oh, my god, yes.

Leo: And aren't you now glad that you didn't?

Steve: Oh, Leo, yes, exactly, and that was the point I was going to make is their website looks like a cartoon. And that's what I wanted to get away from. The last thing you want in a domain registrar is excitement. Things are rather excited over at GoDaddy right now. No, thank you. What you want from a domain registrar is a great deal of boredom.

Leo: Boring. Boring.

Steve: Just, yes. Just do that job and don't go jumping up and down and trying to be more because, boy, you open yourself to this kind of mess.

Leo: Yeah.

Steve: Okay. A heads-up, an important heads-up for our Netgear owners. I know we've got them. Last week Netgear released a round of patches to remediate a high-severity remote code execution vulnerability affecting 61 different models. I've included a table of the impacted routers in the show notes below. But really, I wouldn't bother, like, looking for your model there. It's, like, all of them. All of our listeners who are using Netgear routers should make a point of checking in right now, like after this podcast, for any available update.

This one is another Universal Plug and Play vulnerability, you know, UPnP, which when exploited would allow remote attackers to take control of a vulnerable system. The good news is most routers won't be exposing their vulnerable UPnP ports on the public Internet. On the other hand, GRC's UPnP Internet Exposure Test has counted 55,166 positive Internet-facing tests since I put it online. But I'm not logging unique IPs. I do keep a most recently seen IP list. I can't remember now how long it is and how I expire it. But it's like a day. The idea was I didn't want to be double-counting ShieldsUP! visitors if they did multiple tests, like trying to get their routers configured right. And this thing, and all of the ShieldsUP! system reshares that same MRU list so that it doesn't double count. So 55,166 without recent double counting.

Okay. But anyway, hopefully, if you have a Netgear router, even if you don't have the UPnP service bound to your WAN interface, all of these routers will have UPnP running on their LAN interface by default. And that exploitation is still possible. Exploitation could be by way of a malicious script running on a browser inside the LAN, thus accessible to the router's internal LAN interface at a known IP, right, the broadcast, the gateway IP and a known port, what is it, 1900 is UPnP. And then that would be used to make the router remotely accessible. So if you would go to a site that was hosting JavaScript or a malicious ad, potentially, that is running script on your browser. We've talked about this, techniques used for browsers to access the unprotected LAN port on routers. In this case it is trivial to do this.

In recognition of the severity, it's assigned a CVE of 2021-34991 with a CVSS severity of 8.8. It's a pre-authentication, meaning you don't need to authenticate, buffer overflow flaw which appears to be present in pretty much all of Netgear's small office/home office routers, and it can lead to remote code execution at the highest privileges. The vulnerability stems from the fact that the UPnP daemon accepts, by design, unauthenticated - because remember Universal Plug and Play is an unauthenticated protocol because the idea is it's just going to be, like, automatic. So you can't have to log in to the router's UPnP or it wouldn't be automatic. Thank you, Microsoft.

Anyway, unauthenticated by design, HTTP SUBSCRIBE and UNSUBSCRIBE requests, which are event notification alerts that devices use to receive notifications from other devices when certain configuration changes such as media sharing occur on the network. But there's a memory stack overflow bug in the code that handles the UNSUBSCRIBE requests which enables an adversary to send a specially crafted HTTP request and run malicious code on the affected device, including resetting the admin password and delivering arbitrary payloads. And as we know, HTTP requests are the things that web browsers routinely generate.

So the idea of JavaScript doing this is not farfetched. Once the password has been reset, the attacker can then log in to the web server and modify any settings or launch further attacks on the router's internal web server. So again, if you're using Netgear, just it's time to go see if there's an update. As of last week, there is, for 61 different routers. And I'm seeing all the WNDR routers, you know, the WNDR 3000 or the 3300, 3400, 4000, 3500, I mean, just...

Leo: I don't see the Orbis in here.

Steve: No.

Leo: And that's their mesh solution. Those are RBL, I think. So that's good news. But everything else, yeah.

Steve: Yeah. So the guy who discovered and reported the trouble noted that: "Since the UPnP daemon runs as root, the highest privileged user in Linux environments, the code executed on behalf of the attacker will run as root, as well." With root access to a device, as we know, an attacker can read and write, do whatever they want to traffic, modify configuration, so forth. So, again, update.

Leo: Okay, Steve. That bottle's almost empty. You'd better wrap this up here.

Steve: HTTP Request Smuggling, also sometimes called HTTP Desync Attacks, take advantage of the fact that much of today's modern Internet is far more complex than just a web client connecting to a web server. Any website hosted, for example, by Cloudflare provides an example of just how complex the Internet has become. Many of today's websites and web hosting applications employ chains of HTTP servers between their users and the back-end application logic. Users' requests are first received by a front-end server. It might be performing web filtering, load balancing, reverse proxying and/or caching. Whatever the case, this front-end server then forwards requests to one or more back-end servers. And not only is this type of architecture increasingly common today, in many cases it's fundamental to cloud-based applications. It's built-in. The assumption is that's the way that web content is being delivered. You know, CDNs operate similarly, very much like Cloudflare.

For the sake of expediency and efficiency, the front-end connections to back-end servers are persistent and long-lived. This saves a huge amount of connection setup and teardown time. And the protocol is very simple, with the front-end sending HTTP requests one after another to the receiving server on the back end, which parses the HTTP request headers to determine where one request ends and the next one begins. In other words, to determine the request boundaries it's necessary to parse the headers. And when this is being done, it's crucial that the front-end and the back-end systems agree about the boundaries between requests. Right?

So the front-end system is building these, you know, receiving them and doing whatever it's doing, and then forwarding them. And it's got a persistent linkup to the back-end, and it's just sending requests down the pipe, one after the other. Naturally, the back-end is receiving these requests from its end of the pipe coming from the front-end, and it needs to understand the start and the end of the requests. If that doesn't happen, if the systems do not agree, an attacker might be able to send an ambiguous request that gets interpreted differently by the front-end and back-end systems, which is of course exactly what can be made to happen, and we're going to explain how.

HTTP Request Smuggling is a slippery and tricky technique which deliberately manipulates the Content-Length and Transfer-Encoding headers of multiple HTTP requests in such as way that the various servers involved in servicing the requests get confused about the successive HTTP request boundaries. And they get confused in a way that allows clever attackers to achieve web cache poisoning, session hijacks, cross-site scripting, and even web application firewall bypasses. And unlike many of the interesting technological attacks and tricks we often discuss here, these are not theoretical.

Last week, as I mentioned at the top of the show, one of the vulnerabilities that was being exploited used HTTP Request Smuggling as part of its attack chain. These are the real deal, and they're getting a lot of attention right now because believe it or not, this was first mentioned in 2005. And because we don't have today's - we didn't then have today's multi-server linked cloud-based connectivity, it was like, eh, you know, kind of a - then it was theoretical. Not anymore.

Okay. So I mentioned the Content-Length and Transfer-Encoding headers. HTTP Request Smuggling vulnerabilities arise because, unfortunately, the HTTP spec provides two different ways to specify where a request ends. That is, the length of the body which is appended after the headers. Remember that an HTTP Request is a bunch of headers like the name of the host, if modified since, where you're able to say, okay, you know, I want you to send this to me only if it's been modified since a certain date because the client that's asking already has a copy in its cache with that date, so it only gets things that are changed. You know, so this is the so-called metadata of HTTP Requests. And of course, famously, cookies are also in the metadata.

Okay. So believe it or not, as I said, the HTTP spec provides two completely different ways to specify where a request ends. So you have - and remember this is just ASCII text. It's going serially down a line. So you have a line of characters, then a character return line feed; the next line of characters, character return line feed; the next line of characters and so on. So how do we know when a request ends?

The Content-Length header is very straightforward, and it's by far the most common method. It simply specifies the length of the message body in bytes. So, for example, you'd have, in a POST query, you'd have POST and then space /search, if that was the - if the URL was search at this site. And then you'd specify the host name that you're wanting to send this to, the content type.

Then you have Content-Length colon, and say that you had it as - say it was 11. So you then have a blank line to separate the headers from the beginning of the body. And then, since this thing says Content-Length 11, the server interpreting this will accept the next 11 characters as being the content for this query. And what's significant about that is that then the 12th character is absolutely considered to be the first character of the next query. In other words, the thing specifying the boundary between the first and the second query is the Content-Length header for the first one which says exactly how many characters that follow are part of that query. And the implication is, and after that, it's a next query.

And if it surprises somebody that that's the way the world works, I mean, like there isn't some special end-of-query character reserved or something, it's like, yeah, there isn't. You know, and pretty much things work except when they don't. And this is an example of how badly it can be broken. So in this example we have an 11-character string, just for the - I'm looking at the show notes. But for those who aren't seeing them, q=smuggling, S-M-U-G-G-L-I-N-G. So that's 11 characters. And the end of that ends this query.

Okay. But I said there were two ways of specifying the length. The use of a Transfer-Encoding header is also completely valid. It's a different way, a completely different way of specifying the end of a query. And it, too, can be used to specify essentially the structure of the HTTP message body. The transfer encoding method is called "chunked." When the Transfer-Encoding header specifies chunked encoding, this means that the message's body consists of one or more chunks of data where each chunk consists of first the chunk size, which is specified in hex, as a hex count, that is, it's a hexadecimal value that's ended with a new line, so character return line feed, followed by that many characters. And so you have a chunk size and then a chunk.

Then you might have the number of characters specified in hex by the chunk size, funky as this sounds. And then you have another chunk, which could be again another chunk of some length. And you can have as many chunks as the sender wants to create. The message ends when the thing that's interpreting this query encounters a chunk declared of zero size.

So again, if we were to be sending this q=smuggling line instead of using Content-Length, saying the Content-Length, saying the Content-Length is 11, we saw that we're going to use the Transfer-Encoding style with the chunked method. So after the blank line following the headers is a B, which is hex for 11; right? Ten, you know, you have nine, then A, then B. So you have nine, 10, 11. So there's just a B on a line by itself that tells the thing that is reading this HTTP query that here comes the 11 characters in the first chunk. And so they read q=smuggling. Then the next line is a zero, which says that that's the end of the message because the next chunk, there isn't any, it's of zero length. And so that tells the thing that has received this query that the message has ended.

Okay, now, first of all, the students of this podcast will immediately see how fragile this chunked encoding is. I mean, it is ugly. For one thing, it's doing a bit of interpretation. We know how we feel about interpretation. And we know how we feel about interpreters on this podcast. But the big no-no from an architectural design standpoint is that it is mixing control metadata in with the data. We've previously seen, for example, how the immensely popular and powerful "printf" function, which is present in so many coding languages, suffers from a similar design vulnerability by mixing data and control metadata, right, those little format escape characters, into the same string. As we saw in that case of the Apple flaw, if an attacker can get control of that string, they can get up to a great deal of mischief.

So as for HTTP, many researchers are unaware that chunked encoding is even, like, still around anymore, and that it's valid in HTTP requests because web browsers don't use chunked encoding in their requests. They use Content-Length. And it's normally only seen in server responses. But it's in the spec, and many servers support it. If they're going to be HTTP spec compliant, they must support it. Now, here's the problem. Since HTTP provides two entirely different methods for specifying, or I guess I'd say for obtaining the length of HTTP messages, and one is quite fragile and susceptible to abuse, it's possible for a single request to use both methods.

And this can be done in such a fashion that they conflict with one another. This possibility was understood by HTTP's designers. You know, they had the Transfer-Encoding method and the Content-Length method. And they realized, okay, these do different things, but they also sort of do the same thing. So they attempted to prevent the problem of this kind of like header collision by simply stating that if both the Content-Length and the Transfer-Encoding headers are present, the Content-Length header should be ignored.

Now, unfortunately, that means that if they're both present, the more fragile of the two wins the contest. And it turns out that while this simple exclusion might be sufficient to avoid ambiguity when only a single server is in play, when two or more servers are chained together, bad stuff can happen. And as I said, in today's Internet it's very often the case that you've got multiple servers chained together. Bad stuff happens because some servers don't support the Transfer-Encoding header in requests at all, and some servers that do support the Transfer-Encoding header can be induced not to process it if the header is obfuscated in some way. And I'll explain in a second.

And here's the key. If the front-end and back-end servers behave differently in relation to the perhaps obfuscated Transfer-Encoding header, or Content-Length versus Transfer-Encoding, they might disagree about the boundaries between successive requests, which enables Request Smuggling vulnerabilities. Okay. So Request Smuggling attacks are created by placing both the Content-Length header and the Transfer-Encoding header into a single HTTP request and manipulating them so that the front-end and back-end servers will process the request differently. And naturally the design of a successful attack depends upon the behavior of the two servers.

So attacks come in three forms depending upon the characteristics of the specific servers being attacked. There's the CL.TE attack. And of course that stands for Content-Length/Transfer Encoding. The CL.TE attack: When both headers are present, the front-end server obeys the Content-Length header, and the back-end server obeys the Transfer-Encoding header. You have the reverse, the TE.CL attack. When both headers are present, the front-end server uses the Transfer-Encoding header; the back-end server uses the Content-Length header. And then you also have the TE.TE, now, which might seem strange. But when both headers are present, the front-end and back-end servers both support the use of the Transfer-Encoding header, but one of the servers can be induced not to process it by obfuscating the header in some way.

Okay. So how does this work? Let's take a look at the first case, the CL.TE attack where, when both headers are present, the front-end server, the first server to see the request, obeys the Content-Length header, and the back-end server uses the Transfer-Encoding header. So in this request, remember, both headers are present. So in this example we have Content-Length specified as 13 and Transfer-Encoding as chunked. So the body starts with a numeric zero on a line by itself, and then a blank line to end that zero, and then the word "smuggled." So what happens?

The front-end server - oh, I'm sorry. And also we have - so Content-Length 13 and Transfer-Encoding is chunked. But the first server, remember, it obeys Transfer-Length encoding. So it says, okay, there's going to be 13 characters following, so that's the zero, the Character Return Line Feed for the new line, and then this word "smuggled." That's 13 characters, so that's what it sends. It forwards the entire request as received to the back end. But the back-end server is more compliant with the HTTP specification, which remember, if both are present, Transfer-Encoding wins. So it ignores the Content-Length header and instead processes the Transfer-Encoding header as it's supposed to. It therefore treats the message body as using chunked encoding.

So what happens? The first thing it sees is that zero on a new line, that zero by itself and a new line, which tells it this message ended. That is, you know, you can send a post or a query or something with no body. So it sees this 13-character body message as a query that had no body. Which means it starts parsing what it assumes is a new query with the word "smuggled." Of course, this is just our example. What is actually there is another HTTP Request; right? And the first server didn't see that second HTTP Request which is being smuggled because the Content-Length enveloped the entire thing. So it just thought that was some, I mean, the body of a query can be anything you want. So it's not - it's taken as a literal and just passed along.

But the second server, because of that zero and the blank line and the fact that it's obeying Transfer-Encoding, it terminates that first query and starts reading immediately afterwards as the next query, which allows someone to smuggle a query past the gateway, past that front server. It slips right through. Therefore, if our attacker had embedded some sort of useful query into the end of the first query, the front-end server, as I said, would not have seen it. It would have treated it as query data and passed it along. The back-end server would have seen that hidden embedded query as its own freestanding second query and would have acted upon it. So obviously, if the front-end server is examining requests and functioning as an HTTPS firewall, a web firewall, a caching server where it needs to know what's gone through, this technique slips a query to the back-end right through a front-end firewall.

Now, it turns out the reverse of this attack, the TE.CL, works just as well. It can be used when the front-end server uses the Transfer-Encoding header and the back-end server uses the Content-Length header in the presence of both. Basically, the attack is, the Content-Length in this example is 3, and the Transfer-Encoding is chunked. And then we have the blank line at the end of the headers, then the number 8 and the word "smuggled," and then a zero. So of course for the chunked encoding, 8 is the number of letters in the word "smuggled" and then the zero terminates the chunked encoding. Since the front-end server honors chunked encoding, it will interpret the message's two chunks as a single message and will forward the entire query to the back-end server. Right? Because that's just a standard chunked encoding. It ignores the Content-Length of 3.

But the back-end server obeys Content-Length and not the Transfer-Encoding. So it sees the length of 3. Well, that gives it the 8, and the character return line feed. And so that's the end of the query that it sees, and it starts processing as a new query the word "smuggled," or what would actually in practice be another HTTP query of some kind. So again, it is trivial. If you've got two servers in a chain disagreeing about how they're going to handle the content, basically how they determine the message query boundaries, you're in trouble.

And the TE.TE vulnerability, turns out it's possible, there are some servers which will still process a malformed Transfer-Encoding header. Like it should be Transfer-Encoding colon space chunked. Sometimes you could do encoding space colon space chunked. Some servers accept that. Some don't. Or you could use colon tab chunked. Again, some will regard that as white space and go, okay, fine. Some will look at it and say, hey, that doesn't abide by the spec. We're not taking it. So the point is, if you put two transfer encoding headers in the same query and format them differently, you can again get differential treatment of the encoding of the message. And that's really what this comes down to, right, is some difference, differential between what two servers in the chain of servers processing the query do, and that allows all of this to happen.

So I considered taking this to the next step by demonstrating how a smuggled request could be leveraged into various forms of devastating web attacks. But I think I've pushed a predominantly audio podcast about as far as I can. Anyway, so...

Leo: You need a whiteboard, Steve, or a chalkboard or something.

Steve: Yeah, that is true. These, as I mentioned, these potential problems were first discovered and documented back in '05. Nobody was really that concerned. Researchers from Northeastern University and Akamai Technologies have written a paper titled "T-Reqs" - okay, a little cute there - "HTTP Request Smuggling with Differential Fuzzing" which was just presented during the 2021 ACM SIGSAC, which is the Conference on Computer and Communications Security.

Their paper's abstract explains: "HTTP Request Smuggling" - they abbreviate it HRS - "is an attack that exploits the HTTP processing discrepancies between two servers deployed in a proxy origin configuration, allowing attackers to smuggle hidden requests through the proxy. While this idea is not new, HRS is soaring in popularity due to recently revealed novel exploitation techniques and real-life abuse scenarios.

"In this work we step back from the highly specific exploits hogging the spotlight and present the first work that systematically explores HRS within a scientific framework. We design an experimental infrastructure powered by a novel grammar-based differential fuzzer, test 10 popular server/proxy/CDN [Content Delivery Network] technologies in combinations, identify pairs that result in processing discrepancies, and discover exploits that lead to HTTP Request Smuggling. Our experiment reveals previously unknown ways to manipulate HTTP requests for exploitation, and for the first time documents the server pairs prone to HRS."

And I have the chart that they produced as the last page of the show notes. And it is really interesting. They tested CloudFront, Cloudflare, Akamai, Varnish, Squid, HAProxy, ATS, Tomcat, NGINX, and Apache. So those form the lines on one side of the table. The exact same set form the lines on the other side of the table. And so the table is populated with symbols representing where the entry point server along the bottom fed its data to the exit point server enumerated down the left-hand side. The symbols show what happened.

So, like, there are a whole bunch of large gray circles for Tomcat where v1.0 chunked encoding was effective when Tomcat was connected to most of the exit points. Let's see. All but, oh, there's also Varnish. All but Varnish and itself, there is a blank diagonal because you're not going to get differential handling when the same thing is talking to another instance of itself. But, you know, CloudFront has a number of purple dots which is double transfer encoding presences for a bunch of senders. You know, there's stars on Various Method Version Combinations.

So anyway, the diagram shows what the researchers found when these 10 popular servers, CDNs, and proxies were fuzzed with a wide range of HTTP header mutations. That's when they talked about a novel grammar-based differential fuzzer. Basically, you know, fuzzing is just throwing everything at the wall and see what happens. And so they produced a technology. They automated the discovery of things that different servers connected to each other would handle differently. One of the items that I noted was the Double Transfer-Encoding, which suggested duplicate headers can also create vulnerabilities.

So I have a link, for anybody who's interested, to their entire paper. I think it was 19 pages. They did a beautiful job. And the good news is this is now getting a lot of attention. Everybody is looking at it. When this is done, hopefully, I would say, well, I would love to say a month from now, maybe six months from now or a year from now, this chart's going to look different because this is now on everybody's radar. And it's going to be, clearly, this is not, as I said, just theoretical. Bad guys are already using this to essentially smuggle queries past border defenses and take advantage of that and to abuse these services. So another very cool piece of Internet technology. Been sitting there the whole time. Nobody really paid any attention to it until recently.

Leo: And that's what you're here for, to pay attention to stuff so we don't have to. Steve Gibson. He's the man of the hour. Well, or the hour and a half, two hours, thereabouts. Always on a Tuesday I look forward to this. And I know all of you do. We do the show, and you are invited to watch us do it live every Tuesday, right after MacBreak Weekly. So that's, you know, it varies, but it's around 1:30 to 2:00 p.m. Pacific, 5:00 p.m. Eastern, 22:00 UTC, at live.twit.tv. If you're watching live, chat live at irc.twit.tv.

You'll find Steve at his website, GRC.com. That's where his bread and butter lives, SpinRite, the world's finest mass storage maintenance and recovery utility. If you've got an SSD or a hard drive, you need SpinRite. You should get a copy right now. 6.0 is out. But 6.1 is imminent, and you'll get an upgrade for free if you buy 6.0 now. And you can participate in the development of 6.1. There's a forum, a very active forum at GRC.com. Leave feedback for Steve at GRC.com/feedback. You should browse around, though. It's a great website, lots of great stuff, including the show.

Steve has a couple of unique versions of the show. Going back 15 years, you said we should have a 16Kb version of the show for people, like people who live in Australia and have metered connections and so forth. So that is the smallest audio file available. It's a little scratchy, but it's small. It's quick. It's an easy download. He's got that, does it faithfully every week, transcodes this into that. He's got the 64Kb he starts with. That's also there. He also has transcripts, and that's really a nice thing that Steve's been doing for some time. He pays Elaine Farris to make really nice human-readable and human-written transcripts.

Steve: Searchable.

Leo: Which are searchable, which means you can jump to anything, anywhere in the entire canon of Security Now! episodes, all 846. All of that's at GRC.com. Steve's on the Twitter, @SGgrc. If you want to DM him there, you can also leave a message there. Those DMs are open.

We have copies of the show, 64Kb audio plus video at our website, TWiT.tv/sn. You can also watch, there's a YouTube channel devoted to Security Now!. And I might encourage you to subscribe. I know you don't want to miss an episode. And you can do that in your favorite podcast client. And if your client allows for reviews, do us a favor. Leave a five-star review. Tell the world all about this guy. He's a precious natural resource, the fifth head on Mount Rushmore, I think. Steve Gibson, thank you so much. Have a great week, and we'll see you next week on Security Now!.