Buffer Overruns

Description: In one of their more "aggressively technical" episodes, Steve and Leo discuss the pernicious nature of software security bugs from the programmer's perspective. They explain how "the system stack" functions, then provide a detailed look at exactly how a small programming mistake can allow executable code to be remotely injected into a computer system despite the best intentions of security-conscious programmers.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-039.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-039-lq.mp3

Leo Laporte: Bandwidth for Security Now! is provided by AOL Radio at AOL.com/podcasting.

This is Security Now! with Steve Gibson, Episode 39 for May 11, 2006: Buffer Overruns. Security Now! is brought to you by Astaro, makers of the Astaro Security Gateway, on the web at www.astaro.com.

Steve Gibson is on the line, and we are ready to talk about one of the number one, in fact I think the prime cause of security flaws in software today. Hello, Steve.

Steve Gibson: Hey, Leo. Great to be with you.

Leo: It's great to talk to you. Things are going very well. Summer has come, and the sun is shining, and I'm moving offices. And if you only saw how everything is teetering on chairs and tabletops. I've kludged together a little recording setup so we can get this done, and then...

Steve: Well, we're on Episode 39.

Leo: Can you believe it?

Steve: And 52 is within sight, you know?

Leo: That's amazing.

Steve: Our first full year of Security Now!, yeah.

Leo: Are you running out of topics?

Steve: No. Actually, the user feedback is fantastic because it keeps me thinking about things. And if I don't take the topic directly, it gives me an idea for something else. So I've got an outline that I maintain. Like whenever something occurs to me, I write it down because I don't want it to get away from me.

Leo: Great.

Steve: And I think we've got lots of things to talk about. And, you know, from the feedback I'm seeing, people are really enjoying this, Leo. So I'm so glad we're doing it.

Leo: Great news. Well, today we talk about buffer – you can either call them "overflows" or "overruns."

Steve: Yeah, I guess "overrun" is technically a little more accurate. As you mentioned, this buffer overrun is this pernicious problem that is the way so many of contemporary security flaws are exploited, essentially. And this is going to be one of our heavier-duty episodes. It's one of those, okay, now, focus.

Leo: Have a cup of coffee now.

Steve: But I promise, as we have achieved before, that if people listen carefully and get what I'm talking about, there's another one of those a-ha experiences here. People by the end of this are going to understand why it seems so difficult to write secure software with exactly how these sorts of exploits occur. So I'm really excited about this No. 39.

Leo: So what is a buffer overrun?

Steve: Well, we need to start and lay a little bit of foundation here. One of the things that is happening is programming in general is becoming incredibly complex. And programming is moving towards sort of a component model where programmers are no longer writing everything themselves from scratch. They're inherently, in order to develop the kinds of programs that are interesting to people, they have to use other people's work. You know, it's like standing on the shoulders of giants and reaching higher than you could by yourself. The problem is, those giants may not have been concerned about security. So more and more, as we move forward, people are using other people's code, code written by, you know, like whether it's libraries or it's an increasingly all-encompassing operating system underneath you, I mean, we all know just from our own experience that, for example, XP is huge. Even Linux is growing. I think it's grown four times larger in the last five years, that is, Linux has. I mean, and the reason is all these new services and features and facilities are being added that programs running on top of them can use. And then of course in the Windows world we have this whole new .NET thing that has come along recently. And so there's a whole 'nother programming environment.

And so essentially what happens is a programmer ends up relying upon the function of many

subsystems that they themselves did not write. And so they're having to assume and presume – I mean, they have to – that those things are working correctly. So in many cases it's not even their fault, although their application, which relies on these subsystems, ends up bearing the brunt of responsibility because, you know, it was their application that had the problem, even though the actual flaw may have been sort of somewhere else or in their communication with these other subsystems. So this component programming model that we see more and more often is part of the problem.

You know, my programs, the stuff I write, are pretty simple. And I happen to be sort of a dinosaur, as we know, writing these things in Assembly language. To a much lesser degree I depend upon other things because I'm writing simple programs, and I'm sort of staying away from, you know, these really huge projects which are really no longer feasible for one person. We're talking in modern corporations producing programs they've got teams of programmers. And there's another problem is, when it's just me writing a program, I know all aspects of what I'm writing. I've got it outlined, I've got it in my head, you know, I wrote it all. So there's no communication problem between me and myself. But in team efforts you do have sort of, you know, this person will be writing this chunk of the system, and somebody else will be writing another chunk which is very different. And these separately written chunks have to interact. Well, they'll produce specifications typically for the way their parts fit together. But mistakes in that communication, it's like, oh, I thought you were going to this. And the problem, of course, is when these mistakes are not found, there's presumptions that people made and, again, these boundaries between separate pieces of work, whether it's a commercial component library or it's something that somebody else in your own organization working on the same project wrote. Any kind of misunderstandings can cause sort of a mismatch.

And what we're going to look at here in detail is exactly, I mean, really exactly how that happens. But so it really does – it does sort of blur the lines of responsibility. And it explains why, even with the best of intentions, I mean, even people now focused on security, if other portions of the system were written without a focus on security, which really does require a special sort of mindset, or if they were written before security was an issue and they're still being used, this is a problem.

So, I mean, a perfect example is, you know, at the beginning of the year we all tumbled around on that Windows Metafile issue. And assuming, you know, Microsoft's stance, they understood now that what they were doing before should absolutely no longer have still been done, but it was still being done. So there was old code that had been moved along from Windows 98 and NT up into modern times, where security is a much bigger issue today than it was before. And there was a gotcha, you know, in the process.

The other thing that goes on with programming, and I want to discuss this just before I get into the details of how the stack and buffers and things work, is the programmer's mindset. I know as a programmer that what I'm trying to do is get my code to work. That's a completely different mindset than somebody who is attacking my code, trying to find a way for it not to work. And I know, for example, you know, when I'm writing text – I wrote, for example, the Tech Talk column every week for eight years for InfoWorld magazine, years and years ago. I would write a column, and I would hand it to a couple of friends of mine, or actually they were employees, and have them proof it. The point is, I can't proof my own column because I know what I meant, and my eyes won't see the words that are wrong. You know, they just won't. I can read it three times really carefully and not see the mistake. Someone reading it who doesn't have sort of an emotional buy-in will read it and say, oh, Steve, you meant this word. And I look at it, I go, oh, how did I not see it?

Leo: Right, right.

Steve: But it's ego buy-in. I want it to be accurate. I know what I meant. And so my brain just sees it as I meant, not as I wrote. Well, exactly the same thing happens with code, maybe even

to a greater degree, due to the nature of how complex the practice is. And of course it's the complexity that is what intrigues programmers. It's why many, so many of us really enjoy programming is it's an intellectually engaging thing. But what I find when I'm debugging a program that isn't working is I can stare at the code, it looks just fine to me. I mean, no matter how much I look at it, it looks fine. So it's when a debugger – you single-step through the code, and you see it do the wrong thing. I mean, you stare at it, and you go, oh. I mean, suddenly, I mean, you have to have your face rubbed in the problem before you get it. There's that level of sort of like assumption of correctness. It's an interesting experience for a programmer to have where, when the debugger just says, look, moron, this is wrong, and it's like, oh, why, you know, how did it write it that way? Why didn't I see it that way? It just – it's the kind of thing that happens all the time.

So my point is that anyone writing code is trying to get it to work. In most situations they're in a team, they're under deadline pressure, these programs always take longer to write than anyone expects, just because that's the nature of the beast. People are putting in long hours, and they're desperate to check that code in and say, okay, I'm done with my part, or this is now working. And so everything about that process is getting it to work as opposed to looking at how to break it, which is a fundamentally different, I mean, radically different way of looking at the same code. And that, of course, is the perspective of the malicious hacker who is trying to find a way into a system, trying to find a way to break code, is the hacker is looking at it not thinking how wonderful it is that it works. They're specifically looking for ways to get it not to work, ways where assumptions which the programmer had, or the team, or the programmer communicating to something they didn't write, a third-party object or module or component, where those assumptions have broken down. And in those little interstices you can get a foothold and cause security problems. That's literally what it's about.

[Phone ringing]

> **Leo:** That's a programmer calling, saying thank you for making all these excuses for me, Steve. I can tell you're a programmer. But really, it also is a mistake, and a fairly serious mistake. And sometimes I wonder why they're still making these mistakes.

Steve: Well, it's funny because Steve Ballmer, apparently, who's now the president of Microsoft...

> **Leo:** Oh, makes him crazy.

Steve: Oh, it does. And there was one famous outburst, I don't remember exactly when it was, I think it was after XP – remember Microsoft claimed that XP was going to be the most secure version of Windows that had ever been created. And I objected as a security person to that statement because you can't claim that something is going to be more secure. History has to judge that retrospectively and decide whether or not it was true. And of course until Service Pack 2, which really, really made some major changes to XP, Windows XP was the least secure Windows that had ever been created. I mean, all kinds of problems arose from all the new code that had been added to XP. I mean, it's inevitable. So anyway, so apparently Steve Ballmer once went in to a group of programmers and screamed out loud, you know, "Why can't we fix these buffer overruns?"

And so let's talk about exactly what this is, how it's possible for hackers outside of a system to basically inject their code into the system. To understand this it's necessary, I mean, at the level of detail that I think people will really be fascinated by, it's necessary to understand something known as "the stack." You know, computers have memory. And it's possible to ask the computer to give you a block of memory, sort of borrow it from the system for your own uses, for whatever purpose, like to store data in, to accept input from a user, to assemble

something that you're going to send out, for whatever reason, a so-called buffer of memory. And then when you're done with it you free it, or release it back to the system.

Well, there's an architecture that has developed in our contemporary computers which is known as "the stack." And what the stack is, I don't find it actually useful to use sort of the stack of plates in the cafeteria model, the idea being that you put plates in this little spring-loaded thing, and the plates go down, and then you take them off, and they pop back up. I mean, that's sort of the analogy that is used sort of in a crude way, but it doesn't give us what we need in order to understand what happens with buffer overruns. So...

Leo: Well, it is important, though, just so you understand that it's a last-in, first-out stack. I mean, that's how data goes in and goes out.

Steve: Kinda.

Leo: It's not in order.

Steve: But actually, I mean – well, okay. The way it really works is the reason there's really a problem.

Leo: Oh, I see, all right, okay, I know where you're going. All right.

Steve: So what the stack is, it's a large region of sort of uncommitted memory. So think of when a program is started up, the system, the operating system, gives this running program its own stack, which is – and so just call it a "stack" as sort of an abstract term. What it actually is is a long buffer, actually almost a bottomless buffer that you don't need to worry about running out of. If people want to picture this, think of like maybe, I don't know, an unwound, long roll of toilet paper, or just a really long banner, for example, oriented vertically from the top down. And for reasons that we'll see in a second, memory is allocated in this stack from the top downwards. So when the stack is empty, that is, it doesn't contain anything, there's a pointer to the very top of the stack. It's at the top because there's nothing above it. And if the program wants to allocate some memory from the stack, it's very simply done. The "stack pointer," as it's called, is moved downwards by the amount of memory that the program wants to allocate for its use.

Leo: The pointer is pointing to the next available space.

Steve: Right. And so if the pointer is moved downwards by a certain amount, then what the pointer is now pointing at is the beginning of that amount of space that was just allocated on the stack. So if you visualize this pointer being moved down like by a thousand bytes, then from that point upwards there is now a thousand bytes, the beginning of which is pointed to by this pointer. So the way this system works is many different things use this stack, sort of all at the same time. It's sort of a general purpose scratchpad. So the program might move it down to allocate some space, then move it down a different amount to allocate some more, and down a little more to allocate, you know, like three different regions of buffer. And then, as these things are no longer needed, the pointer is moved back upwards, back up toward the very top, where the stack would then again be empty. So it's a very convenient system from a programming standpoint, and it's inexpensive in terms of the technology being used. That is to say, it's virtually instantaneous to move – just to change the value of this pointer by a certain amount. And because it's so efficient, it's the system which has come into virtually universal

practice in modern computers.

Leo: It's used very heavily in subroutine calls. You save a context, you jump into the subroutine, and then you could pop the context back out, and...

Steve: Well, in fact, that's a perfect segue to explaining how a subroutine is essentially called or invoked, which is where this becomes a critical problem.

Leo: Now, you manage it by hand because you're an Assembly language programmer. But compilers do this all automatically. The C programmer doesn't see the stack particularly.

Steve: Right. They're not at all aware of it. But it's still happening in the background and is the source of vulnerability. What happens when a program is running along and wants to call a subroutine, a subroutine sort of just being sort of a chunk of code which has been written and is standing by to perform a certain function. And that code might be called by many different other locations in the program. For example, say that it was a little piece of code to turn everything into uppercase. And you might want to do that if you're searching for something, and it's much faster to search all uppercase or all of a known case than it is to do a mixed case search. So you might want to just turn a buffer of text into all uppercase. And many places in your program you would have the occasion to do that. So you only write this code to do the uppercase conversion once, and then you call that function, that subroutine, whenever you want that to be done for you. So that's sort of an example of the power of a subroutine or a function call is it provides resources that the program can use wherever it is.

Well, when you call this code to execute the function, the code somehow needs to return control to you. It needs to come back to where you called it from. And since you might be calling it from many different places in the program, that return to you can't be sort of fixed. It can't be hard coded. It has to be dynamic. So the way the system handles a so-called subroutine call, or a function call, is when you call the subroutine, the address of the next instruction below that call is put on the stack. It's pushed on the stack, as we say, meaning that that stack pointer is moved down just by 4 bytes, by 32 bits. And at that location the address of the next instruction below where the call was made is stored.

Leo: It's called the "return address."

Steve: The return address, exactly. And so then the computer jumps to the location you have called this subroutine and begins to execute the code. Now, the subroutine will use the same stack you're using. That is, say that the subroutine for its work, for example, it's going to take the text buffer you give it and create a new buffer that's going to be uppercased, for example. So it could take the stack pointer and move it down by however much memory it needs and then use that region of the stack from where the stack pointer now is upwards, as long as it wants. When it's done using it, it puts the stack pointer back where it was. And in order for the subroutine to return to you, it literally executes an instruction called "return." And that instruction pops that return address off the stack, meaning it moves the pointer back up. Now it has those 32 bits that were saved there, which is the instruction address of the location below where this subroutine was called from. So the processor jumps from that address and continues executing sort of seamlessly.

So in the main flow of the code you call the subroutine to, for example, uppercase a buffer. It does whatever work it needs, maybe borrowing some of the memory from the program's stack as a scratchpad for doing whatever it's doing. It then releases that memory, which puts the stack pointer back where it was when it first got control. Then the subroutine returns, using the

value stored on the stack, returns to where it was called from. So that's the whole mechanism.

Now, how does this break? How does it fail? A really interesting example is where some programmers are paying attention to the sign of their data and others are not.

Leo: Positive or negative, whether the number is positive or negative.

Steve: Exactly. In binary – let's talk a little bit about how sign is handled because that will factor into this. And then people are going to have one of those a-ha experiences here pretty quickly. In binary numbering, all bits being 0 means 0. And as you begin turn the bits on in the binary sequence, by sort of universal agreement that represents increasing values. In 32 bits – which is what most of our processors still are, those who haven't moved up to 64 bits, but we'll take the 32-bit case in this instance – in 32 bits you have from basically 0 to a value a little bigger than 4 billion. And then you get to all ones. And then if you add one more to that, sort of it overflows, or wraps around back to 0. Well, that's a so-called "unsigned value." And it's regarded, for example, in C it's called a UINT, an unsigned integer.

Leo: And it's always positive. It can't be negative.

Steve: Well, exactly, because every bit combination between 0 and that maximum 4 billion, that represents a positive value from 0 up to that 4 billion quantity. But in many instances in programming it is useful to have a signed value, that is, where you do have negative – you're able to represent negative quantities as well as positive ones. So the way this is handled in the actual storage in the computer is that this 4 billion space that 32 bits gives us is chopped in half. Half of them are the positive, from 0 to 2 billion, and the second half are negative. And it actually goes from -2 billion back down towards 0. So, for example, if we count up towards – with a signed counter we count up towards larger values, we'll get to something above 2 billion. And if we go one more, what happens is – and actually that maximum value has the top bit being 0 and all the rest are ones. We then add one more to that, which all these ones overflow, making the top bit a 1. And actually that's called the "sign bit" in a signed number. And so when that top bit is a 1, that means that the rest of the bits represent a negative quantity.

So, okay. So imagine that some code, a program, wants to accept some data from another program, from a user, from a script running on a web page, from wherever. It wants to make sure, though, that the data that it's receiving will fit within the buffer that it's allocated. And so say the programmer says, okay – say that it's a URL that is going to be accepted at this part of the code. The programmer thinks, okay, how long can a URL probably be? Well, maybe a thousand characters. You know, that's a really long URL, even by today's standards. So the programmer allocates a thousand-character buffer by moving the stack pointer down, as we've seen, and then has access to the thousand characters from where the stack pointer is upwards. So that's sort of the programmer's scratchpad area.

Now the programmer – say that this URL has a length, it's provided with a length, and then the data. Well, the programmer checks the length that is declared to make sure that it's less than a thousand characters long to make sure that the data will fit within this buffer. And if so, then the programmer might call another subroutine to copy the provided data into the buffer. Well, if the programmer made a mistake, and just not even thinking about it was thinking that the value being provided was a signed value, which really doesn't make any sense because it makes no sense to have a negative length on data. Data is always going to be an unsigned length. So, but just because the programmer's used to typing INT, INT, INT, which stands for integer, which is inherently a signed value, instead of UINT, which is unsigned integer, if the programmer declared this value, suspected – or, sorry, expected that the value were going to be signed, or just didn't think about it, then if somebody malicious declared the value to be -5, then when the programmer compares the length to the length that's being declared for this URL

to the buffer size he's allocated, 1,000, you know, is -5 less than 1,000? Well, yes, it is, if it's a signed value, because -5 is less than 1,000.

So the programmer says, okay, this URL I am about to store will fit within my thousand-byte buffer. So he hands – the programmer now calls a subroutine to copy the data from wherever it's coming into its buffer. So he provides the -5 and the address of the buffer to a subroutine. Well, now the subroutine, which somebody else has written, knows that lengths cannot be negative. I mean, that makes no sense. So in the subroutine the length is handled as it really should be, as an unsigned value. Well, we know because we just looked at how values are stored in binary, a negative value, if it's treated unsigned, is actually a very huge positive value. So the subroutine which has been asked to copy the URL sees a really large value as the amount of data it's supposed to copy. Which it says, okay, if that's what the guy who called me wants me to do, that's what I'll do. So the subroutine copies this URL that might be, for example, 4K, because the – or say that the value is shown as -4K, but the original program did the comparison, said is -4K less than 1K, the buffer size? Yup, that 's less, no problem. Well, the subroutine which is doing the copying realizes this is unsigned, I mean, treats it as an unsigned value, which means it's a really huge value. That program copies the data way more than the program that called it expected, for example, 4K worth of data, trying to fit it into this programmer's provided 1K buffer. Well...

Leo: So the problem is because they're thinking of it as a signed value, but it really is an unsigned value. It's too big for them.

Steve: Exactly. And some clever hacker somewhere realized that the code that was calling this copying subroutine was written inaccurately and that, for example, telling it it was -4K would slip by its guard. It would get under...

Leo: Actually, the way they find this out is by trial and error. They just try it until they find something that breaks, really.

Steve: Very often. So now in our model we've called a subroutine, and it has written more than the 1K that we expected. Now, remember that to create this 1K buffer we moved our stack pointer down, making a region above the stack pointer that is a thousand bytes' worth available. Well, if this more than 1K is copied, if, like, 4K were copied, it would fill up the 1K and continue on upwards, overwriting all kinds of, I mean, whatever else was on the stack above that location. Really, I mean, wiping things out. So at that point control is returned to the program that called it, which doesn't realize anything really bad has just happened. It finishes up its work and releases that 1K buffer it allocated, which moves the stack pointer back up by a thousand bytes. But now, instead of the next thing on the stack being the return instruction, remember that if this is going to then return to whoever called it, it's going to – the stack is popped, and the value on the stack is the address to be returned. Well, that's been overwritten maliciously on the stack so that the return value is no longer accurate. If a programmer who was trying to exploit this did his job correctly, what's there is instead the address of other code right there on the stack, which will then get executed.

Leo: Oh, very clever.

Steve: And that's exactly the way somebody from outside a computer system could take advantage of a little mistake, you know, just literally the letter "U" for unsigned was left off of a variable declaration in C. And since C, as you mentioned, Leo, C handles these things for programmers, when C sees that it's an INT comparison as opposed to an unsigned integer comparison, that's the way C compiles the code, which causes this negative value to slip under

the radar of a programmer who was really trying to do the right thing. They were trying to say don't copy more than a K, making sure that a URL is a K byte or shorter. But in fact, because of this missing unsignedness, a subroutine that did the copying thought it had permission to copy 4K. And then...

Leo: Sometimes it's even worse. I mean, sometimes the programmer uses an unchecked string copy without a range on it and just kind of blasts data in there by accident. So there...

Steve: Well, you're right. It's certainly the case that maybe the programmer wasn't checking it all.

Leo: Right.

Steve: As you said, that can happen. But even when you're, like, trying to do the right thing, it's possible to end up with code which is insecure in a very weird way. Because think about it. As long as you provide a positive length, it's going to work. You could try to give him 4K, a positive 4K...

Leo: And he'll reject it. He'll say...

Steve: His code would say, nope.

Leo: What do you think, I'm stupid?

Steve: Exactly. That's too big.

Leo: Right.

Steve: But because of this mistake of just sort of handing responsibility off between different parts of the system, it's possible to find these. And so, literally, code was written onto the stack, which is then executed. And at that point your computer is in the hands of a hacker. It is potentially taken over. And this is the way all those worms that we used to be having, Code Red and Nimda and all these things, where no – I mean, and those, for example, were mistakes in Microsoft's Windows server that had these kinds of vulnerabilities that allowed worms to just literally inject their own code without the system's knowledge or permission into the operating stack of the server, and then run that code and take it over.

Leo: It's pretty impressive, though, accomplishment, even on the hacker's part, because it isn't just kind of randomly sticking code in there. They've got to figure out where to put it. They've got to figure out where the return address would be and make sure it jumps into another area where it's their code. I mean, I imagine there's a lot of trial and error in there, and they're very clever at doing this.

Steve: Oh, Leo, I mean, this is advanced programming to be a hacker like this. There's no doubt about it. I mean, to do this you have to know the system at a level below – certainly

below the so-called "script kiddies," and even really below the level of most C-style programmers who are taking advantage of the convenience of the C language being an abstraction of the machine. And it's that abstraction that allows their code to be portable among different architectures, where the compiler is doing all of the detailed work. Well, hackers, in order to exploit this level of flaw, I mean, they're absolutely operating down at the machine level and, I mean, truly knowing what they're doing.

Leo: It's very impressive. And amazing that it happens so often, given all of the things that have to go wrong and all the things that have to go right to make it work.

Steve: Well, it's again, you know, harkening back to the beginning of this podcast where I really wanted to give people, I mean, not to make excuses, but honestly to explain why these problems keep happening.

Leo: Right.

Steve: It's just this is a, you know, programming is a complex task which is becoming increasingly complex. And we're relying on many more subsystems that we didn't ourselves write. So inherently you make assumptions about how something you're going to use, some subroutine library, you know, the operating system, whatever, you make assumptions about what it's going to do. And when those don't quite match up, that creates a little opportunity for error. And, you know, programs literally have to be perfect in order not to have any bugs like this. And it just – it's incredibly difficult to make them so.

Leo: It is. It's remarkable. Well, once again you've explained the inexplicable.

Steve: Well, one bit of good news I want to talk about just briefly to wrap up this topic is something which was introduced in Windows XP Service Pack 2, which is – the acronym is DEP, stands for Data Execution Prevention. And it's one of the things sort of in answer to Steve Ballmer's war cry or frustration cry about why they can't solve this problem. Because one of the things that's interesting is the stack that we've been talking about is really only for storing data. It's not for storing code.

Now, back in the dawn of Windows, literally Windows 3.1, way back, you know, the early Windows, before we had graphics accelerator chips, the very clever guys who wrote GDI, the Graphics Device Interface portion of Windows, they, in order to make the display work quickly, they would put code on the stack, that is, literally, Windows would write a program to move data quickly from one place on the screen to another, back before the hardware in the display adapter would do that for them. So there is some history of code running, I mean good code, deliberate code, running on the stack. But in general, and certainly much more so in modern times, the stack should really only contain data. It ought to be temporary variables. It ought to be subroutine return addresses, as we were talking about, or buffers that are being allocated by the program for its temporary use, which it then releases. So there really isn't a reason to allow the stack to be executable.

Well, a very cool feature in the latest processor hardware from AMD and Intel is available to essentially mark the stack as non-executable code – wait a minute, sorry, non-executable memory. So that if a buffer overrun occurred, and the processor attempted to jump into the data on the stack, they would immediately throw up a system exception and not allow that to continue, thus completely shutting down the whole buffer overrun problem.

Now, this Data Execution Prevention has been available since Service Pack 2. Only the newer

Intel and AMD hardware supports it. There's a sort of a weaker, software-only version which can be turned on, which provides some protection. The other problem is that there are some programs that won't run if their stack is locked down and not allowed to execute because of some of the tricks that the programmers or the compiler are playing. So there's a little bit of a transition phase here where – and Windows handles that because, if you've got one of these DEP-enabled processors, and you've got it turned on, you are able to make exceptions for known programs that have a problem with this.

Anybody who wants to know more about it can just Google "Data Execution Prevention," or "DEP," and learn about it. It'll take you right to Microsoft's pages where you can see how to turn this on if you haven't messed around with it. It's a really nice step forward which is going to, you know, reasonably help, you know, this whole class of security troubles. And, you know, if we'd had it years ago, things would be a lot better than they have been. But at least, you know, future hardware will support this. Future software can be expected to be compatible with it. And I'm not making excuses for programmers again. I mean, we do make mistakes. They're just so hard to prevent, especially when it's not even in our code, it's some code that we're relying on that somebody else wrote. So this is a really nice sort of prophylactic measure to protect your system from remote code injection exploits.

Leo: Yeah. Also, by the way, there are versions of this for Linux and some versions of BSD. And it's built into the hardware of the processor, although the operating system has to enable it.

Steve: Exactly.

Leo: Well, great stuff, Steve. I really appreciate your taking the time to make this clear. It's something that, you know, I've tried to talk about in the past. We've had experts on the shows trying to explain it. But you did the best. Very, very clear. Thank you.

Steve: Well, you know, as a programmer, and because I'm down at the Assembly language level, I'm dealing with the stack on an intimate basis, so...

Leo: You make your own stacks.

Steve: Exactly.

Leo: Your own stack frames. Well, hey, by the way, I don't know if you saw it in the show notes or the comments to our show notes from last episode, but there was a great – did you read – UnlivedPhalanx wrote a really nice – I guess you did because you responded to it – a really nice comment about SpinRite. He said, "I just wanted to write in here about how amazing SpinRite really is. I had a hard disk with a tax refund and a recent insurance claim go completely bonkers to the point where the moment you turned on the computer it would crash. It was no longer accessible by any other means. In under four hours SpinRite restored the drive to full working order and allowed me to save every single bit of information on the drive." Whoo. "So when you hear Leo talk about SpinRite, he isn't just plugging a friend, he's telling the honest truth." And I am. SpinRite is the single best disk recovery tool ever written.

Steve: I saw that. I mean, obviously SpinRite supports me, so I love it, and I depend upon it. But, you know, we get this email from people like the posting that you just read from last week

where they really needed their stuff saved. And I just – it just really warms my heart when...

Leo: I bet it does, yeah.

Steve: Yeah. When I'm able to help people like that.

Leo: You can see more of these testimonials at SpinRite.info. And of course you can get your copy of SpinRite – everybody should have one – from GRC.com. That's where Steve hangs his hat. You'll also find at GRC.com the show notes for this show and all the past Security Nows, including 16KB versions for the bandwidth impaired and Elaine's great transcripts. She came back from her trip safe and sound, I hope.

Steve: Yup, and she's ready to make a transcript of this one.

Leo: Transcribe some more. Welcome home, Elaine, glad you made it back. [Thanks, Leo.] And that again is GRC.com/securitynow.htm. And of course I want to thank our wonderful sponsor, the Astaro Corporation, the makers of the Astaro Security Gateway software. You can get it at Astaro.com. There's a free version for home users which is fantastic. If you've got an old machine, put it on there, and it turns your system into a firewall that is, bar none, the best. They support open source. It's open source, so it's great. They also sell hardware. I have their Astaro 120, their gateway, and it is really great. Highly recommended. Astaro.com. I said "Astaero," and they want me to say "Astaro." So I'm going to say "Astaro."

Steve: I think as long as people go there, they probably won't be complaining much.

Leo: Astaro. Hey, Steve, thank you so much. What are we going to do next week?

Steve: I haven't even looked at my notes, but I'm sure we'll have something good.

Leo: Well, you know, folks, keep posting comments and suggestions and questions for Steve because, as always, I think he's inspired by the things you ask and suggest to him. Actually next week will be Episode 40, so it'll be your questions and answers next week.

Steve: Oh, that's right. Perfect. Well, there's the answer to that question, Leo.

Leo: It was a simple one. Thank you, Steve. Have a great day, and let the brain cool down a little bit now. I think an ice bath would be a good thing. I know I need one.