DAM BUFFER OVERFLOW!

It's the single most prevalent, most damaging, and least-understood security issue threatening enterprise networks. So why, after nigh-on 20 years, is this supposedly simple programming error still such a serious problem? Chris Anley explains.

FROM THE INFAMOUS Morris worm1 in 1988, to the Zotob worm in August 2005, the buffer overflow has been *the* issue causing most of network administrators' security headaches.

Yet three key issues remain misunderstood: how buffer overflows happen, how attackers use them to break into networks, and what can be done about them from the perspective of network administrators, developers and management. Several attempts have been made to cure the problem – no remedies have proved totally effective. For simplicity, this technical discussion will be limited to the Intel x86 processors, since the mechanics of buffer overflows can vary significantly for different types of processor.

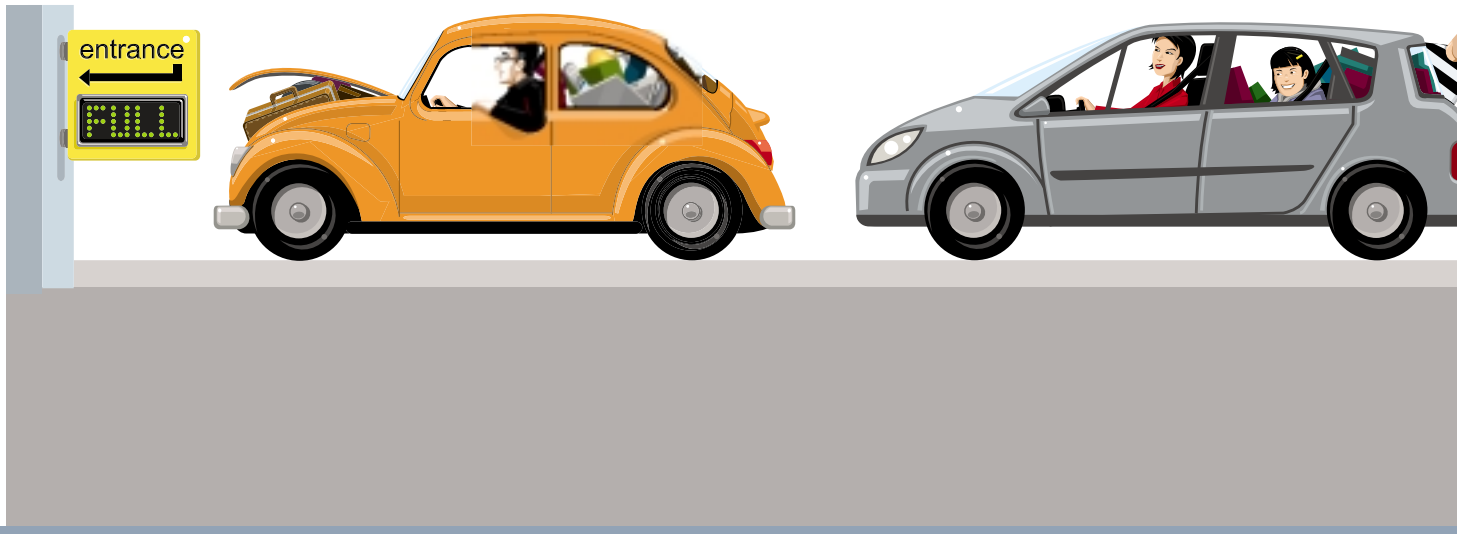
A buffer overflow occurs when data is copied into a buffer that is too small to hold it. Think of a very small program that asks you to enter your name. The programmer thinks a name will not be longer than, say, 50 characters, so they set aside 50 bytes of memory to store your name when you type it in.

All's well if you type in 'Chris' – it fits into the buffer. But what happens if you had a name that is longer than 50 characters, breaking the initial assumption made by the programmer? Or worse, what if you deliberately filled the buffer up with too many characters?

When too much data is copied into a buffer, the data after the buffer is overwritten with data specified by the attacker. Depending on what that data was, the effects can be catastrophic for the security of the program in question. The two most common effects are 'redirection of execution' and 'modification of data'.

With redirection of execution, the attacker causes the program to 'jump' into a small, machine-language program that they supply in their data, and which can essentially do anything the attacker wants.

With modification of data, the attacker modifies data adjacent to the buffer in question, causing the program to behave differently. One simple real-world example of this was a buffer overflow in the Solaris /bin/login



DEVELOPING SCAPEGOATS

It's easy, and fairly common practice, to blame software developers for these bugs, and, when software vendors are hit by a damaging buffer overflow problem, people can lose their jobs. But the reality is that buffer overflows can be hard to spot; indeed, they have affected code written by the most skilled and security-aware developers.

It makes little sense to get rid of someone who has just learnt one of the most valuable secure coding lessons there is, only to replace them with someone who is bound to be less personally aware of the danger.

The best remedies are education, good review procedures, and sound architectural controls. In the absence of these, external security code reviews – or an internal security review – may be sufficient.

Why are buffer overflows so hard to spot? Here is a run-down of some more common causes of overflows in C and C++, illustrating that it's not all about easily-spotted calls to 'strcat' or obvious mistakes.

- Copying data into a buffer without checking its length first: strcat, strcpy, memcpy, etc.
- Using sprintf to substitute a string into a buffer: sprintf(buffer, "The string is: %s", string).
- Mistakenly using the source size, rather than the destination size, in a length-limited copy: strncat(dest, src, src_size).
- Assuming a fixed minimum size for some data and subtracting that size to get to a 'payload'. This often leads to integer wrap-around, whereby subtracting from a small value results in a large value.
- Using a C++ stream input or output function (<<, >>) with a fixed size buffer as the destination.
- Reading the size of some data from user-supplied input and trusting it, thereby copying more data into a buffer than the buffer can hold.
- Reading the size of some data from user-supplied input, into a signed integer. If the integer is negative, it will pass length checks (e.g. x < 255) but then when passed to a memory copy function (strncpy, memcpy) the size is cast to an unsigned integer, and becomes very large (-1 becomes approximately 4 billion, for example).
- Mistakenly supplying the number of bytes as the buffer size to a wide-string function that expects the number of characters, or vice versa (e.g., WideCharToMultibyte, MultiByteToWideChar)
- Calling a function that returns a size if it succeeds, or a negative value if it fails, and using the return code as the offset into an array (e.g., revs, sprint)
- Forgetting to allow space for the terminating null when copying a string (mercy, stinky, sprint).

This last point is especially tricky; the number passed to the function is the number of characters to be copied. If the number of characters in the source string is equal to or greater than the size of the destination, the null terminator will not be appended to the buffer.

If you were to copy or print this string, you would see all of the data that happened to be stored after the string, until you reached a null character.

This results in a string that may actually be larger than intended, but whose length has supposedly been guaranteed to be the size of the destination buffer. Also, in some cases it is possible for an attacker to gain control of the flow of execution just by overflowing a buffer by a single byte.

illustrations: Chris Robson

SECURITY BRIEFING: HOW ATTACKERS USE BUFFER OVERFLOWS

Attackers take advantage of buffer overflows by writing an 'exploit'. This is typically a small piece of malicious code that will deliver data to the buffer in question – perhaps in a file format (say, a malformed .mp3 file), or maybe via a network protocol such as HTTP. The data the attacker sends will typically redirect the flow of execution into the data the attacker supplied – effectively, the attacker gets to upload and run a program on your computer. The mechanics of this can get complicated, but here is a brief rundown of the two most common ways.

ATTACK WAY 1: The attacker overwrites the 'saved return address' on the stack with a value those points to their data.

Whenever the processor makes a procedure call (using the 'CALL' operation) it stores its current location on the stack. When a buffer on the stack is overflowed, the attacker gets to overwrite the saved return address. When the processor executes the corresponding 'RET' instruction to return from the function, we will 'return' to the attacker's code. On Windows systems, the attacker will normally overwrite the saved return address with the location of an instruction that will 'jump' to or 'call' their data, since the location of the stack in Windows systems is less predictable than UNIX systems.

ATTACK WAY 2: The attacker overwrites heap management structures, allowing the writing of a 32-bit value to the 32-bit address of their choice.

Typically, the attacker overwrites a function pointer with the address of a buffer they have supplied. When the function is called using the over-written pointer, their code executes. The attacker gets to write the value of their choice to the location of their choice because of how heap management structures are laid out. Imagine that blocks of memory allocated on the heap are represented by a linked list, and each block of memory on the heap has a linked list item with a forward and a backward pointer; you can see how this arbitrary overwrite might happen.

When you remove a block from the list (i.e., free a block of memory), you use its forward pointer to determine the address of the next block. You then update that block's backward pointer to point at the freed block's backward pointer. As the attacker controls the freed block's forward and backward pointers, we can specify both the value we want written, and where we want to write it. This is a much-simplified explanation, but this is, in essence, how most heap-based buffer overflow exploits work.

The diagram below is a simplified representation of this; the solid areas contain data supplied by the attacker, the leftmost hatched area is the value we wish to write (which is also controlled by the attacker), and the rightmost hatched area is the location the value gets written to.

Exploits can have various effects. In worms, the effect is normally to attempt to pass the worm on to other, randomised IP addresses and perform some malicious operation on the infected host. More specific, targeted exploits might allow an attacker to run commands on the targeted machine via a command shell, or allow the upload and execution of a larger program that gives control of the targeted machine.

Once one person has bypassed a defence, all attackers can make use of that bypass

program, accessible via the Telnet daemon, whereby an attacker could log on as the user of their choosing by simply setting an environment variable ("TTYPROMPT") to a long string2. The TTYPROMPT data would overflow the buffer allocated for it, and overwrite an integer flag that determined whether or not the user's password was checked.

Exploit Prevention Mechanisms

Various companies and organisations have created mechanisms that attempt – with varying degrees of success – to prevent the execution of buffer overflow exploits. The C++ compiler in Microsoft Visual Studio now includes a mechanism that reorders variables on the stack to reduce the impact of stack overflows, so that if a text buffer overflows, it generally won't overwrite a pointer or an integer in the same stack frame.

It also places an unpredictable value (a 'cookie') on the stack to protect the saved return address and base pointer; the compiler adds code to check whether this value has been changed before executing an RET instruction. These measures all help to protect against stack overflow exploits.

The heap management functions in Windows also perform security verification before allocating and freeing memory, to make heap overflow exploits harder to write. In the Unix world, the GCC compiler has a number of similar add-ins such as the FORTIFY_SOURCE mechanism, and the IBM SSP mechanism that implements stack variable reordering and a stack 'cookie' to detect when the saved return address or frame pointer have been overwritten.

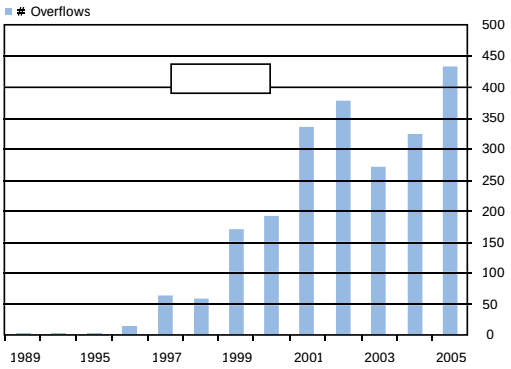
Several classes of overflow that they don't protect against. Buffer overflows where a function pointer in a structure is overwritten, for example, or references to data stored in parent stack frames.

The recruit (www.grsecurity.net) project is an extensive set of security patches applied to Linux, including the Pox Address Space Layout Randomisation mechanism, which randomises the locations of parts of memory that attackers may need to rely on in order to get their exploits to work. The concept of changing the permissions on pages of memory, to make data pages non-executable, has recently been implemented in Windows, and has been around for a while in other operating systems (the noexec_user_stack option in Solaris for example, and the Write or Execute only patches applied by PaX). Since attackers upload data which is subsequently executed, this would seem to be a bullet-proof protection against buffer overflow exploits but again, this is unfortunately not the case.

By performing something known as a 'return to libc' exploit, an attacker can execute code that already exists on the system by setting up variables on the stack and then 'returning' into a well known function such as system (that executes a command line).

Combining non-executable data areas with other defences can make it more effective. For example, Address Space Layout Randomisation is effective where an attacker only gets one attempt at exploiting an overflow before the target program crashes – but it's less effective if the target program is automatically restarted, or operates a 'pool' of worker processes, as with the Apache web server. A paper from Stanford demonstrated a working return-to-libc exploit on a machine with both the non-executable data and Address Space Layout Randomisation protection (www. stanford.edu/~blp/papers/asrandom.pdf).

The difficulty with these technologies, as with most defensive technologies, is that they are available to »



the attacker as well; a skilled attacker will study the system, create some means of bypassing it, and publish it.

Once one person has created a means of bypassing the defence, all attackers can make use of that bypass in their own exploits. That said, each new step taken towards providing protection against overflows is a positive one – the key when deploying these technologies is to understand their limitations.

Attacking the attackers

Underlying these figures is a constant battle between software vendors, who are trying to improve their development process and remove sources of buffer overflows, and an 'attacking' community comprised of security researchers, interested users, and criminals, who are attempting to expose the overflows in the products, and discover new and more effective ways of finding and exploiting overflows.

The number of buffer overflows being found is increasing, so the source of the problem is persisting; it must be faced that programs will almost certainly continue to have overflows. While exploit prevention technologies are improving, there is no current technology that offers total protection against buffer overflow exploits.

As exploit prevention mechanisms mature, attackers will be forced to take advantage of features specific to the application they're attacking in order to exploit overflows. Until now, most buffer overflow exploits have used generic mechanisms – such as overwriting the saved return address – that can be applied in almost all cases. As technologies such as Data Execution Prevention, stack cookies and Address Space Layout Randomisation become more common, attackers will be forced to move away from generic ways of writing exploits into the murkier depths of the specific behaviours of the target application. It's likely that some classes of buffer overflow will cease to be exploitable; it's unlikely that the entire problem will be solved in the immediate future.

Protect and survive

If you develop software, or are in control of a development organisation, the best thing is to educate yourself and your staff. Consult the Internet, and any (recent) security books addressing the subject in detail.

Another good practice is to carry-out code security reviews. If you have the skills within your organisation to do this, so much the better; otherwise, there are security consultancy companies out there that can help you in this regard.

The final step is to invest in some compile-time assistance. If you are using GCC, apply the FORTIFY_SOURCE patch and compile your code with it enabled. If you are using Microsoft Visual Studio, enable the /GS flag to turn on stack protection. There are a number of standard development practices that can help; for instance, always compiling at the highest warning level, and taking care to avoid dangerous string handling functions.

Code quality policies like these should be policed manually: developers are apt to devise inventive ways of avoiding the automated code quality mechanisms. This can have the unexpected effect of making serious string handling problems harder to detect, so some kind of manual enforcement is helpful.

If you are a network administrator, the challenges you face can be equally daunting. Effective patch management is important, and a good set of standard build procedures for the various operating systems, web servers, and databases you have in your network can be invaluable. If you are considering making use of an exploit prevention technology either at the Operating System level, or as a stand-alone Intrusion Prevention System, it's a good idea to be aware of exactly which classes of exploit the system will prevent.

None of these products provide a total solution. It's unwise to lower your guard by investing less in the other, more traditional security measures, such as segmented network design and secure build guidelines for the major platforms that you have to manage.

Chris Anley is a director at NGS Consulting

Left: You can see the number of buffer overflow vulnerabilities reported per year since 1989, with 434 as of October 2005. The data was extracted from the National Vulnerability Database, a project sponsored by the National Institute of Standards and Technology (NIST) in the USA. The general trend is clearly that the number of overflows reported per year is increasing, though there aren't enough data points to determine how quickly the number of overflows discovered is increasing.