SECURITY NOW!

Transcript of Episode #205

Lempel & Ziv

Description: Steve and Leo examine the operation of one of the most prevalent computer algorithm inventions in history: Lempel-Ziv data compression. Variations of this invention form the foundation of all modern data compression technologies.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-205.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-205-lq.mp3

INTRO: Netcasts you love, from people you trust. This is TWiT.

Leo Laporte: Bandwidth for Security Now! is provided by AOL Music and Spinner.com, where you can get free MP3s, exclusive interviews, and more.

This is Security Now! with Steve Gibson, Episode 205 for July 16, 2009: Lempel-Ziv. This show is brought to you by listeners like you and your contributions. We couldn't do it without you. Thanks so much.

Get ready, it's time for Security Now!, the show that covers your privacy, your security, protects you online. Our protector, of course, the great Steve Gibson of the Gibson Research Corp., GRC.com, the inventor of SpinRite, the discoverer of spyware, and a lover of great cabernet. What is it, cabernets?

Steve Gibson: Cabernet, yup. Red wine. Good for you, too.

Leo: It is good for you.

Steve: Yup.

Leo: Do you like dark chocolate, as well?

Steve: Huh?

Leo: Do you like dark chocolate, as well?

Steve: Dark chocolate is very good for you, too.

Leo: I know. See, if I could live on red wine and dark chocolate, I think - hey, Steve. Good to see you again.

Steve: Great to be back with you, Leo. You are, as our listeners are hearing this, you're floating somewhere out in Asia.

Leo: Let me see. I could check the date. You know, I am the opposite of a private person, as you probably have gathered. And I have posted my entire itinerary, including everything you'd want to know except maybe the phone number to call.

Steve: So this is airing on July 16, Episode 205.

Leo: July 16. Oh, that'll be a sad day. I'll only have a day left on my cruise. We will be in the China Sea, sailing back to China for our departure. So when you're listening to this, think of me crying over the railing.

Steve: Well, we are recording this because I refuse to miss any weeks. I don't want to leave our listeners stranded. We're recording this several weeks in advance. So we don't have any security news. However, we're not going to miss any news. The episode after this one, when you're back, will be the mega security news update, where we'll be catching up on three weeks of everything that went on. And I have a feeling there'll be enough that will have happened that we can do a whole episode just on everything that happened while you were out traveling.

Leo: Let's look on the bright side. Maybe I'll come back, and nothing will have happened.

Steve: [Laughing]

Leo: There'll be no security - yeah, we laugh. Not possible. Not possible.

Steve: No. Not these days.

Leo: Not these days. So what are we going to do today?

Steve: We're going to do the second of our two episodes interestingly enough named after people, because it's about sort of a fundamental concept or technology or basically computer science breakthrough that is named after the people who invented it. In this case, the initials are "L" and "Z," for Lempel & Ziv, who were two researchers, I think they were at IBM at the time. I remember reading their original patent. And I think it was assigned to IBM, although I didn't dig back prior to the show and verify that, because the history is tangled with patents and other people doing modifications to their concept. But basically this is the way data compression is done. It was done in a whole bunch of other ways that were less good until 1977, when Lempel & Ziv told the world how to do this.

Leo: Now, I know a little bit about this because when I was a younger fellow and still could program my way out of a paper bag, I wrote a program called Mac Arc. Remember before ZIP there was ARC?

Steve: Yes.

Leo: And there was no ARC for the Mac. This was, like, '84 or '85 when the Mac had just come out. And we - I had a BBS, and we wanted to be able to compress files for the Mac. And there was no real easy way to do this. So I took the source code for ARC on the PC, which was I guess open. We didn't know about open source at the time, but the source code was available, including the Lempel-Ziv algorithms - I think it was table-based, as I remember - and ported it over to the Macintosh. I never got to the compression part, just the decompression. But even that was better than nothing. So I remember doing this, writing this algorithm out. It's clever.

Steve: Well, it's way cool. And so we're going to take our listeners through a little bit of sort of starting at how you might think about compression. And essentially we'll go through various stages of different types of compression, up until these guys said, okay, here's how it's done. And boy, were they right.

Since we don't have any security news or errata, I have nothing but sort of a fun SpinRite story to share. This was actually a posting in the GRC newsgroups that I ran across. And it's like, oh, there's someone to talk about SpinRite. Jacob Janzen is the person who wrote this. And he said, "SpinRite has saved my bacon," as he put it, "three times now, and also saved my mother's business from losing all her data, and saved us $1,500 in recovery costs. He says, "Of course I have purchased a license, which is kind of a funny story you might enjoy.

"So four years ago I was playing around with Partition Magic, and it screwed up halfway through a partition adjustment. I was floored. All of my source code," so I guess this guy's a programmer, "all of my source code was on the machine, and I hadn't backed up in a week. Which is a ton of code to lose. So I ran to my brother's house and consulted him. We determined that we needed an adapter to plug the laptop drive into his PC so we could talk to it. And his first suggestion was, 'Buy SpinRite,' which I did on the spot.

"The funny thing is, when we mounted the drive, we decided to boot into Windows first, to see if we could see any data and copy it right away. And bloody Windows fixed the broken partitioning. I then ran SpinRite anyway as a precaution, and have been using it routinely ever since. And as mentioned above, it has saved me personally three times." He has in parentheses, "(Those darn laptop drives)," he says, "and saved my mother's business, too. She now has Jungle Disk backing up all her data nightly.

"Oh, and a friend of mine in the Canadian Armed Forces used SpinRite to save a hard disk drive in the field, albeit on training, no lifesaving events, although they did manage to save face by having access to the navigation. So SpinRite has earned me a case of beer." He said, "My buddy balked at buying it, but after it saved his bacon, he bought me a case of beer in thanks. $1,500 in savings for my mom and relief from endless suffering via restored hard disk drives."

Leo: Well, I hope that not only did he buy a case of beer, he bought a copy of SpinRite.

Steve: Oh, I think he did that first. Yeah, he said his buddy balked at having to buy one.

Leo: Right.

Steve: But when it worked for him and saved him, then he was glad.

Leo: Yeah. Yeah. There you go. SpinRite is a must-have. If you have disk drives, you kind of have to have SpinRite. We use it all the time around here, and it's very handy.

Steve: It's fun to have Colleen as a believer, having seen, you know, she saw it first-hand, so...

Leo: We won her over, we really did.

Steve: That was very cool.

Leo: All right. We get to Mr. Lempel and Mr. Ziv in just a little bit, the creators of the most amazing compression algorithm ever. There have been successors; right? Are they all based on the same notion?

Steve: They're all based on this concept. I mean, very much like we talked about two weeks ago with Boyer & Moore, where once this, like, this revelation - in that case it was searching from the end of the pattern forward - once you get that it's like, oh, I'm never going to do it any other way. Well, similarly, Lempel & Ziv have a key concept which, as you said, many people have extended in different ways because theirs is so fundamental that there's all kinds of things, different things you can do with it, different ways, like twists on, you know, variations on the theme, but the theme has survived. And basically, well, not even basically, all compression today is done with Lempel & Ziv at its core.

Leo: Isn't that neat.

Steve: And we're going to explain what that is.

Leo: All right, now, let's talk about Mr. Lempel and Mr. Ziv.

Steve: Well, we'll wind back first and talk about the things, the approaches that led up to it. First of all, all of this is data compression. And this occurred to me, not only because it's another sort of fundamental core piece of computer science, but of course a couple weeks ago we were talking about SecureZIP, ZIP and ARC, and the things that PKWARE does, and basically compression everywhere. When you right-click in Windows and say you want to compress a file, back in the DOS days there was - remember the company Stacker?

Leo: Oh, yeah, Stacker.

Steve: Yeah, Stacker was doing partition compression. And they had LZS was their technique. So obviously data compression has been around for a long time and has been interesting to people. And, I mean, the audio that we're doing, now, we're doing, when we compress with MP3, we're not doing what's called "lossless" compression. Similarly, when you compress an image with JPEG, both MP3 audio compression and JPEG image compression are lossy compression, meaning that when you decompress it, you do not get back exactly what you put in. You get back something that's good enough. The picture looks good enough, not too blurry. The audio sounds to your ear indistinguishably different from what went in, yet it was much smaller while it was compressed.

So there's two classes of compression, lossy compression and lossless compression. With lossless compression you get back precisely what you put in. And that's the type of compression we're talking about. That's what LZ compression does, and many other types of compression that led up to it.

Leo: It makes sense. It makes sense because an analog thing like sound or a picture or video, you can degrade them a little bit. But you can't degrade binary code. You can't take your word processing document and take out the E's. It's not going to work.

Steve: Right.

Leo: So there's some things you have to compress losslessly.

Steve: Yes. And we've also talked about, in the context of security, we've talked about the need to compress something before you encrypt it. Because, as we know, good encryption turns regular plaintext into pseudorandom noise. Well...

Leo: Right. Which is not very well compressed.

Steve: Well, actually pseudorandom noise, one of the really interesting tests for the quality of pseudorandom data, that is to say, how random is it, is how compressible it is. Because truly random data won't compress at all. You can't compress it. You can't find anything meaningfully redundant in the data. And that's the key. In every case the lossless compression where we're talking about getting back exactly what you originally gave the compressor, it involves redundancy. It involves something that is repeating.

And so, for example, the simplest, most sort of obvious compression is known as, it's very simple, it's known as run-length compression. So that was, for example, what faxes used in the early days. If you were feeding a fax in, and there was a line of black that went all the way across the page, instead of sending individual bytes of black code, for example, the sender would say, wait a minute. It would sort of - it would store up what was coming so that it could look ahead in its buffer before it sent it out. And it would see a whole scan line across the fax of black pixels. And so instead it would have a code that said I'm going to send you a count instead of the actual data. And then it would send a count, and then the data that's to be repeated that many times.

Well, so that's just like three symbols - an escape character, a count, and then what it is you want to send, instead of, you know, who knows how many hundred individual pixels across the page. So, and then the same is true with white space. Faxes, for example, have lots of white space. So the scanner in the fax would scan ahead, be able to look sort of into the future to see how much, you know, how many scan lines, how many individual pixels of white it's going to encounter before the first pixel of black. And it would just - it would compress that by saying, okay, here comes 3,000 white pixels. So there you go. So it would send that across the line.

At the other end, the receiver would be receiving this and see the escape character, meaning, oh, instead of actual data, I'm about to get a count and then the data. And it would say, oh, here comes 3,000 whites. Well, it would expand that as the fax is coming out of the receiving end, back into a whole bunch of white space before the next line of text starts. So run-length compression is very effective in some contexts, except it requires obviously lots of the same thing in a single run. And if you don't have that, for example, if you just had the word "the space the space the space the space," well, you're got t-h-e space t-h-e space t-h-e. So no character is repeated. Yet it's obvious to us looking at it that there's a lot of redundancy there.

So the next approach that was taken, sort of in sort of fundamental compression technology, was developed by an MIT student back in 1952, a guy named David Huffman. What he recognized was that different - that one way to represent data in a smaller space was to represent those things that occur most often with a shorter code, and the things that occur less often with a longer code. Well, my favorite example of this actually way predates David's work. What he did, what Huffman did, was come up with a mathematically precise, scientific way of taking any alphabet with known frequency characteristics and developing an optimum encoding for it. But somebody back in the 1840s had a similar problem. His name was Samuel Morse.

Leo: Oh, Mr. Dit Dit Dit Dot Dot Dot Dash Dash Dash.

Steve: Exactly. The Morse code was designed - so Samuel Morse comes up with this way of having a telegraph key and a wire heading out West somewhere, and a little clanker at the other end, an electromagnet with an armature that would - and all he could do was press the key and go clankety-clank-clank at the other end. So he says, okay, I've got that now. How do I send English through this thing? And what he realized was he needs to encode English in an efficient way. So he decides, okay, I can send short events and long events, which are called dots and dashes. And so I'm going to have to do groups of those to represent characters, and he's not going to have highly trained people at each end, so it's got to be basically a system where someone can look at a page, it was called a "telegram" I guess back then, and by just sight-reading it, turn this into some impulses which go over the wire that somebody at the other end can be transcribing, very much like Elaine does, what comes out the other end.

So he said, okay, well, I'd like the characters in English that occur the most often, they should be the shortest. So not surprisingly, "E" is a dot. And "T," which actually occurs more often in English than most people would think, is a dash. Just, you know, "E" is one dot; "T" is one dash. "A," that occurs often, but not as often as "E," it's dot-dash.

Leo: Do you know how he figured out the frequencies of English letters?

Steve: No.

Leo: Kind of an interesting footnote to this.

Steve: Really, how?

Leo: Well, if you think about it, in those days, newspapers were typeset. And they made individual characters, cold type characters, for each letter in the newspaper.

Steve: Nice.

Leo: So they analyzed the bins. They actually got it a little bit wrong.

Steve: See how many E's you had to have in order to typeset the typical page.

Leo: Exactly. Isn't that clever?

Steve: Very nice. Very nice. So "A" is dot-dash. "N" is dash-dot. "I" is dot-dot. "M" is dash-dash. And to give you an example, the other end of the spectrum, "Q" is dash-dash-dot-dash. So a long pattern for the things that don't occur very often, and a shorter pattern for the things that do occur the most often.

So what Huffman came up with was a way that you could take a block of text and count the number of each of the different characters and build an efficient, like, the most efficient, the provably most efficient encoding such that those characters that occur the most often will have the shortest representation.

And here we're talking bits. So, like, "E" might be one zero, and "T" might be one one. And then the next longest one might be zero one zero and so forth. So the idea being that you break this byte orientation. Notice that normal ASCII, for example, where we use an alphabet of just typically alphabetic characters and numbers, we're storing that in a byte because it's convenient to do with a computer that moves things around in eight-bit lumps, which we've been using for, you know, decades now. But that's inherently inefficient because a byte gives us, as we know, 256 possible combinations. Yet if we just have a text document, we're only using typically the alphabet is 26 times two for upper and lowercase, plus numbers and symbols, many, many fewer than 256. So bytes, eight-bit characters, are an inefficient way of representing text.

What simple Huffman coding does, when it's applied, is to essentially re-encode this fixed-length eight bits per character in a variable length token, much as Morse code uses variable length dots and dash strings in order to represent a message, taking advantage of the fact that the shorter ones represent symbols that occur the most often. So Huffman coding was another step forward.

And then time went by. People were using computers. And compression, the need to compress things efficiently was continuing, it continued to be important, certainly as it is for us today. We're all using ZIP and Gzip and GIF images and PNG images, all which are lossless compression based. So the next innovation was the move towards dictionaries. The idea was that, if you had a dictionary, and each end had this dictionary, instead of sending the individual symbols through or storing the individual symbols, you could instead store pointers into the dictionary. So if you had, for example, a dictionary of English, and it made sense based on what it was that you were trying to compress, you could, instead of sending the individual symbols, you could just send the location in the dictionary of the word that you wanted.

So now a pointer represents a whole word instead of all the individual characters being sent. And if you think about it, I mean, even though there's lots of words, as soon as you have four characters, well, if the characters started off being eight-bit bytes, you've got four of those, so that's 32 bits that you've consumed using four byte-size characters. Well, we know that 32 bits gives us four billion possible combinations, which means that with those same four characters, those four bytes, you could point to any of four billion words. That's a lot of words. And you could also, of course, always have, like, some extra code that's reserved, saying, okay, this isn't in the dictionary, so we're going to give this to you a character at a time for those exceptions.

And so that was a nice approach. That was an effective approach for some classes of use. The problem is that you would have to have dictionaries that matched. You'd have to - the decompressor or the person at the receiving end would have to have a dictionary. Otherwise all they've got is gibberish. So there developed this notion of a dynamic dictionary where you would scan through the content that you wanted to compress or that you wanted to send, and on the fly you would build a dictionary. That's sort of what Huffman was doing with his symbols. There he's encoding - he's looking at the frequency distribution of symbols and encoding each symbol in the minimum space possible.

In dictionary compression you're taking larger chunks, larger pieces of source, like words, for example, using spaces as the delimiting boundaries, and you're saying, okay, the word "the" occurs this many times. The word "apostrophe" happened once. And so you preprocess the content that you want to compress or send, look at the whole thing, perform an analysis, and create a dictionary just for that single use, that is, the dictionary would be optimal for this particular content that you want to send.

The problem is that, while that's a nice approach, it means that you have to, in order for the receiver to understand what you send, that is, the pointers into this custom dictionary, you've got to send the dictionary. So to transmit this message you send the dictionary, then all the pointers. Or, if you were compressing something statically, the whole front, the whole, like, the header of what you were compressing would just be the dictionary. And then what would follow would be pointers back into the dictionary.

So that was, you know, another step forward. The brilliance of what Lempel & Ziv came up with in 1977 won them a patent and, I mean, as we said before, this is an approach which has been used ever since in variations. So here's what they realized. They came up with a way of either statically compressing, or they actually described it in terms of a transmission channel. And I've used that example a couple times here with Morse and the telegraph, or with a fax machine where you've got a sender and a receiver, and some transmission channel, and you're wanting to minimize the bandwidth you need, minimize the number of characters you send, which is to say you want to compress what you're sending through the channel so that it can be decompressed at the other end to reconstitute the original text.

Now, really, compressing a file is the same way, where the channel is just your hard drive. So you compress it to this static file which at some point later you're going to decompress. So the idea of compressing a file statically and sending it through a channel, they're equivalent for our purposes. So what Lempel & Ziv realized was there was a way that they could sort of use this concept of a dynamic dictionary, and at the same time avoid having to ever send it. Which is, like, what? How can you possibly avoid sending a dynamic dictionary, if that's what you're going to use?

Well, they came up with a way of incrementally constructing the dictionary on the fly. And the idea is that both the sender and the receiver start with an empty dictionary. We can think of it as a buffer or a dictionary which is empty. So the sender looks to see whether the character they want to send is in the dictionary. Well, since it's empty, it's not. So they send the character. Then, and the receiver...

Leo: Oh, I get what's going on.

Steve: Uh-huh.

Leo: They're going to build it.

Steve: They build it. So they put the character that's not there in the dictionary, and they send it. The receiver sees that they've received a character, so they put it in the dictionary. So then they come to the next character. Is this in the dictionary? If not, stick it in the dictionary and send it. And the receiver receives the character, puts it in the dictionary. So what you're going to do is, you end up essentially building a linear buffer of everything you've sent before, until at some point, say that like you've sent the word "the" in the past, in the recent past, and you have the ability also to look ahead. So you get a "T," and you're able to look, because you know what it is you're sending, you see a "T," followed by an "H" and an "E" and a space. And looking in the dictionary you realize, hey, we sent that before. We've already sent T-H-E space. And so you match as much of what you're about to be sending as what you've already sent to find the longest string that exists in the dictionary, and instead you send a pointer to it. So you come up with a compact way of saying, here in the dictionary for this many characters is what I mean. And that you send...

Leo: Clever.

Steve: ...instead of the T-H-E space. The receiver, because they've been essentially building a synchronized dictionary - and that's what's so cool is that, by playing by the same rules, at the other end of this channel, whether it's a real-time communications channel - and, by the way, like the V.42 modem spec used Lempel-Ziv compression. That was how we had compression in the modems we were using before we finally stopped using modems. Of course people still do today. And so you've got it, literally are doing this on the fly. And so whether you do it in real-time in a communications channel or storing a fie, when you use PKZIP, when you use ZIP, or Deflate in UNIX, Deflate is the same thing, it is this Lempel-Ziv technology where the receiver then receives this little token which says look here in your buffer for these many characters and expand that.

And so that's the way this works. The beauty of this also is that the buffer is of a fixed length. So at some point you start losing the stuff that's really old. And it's interesting how well this works because, if you think about writing, like, a long document in English, the subject of what you're writing about sort of - there's like a locality. You have a couple paragraphs talking about this. Then you start talking about that. And you're talking about something else. And so there's local context which is sort of relevant to the recency of the text that's just come before.

So this Lempel-Ziv approach of having sort of a sliding buffer of the past, the older stuff that may no longer, like there's less chance of redundancy, it sort of leaves the end of the buffer, and new stuff is coming in on the front that you've more recently sent. So the chances of talking about what you're talking about redundantly is much higher than talking about something that, you know, you were referring to a few pages ago. So the fact that that sort of left the history buffer, sort of it dynamically adjusts in a really elegant way.

And so that's the fundamental concept, which is just - it's just beautiful because the sender and the receiver, or the compressor and the decompressor, they build, they sort of dynamically build this dynamic dictionary. And the more redundancy you have, the greater chance of finding a match that's in the dictionary, so you can just send pointers to the other guy's dictionary, which you know is the same as yours because they've been building it as you have. It's just beautiful.

Leo: Yeah. Very slick. Very elegant.

Steve: And then all these variations. They published a paper in '77, their first approach. They called it "A Universal Algorithm for Sequential Data Compression." And a year later - and that was called LZ77, which is the date of their paper. Then LZ78 a year later. They did another paper called "Compression of Individual Sequences via Variable Rate Coding." And so they began to get fancier with the way you encode these positions and lengths in the dictionary.

And essentially everybody who's come afterwards has come up with various twists on that. You know, this LZW stood for Lempel-Ziv-Welch. A guy named Terry Welch in '84, so, what, six years later, produced a paper, a technique for a high-performance data compression, where he referred to the Lempel-Ziv approach, but he said I've come up with a different way. I'm going to - the Lempel-Ziv approach, you still needed to send characters across the line because you started with empty dictionaries.

So Welch said, wait a minute, let's preload the dictionaries with the whole set of symbols so we'll never have to send a character. We can always only send pointers. Because we know if we put the whole character set in each dictionary, the sender and the receiver, to start with, then we know that all the characters we could ever encounter will be there somewhere. And so it's that kind of variations on what Lempel & Ziv did that people have come up with. But fundamentally the concept was something that would dynamically adapt where the sender and receiver or the compressor and decompressor were able to follow rules such that the symbols, the pointers that they were sending to refer to entries in the dictionary would always be synchronized, even though the dictionary was being built on the fly.

Leo: Neato.

Steve: And of course it's also been a patent mess.

Leo: Yeah. Let's talk about that. So did Unisys own the patent? Did they work for Unisys?

Steve: I think they were with Sperry at some point.

Leo: Sperry Univac, which became Unisys.

Steve: Exactly. And so they licensed this to Sperry that became Unisys. And without really knowing any better, CompuServe, remember the old bulletin - the big commercial bulletin board system, CompuServe...

Leo: Are they gone? I guess they are.

Steve: Oh, yeah.

Leo: H&R Block owns them.

Steve: Right, right.

Leo: I guess they're gone.

Steve: Which was strange. It was like, okay, well...

Leo: Well, you know what it was, it was a timeshare that had extra time. And so they said, well, let's try letting other people use it, pay to use it.

Steve: And I think in fact - I think they were running on CDC Systems because...

Leo: Oh, wow. I think you're right.

Steve: ...the userID that CompuServe gave people...

Leo: 75106,3135.

Steve: Exactly.

Leo: I happen to remember that.

Steve: I think those were CDC usernames...

Leo: Oh, that explains it.

Steve: Like, log-on approach.

Leo: How ridiculous was that?

Steve: And that's what was, like, surfaced as your userID. Well, they did the GIF image format, or GIF, depending upon who you talk to, G-I-F, which was a lossless image compression. And they based it on Terry Welch's LZW compression, not knowing that it was patent encumbered. And this all got very popular. Years went by. And then after the fact, historically, Unisys said, you know, it just occurred to us we have a patent on everyone's GIF images, I mean, on the technology to compress and decompress. And the way patents work is, even if you didn't write the software, if you've got the ability to display a GIF image, you've got the algorithm in your machine. And every time you display a GIF image, you're violating the patent unless you have a license to do that. And so...

Leo: There was a complete freakout on the Internet over this.

Steve: Oh, people, oh, I mean, there was this righteous indignation, and people were - they were talking about burning all GIFs and, I mean, really upset that Unisys was choosing to assert their patent rights. Although you could argue they had the full right to do so.

Leo: Sure they did. It was CompuServe's mistake.

Steve: Yes, exactly. And then of course what happened was the PNG format...

Leo: Portable Network Graphics, I think.

Steve: Yes, exactly. And it was deliberately not patent encumbered. They used a different variation on compression that didn't stomp on the Welch patent, the LZW patent. Probably by that time Lempel & Ziv's had expired. Because I remember that their patent was expiring a lot sooner. And even LZW did expire in the middle of June. In 2003 June LZW became - went into the public domain. Because that's one of the nice things about patents is they're limited to 17 years. And after 17 years there's been full documentation of the technology published in the patent, which then becomes completely free for anyone to use.

Leo: Really interesting stuff. I love the algorithm stuff. And of course this time it ties very closely into computer history.

Steve: Yeah.

Leo: I mean, it really - this one made a huge difference, and everybody knew about it. Steve, you're the best. If you want to get more Steve, there's lots of it at GRC.com. His software, of course, SpinRite, the world's best hard drive maintenance and recovery utility. And he also publishes a bunch of free software you can get at GRC.com. And also services like ShieldsUP! where you can test your router. You'll find 16KB versions of our show there, as well as complete text transcripts, searchable text transcripts at GRC.com. And next week's a Q&A segment, so you might want to get to GRC.com/feedback if you've got any questions or comments or suggestions, and we incorporate those into the show, every other show.

Steve, I will see you next week. We'll be back live recording the show on Wednesdays: 11:00 a.m. Pacific, 2:00 p.m. Eastern time, 1800 UTC. So you can watch at live.twit.tv, if you'd like to see us do it live. But you can always listen. These shows are available on iTunes, on the Zune Marketplace, everywhere podcasts are available, absolutely free.

Steve: And we will have jumped from July 1st, when we're recording this, Leo, all the way to the 22nd, when you're back from your trip. And we'll pick up from there.

Leo: See you in three weeks.

Steve: Thanks.

Leo: Or next week [laughter].

Steve: Exactly.

Leo: Depending on how you're enjoying this show. Thanks, everybody. We'll see you next time on Security Now!.

Steve: Bye bye.