STEVEN ALEXANDER

defeating compilerlevel buffer overflow protection



Steven is a network test engineer at Front Porch in Sonora, CA. He gets to break things and shoot Nerf guns at people.

■ alexander.steven@sbcglobal.net

BUFFER OVERFLOW ATTACKS ARE THE

most popular method intruders use to gain remote and privileged access to computer systems. Programs that fail to use appropriate bounds checking can allow an attacker to write data beyond the intended boundaries of a buffer and thus possibly corrupt control structures in the program. This enables an attacker to execute arbitrary code with the same privilege as the victim process. An attacker's preference is usually to overwrite the saved instruction pointer that is pushed onto the stack before a function call or to overwrite a function pointer that will be used later in the program.

It is also possible to use these attacks simply to overwrite other data. This kind of attack is harder to prevent but, fortunately, is less common than the previous type and is not discussed here.

Buffer overflows first gained attention with the release of the famed Morris worm which exploited a buffer overflow in fingerd [1]. Despite the attack used in the Morris worm, buffer overflows did not become popular until the release of two papers that detailed the discovery and exploitation of these vulnerabilities [2,3].

This paper discusses vulnerabilities in two compiler-level protection mechanisms, StackGuard and Point-Guard. While this paper takes a critical look at both of these solutions, it does not intend to make them seem insignificant. The attacks described in this paper help to show how StackGuard and PointGuard should be complemented to construct a more complete protection system.

The reader should also note that PointGuard has not been publicly deployed. It was presented at the USENIX Security Symposium in 2003. The design might be changed before its release to correct functionality problems with some real-world software [4].

The reader should also note that StackGuard has reverted from the more advanced random XOR canary protection method analyzed here to the simpler terminator canary [5]. The justification for the change is that the attack method that prompted the change also enables an attacker to manipulate a program in ways that StackGuard cannot, and was not designed to, protect against. Because StackGuard has reverted to a weaker method and PointGuard is not available, the attacks in this paper are mostly of importance to the designers of new protection methods and have little consequence for currently deployed systems.

Exploiting a Buffer Overflow

To understand how a buffer overflow exploit works, we must first understand how a function call occurs:

- 1. The calling procedure pushes any function arguments onto the stack in reverse order.
- 2. The calling procedure executes a "call" instruction, which pushes the address of the next sequential instruction onto the stack and tells the processor to transfer execution to the target function.
- 3. Assuming that frame pointers are being used, the called function pushes the old frame pointer onto the stack and copies the value stored in the stack pointer over the frame pointer. Then, the stack pointer is decremented (the stack grows down) to make room for local variables.

Figure 1 show the stack layout for a called function with a single variable (a character array).

Function Arguments
Saved Instruction Pointer
Saved Frame Pointer
buf[255]
•••
buf[0]

FIGURE 1: STACK LAYOUT FOR A CALLED FUNCTION WITH CHARACTER ARRAY ARGUMENT

The function epilogue consists of popping the saved frame pointer from the stack and executing a return instruction. The return instruction causes the processor to pop the saved instruction pointer from the stack into the program counter and begin execution at that address. The saved instruction pointer is supposed to hold the instruction address that was saved on the stack in step 2 above.

Consider the following code:

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    char buf[256];
    if(argc < 2) {
        printf("Oops.\n");
        return -1;
    }
    strcpy(buf, argv[1]);
    return 0;
}</pre>
```

This snippet of code is vulnerable to a trivial buffer overflow attack. The strcpy function does not perform bounds checking (unlike its cousin strncpy), so the program will copy characters from argv[1] to buf until the program crashes or strcpy encounters a null character, \(\mathbb{O}\). An attacker could find a way to provide a carefully crafted input that will cause this function to execute his own code instead.

First, such an attacker would assemble a small bit of code that will do something useful such as the semantic equivalent of exec ("/bin/sh"). Such code is usually

referred to as "shellcode" since the popular use is to execute a command shell. Shellcode can be used to do more complicated things, such as open a network connection or add a new root user. There are some restrictions as to how this code can be constructed. For instance, there cannot be any null characters in the resulting machine code. Aleph One discusses constructing workable shellcode [3]. There is quite a lot of shellcode available online so, unfortunately, aspiring exploit writers don't have to start from scratch.

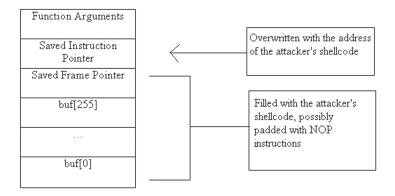
In order to execute some shellcode, an attacker provides the code as a part of the input to a vulnerable program. The attacker crafts the input so that it will exceed the bounds of the allocated buffer and overwrite the saved instruction pointer with the address of the provided shellcode. If the attacker does not know the exact address at which the shellcode will be stored, he can prepend a series of null instructions (NOPs) to the shellcode. If the provided address points to any location within the series of NOPs, execution will continue through the NOPs and eventually reach the shellcode. If the attacker does not know the exact location of the saved instruction pointer (common if the attacker doesn't have access to the source code), he may duplicate the shellcode address several times. In such a case, it might take the attacker a few tries to overwrite the saved instruction pointer on the correct 4-byte boundary.

It might also take the attacker a few extra tries to guess the correct shellcode address. The address at which the shellcode is stored is usually not difficult to guess, even in black-box analysis, since the stack begins at a known location. This does not hold true if the target program runs on a system with good stack randomization. Figure 2 shows the attacker's input layout. Figure 3 depicts the manner in which this input corresponds to the function stack layout.



FIGURE 2: ATTACKER'S STACK LAYOUT AFTER INPUT

Higher Addresses



Lower Addresses

FIGURE 3: EXPLICATION OF FUNCTION STACK LAYOUT WITH ATTACK

Solutions, a Survey

Many methods have been proposed to prevent the execution of buffer overflow attacks [6], some of which are discussed here. Papers about several solutions and attacks are available on Purdue University's SmashGuard buffer overflow prevention page [7].

OS-LEVEL

NON-EXECUTABLE STACK

One of the first methods, the non-executable stack, was proposed by Solar Designer [8]. A non-executable stack prevents the standard buffer overflow attack which modifies the saved instruction pointer so that it points at the attacker's shellcode. The attacker's shellcode is normally stored in the same stack-allocated buffer that was overrun to change the instruction pointer. If the stack is non-executable, the attempt to resume execution at this location will fail.

This defense can be defeated by injecting executable code into other data areas, such as the standard .data and .bss sections. The defense was also defeated by Solar Designer [9] and Rafal Wojtczuk [10] using the return-into-libc method. In this method, the saved instruction pointer is modified so that the program will return into an instruction sequence in the *C* library. It is not necessary that the instruction pointer direct execution to the beginning of a function in the *C* library. Often, an attacker will wish to point at a call to system() inside one of the *C* library functions. An attacker can manipulate the stack so that his provided arguments will be used in the call to system().

PAX/ASLR

Randomizing the base address at which libraries are loaded can hinder returninto-libc attacks (used to defeat non-executable restrictions such as those in Solar Designer's stack patch). This technique was introduced in [11] and used by ASLR in PaX [12]. In early versions of PaX, an attacker could defeat this by instead returning into the Process Linkage Table (PLT) [13,14]. The PLT is used to resolve libc (and other) function addresses automatically. Currently, PaX can also randomize the executable base for ELF executables [12]; this prevents the return-into-PLT attack. There is another attack that can be used against PaX/ASLR with the randomized executable base in effect [15]. The attack uses a partial overwrite of the saved instruction pointer to gain control over the arguments passed to printf, which allows an attacker to discover information about the randomized library base using a format string attack so that a normal returninto-libc attack can be performed. When used with PaX, StackGuard and Pro-Police/SSP can both prevent these attacks. The OpenBSD project has implemented W^X, which uses techniques similar to PaX. OpenBSD also uses address randomization and ProPolice/SSP.

COMPILER-LEVEL

STACKGUARD AND PROPOLICE/SSP

Another possible solution was proposed by Crispin Cowan and is used in Stack-Guard [16,5]. StackGuard places a canary value between the saved frame and instruction pointers and the local function arguments. Figure 4 shows the revised stack layout. The canary value is set in the prologue to each function and is checked for validity in the epilogue. If the canary value has been modified, a

handler function is called and the program terminates. A direct attack will overwrite the canary value before it overwrites the saved instruction or frame pointers. Any of three types of canary can be used: a terminator canary, a random canary, or a random XOR canary.

Function Arguments				
Saved Instruction Pointer				
Saved Frame Pointer				
Canary				
buf[255]				
buf[0]				

FIGURE 4: REVISED STACK LAYOUT WITH CANARY

A terminator canary contains multiple terminator values, such as a NULL byte or newline, which are used to indicate the end of a string in the various *C* library string functions. Because these values are used to terminate a string, an attacker cannot avoid changing them with a direct buffer overrun. It is possible to repair a terminator canary if an attacker has the opportunity to perform multiple overruns in one function. The first overrun can be used to change the instruction pointer and the subsequent overrun can be used to repair the canary by lining up the terminator in the string with the corresponding value in the terminator canary.

A random canary is a random value chosen at runtime. The random value is stored in a global variable and is used for each function in a program. It is stored in the same manner as the terminator canary. It is assumed that an attacker will be unable to overwrite the global value or to cause the program to leak the value. In some circumstances it is possible, however, to force the program to leak the random value using a format string attack. Overwriting the global variable is not useful since an attacker could just as easily overwrite a function pointer (.got entry, .dtors, etc.).

The random XOR canary was introduced into StackGuard to prevent an attack published in *Phrack Magazine* [17]. Rather than directly overwriting the canary and saved instruction pointers, an attacker can overwrite a data pointer that will be used later in the function as the destination for a string or memory copy that uses attacker-supplied data. The attacker can modify the pointer so that it points directly at the saved instruction pointer. When the attacker's data is copied to that address later in the function, the saved instruction pointer will be overwritten without modifying the canary.

With the random XOR canary, a random value is again generated at runtime and stored in a global variable. Rather than storing the random value on the stack, the random value is XORed with the saved instruction pointer and the result is stored on the stack. During the function epilogue, the saved canary is XORed with the random value and the result is compared to the saved instruction

pointer. If the values do not match, the handler function is called and the program terminates. The maintainers of StackGuard have reverted to using the terminator canary because the attack used to defeat the terminator canary can also be used to corrupt other important values such as function pointers.

SSP, previously known as ProPolice, is based on StackGuard and uses a random canary [18]. SSP offers several improvements over StackGuard, however, and is more difficult to defeat. SSP reorders local function variables so that pointers are stored below buffers in memory (i.e., higher on the stack). This rearrangement prevents an attacker from successfully employing attacks such as the one used to defeat StackGuard. There is a limitation to this: the variables within a data structure cannot be reordered, so it is possible for an attacker to exploit a buffer overflow within a data structure and overwrite a pointer value within that same structure. This does not seem (to me) to be a common problem.

SSP also copies function arguments to the local stack frame. An attacker can target the arguments of a function if they will be used inside the function after he modifies them. In some cases, an attacker can use them (perhaps by overwriting a pointer value) to write arbitrary data to any writable location in memory. The canary value will be overwritten but, since an attacker can write anywhere, he could also overwrite the address in .got of one of the functions used in the handler function that is called to terminate the program. By copying the function arguments to a local memory area below the local variables, SSP prevents this.

StackGuard and SSP cannot prevent attacks that occur in heap memory [19,20,21]. Early versions of StackGuard did not attempt to protect the saved frame pointer. If the frame pointer is not protected, StackGuard can be bypassed by taking control of the stack frame [22].

PointGuard

PointGuard protects pointer values inside programs, a technique that promises much better protection than using StackGuard alone [23]. PointGuard works by XOR-encrypting pointer values with a random value determined at runtime and stored in a global variable. Code is added to a protected program to decrypt pointer values automatically before each use. Pointer values are decrypted only in registers, and the decrypted pointer is not stored in memory. Without knowledge of the random value used to encrypt the pointers in a program, an attacker cannot overwrite a pointer and hope for a meaningful decryption. If an attacker overwrites a pointer hoping to point to an exact location, his chances are 1 in 2^{32} , or about 1 in 4 billion. An attacker has a much better chance if he is trying to point a function pointer at NOP-padded shellcode, but even with a 1-kilobyte NOP buffer, his chances are only about 1 in 4 million. Dereferencing a random pointer value is likely to cause a segmentation violation, which will cause the targeted program to exit and dump core.

Unlike StackGuard and SSP, PointGuard does provide protection against heap attacks. Note that in order to provide protection against malloc and free attacks, libc must be compiled with PointGuard. Unfortunately, PointGuard can be defeated using format string attacks, as discussed on Bugtraq [24] and using an attack detailed below. An implementation of PointGuard has not been publicly released.

Format String Vulnerabilities

Format string vulnerabilities arise when functions that accept format strings and a variable number of arguments (e.g., printf) are used without a programmer-

provided format string. If the function is used to process user-provided input, a malicious user can supply his own format string. An attacker can use specially crafted input to leak information from the victim program (most likely by walking the stack) or to overwrite arbitrary data. (For an introduction to format string exploits, see [25]; for more advanced techniques, see [26] and [27].)

In the printf family of functions, data can be overwritten using the %n format specifier. The %n specifier stores the number of bytes that printf has written so far at the provided address. An attacker can use this feature to overwrite a pointer (including a function pointer), a saved instruction address, an entry in the Global Offset Table (GOT) [14], or any other value in memory that can be changed to aid an attacker in diverting a program's execution or elevating privilege.

While some RISC systems have alignment requirements for writes that use the %n specifier, Intel-based systems do not. Because of this, the %n specifier can be used multiple times, with each write operation targeting an address just one byte higher than the previous operation. In this case, only the least significant byte of each count is used to construct a new value for a 32-bit word. This technique has the consequence that it will also overwrite three bytes adjacent to the target value. This is usually not a problem for an attacker. If, for instance, an attacker uses this method to overwrite a saved instruction pointer, the first three bytes lower in the stack (at a higher memory address) will be corrupted. Normally, this value will be one of the arguments passed in to the current function. If this value is dereferenced after the attacker corrupts it, the program may crash. If, on the other hand, it is not, the attacker can cause the program to execute arbitrary code when it exits from the current function. If this is a problem, the attacker need only overwrite another value, such as _atexit or a GOT entry, instead.

Attacker-provided format strings can also be used to leak information from the currently running program. The %iii\$ specifier is extremely useful in this regard. In this specifier, iii is the number of the argument to print; for instance, %2\$08x will print the second argument on the stack in zero-padded hexadecimal format. This can be used to "walk" the stack or to print arbitrary values directly. This technique was crucial in gathering information for the return-intolibc exploit used to defeat PaX [15]. In that particular case, the least significant byte of the saved instruction pointer was overwritten by a buffer overflow to cause a vulnerable function to return directly to a printf call in the middle of that same function. In doing this, the author was able to cause his own arguments to be provided to the printf function instead of those that were hard-coded into the program. The author used this technique to force the program to leak the information necessary to execute a return-into-libc exploit on a PaX protected system with ASLR. The target function was not otherwise vulnerable to a format string attack. The format string attack was made possible only by the buffer overflow, which prevented the correct values from being placed on the stack before printf was called.

A New Weakness in PointGuard

In addition to the previously discussed vulnerability to information leaking with format strings, PointGuard is also vulnerable to buffer overflows and to data manipulation with format strings. The claim given in the PointGuard paper [23] is that an attacker can destroy a pointer value but cannot produce a predictable pointer value. This is not completely true.

PointGuard is weak because pointer encryption is achieved by using a bitwise exclusive-OR operation rather than a more complex nonlinear operation. Because of this, any byte of the encrypted pointer that is not overwritten will

still decrypt correctly. This enables an attacker to make use of partially overwriting a pointer. If an attacker can find a situation in which it is advantageous to redirect a pointer toward a location whose most significant one to three bytes are the same as the location that the pointer originally referenced, he can, by brute force, attempt to redirect the pointer to this new location with far less effort than would be required to brute-force a 32-bit value.

On little-endian architectures, an attacker can use a simple buffer overflow to overwrite the least significant bytes of a pointer value, since the least significant bytes are stored at a lower address and thus overwritten first. Using format string attacks, which allow considerable flexibility in the way a value is overwritten, an attacker can bypass PointGuard on both little-endian and big-endian systems.

Consider the following code, a variation of the vulnerable "straw man" program included in the PointGuard paper:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define ERROR -1
#define BUFSIZE 64
int goodfunc(const char *string) {
   printf("%s\n", string);
   return 0;
}
int main(int argc, char **argv) {
   static char buf[BUFSIZE]:
   static int (*funcptr)(const char *str);
   if(argc <= 2) {
      fprintf(stderr, "Usage: %s <buf> <goodfunc arg>\n",
      arqv[0]);
      exit(ERROR);
   funcptr = (int (*)(const char *str))goodfunc;
   memset(buf, 0, sizeof(buf));
   strncpy(buf, argv[1], strlen(argv[1]));
   (void)(*funcptr)(argv[2]);
   return 0;
```

I compiled this code on an Athlon XP running FreeBSD 4.9. When the executable is loaded, the goodfunc function is located at 0x080485c4 and the buf buffer is located at 0x08049940. An attacker who loaded buf with his own executable shellcode would only need to overwrite the two least significant bytes of funcptr correctly in order to execute his code instead of goodfunc. Since those two bytes are XOR-encrypted with 16 random bits, an attacker who overwrites the two least significant bytes of funcptr will have a 1 in 65,536 chance of redirecting that pointer to the beginning of his shellcode. While this might be difficult to accomplish remotely before the attack is noticed by a system administrator, such an attack could be accomplished locally without any trouble.

Obviously, this example is contrived and does not necessarily provide a realistic memory layout for a real-life program. Instead, let us consider the layout information (see Figure 5, below) from three real, privileged programs from FreeBSD 4.9: lpr, ftpd, and rcp. In the case of lpr, redirecting a vulnerable pointer in a PointGuard-protected instance of the program would require an amount of effort similar to our example, since the .text and both data sections are located inside the same 16-bit segment. The other two programs would be more difficult to subvert since the data sections and the .text section share only the most significant eight bits of their addresses. An attacker would thus be required to over-

write the lower 24 bits of a function pointer in order to redirect it to his injected shellcode in one of these sections.

The odds of an attacker providing a 24-bit value that will correctly decrypt to the address of his shellcode are slightly better than 1 in 17 million. The outlook for an attacker is not quite so bleak, however. If the attacker is able to place shellcode in more than one location or to prepend a long series of null operations (NOP) to his shellcode, he can increase his odds tremendously.

Assume that an attacker's shellcode is only 50 bytes (a number well within the normal range). Further, assume that he is able to place this shellcode at the end of a one-kilobyte buffer after padding the buffer with NOPs. The attacker's odds increase to one in 17,000. In some situations, the attacker may be able to construct an even longer series of NOPs by having access to a large character array or by overwriting several data structures with the NOPs and shellcode without that data being molested before the altered function pointer is dereferenced. In highly favorable situations, an attacker might be able to guess a correct value with only a few thousand guesses on average. Clearly, such situations do not correspond with the argument in the PointGuard paper that an attacker cannot meaningfully corrupt a pointer without knowledge of the PointGuard encryption key.

In general, the complexity of guessing a value that will successfully cause a function pointer to reference NOP-padded shellcode is $2^{(X-\ln(number\ of\ NOPs))}$ where X is the number of bits guessed.

On little-endian systems, the security of PointGuard can be improved slightly by rotating a pointer value one byte to the left after the XOR encryption and rotating it back before the XOR decryption. In most situations, this would force an attacker to overwrite the entire 32-bit value. Using format string attacks, it would still be possible in some circumstances to overwrite only the least significant three bytes. Still, such situations are likely to be far more rare than those in which an attacker can corrupt a pointer with a simple buffer overflow. Unfortunately, such a change is likely to at least double the current performance penalty imposed by PointGuard.

Program	.text	.data	.bss
/usr/bin/lpr	0x0804964c	0x0804f140	0x0804f400
/usr/libexec/ftpd	0x0804a974	0x08059ce0	0x0805a560
/hin/rcn	0x080480b8	0x08081ae0	0x080831a0

FIGURE 5: ACTUAL MEMORY LAYOUTS FOR THREE COMMON PROGRAMS

A New Weakness in StackGuard

In this section, all references to StackGuard should be interpreted to mean StackGuard with the random XOR canary [5,17].

StackGuard has a weakness that corresponds to the previously discussed vulnerability in PointGuard. The random XOR canary is the result of exclusive-ORing a random canary value (generated at runtime) with the saved instruction pointer. The result is stored on the stack after the saved instruction and frame pointers and before the local function variables. Code in the function epilogue exclusive-ORs the saved canary with the random value (thus canceling out the effect of the random value) and compares the result to the saved instruction pointer. If the two values do not match, the program exits.

Since exclusive-OR is a bitwise operation, if only some bytes of the saved instruction pointer are modified, then only the corresponding bytes of the saved canary value need to be modified. The bytes of the saved canary can be overwritten with any random value.

This weakness is more difficult to exploit than the one in PointGuard. The conditions that must exist in a program's code for exploitation to be possible are more specific. The value used to overwrite the saved canary must be equal to the result of exclusive-ORing the pertinent bytes of the random canary and the attacker-supplied instruction pointer value, since a direct comparison is used in the function epilogue to determine whether the exclusive-OR of the random canary and the saved instruction pointer match the saved canary. It is still possible to overwrite the saved canary value with any random or fixed value, since the random canary used by the program changes with each execution.

PointGuard offers more room for error because it does not perform a direct comparison; instead, PointGuard allows the pointer to be dereferenced, under the assumption that a corrupted pointer will decrypt to a random value and most likely reference an invalid memory region, which will cause the program to crash. With PointGuard, an attacker can inject NOP-padded shellcode, which allows him the opportunity to guess a value that will decrypt to any location within the series of NOPs (or the first useful instruction in the shellcode).

If a format string overwrite attack is used to circumvent StackGuard, the attack is fairly straightforward. The attacker uses the %n modifier to overwrite all or part of the saved instruction pointer with a newly constructed value of his choosing. The attacker also uses the %n modifier to overwrite the corresponding bytes of the saved canary with any random or fixed value (probably fixed).

If an attacker overwrites the entire saved instruction pointer, he must also overwrite the entire saved canary. In this case, his attack has less than a 1 in 4 billion chance of success. An attacker's goal will be to find a situation in which he is able to inject code at a location that shares one or two significant bytes with the value of the original saved instruction pointer (as in the above PointGuard attack).

In order to bypass StackGuard using traditional techniques, an attacker must use a buffer overflow to overwrite the least significant bytes of the saved canary value. The attacker can overwrite these bytes with any value, fixed or random. He must also overwrite a data pointer so that it points directly at the saved instruction pointer. This modified data pointer must later be used as the destination for a string or memory copy that uses user-supplied input. An attacker will use the string or memory copy to point the saved instruction pointer at his shellcode (or to perform a return-into-libc attack). The affected data pointer must point directly at the saved instruction pointer; there is no margin for error as when attempting to point at NOP-padded shellcode.

Assuming that an attacker is successful in overwriting a pointer value and that he uses the corrupted pointer to correctly overwrite the saved instruction pointer, this attack will fail in each instance that the saved canary value is not equal to the exclusive-OR of the random canary (generated at each execution of the program) and the attacker-supplied return address. Since the random canary changes with each execution of the program, an attacker can supply any fixed or random value to overwrite the least significant bytes of the saved canary and will eventually succeed.

Consider the following source code:

```
int main(int argc, char **argv) {
    char *ptr;
    char buf[256];
    ...
    strcpy(buf, argv[1]);
    do_some_parsing(buf);
    strcpy(ptr, buf);
}
```

This program is vulnerable to a standard buffer overflow attack; an attacker can provide input that will be copied beyond the boundaries of the buf array, potentially overwriting the saved frame or instruction pointers that are stored on the stack between the function arguments and the local variables.

StackGuard will prevent a generic buffer overflow attack against this code. If an attacker attempts a standard buffer overflow attack against the saved instruction pointer, the canary value will be overwritten, StackGuard will detect the modification in the function epilogue, and the attack will fail.

In the versions of StackGuard that use a random or terminator canary, the previously published attack [17] applies and ptr can be overwritten instead. Instead of attacking the saved instruction pointer, an attacker can use a stack overflow to modify ptr so that it points at the return instruction pointer. The second strcpy operation will then overwrite the instruction pointer with the contents of buf without modifying the canary.

Although the random XOR canary prevents a direct application of this attack, the attack remains possible with some modifications. An attacker can still use ptr to overwrite part or all of the saved instruction pointer. In addition, he will have to overwrite the bytes of the canary that correspond to the bytes of the instruction pointer that he modifies. If he modifies every byte of the saved instruction pointer, his chances of success are slim, because he will have to overwrite the entire canary and will have less than a 1 in 4 billion chance that he will overwrite the canary with the correct value.

To improve his chances, the attacker will have to inject code into the .bss or .data memory regions, which often share the one or two most significant bytes of their addresses with the .text section. Alternatively, the attacker can attempt a return-into-PLT attack, since the .plt section often shares the most significant bytes of its address with the .text section. By using a return-into-PLT attack or injecting code in the .bss or .data sections, an attacker can redirect control of the program by only partially overwriting the saved instruction pointer and, consequently, only partially overwriting the saved canary.

This attack is an extension of the technique used to bypass StackGuard [17]. In the *Phrack* article, a string pointer was overwritten using a simple stack overflow. The pointer was later used as a destination pointer for a string copy which overwrote the saved instruction pointer with the location of either attacker-supplied shellcode or the address of a libc function (for a return-into-libc attack). The attack in this paper carries the restrictions that the saved instruction pointer should only be partially overwritten in order to ensure a reasonable chance of success and that corresponding bytes of the saved canary value must also be overwritten.

Conclusion

The attack against StackGuard is easy to ameliorate since it depends on exact knowledge of the location of the saved instruction pointer on the stack. Runtime and load-time stack randomization [11] greatly increase the difficulty of this attack. Load-time stack randomization can be implemented with only a few lines of code on most operating systems [28]. The difficulty of this attack is multiplied by the amount of stack randomization applied. Thus, if an attacker constructs an exploit that has a 1 in 16 million chance of success (he modified three bytes) and the attack is used against a system that uses 10 bytes of stack randomization, the chance of success drops to less than 1 in 16 billion. PaX uses 24 bits of stack randomization; the code published in ;login: uses 18. Load-time stack randomization carries a negligible performance penalty at load-time and does not affect runtime performance at all.

The attack against StackGuard is not possible when PointGuard is used. Under most circumstances, this attack would not be possible if StackGuard used local variable reordering as in ProPolice/SSP.

The use of buffer overflows against PointGuard is possible only under specific circumstances. SSP's local variable reordering has no runtime performance penalty and would make these circumstances extremely rare. FormatGuard can likewise protect PointGuard against most format string attacks [29]. Unfortunately, FormatGuard protects only calls to the C library. Programs such as wu-ftpd, which use an alternative implementation of printf, would not be protected. In some circumstances, the combination of SSP, PointGuard, and FormatGuard would still be vulnerable. Replacing SSP with StackGuard makes the combination even weaker.

The non-executable restrictions imposed by PaX and W^X would make the attack against PointGuard difficult because an attacker would not be able to execute code injected into the .data or .bss sections. The various memory randomization features of ASLR would make it even more difficult for an attacker to meaningfully redirect a pointer value.

The use of compiler-level stack protection, as in StackGuard and SSP, along with PaX, can defeat the attacks that have been published for defeating PaX alone. Some more advanced variations on these attacks may be possible, but the pointer protection offered by SSP's local variable reordering is likely to prevent most of them. Even without the benefit of StackGuard or SSP, the attacks against PaX are more difficult than the above attack against PointGuard.

Pointer encryption, canary protection methods, and execution restriction mechanisms have all been shown to be vulnerable to various attacks. The risk of a successful attack against these systems can be reduced if a host intrusion detection mechanism such as Segvguard [13] is used to prevent a program from executing after some number of crashes. A mechanism such as Segvguard is necessary to complement PointGuard, PaX, W^X, or any address space randomization.

REFERENCES

- [1] Eugene Spafford, "The Internet Worm Program: Analysis," Computer Communications Review (January 1989).
- [2] Mudge, "How to Write Buffer Overflows" (October 1995) http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html.
- [3] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack Magazine* 49 (November 1996), http://www.phrack.org/49/P49-14.
- [4] Crispin Cowan, personal communication, January 2004.
- [5] Crispin Cowan and Perry Wagle, "StackGuard: Simple Stack Smash Protection for GCC," *Proceedings of the GCC Developers Summit* (May 2003).
- [6] Crispin Cowan et al., "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000
- $\label{thm:continuous} \parbox{0.5cm} \parbox{0.5cm} See \parbox{0.5cm} \parbox$
- [8] Solar Designer, "Non-Executable User Stack," http://www.openwall.com/linux/.
- [9] Solar Designer, "Getting Around Non-Executable Stack (and Fix)." http://www.securityfocus.com/archive/1/7480.
- [10] Rafal Wojtczuk, "Defeating Solar Designer's Non-Executable Stack Patch" (January 1998), http://www.securityfocus.com/archive/1/8470.
- [11] Monica Chew and Dawn Song, "Mitigating Buffer Overflows by Operating System Randomization," *Tech Report CMU-CS-02-197* (December 2002).
- [12] See http://pax.grsecurity.net/docs/index.html.
- [13] Nergal, "The Advanced return-into-lib(c) Exploits: PaX Case Study," *Phrack Magazine* 58 (December 2001), http://www.phrack.org/phrack/58/p58-0x04.

- [14] John R. Levine, Linkers and Loaders (San Diego: Academic Press, 2000).
- [15] Anonymous, "Bypassing PaX ASLR Protection," *Phrack Magazine* 59 (July 2002), http://www.prhack.org/phrack/59/p59-0x09.
- [16] Crispin Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," 7th USENIX Security Symposium (January 1998), pp. 63–77.
- [17] Bulba and Kil3r, "Bypassing StackGuard and StackShield," *Phrack Magazine* 56 (May 2000), http://www.phrack.org/phrack/56/p56-0x05.
- [18] Hiroaki Etoh, "ProPolice: GCC Extension for Protecting Applications from Stack-Smashing Attacks," IBM (April 2003), http://www.trl.ibm.com/projects/security/ssp/.
- [19] Matt Conover, "w00w00 on Heap Overflows" (January 1999), http://www.w00w00.org/files/articles/heaptut.txt.
- [20] Michel Kaempf, "Vudo Malloc Tricks," *Phrack Magazine* 57 (August 2001), http://www.phrack.org/phrack/57/p57-0x0b.
- [21] Anonymous, "Once upon a free()," *Phrack Magazine* 57 (August 2001), http://www.phrack.org/phrack/57/p57-0x0c.
- [22] Gerardo Richarte, "Bypassing the StackShield and StackGuard Protection" (April 2002), http://www1.corest.com/corelabs/papers/index.php.
- [23] Crispin Cowan et al., "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities," 12th USENIX Security Symposium (August 2003), pp. 91–104.
- [24] Crispin Cowan, "Re: PointGuard: It's not the Size of the Buffer, it's the Address." http://www.securityfocus.com/archive/1/333988.
- [25] Pascal Bouchareine, "Format String Vulnerability" (July 2000), http://www.hert.org/papers/format.html.
- [26] scut and Team Teso, "Exploiting Format String Vulnerabilities" (September 2001), http://www.team-teso.net/articles/formatstring/.
- [27] gera and riq, "Advances in Format String Exploitation," *Phrack Magazine* 59 (July 2002), http://www.phrack.org/phrack/59/p59-0x12.
- [28] Steven Alexander, "Improving Security with Homebrew System Modifications," ;login:, vol. 29, no. 6 (December 2004), pp. 26–32.
- [29] Crispin Cowan et al., "FormatGuard: Automatic Protection from printf Format String Vulnerabilities," 10th USENIX Security Symposium (August 2001).