Chapter 18

Improving security with patches

Computer systems become safer and safer over the course of time. The security holes that are discovered are patched immediately, and as a result the code quality rises continuously. A hacker has very little time to exploit a gap in vulnerable software. If he doesn't manage to do this before the security hole is patched by the administrator, his entire effort in writing an exploit will vanish into thin air. Even if the administrator is lazy and forgets to update the software, the ready-made scripts contained in current distributions will do it for him.

But what happens if a hacker finds an error and doesn't tell anybody about it? In such a situation no update will protect us, nor will keeping up-to-date with security-related discussion boards. Only special patches can protect us against attacks on undocumented errors, increasing the overall security of our system, not just that of individual applications.

As we already know, every operating system possesses a kernel. The kernel is responsible for communication with the hardware, allocation of memory, and other operations requiring full access to the processor. It's the kernel that manages all system processes, performs memory operations, and classifies their execution. Let's assume that a program that has been started up contains an error. A hacker exploits it, takes control over the program, and gains its rights. If the process was started up by a root user, the hacker gains administrator privileges. There are several things we can do to protect ourselves against such an incident:

a) We can patch the program containing the error.

b) We can patch the system in such a way that it will protect against this type of attack.

The first solution is effective if anybody, apart from the hacker, knows about the existence of the error. Otherwise, we won't even be aware of the fact that the hacker has used this program to break in. The second possibility definitely gives better results. Instead of patching each program, we can obtain the same effect protecting the whole system against attacks of this kind. The term "system" instead of "kernel" has been used here on purpose, because we can divide this type of protection into several types:

a) Protection on the system kernel level.
b) Protection on the compiler level.
c) Protection on the library level.

Let's stop and think how we can protect our program against attacks from the kernel level. The attack most frequently encountered against applications is the buffer overflow attack, which was described in an earlier chapter of this handbook. It consists in overwriting parts of the program memory and starting up one's own code. This code is most frequently used to start up the shell with root privileges and is transferred to the program through its arguments. The arguments of each process are located in the memory area known as the stack. The hacker will aim to start up his code from this memory segment. As we know already, the kernel manages the process memory. The kernel defines some flags (tags) that specify whether a specific memory fragment can be written, read, or started up. In the standard configuration of the Linux kernel, the stack contains each of these flags. Therefore, there is nothing to stop someone from starting up their own code located on the stack. The main task of security patches written for the kernel is to configure the process memory to prevent the startup of user code on the process level. This and many other useful functions are contained in two popular patches, which will be described later on in this chapter.

We know already that we can protect the process memory to prevent a hacker's code from starting up. However, we can protect our programs so that it will generally be impossible to change the function of the process. The most frequently overwritten memory area, which decides which code executes the program, is the stack frame. It contains information about from which function the currently running procedure was called. Because of this, a return to the main function, after terminating the one currently running, is possible. However, if for some reason, due to a programmer's error, the hacker can manage to overwrite the stack frame with his own data, he can cause the program to jump to the harmful code, and not to the main function. The security patches applied to the compiler can protect us against overwriting of the frame. In this way, any program we compile won't be susceptible to the buffer overflow error. We will have a closer look at this protection method a little bit later.

When we program in the C language, we have many standard functions at our disposal, such as printf(), strcpy(), and gets(). They are not located in the code of our program, but in the library of the C language, which is libc. This is a dynamic library, which means that it is loaded automatically after starting up our program, so it can use the library functions. It is mainly because these functions don't contain any protections that the buffer overflow and format string attacks are possible. There are libraries available that check the security of the transferred parameters of these functions, and by using them it is possible to avoid overwriting important data. We will discuss this method in more detail later in this chapter.

The first method of system protection, which we will look at more closely, will be security patches applied to the kernel. These have become more and more popular, and are now standard in many distributions of the Linux system.

Grsecurity

Grsecurity is without doubt the most complex and popular patch for the system kernel. It uses a memory protection system called PaX. We can use it as a separate patch, downloading it from:

```
http://pax.grsecurity.net/
```

However, it's better to invest in the whole grsecurity packet, of which the memory protection system is only a small part. The following are the main elements of this packet:

a) Memory protection through PaX
b) Randomization of memory and the TCP/IP stack
c) Limited display of processes
d) Increased protection in the chroot environment
e) Protection against a race condition in /tmp

We will have a closer look at each of these elements during the configuration of our patch. But first we will consider the installation process.

Therefore, we download the newest version of the Linux kernel. It can be found at:

```
http://www.kernel.org
```

Currently the newest version from the 2.6 branch is 2.6.29.4. Our tests will be performed on this kernel version. We go to the folder /usr/src, where we should store the system resources as standard. It is good to stick to this rule. Next, we download the compressed resources using the wget tool:

```
wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.29.4.tar.bz2
```

After a while we will have the resources for our new kernel. Now it's time to equip ourselves with the grsecurity patch. Therefore we go to the project page and download the patch appropriate for our kernel (that is, 2.6.29.4):

```
wget http://www.grsecurity.net/grsecurity-2.1.14-2.6.29.4-200905292205.patch
```

Patching and configuring a new kernel

We have now all the required elements. Now it's time to unpack the system sources and to apply the patch:

```
bash-2.05b# tar xfj linux-2.6.29.4.tar.bz2
bash-2.05b# patch -p0 < ./grsecurity-2.1.14-2.6.29.4-200905292205.patch
patching file linux-2.6.29.4/Documentation/Configure.help
patching file linux-2.6.29.4/Makefile
patching file linux-2.6.29.4/arch/alpha/config.in
patching file linux-2.6.29.4/arch/alpha/kernel/osf_sys.c
(...)
patching file linux-2.6.29.4/net/netsyms.c
patching file linux-2.6.29.4/net/socket.c
patching file linux-2.6.29.4/net/sunrpc/xprt.c
patching file linux-2.6.29.4/net/unix/af_unix.c
bash-2.05b#
```

The patch program, logically enough, applies patches. We transfer the patch content to the patch program's standard input (using the tag <), where it applies the patch to the application. If the program returns an error, it can mean that the version of our patch is different from the system version on which we want to install it. In such a case, we have to go the grsecurity page and make sure we have the appropriate patch.

Now we go to the grsecurity configuration. The best solution will be to download an example configuration file from the page:

```
http://www.grsecurity.net/generic-config
```

This contains the most frequently used options. After downloading this file to the folder /usr/src we go to the directory with the system resources (/usr/src/linux-2.6.29.4) and call the command:

```
make menuconfig
```

After terminating the compilation process we will see the console menu. At the very bottom we choose the option:

```
Load an Alternate Configuration File
```

Next, we enter the path to our configuration file (/usr/src/generic-config). We have to configure the kernel depending on the hardware that we have. It's worth reviewing all the sections so we won't forget to compile the services of important devices. We will focus on the last section, Grsecurity. The star between brackets [*] means that an option has to be selected. If there is no star, we deselect the option. The following fields are located in this sector:

```
a) [*] Grsecurity – We select this to activate the grsecurity service in the kernel.
b) (Customized) Security level – Options regarding the grsec protection level
      ( ) Low – Low protection level
      ( ) Medium – Medium protection level
      ( ) High – High protection level – in this case some programs such as X
          Window System can stop working.
      (X) Customized – Protection level defined by the user. We select this one.
c) PaX Control ---> These options adopt the loading of system binaries for the PaX
patch.
      [ ] Support soft mode
      [*] Use legacy ELF header marking
      [*] Use ELF program header marking
          (none) MAC system integration
d) Address Space Protection ---> This selection allows us to configure some patch
options.
      [*] Enforce Non-executable pages (NEW) – Forcibly creates non-executable areas
          of memory.
      [*] Paging based non-executable pages (NEW) – Method of creation of non-executable
          memory pages.
      [*] Segmentation based non-executable pages (NEW) – As above.
      [ ] Emulate trampolines (NEW)
      [*] Restrict mprotect() (NEW) – Does not allow programs to change the rights of
          the memory pages (e.g., in order for them to be executable).
      [ ] Disallow ELF text relocations (DANGEROUS) (NEW)
      [*] Address Space Layout Randomization (NEW) – Randomization of addresses.
      [*] Randomize kernel stack base (NEW) – Randomizes the address of the kernel
          stack.
      [*] Randomize user stack base (NEW) – Randomizes the address of the user stack.
          In this way the exploitation of the stack overflow errors is rendered very
          difficult.
         At each startup of the program, the stack address changes, and at the same time
         the address of the cracker's injected code changes.
      [*] Randomize mmap() base (NEW) – Randomizes the address returned by the
          function mmap(). In this way the exploitation of heap overflow errors is
          rendered very difficult.
      [*] Randomize ET_EXEC base (NEW) – Randomizes the mapping location of the
          program started up. As in the previous cases it serves to make the hacker's
          actions more difficult.
      [*] Deny writing to /dev/kmem, /dev/mem, and /dev/port – Using the devices
          /dev/kmem, /dev/mem the root user has access to the system kernel memory,
          which means that he can modify them freely. If an attacker manages to gain
          such privileges, he can, using these devices, introduce a backdoor to the
          kernel.
          The device /dev/port makes possible the access to the computer hardware, as
          well as to the disk and to the RAM memory. Exploitation of this file can also
          contribute
```

```
    [ ] Disable privileged I/O
    [*] Remove addresses from /proc/pid/[maps|stat] – In the file /proc/self/maps
        the addresses of the uploaded libraries of the process currently started up, are
        located. One of them is for example libc. Thanks to this address, it is
        possible to perform the attack return into libc, against which even a non-
        executable stack cannot protect fully.
    [*] Deter exploit bruteforcing – Detects an attempt of use of brute-force
        exploits.
    [ ] Hide kernel symbols
e) Role Based Access Control Options ---> Allows the configuration of the RBAC control
        system.
    [*] Hide kernel processes – Hides processes belonging to the system kernel.
        (3) Maximum tries before password lockout – Maximum number of attempts to enter
        the password before it is blocked.
        (30) Time to wait after max password tries, in seconds - Time to wait to
        introduce the password.
f) Filesystem Protections ---> Functions related to the file system protection.
    [*] Proc restrictions – Thanks to this option, users cannot spy on each other's
        processes through the /proc file system.
    [ ] Restrict to user only
    [*] Allow special group – Using this option, we will be able to choose a
        group, whose users will see all processes in the system.
        (1001) GID for special group – The number of this group is in our case 1001.

    [*] Additional restrictions – Using this, normal users won't be able to see
         information on the hardware, which are located in the files in the folder
        /proc (e.g., /proc/cpuinfo).
    [*] Linking restrictions – This option forbids creating symbolic links in the
        /tmp folder. This protects against race condition attacks.
    [*] FIFO restrictions – As above, it protects against race condition, but this
        time with use of FIFO queues.
    [*] Chroot jail restrictions – Options restricting the rights of a user located
        in a separated chroot environment. The information on using chroot can be
        found on the system manual pages (man chroot). All options below concern it.
    [*] Deny mounts – Disallows mounting devices.
    [*] Deny double-chroots – Disallows executing a double chroot.
    [*] Deny pivot_root in chroot – Disallows use of the pivot_root() function.
    [*] Enforce chdir("/") on all chroots – Assumes „/" as main directory of each
        chroot.
    [*] Deny (f)chmod +s – Disallows setting of the suid and sgid bits.
    [*] Deny fchdir out of chroot – Disallows using the fchdir function. Allows
        preventing a situation, in which a chroot by-pass is possible.
    [*] Deny mknod – Disallows creating devices.
    [*] Deny shmat() out of chroot - The shmat() function is used to add a memory
        segment. The program in chroot could add a program segment, which is not to be
        found in it and draw information from it. Therefore, this function is blocked.
    [*] Deny access to abstract AF_UNIX sockets out of chroot – Blocks access to
        the AF_UNIX sockets, because they allow communication between processes.
    [*] Protect outside processes – This function completely isolates the process
        in the chroot environment. It cannot send signals or perform any other
        operations on processes that are not within it.
    [*] Restrict priority changes – Disallows changing the process priority.
    [*] Deny sysctl writes in chroot – Disallows using the sysctl function.
    [*] Capability restrictions within chroot – With this option the process
        working in chroot will loose all privileges available to the root user.
```

```
g) Kernel Auditing ---> These options don't grant security by themselves, however, using
         them we can obtain a lot of useful information on the system functioning.
         They allow logging various function calls (sometimes dangerous).
     [ ] Single group for auditing – After selecting this option we can set the
         number of the group to be logged.
     [ ] Exec logging – Enables logging of all calls of the exec() function. It is
         executed very frequently during the system functioning, therefore, it is not
         worth selecting it.
         It will only cause us to be bombarded with useless information.
      [*] Resource logging – Logs all cases, in which the process exceeds the access
         limit to the system resources.
      [ ] Log execs within chroot – Logs start up of the programs in the chroot
           environment.
      [ ] Chdir logging – Logs changes to the current directory.
      [ ] (Un)Mount logging – Logs mounting and unmounting of devices.
      [ ] IPC logging – Logs when the IPC mechanisms are used by the processes (they
           allow communication between processes).
      [*] Signal logging – Logs all sent signals.
      [*] Fork failure logging – Logs the calls of the fork() function, which end with
           an error.
      [*] Time change logging – Logs changes to the system time.
      [ ] /proc/<pid>/ipaddr support – The option creates a new file in the /proc/pid
          folder, which shows sockets opened by a given process.
      [ ] ELF text relocations logging (READ HELP) – Option connected with the change
           of the content of the programs compiled.
h) Executable Protections ---> These functions regard the functioning of the system
                               processes.
      [*] Enforce RLIMIT_NPROC on execs – Checks if the process limit has been
          exceeded when the exec() function has been called. As standard such a
          test is performed only when using the fork() function.
      [*] Destroy unused shared memory (NEW) – Destroys the shared memory, which is  not
          used by any process.
      [*] Dmesg(8) restriction – After switching on this option a normal user won't be
           able to display more than 4kb of the kernel message buffer. To display it
           the dmesg program is used.
      [*] Randomized PIDs – This option chooses a random PID number for the newly
          created processes. As standard they are chosen in sequence.
      [ ] Trusted path execution – After choosing this option, we can determine the
           number of the group, which the set limits won't regard.
i) Network Protections ---> This section allows randomization of some elements of the
                            TCP/IP stack. It can allow, for example, protection against
                            the remote identification of the system version.
        [*] Larger entropy pools – Increases the memory size used by the system, and is
        useful for generating of random values.
        [*] Truly random TCP ISN selection – After activation of this option the field
        initiating      the TCP sequence will be chosen randomly.
        [*] Randomized IP IDs – The id field of the IP header will be chosen randomly.
        [*] Randomized TCP source ports – The source ports will be random while making a
        connection, not, as is standard, in sequence.
        [*] Randomized RPC XIDs – The XID field of the RPC request will be chosen
        randomly.
        [ ] Socket restrictions – These options will allow limitations on the creation
        of sockets.
        We can for example define a group that will be able to create sockets for
       servers or clients, whereas we can prevent the rest of the users from doing this.
j) Sysctl support ---> This option allows activation of the sysctl service for
   grsecurity. In this way we will be able to modify the functioning of the patch right
   after compiling the kernel.
        [ ] Sysctl support – At our discretion, we can activate this option or not. If
        we activate it the root user will be able to change the grsec configuration
        with use of the sysctl command, which also is potentially dangerous.
```

```
k) Logging options ---> Options related to the frequency of sending messages (logs).
        (10) Seconds in between log messages (minimum) – Frequency of writing logs.
        (4) Number of messages in a burst (maximum) – Maximum number of messages
        given simultaneously.
```

These are all the options of the grsecurity packet. As we can see it is a huge packet, significantly improving system security. We can quit the configurator, and when prompted we should save the configuration.

In case we use a generic configuration file, before compiling we need to copy generic-config to the root directory /usr/src/linux-2.6.29.4/ under the name .config.

```
bash-2.05# cp /usr/src/generic-confi g /usr/src/linux-2.6.29.4/.confi g
```

Attention.
To compile a new kernel with the grsecurity patch in Training Operating System successfully we also need to update binutils package to version 2.18. The latest version of the source can be found at:

```
http://ftp.gnu.org/gnu/binutils/
```

The process of configuring and compiling the binutils package can be conducted as follows:

```
bash-2.05b# tar xfj binutils-2.19.tar.bz2
bash-2.05b# cd binutils-2.19
bash-2.05b# ./configure --prefi x=/usr
bash-2.05b# make
bash-2.05b# make install
```

The Training Operating System includes a standard version of gcc v4.1.1 compiler. However our 2.6.29.4 kernel requires a newer version of gcc (eg v4.4) or carrying out certain modifications in one of the headers. Therefore we can download a new version of gcc from repository or delete line 5, 6 and 7 from the file /usr/src/linux-2.6.29.4/include/linux/compiler-gcc4.h. Nevertheless, please remember that this is an experimental modification and it is better to obtain the latest versions of the required applications.

Compiling a kernel with grsecurity patch

Now we can start the process of compilation of our new system. If the reader is familiar with the subject of kernel compilation, he can without hesitation skip this part.

The compilation process can take several hours, depending on the speed of the computer. We give the command:

```
make bzImage
```

After executing this command we can leave the machine alone for some time. The image of our kernel will be found in the directory /usr/src/linux-2.6.29.4/arch/i386/boot/bzImage.

Before we take care of this, we have to compile and install the kernel modules, executing a series of commands:

```
make modules
make modules_install
```

The duration of this process depends on the number of modules selected during the kernel configuration. Next, we compile the kernel image to the /boot folder with the command:

```
bash-2.05b# cp /usr/src/linux-2.6.29.4/arch/i386/boot/bzImage /boot/linux-2.6.29.4-grsec
```

Depending on whether we compiled the modules into the kernel or not, we start up the geninitrd program. This creates a file with modules that should upload right after the kernel, before mounting the disk (i.e., the most important drivers).

```
bash-2.05b# mkinitrd /boot/initrd-2.6.29.4-grsec 2.6.29.4-grsec
```

Next, depending on the bootloader used, we create an entry in the configuration file. Below an example entry to the file /etc/lilo.conf:

```
image=/boot/linux-2.6.29.4-grsec
```

```
root=/dev/hda1
label=2.6.29.4-grsec
initrd=/boot/initrd-2.6.29.4-grsec
```

After adding an entry we start up the lilo program to confirm the changes:

```
bash-2.05b# lilo
Added 2.6.29.4-grsec *
Added 2.6.29.4-old
```

If you are using Grub boot loader, similar configuration entry in /boot/grub/menu.lst could look like this:

```
title 2.6.29.4-grsec
root (hd0,0)
kernel /boot/linux-2.6.29.4-grsec root=/dev/hda1
initrd /boot/initrd-2.6.29.4-grsec
```

All that's left to do is give the reboot command. It is also good to leave the old kernel image in case the system won't load, for example due to not selecting the appropriate drivers in the configuration file. Therefore we restart the system and load the new kernel:

```
bash-2.05b# reboot

Broadcast message from root (pts/2) (Wed Jan 19 13:48:27 2009):
The system is going down for reboot NOW!
(...)
```

From outside, the new system doesn't differ much from the old one. To test if the patch really works, we will use the PaXtest program. It will test the susceptibility of our system to various kinds of attacks. Currently its newest version is 0.9.7-pre4:

```
bash-2.05b# wget http://pax.grsecurity.net/paxtest-0.9.7-pre4.tar.gz
```

Then we unpack the archive and compile the sources using a series of commands:

```
bash-2.05b# tar zxf paxtest-0.9.7-pre4.tar.gz
bash-2.05b# cd paxtest-0.9.7-pre4
bash-2.05b# make generic
```

```
make -f Makefile.generic
make[1]: Entering directory '/home/user/paxtest-0.9.7-pre4'
(...)
(...)
(...)
sh genpaxtest anonmap execbss execdata execheap execstack mprotanon mprotbss mprotdata
mprotheap mprotshbss mprotshdata mprotstack randamap randheap1 randheap2 randmain1
randmain2 randshlib randstack1 randstack2 rettofunc1 rettofunc1x rettofunc2 rettofunc2x
shlibbss shlibdata writetext
make[1]: Leaving directory `/home/user/paxtest-0.9.7-pre4'
bash-2.05b#
```

So, let's test the functionality of the newly applied patch. We start up the paxtest program located in the current directory:

```
bash-2.05b# ./paxtest
PaXtest - Copyright(c) 2003, 2004 by Peter Busser <peter@adamantix.org>
Released under the GNU Public Licence version 2 or later

It may take a while for the tests to complete
Test results:


Executable anonymous mapping                      : Killed
Executable bss                                    : Killed
Executable data                                   : Killed
Executable heap                                   : Killed
Executable stack                                  : Killed
Executable anonymous mapping (mprotect)           : Killed
Executable bss (mprotect)                         : Killed
Executable data (mprotect)                        : Killed
Executable heap (mprotect)                        : Killed
Executable shared library bss (mprotect)          : Killed
Executable shared library data (mprotect)         : Killed
Executable stack (mprotect)                       : Killed
Anonymous mapping randomisation test              : 16 bits (guessed)
Heap randomisation test (ET_EXEC)                 : 13 bits (guessed)
Heap randomisation test (ET_DYN)                  : 25 bits (guessed)
Main executable randomisation (ET_EXEC)           : No randomisation
Main executable randomisation (ET_DYN)            : 17 bits (guessed)
Shared library randomisation test                 : 16 bits (guessed)
Stack randomisation test (SEGMEXEC)               : 23 bits (guessed)
Stack randomisation test (PAGEEXEC)               : 23 bits (guessed)
Return to function (strcpy)                       : Vulnerable
Return to function (strcpy, RANDEXEC)             : Vulnerable
Return to function (memcpy)                       : Vulnerable
Return to function (memcpy, RANDEXEC)             : Vulnerable
Executable shared library bss                     : Killed
Executable shared library data                    : Killed
Writable text segments                            : Killed
bash-2.05b#
```

As we can see, PaX doesn't allow us to start up the code in any segment of the memory. A program wanting to perform such an operation is immediately closed (killed). The only attack that could be executed is a return to the function belonging to the program (or its library). Unfortunately the grsecurity patch doesn't protect us against an attack of this kind. If the program itself contains a code that could, for example, start up the bash shell, the attacker will be able to perform a jump into it and to start it up, after overwriting the required memory areas. PaX cannot forbid starting up potentially dangerous functions, because it is a part of the program code. The attack technique called return to libc library is based around this gap. Instead of overwriting memory with the address of his code, which won't be started up anyway, the hacker overwrites it with the address of, for example, the system() function located in the libc library. In this way he can perform any command through the bash shell. Luckily, grsecurity contains a system of randomization of addresses of the loaded libraries, which makes the attack more difficult, although still possible – the hacker must only dedicate more time than usual to the attack.

Now our system protects processes against buffer overflow attacks on the kernel level. It is time to secure the next important element – the compiler.

Stack-Smashing Protector

The C language is over 30 years old, and is one of the oldest programming languages. Despite its age it is still clearly the most frequently used language in big projects, where performance is a priority. This language is compiled, meaning converted by a program called a compiler to a language understandable to a processor. The compiler takes care so the code is the smallest possible and has the best performance. Therefore, it doesn't contain any supplements, including those related to security. Many people think that we have to deal with so many errors because of the fact that many programmers still use C. There is some truth in this. Higher level languages, such as Java and Python, secure programs against overwriting the stack frame (or don't use it at all). However, they themselves have many disadvantages, notably lower performance. So it is good to program both securely and effectively. But, as commonly known, this is a big challenge for the programmer. Even programs like the Apache server and PHP, which are widely used and well respected, have had buffer overflow errors in their history. In certain situations they are difficult to avoid. Recently, patches for the C language compiler, GCC, protecting the program against such attacks, have become popular. Tests show only a marginal reduction in program performance, while providing much greater security. Therefore the Stack-Smashing Protector patch has also been included in many distributions of the Linux system in which security has priority.

So let's apply this patch to our compiler and to see how it works. At the beginning we download it from the publisher's page:

```
http://www.research.ibm.com/trl/projects/security/ssp/
```

On this page we can also find much useful information on the project. Currently the newest SSP version is 3.4.4; we have to put it on the GCC version with the same number.

```
bash-2.05b# wget
http://www.research.ibm.com/trl/projects/security/ssp/gcc3_4_4/protector-3.4.4-1.tar.gz
```

It doesn't take a lot of space, and the downloading process is complete after a short while. Now, it's time to download the GCC compiler sources. We get it from:

```
bash-2.05b# wget ftp://sunsite.icm.edu.pl/pub/gnu/gcc/gcc-3.4.4/gcc-3.4.4.tar.bz2
```

We now have all the required sources. Now we have to unpack the downloaded GCC compiler sources and to apply the SSP patch:

```
bash-2.05b# tar jxf gcc-3.4.4.tar.bz2
bash-2.05b# mv protector-3.4.4-1.tar.gz gcc-3.4.4
bash-2.05b# cd gcc-3.4.4
bash-2.05b# tar zxf protector-3.4.4-1.tar.gz
```

We have unpacked the GCC sources and transferred the patch into the folder where they are located.

All that remains to be done is to apply the patch using the standard command:

```
bash-2.05b# patch -p0 < gcc_3_4_4.dif
patching file gcc/Makefile.in
patching file gcc/c-cppbuiltin.c
patching file gcc/calls.c
patching file gcc/combine.c
patching file gcc/common.opt
patching file gcc/configure
patching file gcc/cse.c
patching file gcc/explow.c
patching file gcc/expr.c
patching file gcc/flags.h
patching file gcc/function.c
patching file gcc/gcse.c
patching file gcc/integrate.c
patching file gcc/libgcc-std.ver
patching file gcc/libgcc2.c
patching file gcc/loop.c
patching file gcc/mklibgcc.in
patching file gcc/optabs.c
patching file gcc/opts.c
patching file gcc/reload1.c
patching file gcc/rtl.h
patching file gcc/simplify-rtx.c
patching file gcc/toplev.c
patching file gcc/tree.h
patching file gcc/config/t-linux
patching file gcc/config/arm/arm.md
patching file gcc/doc/invoke.texi
bash-2.05b#
```

No information about any error has appeared, meaning that the patch has been applied correctly. Now we have to configure GCC so it uses the stack protection option. Therefore we create a folder in which the compiled files will be located and we go to it:

```
bash-2.05b# mkdir obj; cd obj
```

Then we unload the configure program with the options described below:

a) --enable-languages=c,c++,objc – We want GCC to compile the languages C, C++, and Object C. We use this option only to shorten the compilation time of the GCC itself. As standard it serves several languages. But to us the three most common will be sufficient.

b) --prefix=/usr/pp - Tells GCC to place its compiled binaries in the folder /usr/pp.
c) --enable-stack-protector – The most important option for us. It configures GCC so it will be compiled with the stack protection that is using our patch.

```
bash-2.05b# ../configure --enable-languages=c,c++,objc --prefix=/usr/pp
                         --enable-stack-protector
creating cache ./config.cache
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
(...)
updating cache ./config.cache
creating ./config.status
creating Makefile
bash-2.05b#
```

After configuration, the Makefile file has been created, which will be useful for the compilation of GCC. Now, all we need to do is to start up the compilation process:

```
bash-2.05b# make bootstrap-lean
```

The compilation can take a relatively long time, depending on the performance of our computer. GCC is a huge application, and the size of its resources is comparable to that of the system kernel. We can therefore safely go away from the computer for several minutes. After terminating the compilation of our new compiler we can start the installation in the /usr/pp directory:

```
bash-2.05b# make install prefix=/usr/pp
/bin/sh ../mkinstalldirs /usr/pp /usr/pp
make[1]: Entry to the directory `/usr/src/gcc-3.4.4/obj/gcc'
(...)
make[2]: There is nothing to do in `install'.
make[2]: Directory abandonment `/usr/src/gcc-3.4.4/obj/i686-pc-linux-
gnu/libiberty/testsuite'
make[1]: Directory abandonment `/usr/src/gcc-3.4.4/obj/i686-pc-linux-gnu/libiberty'
bash-2.05b#
```

This process shouldn’t take too long, because it consists only of copying files to the target folder.

Let’s have a look which folders have been created in the /usr/pp directory:

```
bash-2.05b# ls
bin  include  info  lib  libexec  man  share
bash-2.05b#
```

Their destination is the following:

```
a) bin             - This folder contains a compiled GCC version with a patch applied.
b) include         - These are the header files for programs written in the language C++.
c) info            - Here help files for the info program are located.
d) lib & libexec  - Libraries necessary during GCC work.
e) man             - Help pages of the compiler man.
f) share           - Local files, used by GCC, e.g., when informing about errors.
```

However, we are interested only in the functioning of our compiler. Therefore we go now to the bin directory and have a look at its content:

```
bash-2.05b# cd bin
bash-2.05b# ls
c++   cpp  gcc  gcov  i686-pc-linux-gnu-g++  i686-pc-linux-gnu-gcc-3.4.4
test.c  core  g++ gccbug  i686-pc-linux-gnu-c++  i686-pc-linux-gnu-gcc  test
bash-2.05b#
```

Here the GCC binaries are located that we will use to compile the test program. The installation process has also created a source code with the name test.c. Let's have a look at it (**/CD/Chapter18/Listings/test.c**):

```
int main(int argc, char *argv[])
{
    char buf[8];
    strcpy(buf, argv[1]);
    return 0;
}
```

This program creates a table of characters with eight elements (char buf[8]). Then it copies to it the content of the first argument transferred by the user, without checking its size. This is a standard example of buffer overflow, and an opportunity for an attack, against which the applied patch should protect us. We should test the program, compiling it using the standard GCC version. But we have to be sure that we have started up the old version of the compiler, and not the new one, which is located in the current directory.

We will refer to /usr/bin/gcc.

```
bash-2.05b# /usr/bin/gcc -o test test.c
test.c:6:2: warning: no newline at end of file
```

We have compiled a test program using GCC without the patch. We don't have to worry about the warning it returns, because this always appears when the source code doesn't contain a new line character at the end. We will now try to transfer such a long character sequence to the test program that it will cause the overwriting of the stack frame:

```
bash-2.05b# ulimit -c 1500000000
bash-2.05b# ./test AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

As we can see, we performed the ulimit command, which orders the programs to generate a memory discharge in the form of the core file after the appearance of an error. Our test application terminated with an error, after entering too many "A" characters in the first argument. Now we can check with gdb and the core file to see why this happened:

```
bash-2.05b# gdb test core
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host= --target=i686-pld-linux"...
Core was generated by `./test AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.

warning: current_sos: Can't read pathname for load map: Input/output error

Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in?? ()
(gdb)
```

We can see that our program overflowed when attempting to access the memory located at address 0x41414141, which corresponds to the "AAAA" characters.

Let’s check how our stack frame looks:

```
(gdb) info frame
Cannot access memory at address 0x41414141
(gdb)
```

After overwriting the frame of the main() function with the “A” values, the program returned to the previous function, thinking that the original frame was located right under this address. It happened this way, because we overwrote the copy of the EBP register that was responsible for this. The program also wanted to jump to the address 0x41414141, because we overwrote a copy of the EIP register. The processor executes the code from this address, which is located in this register.

```
(gdb) print $eip
$1 = (void *) 0x41414141
(gdb) print $ebp
$2 = (void *) 0x41414141
```

As we can see, thanks to overwriting the register copies located in the function frame, erroneous values were entered for both of them. If a code added by a hacker were located under the address 0x41414141, it would be executed. In our case the program terminated the action with an error, because the memory from this address is not available for the process. Attempting access to this memory area led to the process being killed by the system kernel.

Now we know how easy it is to overwrite a function frame and which consequences it can have. The SSP patch should protect us against attacks of this type. Let’s check it. We compile our test program, this time using the patched GCC version (located in the directory /usr/pp/bin, our current path):

```
bash-2.05b# ./gcc -o test test.c
test.c:6:2: warning: no newline at end of file
bash-2.05b# ./test AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
test: stack smashing attack in function main
Interrupted (core dumped)
bash-2.05b#
```

As we can see, this time the program didn’t terminate with an error, but it interrupted itself. Thanks to the security patch, the program detected the overwriting of the function frame and sent a SIGKILL signal to itself, ordering itself to end the action immediately. In this case, a core file was also generated, so let's investigate it:

```
bash-2.05b# gdb test core
GNU gdb 5.2.1
(...)
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x40046841 in kill () from /lib/libc.so.6
(gdb)
```

Our program terminated with the kill() function, which means that this function was the last to be performed.

In the Training Operating System environment, the process of compiling using the SSP could look like this:

```
bash-2.05b# gcc -fstack-protector -o test test.c
test.c:6:2: warning: no newline at end of file
bash-2.05b# ./test AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: ./test terminated
Interrupted (core dumped)
bash-2.05b#
```

The Stack-Smashing Protector security patch not only protects against the buffer overflow error, but also against exploiting this error. Because of this patch the attacker can, at most, only freeze the program using the error, which usually doesn't give him any benefit. A significant security improvement, ensured by SSP, also brings a drop in performance. Therefore its authors decided that the only protected functions would be those that contain the char tables, because it's mainly during operations performed on them that the buffer overflow error occurs. We will now try to modify our test program slightly, so the data are copied to the int table and not to char (**/CD/Chapter18/Listings/test2.c**):

```
int main(int argc, char *argv[])
{
    int buf[8];
    strcpy((char*)buf, argv[1]);
    return 0;
}
```

The strcpy() function assumes only arguments being the char pointers, therefore we had to apply typecasting using (char*)buf in the first parameter. The int variables have the size of four bytes, whereas char has only one. The number of “A” characters we will enter in the test will also have to be greater to overflow the buffer and to overwrite the function frame. We compile and test our program:

```
bash-2.05b# ./gcc -o test2 test2.c
test.c:6:2: warning: no newline at end of file
bash-2.05b# ./test2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAA
Segmentation fault (core dumped)
bash-2.05b#
```

This time the program ended with a memory protection violation error.

Let’s examine the core file:

```
bash-2.05b# gdb test2 core
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
(...)
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x41414141 in?? ()
(gdb)
```

As we can see we managed to overwrite the registers’ copies without problems, despite using the GCC program with the applied security patch for the compilation. Luckily this can be remedied. Stack-Smashing Protector contains the compilation option -fstack-protector-all, which protects all functions, not just those containing the char tables. We will now compile our test program with its use and will test it again:

```
bash-2.05b# ./gcc -o test2 test2.c -fstack-protector-all
test.c:6:2: warning: no newline at end of file
bash-2.05b# ./test2 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA
test: stack smashing attack in function main
Interrupted (core dumped)
bash-2.05b#
```

The program intercepted an attempt to overwrite the function frame and immediately terminated its action. We can check it also using gdb and the core file. The result will be exactly the same as in the case of the char table function. It is therefore good to add this option to each program that we compile.

We now know how to use a patched version of our compiler. We have seen that the SSP patch protects effectively against buffer overflow attacks. However, it would also be good to discover how it really works.

To detect the violation (overwriting) of the stack, SSP uses an additional variable called the guard. This variable is placed in each function, right before the data whose overwriting can have disastrous effects (e.g., the stack frame).

This process progresses in the following way:

a) Declaration of the guard variable immediately under the stack frame for the given function.

```
int guard;
```

b) At the beginning of the function body, a random value is assigned to the guard variable. This is a value generated by /dev/urandom or /dev/random.

```
guard = guard_value;
```

c) At the end of the function a comparison of the guard variable value with the original value occurs. If those values correspond to each other, it means that a buffer overflow has taken place in the program. In such a situation information about the incident is printed and program execution is interrupted.

```
  if (guard != guard_value) {
        /* informing about the incident */
        /* action termination */
  }
```

Let’s have a closer look at an example stack structure, right after starting the execution of a function:

Function arguments
Function return address
Previous frame pointer
Guard
Tables
Other local variables

The overflow can happen only in the location marked “Tables.” The SSP patch is so clever that it alone changes the order of the local variables function so that it is impossible to overwrite anything else.

We have an example of variable declaration below:

```
char *p;
int ok;
char buf[8];
```

If a buffer overflow error were to be found in the program code, it would be possible to overwrite the ok variable and the p pointer, which could have disastrous effects (depending on the function of these variables). Therefore the Stack-Smashing Protector changes the order of the local variables, so the tables will be located immediately under the guard variable – our guard. The code compiled in our new compiler looks then as follows:

```
char buf[8];
char *p;
int ok;
```

As we know already, buffer overwriting in the above case won’t do much, because the guard variable will be overwritten, and this will be detected at the end of the function. Bypassing this type of protection is almost impossible, and only errors enabling us to save data to any memory address (e.g., such as those exploited during a format string attack) can be exploited by the hacker.

Luckily they are rare. The protection method is also visible in the assembler code of each function. At the end it contains usually instructions of this type:

```
0x80487cd <main+73>:        cmp    0x8049e00,%edx
0x80487d3 <main+79>:        je     0x80487e8 <main+100>
0x80487d5 <main+81>:        mov    0xffffffe8(%ebp),%eax
0x80487d8 <main+84>:        mov    %eax,0x4(%esp,1)
0x80487dc <main+88>:        movl   $0x8048c38,(%esp,1)
0x80487e3 <main+95>:        call   0x8048890 <__stack_smash_handler>
0x80487e8 <main+100>:       leave
0x80487e9 <main+101>:       ret
```

At the beginning a comparison of the values located under the address 0x8049e00 with the EDX register value takes place. If they agree with each other, a jump to the address 0x80487e8, the very end of the function (leave instruction), is performed. If these values differ, the __stack_smash_handler function located under the address 0x8048890 is performed. This tells us about the attempt to overwrite the stack. We'll try using gdb to check how both of these cases look. We start up gdb, entering in the call parameter the name of our program (gdb test). We put a break under the address 0x80487cd, where the comparison function is located.

```
(gdb) break *0x80487cd
Breakpoint 1 at 0x80487cd
(gdb)
```

The program will stop running there, and we will check the values. Now we start up the program, with short arguments so as not to overwrite the guard variable:

```
(gdb) r AAAA
Starting program: /usr/pp/bin/test AAAA

Breakpoint 1, 0x080487cd in main ()
(gdb) printf "0x%x\n", $edx
0xc5f93634
(gdb) printf "0x%x\n", *0x8049e00
0xc5f93634
(gdb)
```

To display the given values we used the printf function, similar to that in the C language. As we can see, the read values are exactly the same.

So we continue running the program:

```
(gdb) c
Continuing.

Program exited normally.
(gdb)
```

Nothing happened, and the program terminated normally, according to our expectations. Now, we will try to transfer a long argument to the program, in order to overwrite the stack:

```
(gdb) r AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /usr/pp/bin/test AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x080487cd in main ()
(gdb) printf "0x%x\n", $edx
0x41414141
(gdb) printf "0x%x\n", *0x8049e00
0xbda71c36
(gdb)
```

We see that the EDX register was filled with the value that we entered in the program argument. The same number was also placed under the address 0x8049e00. This is a global guard with a random value, and it is impossible to overwrite. The values are different; therefore, the condition was not met. Let's try to continue running the program:

```
(gdb) c
Continuing.
test: stack smashing attack in function main
Program received signal SIGABRT, Aborted.
0x40046841 in kill () from /lib/libc.so.6
(gdb)
```

The program started up the __stack_smash_handler function, which printed the information on the incident and sent the SIGKILL signal to itself, which is shown by gdb.

LibSafe

In the battle between performance and security in the development of a standard C library, performance won. The library itself is safe, but it makes many functions available that can prove dangerous in the hands of an inexperienced programmer. One of those potentially dangerous functions is gets(). To see for ourselves, we will now try to compile the program below (**/CD/Chapter18/Listings/test3.c**):

```
int main(int argc, char* argv[])
{
    char buf[8];
    gets(buf);
    return 0;
}
```

```
bash-2.05b# gcc -o test3 test3.c
test.c:6:2: warning: no newline at end of file
/tmp/ccj8vnHg.o(.text+0x17): In function `main':
: warning: the `gets' function is dangerous and should not be used.
bash-2.05b#
```

The compiler has told us that the function is dangerous and we shouldn't use it. Besides gets(), there are many other functions of the libc library that can threaten the program. An example is strcpy(), which also doesn't check the size of the introduced buffer.

Let's enter a long character sequence into our program:

```
bash-2.05b$ ./test3
AAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

We can assume that it was compiled without using the SSP patch. As we can see, the stack frame has been overwritten with our buffer.

We'll review the steps we have taken, bringing us slowly but surely to where the attack begins. First we protected the memory against the startup of the hacker's code. Then we secured the program against jumping to an erroneous function by applying the SSP patch. Next, we should secure our programs so an overflow won't happen.

Linux has the ability to intercept references to the libc library functions. This means that we can create our own library containing functions with the same names as those in the libc library, and these will be used by our programs.

LibSafe is an example of such a library. It filters all dangerous function calls. If it detects an attack, it tells us about it immediately and ends the program. Otherwise it transfers the action to the standard libc library. Now we will download, install, and test the safe library. It can be downloaded from:

```
http://directory.fsf.org/project/libsafe/
```

Information on the function, installation, and use of the library are also to be found there. In our example we will use version 2.0. So, we download its resources, as usual to the /usr/src directory:

```
wget http://pubs.research.avayalabs.com/src/libsafe-2.0-16.tgz
```

We unpack the archive and perform the compilation and installation process of the library:

```
bash-2.05b# tar zxf libsafe-2.0-16.tgz
bash-2.05b# cd libsafe-2.0-16
bash-2.05b# make
cd src; make
make[1]: Entering directory `/usr/src/libsafe-2.0-16/src'
gcc -M  util.c intercept.c > dep
gcc -c -o util.o -O2 -Wall -fPIC -DLIBSAFE_VERSION=\"2.0.16\"  util.c
(...)
gcc -o canary-exploit -Wall  canary-exploit.c
gcc -o exploit-non-exec-stack -Wall  exploit-non-exec-stack.c
make[1]: Leaving directory `/usr/src/libsafe-2.0-16/exploits'
bash-2.05b#
```

Due to its small size, this process will take at most a few seconds on a modern computer.

```
bash-2.05b# make install
cd src; make install
make[1]: entering directory `/usr/src/libsafe-2.0-16/src'
install libsafe.so.2.0.16 /lib
(...)
Type y for installing libsafe system wide?[default n] n
install ../doc/libsafe.8 /usr/share/man/man8
make[1]: leaving directory `/usr/src/libsafe-2.0-16/src'
bash-2.05b#
```

After installation the library is ready to be used. In order for LibSafe to be loaded while starting up each program, we have to set the environment variable LD_PRELOAD, which will point to its access path. We do this using the export command:

```
bash-2.05b# export LD_PRELOAD=/lib/libsafe.so.2
```

It is good to add this line to the /etc/profile file so it will be loaded at each login. We will now test the program below, which is susceptible to the buffer overflow attack in the strcpy() function (**/CD/Chapter18/Listings/test.c**):

```
int main(int argc, char* argv[])
{
    char buf[8];
    strcpy(buf, argv[1]);
    return 0;
}
```

The first step is to compile the program:

```
bash-2.05b# gcc -o test test.c
```

Now, let's check if the LibSafe library will be uploaded at the startup of our program. We will use the ldd program for this purpose:

```
bash-2.05b# ldd test
        /lib/libsafe.so.2        => /lib/libsafe.so.2 (0x40015000)
        linux-gate.so.1          => (0xffffe000)
        libc.so.6                => /lib/libc.so.6 (0x40024000)
        libdl.so.2               => /lib/libdl.so.2 (0x40135000)
        /lib/ld-linux.so.2       => /lib/ld-linux.so.2 (0x40000000)
bash-2.05b#
```

We can see that our library is located at the very top of the list, so we can be sure it will be uploaded. We start up the program and transfer a long character sequence to it:

```
bash-2.05b# ./test AAAAAAAAAAAAAAAAAAAAAAAAAAA
Libsafe version 2.0.16
Detected an attempt to write across stack boundary.
Terminating /usr/src/test.
    uid=0  euid=0  pid=3131
Call stack:
    0x40016b1c  /lib/libsafe.so.2.0.16
    0x40016c4b  /lib/libsafe.so.2.0.16
    0x80483c6   /usr/src/test
    0x400390ad  /lib/libc-2.3.3.so
Overflow caused by strcpy()
Annihilated
bash-2.05b#
```

The overflow is identified before it calls the real strcpy() function in the libc library, which is as we expected. Owing to this, no memory area was ever overwritten, and the program terminated, giving the exact location where the error (strcpy() function) was, at 0x80483c6). This information has also been saved into the /var/log/secure file, giving us the ability to identify errors in programs working as invisible services (e.g., WWW server).

a) strcpy(char *dest, const char *src) – Can overflow the dest buffer.
b) strcat(char *dest, const char *src) – Can overflow the dest buffer.
c) getwd(char *buf) – Can overflow the buf buffer.
d) gets(char *s) – Can overflow the s buffer.
e) [vf]scanf(const char *format, ...) - Can overflow the arguments of this function.
f) realpath(char *path, char resolved_path[]) – Can overflow the path buffer.
g) [v]sprintf(char *str, const char *format, ...) - Can overflow the str buffer.

All the functions mentioned above are considered dangerous. Besides the functions listed above, LibSafe also checks the ones operating on formatting characters. An example of this kind is printf(). If, while using it, an attacker tries to change the way a program is executed, it will also be closed, and the user will be warned about the incident.

As we can see, despite its small size the LibSafe library provides a considerable improvement in security and checks for cracker activity from the very beginning of the attack. It is thus worth keeping in the system for daily use. The performance penalty for programs is minimal.

The principles grsecurity programmers apply, are: “People are usually the weakest link in security” and “We should ensure security at every level.”

Therefore, we should remember that security is a process and not a product. We should take care of it continuously. At the same time applying a patch is not enough; it can only make a cracker's job more difficult.

Not even memory protection and the randomization of addresses will ensure perfect security. We should therefore take care of it on many levels:

a) Applying a patch to the kernel.
b) Applying a patch to the compiler.
c) Using security libraries.
d) Activating software.

Even if an attacker can somehow bypass the LibSafe library, he will encounter an error related to the Stack-Smashing Protector. In theory this can be bypassed only using a format string attack; however, it is not possible to perform this because of LibSafe. If, however, we come across a remarkable cracker and he passes through the first two barriers, he will then come across the problem that it is not possible to start up his own code. The protection levels are very interdependent, and each level determines the security of the others. After applying numerous protection systems, it becomes nearly impossible to exploit an error. After protecting the system in this way we can sleep more soundly, as we do not have to worry quite so much about the integrity of our data.