[aadams@securityfocus.com: Linux Kernel <= 2.4.21 MXCSR Local DOS Exploitation]

Source: http://www.derkeiler.com/Mailing–Lists/securityfocus/bugtraq/2003–11/0253.html

---

From: David Ahmad (*da_at_securityfocus.com*)
Date: 11/21/03

```
Date: Thu, 20 Nov 2003 17:10:57 -0700
To: bugtraq@securityfocus.com
```

As far as I know, this bug was not discussed or exploited anywhere
publicly. Also, the technique used to cause the memory copy to
fail is clever and may be useful in other scenarios.

––––– Forwarded message from Aaron Adams <aadams@securityfocus.com> –––––

From: Aaron Adams <aadams@securityfocus.com>
Subject: Linux Kernel <= 2.4.21 MXCSR Local DOS Exploitation
To: vuln–dev@securityfocus.com
Date: Thu, 20 Nov 2003 15:49:34 –0700 (MST)
Message–ID: <Pine.LNX.4.58.0311201547400.12519@mail.securityfocus.com>

As I mentioned in my previous post to the list, I've been looking into x87
FPU stuff lately. While I was tinkering I ran into the MXCSR register,
which reminded me of an old RedHat advisory with a vague description of a
kernel vuln relating to the register. I don't recall anyone ever
discussing this issue really or looking into the implications, although
some people speculated it could be used to trigger a DOS.

So because the info in the RH advisory and kernel changelog weren't too
eye opening:

RH:
Andrea Arcangeli found an issue in the low–level mxcsr code in which a
malformed address would leave garbage in cpu state registers.

Changelog:
MXCSR Handler Unspecified Vulnerability

I decided to look into it a little bit and see what could be done. I'll
start off with a little bit of detail on the MXCSR register since it
obviously pertains to the whole thing.

Starting with the P3, Intel processors included SSE support which amongst
many other things added the MXCSR register to handle SSE status and

control information (and various behavioral flags). MXCSR is a 32−bit register, the MSB 16−bits of which are reserved. Intel specifies that if these reserved bits are written to a general protection fault (#GP) will occur.

Note: For the interested reader, the register also includes the 'DAZ' (denormals are zero) flag which is not supported on processors earlier then the p4 and xeon. If this bit (the 6th) is written to on non−supporting processors, a #GP will also occur.

Because of the reserved bits detailed above, whenever the kernel is setting/restoring data in a process' MXCSR register, the value must be ANDed against 0xffbf to ensure that only 'legal' bits are set.

Knowing these details about the register I had a pretty good idea of what the issue would be and vaguely how to trigger it. So I decided to take a look at the patch and identify the problem. The problematic code is shown below:

>From /arch/i386/kernel/i387.c

```
 static inline int restore_i387_fxsave( struct _fpstate *buf )
 {
        struct task_struct *tsk = current;
        clear_fpu( tsk );
        if ( __copy_from_user( &tsk−>thread.i387.fxsave, &buf−>_fxsr_env[0],
                        sizeof(struct i387_fxsave_struct) ) )
                return 1;
        /* mxcsr bit 6 and 31−16 must be zero for security reasons */
        tsk−>thread.i387.fxsave.mxcsr &= 0xffbf;
        return convert_fxsr_from_user( &tsk−>thread.i387.fxsave, buf );
 }
```

Even without the patch the problem is easy to spot. If when restoring data previously saved with the FXSAVE instruction, __copy_from_user() can be used to store malicious data within thread.i387.fxsave.mxcsr and then fail, the function will return without sanitizing the register. As we know from the details above, if any of the reserved bits are set, this will cause a #GP within the kernel.

Note: This isn't related to the DOS issue, but if you see the code above:
tsk−>thread.i387.fxsave.mxcsr &= 0xffbf;
would this not be erroneous masking if the processor supports the use of the DAZ flag? Just a thought.

So, now that we know what the problem is and know that corrupting the MXCSR register will allow an unprivileged user to crash the kernel, we look into how to go about actually triggering it.

The above code is called when restoring data saved with the FXSAVE instruction. This basically stores the FPU environment, as well as the

MMX/SSE/SSE2 registers. The data is stored as a 512 byte structure, which is define in sys/user.h as:

```
struct user_fpxregs_struct
{
        unsigned short int cwd;
        unsigned short int swd;
        unsigned short int twd;
        unsigned short int fop;
        long int fip;
        long int fcs;
        long int foo;
        long int fos;
        long int mxcsr;
        long int reserved;
        long int st_space[32];
        long int xmm_space[32];
        long int padding[56];
}
```

You can save this data to an arbitrary location, dictated by the operand passed to the instruction.

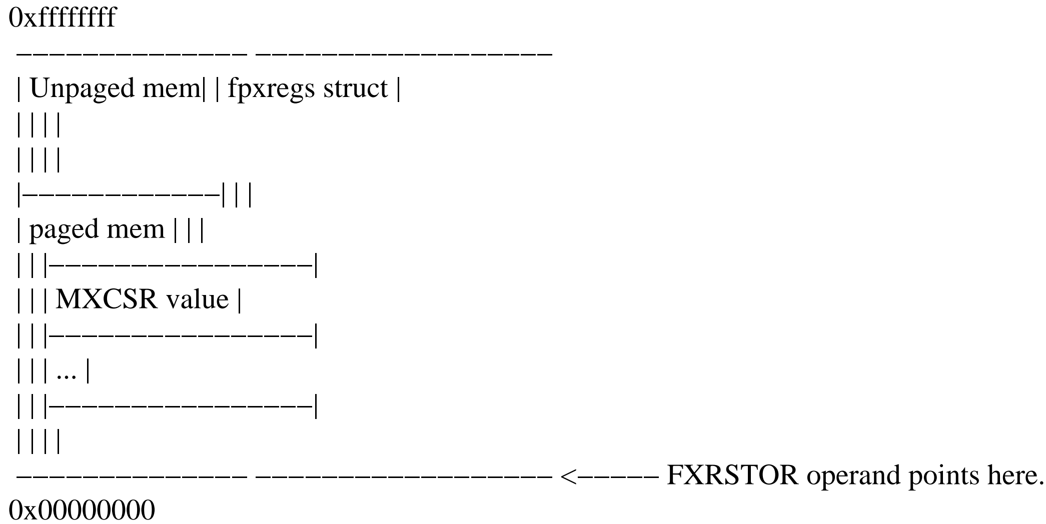
All of the data/registers are restored when a call to FXRSTOR is called, which requires an operand that points to the location of the aforementioned structure. So

Knowing that we can use FXRSTOR to dictate the information restored, we need to simply create a fake user_fpxregs_struct structure in memory containing at least one bit of the reserved 16–bits of the MXCSR value set. As seen above, the MXCSR data is stored in the 24 – 28th bytes of the structure.

With all of the above information we have all the conditions to trigger the bug met save one, which was probably the most interesting part of looking into all of this. Somehow, the __copy_from_user() function must be used to write to the MXCSR variable stored in the kernel, but it also must eventually fail. It took me a little bit to think of a good way to do this, and it actually came to me while I was asleep. I woke up in the middle of the night and said outloud how to do it... kinda wierd. :/

Basically, I created a 'fake' user_fpxregs_struct on the boundary of a page, so that approximately half of the structure is 'pseudo–stored' in unpaged memory. This allows us to reference the start of the structure, located in legitimate userland memory, with a call to FXRSTOR and have enough legitimate data to copy our corrupt value to MXCSR. However, as the copy operation continues it will eventually hit invalid memory and cause the procedure to fail. This will cause the restore_i387_fxsave() function to return and the kernel to attemptt to function with the MXCSR in an illegal state, triggering a #GP and causing a kernel panic.

So, if that little explanation didn't make sense this is kinda how it
works:



So that was a little long winded for something so simple, but even though
it is just a DOS I think it's interesting and it was fun to look into
it. Any additional insight into any of this or corrections would be
interesting too.

As an exercise I'll leave triggering the issue to interested readers.
Hint: It's possible with 3 assembly instructions (less anyone?).

ttyl.

Aaron Adams

----- End forwarded message -----

```
--
David Mirza Ahmad
Symantec
PGP: 0x26005712
8D 9A B1 33 82 3D B3 D0 40 EB  AB F0 1E 67 C6 1A 26 00 57 12
--
The battle for the past is for the future.
We must be the winners of the memory war.
```