Buffer overflows in a Windows environment

Buffer overflow vulnerabilities are commonly exploited by hackers to gain control of an IT system. **Parvez Anwar** and **Andreas Fuchsberger** explain what they are, how they work, and how companies can protect themselves—up to a point.

Royal Holloway University of London

SearchSecurity.co.UK

COMPUTERS GETTING HACKED, information being disclosed, corporations losing billions[1], these are some of the miseries brought by the buffer overflow vulnerability. Over the years we have seen hundreds of articles relating to buffer overflows being published making it a thoroughly researched vulnerability. Well known security researchers have published papers providing a plethora of information, but buffer overflows are still being discovered on a regular basis. The chaos and destruction a buffer overflow can cause can be disastrous if not dealt with at the earliest opportunity. This article will gives an insight into why this vulnerability exists in the first place and what can be done to improve defences against it. In the last decade we have seen various worms[2] taking advantage of buffer overflows to assist in criminal deeds. Worms are malicious programs which silently take over IT systems and carry out malicious activities. As recently as October 2008 another worm was discovered in the wild abusing a buffer overflow vulnerability. This worm not only hid itself in a way not easily visible but also downloaded various malicious files. So what can be done to stop this threat and what measures can be taken to mitigate it? As well as having layers of security providing defence in depth, it is often necessary to protect against buffer overflow attacks directly and understanding exactly what a buffer overflow is goes a long way in diagnosing and investigating a possible attack. Taking early pre-emptive measures on a likely vulnerability without relying heavily on third-party software might make all the difference in protecting an organisation's IT assets.

UNDERSTANDING BUFFER OVERFLOWS

What exactly is a buffer overflow? A buffer overflow is a type of vulnerability in software that could be exploited to gain control of an IT system. A buffer overflow occurs when too much data is inputted into a fixed amount of space set aside by the programmer, commonly called a buffer. If too much water is poured into a glass, the excess water would overflow and create a mess. If too much data is accepted, more than the allocated memory can handle, then the data would over-

flow to the adjacent memory locations corrupting memory and similarly creating a mess. But why do these buffer overflow vulnerabilities exist in the first place? This really comes down to poor programming practices by developers who produce software. Developers use various functions

When a buffer overflow takes place registers get overwritten and the program usually crashes. Crash logs can usually help identify the problem.

to write their software. Some functions are vulnerable as there are no checks in place as to how much data they can handle. C and C++ are two of the most popular languages affected by buffer overflows. Simply using safe functions, which define boundaries of inputted data, can eliminate this vulnerability. Languages such as Java and C# are known as safe languages as they have checks in place which prevent overflows.

To appreciate how buffer overflow vulnerabilities can be exploited we need to have an understanding of processor registers and shellcode.

PROCESSOR REGISTERS: Registers are a type of memory contained in the processor and used by the processor. These are different from main memory where programs are loaded and data is stored. Registers are small portions of memory used for very fast processing. There are a number of registers, each with its own responsibilities. When a buffer overflow takes place these registers get overwritten and the program usually crashes. Observing these register values after a crash will give us some indication as to what had occurred. Simply looking at crash logs usually contains the information we need to diagnose a problem. For a security researcher, observing a crash is a good thing as it gives the researcher a possibility of discovering a new vulnerability.

SHELLCODE: For a buffer overflow vulnerability to be exploited we need to have a basic understanding of shellcode. Shellcode is basically program code in binary processor native code, usually reproduced in hexadecimal format. The purpose of the program code could be to download and execute a virus or a worm. The reason shellcode is used is that code injected in memory has to be

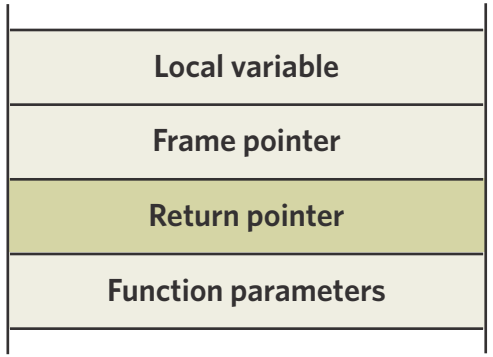
in a format for the processor to understand. When exploiting a buffer overflow vulnerability, the overflowed data including the shellcode is injected into memory and the processor is manipulated to jump to the shellcode. As an example "Royal Holloway University of London" in hexadecimal format will look like:

```
52 6f 79 61 6c 20 48 6f 6c 6c 6f 77
61 79 20 55 6e 69 76 65 72 73 69 74
79 20 6f 66 20 4c 6f 6e 64 6f 6e
```

There are two main types of buffer overflow vulnerabilities, stack-based and heap-based buffer overflows. The stack and heap are areas of memory that programs use for reading and writing data.

STACK-BASED BUFFER OVERFLOWS: When a program uses the stack it creates a stack frame for function calls made by the program. A function is a set of instructions to carry out a specific task such as copying a file from one location to another. When a function call is made, the function is carried out for the program and then the flow of control is returned back to the calling program. The stack frame created holds a number of elements for the called function. First the function parameters are placed onto the stack frame. Next the return pointer is pushed on which holds the current return address. This return address is a location in memory where the program is located so when an operation is finished it knows where to return. Next to go onto the stack frame is the frame pointer which contains the base address of the stack frame. The base address is a location in memory of the stack frame which does not change during the functions operation. Finally the local variable goes onto the stack frame. This local variable contains the information the function needs to operate, such as the name of the file we wish to copy. FIGURE 1 shows a stack frame lay-

FIGURE 1



out. When too much data is inputted, more than the local variable can handle, e.g. entering a very long filename, it overflows towards and past the return pointer overwriting our return address and as a result producing a stack-based buffer overflow.

HEAP-BASED BUFFER OVERFLOWS: The heap memory is structured and works in a different way from stack memory. The heap manager manages the allocation and freeing of heap memory. When a program requires heap memory, the heap manager allocates blocks of memory known as chunks. This memory chunk consists of a chunk header and chunk data. The chunk header stores various bits of information about the chunk, such as the size, memory segment location, whether the memory chunk is free or in use, etc. This chunk header is 8 bytes in size and if the block is free then it contains 8 more bytes of which are its pointers. There are two pointers of 4 bytes each and are known as the forward and backward links. The pointers point to previous and next memory chunks which are managed by the heap manager. The heap manager updates these pointers and keeps note of which chunks are free and which chunks are in use. When a buffer overflow takes place, and a neighbouring chunk exists, the chunk data overflows into the next free heap chunk overwriting these pointers. When this happens a heap-based buffer overflow in the program has taken place.

GAINING CONTROL BY EXPLOITING SOFTWARE

Once a buffer overflow vulnerability has been discovered, it can be only a matter of days before hackers start exploiting the vulnerability to carry out its malicious activities. There are a number of buffer overflow techniques which allow hackers to take over a machine. These involve overwriting registers or function pointers with values that tell the processor to jump to a memory location and process shellcode from that location. Once the shellcode has been processed the processor goes back to its normal operation processing valid instructions. This is known as controlling the "flow of execution".

RETURN ADDRESS: When overwriting the return address, the values held in a control register which holds the return address for the current function are indirectly changed. Every time the program completes an operation it updates this register with a new return address. This return

address is responsible for the program's flow of execution. To obtain control an address needs to be found that can be placed in the register that can then take control of the flow execution back to the memory location where the injected shellcode resides. To recognize what address is needed, the stack memory is examined after an overflow and observations on what memory locations have been overwritten are used. Based on what has been overwritten an address is chosen that is at the beginning of the injected code.

STRUCTURED EXCEPTION HANDLER: Another way to control the flow of execution for stack-based overflows is using the structured exception handler. The Structured Exception Handler or SEH for short is a structure used in handing errors in software. This SEH structure contains two pointers; one which points to the next handler in a chain of handlers and the other is the exception handler which essentially produces the error. When a buffer overflow occurs in a function that is located inside a block of code handled by the SEH structure, the SEH handler handles the error, for example overwriting the return address. In some cases when the allocated memory is overflowed it will continue overwriting the stack and these two pointers. These pointers can be used to control the flow of execution.

UNHANDLED EXCEPTION FILTER: In heap-based buffer overflows control of the flow of execution is obtained by overwriting the pointers on the next available free block header. These pointers are held in two registers. These two registers can then be used to overwrite a function pointer to jump to a memory location containing the shellcode. The **UnhandledExceptionFilter()** function is one of many exception handler functions which can be used to do this. Exception handler functions are used rather than any other functions as it allows control of the flow of execution. If pointers of other functions were to be overwritten, an exception would have occurred preventing the exploit from being successful. Once the pointers of an exception handler function has been overwritten, thereafter, when an exception takes place and the exception handler function is called, the flow of execution jumps to the address given by the modified pointers and executes the injected shellcode.

Writing exploits can be a challenge so knowing a variety of languages can be a tremendous help. It is common to come across certain languages better suited for writing particular exploits[3]. The memory available to be able to inject shellcode

can be limited and the injected code needs to be able to handle restrictions on certain hexadecimal values.

FUNCTION ADDRESSES: Each function has its own memory address describing where it is located in memory. These function addresses are static i.e. they do not change if the version of the software does not change. Static function addresses are generally used in shellcode to make the size of the shellcode smaller. It is normal to encounter a limited amount of memory space so having a small shellcode helps the exploit be successful.

Having static addresses has its drawbacks as it makes the exploit dependant on a specific operating system or version of software.

BAD CHARACTERS: For injected shellcode to work the vulnerable software must not remove or replace any characters from our shellcode. Some applications implement a filtering process which modifies or omits a number of characters from the inputted data when placing it into memory. These characters would no longer be available in the executing shellcode. Successful exploits have been modified to take account of modified input characters.

PREVENTING EXPLOITATION OF SOFTWARE

There are a number of ways buffer overflows can be prevented even if vulnerable functions still exist in code. Ideally developers should replace vulnerable functions with secure functions and follow secure programming practices, but developers are only human and mistakes will happen. The features mentioned below add another preventive measure in software.

ADDRESS SPACE LAYOUT RANDOMIZATION: Address Space Layout Randomization is commonly known as ASLR. This feature randomizes a program's memory address locations every time a program is started. As memory addresses are different, an attacker has no way of predicting what jump address or function address to use in an exploit. This feature has been incorporated in Microsoft's Windows Vista®. For programs to benefit from this feature developers will need to compile the program code and link it with the DYNAMICBASE switch.

HARDWARE-BASED DEP: Data Execution Prevention (DEP) is another protection mechanism which prevents the execution of code from the stack or the heap. It works by marking memory

locations as readable and writeable but not executable. To be able to mark memory locations the operating system requires processors that support NX ("No eXecute") bit. For software to benefit from this feature the program code will need to be compiled and linked with a switch called NXCOMPAT. No matter what method thereafter is used to take to control of the flow of execution back to the shellcode location on the stack or heap, the exploit will be unable to execute. In this way, we can mitigate the vulnerability.

SOFTWARE-BASED DEP: For machines that do not have processors supporting the hardware NX bit, software-based DEP can provide another form of protection. This protection mechanism does not prevent execution of code in memory but instead protects from SEH overwrite attacks. When an exception is raised it checks to make sure that the exception is registered and is considered to be one of the safe exception handlers. If the exception handler is not a safe exception handler, then steps are taken to prevent it from being called, stopping a possible attack. To use software-based DEP protection program code will need to be compiled and linked with the SAFESEH switch.

CANARY: Another simple but effective way of preventing the occurrence of buffer overflows is to placing a canary, also known as a cookie, before the return address in the stack. When a stack-based buffer overflow occurs it overwrites this canary and the return address. Before the program returns, the value of the canary on the stack is compared to the original value stored in memory, and if they do not match then the program closes. Programs will need to be compiled with the GS switch to benefit from this protection.

SAFE UNLINKING AND COOKIES: To protect heap memory chunks Microsoft introduced two new security features. One was to prevent the abuse of the unlinking functionality of the heap management routines. This is known as safe unlinking and requires link pointers (which point to the previous and next memory chunks) to be validated before executing the unlinking process. The other security feature involved adding a cookie in the chunk header. A cookie is a byte in size which contains a value. This value is checked when memory chunks are allocated and freed. If the cookie value is found to be missing or inconsistent then an exception is raised mitigating a potential exploit.

BYPASSING PROTECTION

Despite all these preventive measures, there are still a fair few techniques which can be used to bypass the above protection. Below we explain a number of ways that prevention can be bypassed.

RETURN TO LIBC: Stack-based buffer overflows require executable stack memory for the injected shellcode to be processed. If however the vulnerable software takes advantage of hardware-based DEP protection then execution of shellcode from the stack will not be possible and consequently the use of shellcode is no longer applicable. Unfortunately there is a way around this protection known as "Return to Libc". In return to libc, the return address is overwritten with a memory address of a function already loaded in memory. Using this function the flow of execution can be made to execute a command injected in the stack frame instead of our shellcode. For example, a command could be "calc.exe" which would load up the Windows calculator.

HEAP SPRAYING: Heap spraying technique works by spraying the heap with hundreds of heap memory blocks each containing a block of shellcode. When a stack-based or heap-based buffer overflow takes place, it is possible that control of the flow of execution falls into one of these blocks, which executes the shellcode. Due to the heap protections put in place by Microsoft, the heap spraying technique is an ideal choice for exploitation as it bypasses the existing protection. This spraying technique works well for web based attacks as the heap protection does not extend to program data stored in memory.

NO SAFESEH: For software-based DEP protection the SafeSEH switch will need to be used at compile time telling the linker to add a static list of known good exception handlers in the program for it to use. Unfortunately if every individual program code is not compiled with SafeSEH then protection is not achieved. If these unprotected programs are loaded by software that has been compiled with the SafeSEH switch, it makes the protected software insecure.

CONCLUSION

This article has shown how buffer overflows take place and what effects they can bring. A number of preventive measures can be taken by developers for protecting their code from buffer overflows, but this should by no means thought of as a solution, but rather as a backup measure if acci-

dentally unsafe functions or programming techniques are used in the code. Most of these preventive measures involve recompiling and linking code with various switches. By using the latest compilers provided by development environments such as Visual Studio® 2008, we will be able to use all of the switches mentioned in this article to protect code. Upgrading to Microsoft's Windows Vista® operating system will also be a huge countermeasure against buffer overflows. Windows Vista® alone will greatly diminish exploits from succeeding even if vulnerable code still exists with no preventive measures taken by the developer. It is therefore always recommended to upgrade products frequently to benefit from the new security features being integrated in products.

TABLE 1 below shows that applying one pre-

TABLE 1

BUFFER OVERFLOW PREVENTIVE MEASURES AND BYPASS TECHNIQUES

PREVENTIVE MEASURE	SWITCH/UPDATE FOR PROTECTION	TYPE OF PROTECTION	METHOD TO BYPASS
ASLR	Dynamicbase	Stack and heap	None easily
DEP (hardware)	Nxcompat	Stack and heap	Return to LibC
DEP (software)	SafeSEH	Stack	Heap spraying
Canary	GS (default)	Stack	SEH/heap spraying
Safe unlinking	SP2 for XP onwards	Heap	Heap spraying
Heap cookies	SP2 for XP onwards	Heap	Heap spraying

ventive measure will still allow an attacker to use another technique to obtain control.

It is worth pointing out that just having an up-to-date antivirus software and intrusion prevention software is not enough these days as hackers always find innovative techniques to bypass detection. It is now considered good practice, when a buffer overflow vulnerability has been published and there is a software update to fix the issue, to get the fix deployed as soon as operationally possible. Buffer overflows are not something to overlook as they are here to stay for years to come. ■

ABOUT THE AUTHORS

Parvez Anwar is a Senior Technical Security Specialist working for Verizon. His specialities include the investigation of malware and performing assessments for vulnerablities. As an independent security researcher, he hunts for vulnerabilities in his spare time and his personal web site is www.greyhathacker.net

Andreas Fuchsberger is a lecturer in information security at Royal Holloway, University of London, with research interests in secure software development, intrusion detection, and network security. He teaches the course in Software Security on the M.Sc. in Information Security at Royal Holloway.

SOURCES:

[1]http://news.zdnet.co.uk/itmanagement/0,1000000308,2129738,00.htm

[2]http://vil.nai.com/vil/content/v_100454.htm

[3]The full MSc project report illustrates an exploit bypassing certain restrictions.