Modern Buffer Overflow Prevention Techniques:

How they work and why they don't

Russ Osborn

CS182 JT

4/13/2006

In the past 10 years, computer viruses have been a growing problem. In 1995, there were approximately 2,400 known viruses. This number increased to 82,000 by 2002[1]. These viruses have caused tremendous damage and stress for computer users everywhere. With the development of ubiquitous computing, the virus has spread from servers and desktops to cell phones and handheld personal organizers. Computer viruses have evolved from an annoyance to a threat which can disrupt the functioning of our daily lives.

Traditionally, a buffer overflow used an unprotected or unbounded copy with attacker controlled data to overwrite a buffer on the stack. The attacker's data would be copied down the stack until the return address of the overflowing function was overwritten. At this point, the attacker can gain execution control at a location of his choosing. The majority of viruses use this method of attack; however, many new types of attacks are just as effective as traditional methods.

Modern buffer overflow techniques have become increasingly clever and complicated. Most new methods involve leveraging the same type of unbounded copy mistake as in a traditional attack; however, they accomplish their exploit in a different manner[2]. Modern attacks may simply modify data on the stack (not overwriting the return address), and, in doing so, change the course of program execution to an attacker chosen path. An example might be overwriting a pointer stored on the stack so that it points at an attacker controlled input buffer instead of its original target. Another type of attack involves supplying unexpected data as program input. For example, an attacker can put a

[1] "Intel XD Bit," < http://www.intel.com/business/bss/infrastructure/security/flash.htm >
[2] Baker, Brendan and Pincus, Jonathan. "Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns," IEEE Security and Privacy, 2004 p. 20-27

sequence of control characters in an input string, which will then be processed by the program as if they were trusted commands. As a final example, it has recently been discovered that heap buffers may also be overflowed to the attacker's advantage. Heap allocated blocks have header data before the actual block of allocated memory. When overflowing a buffer on the heap, these header values are overwritten. Most commonly, the attacker is able to use this to cause a wild pointer write to an arbitrary location. While these types of attacks may be the most common of the new breed of attacks, there are undoubtedly many more, clever attacks which take advantage of insecure coding practices.

Not surprisingly, software developers have exerted tremendous effort to avoid coding errors leading to viruses. During Microsoft's security push, they suspended development until all of their programmers took secure coding courses. Many companies have extensive code reviews and red team efforts (internal teams that try to attack the developer's code to find exploitable errors). Certainly, secure coding practice has become much more important for all production software.

In addition to the emphasis on writing secure code, there have been a number of measures developed by both software and hardware vendors to enhance the security of their products. There are two general classes of mechanisms for enhanced security[3]. The first involves modifying compilers to change the structure of data in memory. The second involves additional protection (usually through hardware) of memory, commonly in the form of write or execute prevention.

[3] Crispan Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," 7th USENIX Security Symposium Proceedings, 1998.

The common type of compiler modification is presented by Cowan et al. in their paper, “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” (called either the StackGuard or stack canary system). Here, the authors present a technique for protecting the return address from being overwritten in a standard buffer overflow. Note that, as the name suggests, this approach protects only the stack. The idea could be expanded to the heap, but this has not been done yet. The key to the StackGuard system is that a randomly generated value, called a canary, is placed on the stack just above the return address. When a function prepares to return, it compares this random value with a saved copy located elsewhere (on the heap). If the values do not match, the return address is considered to be invalid and the system does not use this corrupted value.

This method introduces relatively little overhead. It involves only two extra steps. First, the canary must be pushed onto the stack immediately after the return address. Second, the canary value must be checked against the saved value. This can take somewhat longer, as it requires calling a checking function and reading a word from memory. Cowan et al. ran a series of tests with canaries in place and determined that for a simple function that took a single argument, incremented it, and returned, that the runtime was increased by about 70%. However, for longer functions, the overhead is the same, so true system runtime increase would be less. Additionally, the canaries need only be put in functions with potential overflows, meaning that not every function is subject to the increase in runtime. The StackGuard system is currently incorporated into some versions of GCC. Microsoft has adopted a similar system in to their latest Visual Studio Compiler,

and has recompiled their entire Internet Information Services (IIS) module with this protection mechanism.

The StackGuard mechanism certainly protects against many existing viruses. This success is because nearly all viruses depend on overwriting the return address to gain execution control. Consider, however, the other types of attacks discussed above. The StackGuard system (as described in Cowan's paper) does not protect any local variables or pointers. This leaves the system vulnerable to these types of attacks, despite the stack canary. Microsoft has addressed this issue in their latest version of the Visual Studio C/C++ compiler. In Microsoft compiled code, all local variables and pointers are stored after the return address (and therefore after the stack canary). Thus, to overwrite any variables used by the program, the stack canary must also be overwritten. It seems as though the StackGuard system can be modified to protect from all types of stack based attacks.

David Litchfield presents a novel attack to defeat the StackGuard-style protection mechanism[4]. First, assume that we have a buffer overflow attack which can overwrite a return address, but also overwrites the stack canary in doing so. After the program determines that the canary has been overwritten, it invokes the exception handler. This handler is called and deals with the exception as any other software exception would be handled. However, the exception handling is maintained by a series of structures on the stack (the exact format varies depending on the operating system, and may be very different for Linux). Included in this structure is a function pointer to the current exception handler. This function pointer is called when an exception is raised. Now, we

[4] Litchfield, David. "Defeating the Stack Based Buffer Overflow Protection Mechanism of Windows Server 2003," Proceedings of Black Hat, September 8th 2003.

can gain control of a system with stack canaries by overwriting the stack to such a large degree that we overwrite an exception handler function pointer, and have an address of our choosing called instead of the true exception handler. Gaining control in a process with stack canaries certainly requires more effort. However, the attacker can still gain control of the process, despite the StackGuard mechanism.

The StackGuard system of stack canaries provides only a limited set of protections. The only situation in which they prevent a buffer overflow style attack is when there is no exception handler structure on the stack (no 'try… catch' blocks are used), or where this structure is so far away on the stack that trying to overwrite by such a long length generates an exception before the handler can be overwritten. The stack canary system is far from reaching its stated goal of preventing future buffer overflow attacks.

The stack canary system makes the mistake of trying to allow recovery from an attacked process. To remedy the problems with the current stack canary system, only a few changes need to be made. Successfully implementing these changes would make traditional buffer overflows very difficult in the future. The safest thing to do once a buffer overflow has been detected is to exit the process immediately. Clearly, something has gone very wrong. Since the basic assumption about the safety of data on the stack has been violated, an attacked process should just exit. In the case of stack canaries, failure to do so results in an unsuccessful protection system. Additionally, canaries should be placed at the end of every buffer that serves as a copy destination (including heap buffers), and should be checked after every copy. This way, if a buffer is overflowed, the program can halt immediately, avoiding execution of any instructions after the program

enters an unsafe state. This solution, however, becomes increasingly time consuming and begins to resemble a rudimentary version of bounds checking that is already available in other programming languages.

The second common avenue for protection against buffer overflows (including heap overflows) is additional forms of memory protection. This can be done both at the software and hardware level. Cowan et al. address the software level implementation of this idea with their MemGuard protection mechanism[5]. The idea is that every standard memory operation is replaced with a safe, bounds checked alternative. Effectively, this is adding a bounds checking layer to C/C++. Runtime test of the system reveal that it is impractical, with standard runtimes increasing by a factor of 50 to 100.

If bounds checking during runtime at the software level were a desired feature for developers, several modern languages (Java, Ada, etc.) are available to do so. It seems as though modern developers do not feel that the performance penalty brought using these other languages is worth it, leaving such memory protections to the hardware manufacturers. Both Intel and AMD have implemented enhanced versions of memory protection with buffer overflow protection in mind (called XD and NX, respectively)[6][7]. While the specifics of the hardware implementation differ, the protections they offer do not. Each mechanism provides another entry in the page tables specifying whether or not a given page is executable. By doing this, the stack can be marked as non-executable, so

---

[5] Crispan Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," 7th USENIX Security Symposium Proceedings, 1998.

[6] Zeichick, Alan. "Security Ahoy! Flying the NX Flag on Windows and AMD64 To Stop Attacks," <http://www.devx.com/amd/Article/27809>, March 31, 2005

[7] "Intel Execute Disable bit and Windows XP SP2," <http://www.intel.com/business/bss/infrastructure/security/flash.htm>

attackers can no longer specify a return address which is a buffer they control on the stack.

This protection mechanism does prevent some types of attacks. However, all an attacker needs to do is locate a series of bytes which can be interpreted as useful instructions and then be used to gain control in a loaded library (there are many libraries loaded at any given time). For instance, it may be possible to return to a procedure like Winexec or some similar library function which could be subverted to spawn another process or perform some task enabling further exploitation (there is a class of exploits of this type known as ‘return to libc’ attacks). With a large number of executable libraries to choose, and much of the data on the stack controlled due to the overflow, the attacker may be able to select the arguments to whatever function he can find. Some exploits may be made impossible by the addition of these execution controls in the page tables. However, it is unlikely that system managers would consider their system to be safe when a hacker would have the opportunity to still execute any code in the attacked process’ memory space with arguments of the attacker’s choice. As long as an attacker can modify data that is needed by the program to continue its normal function, simply preventing the types of attacks that already exist does not solve the security issues raised by unbounded copies.

An effective memory protection mechanism would require frequent updates of permissions for commonly used memory, and permissions on a word level. This would allow protection of return addresses during a function’s duration until return is invoked, when the area on the stack occupied by the return address could legally be overwritten. This solution, however, is rather unfeasible as it would require a tremendous amount of

overhead due to checking and reassigning protection bits on individual words in memory. If we only wanted to add 3 bits of protection (read, write, execute) for every word, this introduces nearly a 10 percent storage overhead on 32 bit processors. Such protection is likely too costly to be implemented. In general, memory protection mechanisms do not provide an ultimate solution to the buffer overflow problem. They act only as another hurdle for an attacker to overcome in order to gain control.

The current state of the art methods for preventing attacks based on buffer overflows are unsuccessful in their goal of preventing future unbounded copy based attacks. Both compiler based changes to rearrange memory structure and memory protection mechanisms are merely responses to the types of attacks seen previously. As they currently stand, these security mechanisms stop old viruses but do nothing to prevent a clever (or simply well-read) hacker from developing an attack to circumvent the latest protection efforts. Ultimately, the flaw of these security measures lies in their reactionary approach to handling viruses and their lack vision about the alternatives to the classic buffer overflow attack.

The real solutions to the problems these measures try to address are costly, but certainly within reach. Writing secure code seems to be the most straightforward approach. However, time and time again, it has proven difficult to do in its entirety for a whole project. Switching to a secure, bounds checked programming language is certainly viable. Java development is a promising option for those desiring security. Each of the mechanisms discussed here can be modified to deal with the various attacks currently employed, however the performance hit may well be worse than changing to a secure language (and in the case of stack canaries, making the mechanism more effective ends

up adding a bounds checking level to current C/C++ code). It is certainly possible to develop secure code without extraordinary innovation, and hopefully no great catastrophe is needed to prove to developers that virus-safe code is worth the effort.

Works Cited

Baker, Brendan and Pincus, Jonathan. “Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns,” IEEE Security and Privacy, 2004 p. 20-27

Cowan, Crispan et al. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” 7th USENIX Security Symposium Proceedings, 1998.

Litchfield, David. “Defeating the Stack Based Buffer Overflow Protection Mechanism of Windows Server 2003,” Proceedings of Black Hat, September 8th 2003.

Zeichick, Alan. “Security Ahoy! Flying the NX Flag on Windows and AMD64 to Stop Attacks,” <http://www.devx.com/amd/Article/27809>, March 31, 2005

“Intel Execute Disable bit and Windows XP SP2,” <http://www.intel.com/business/bss/infrastructure/security/flash.htm>