TECHNOTE: Driver Loader Library Call `GetDriverInformation`: A Bug & Workaround

By John Miller
john_miller@quickmail.apple.com
Apple Integration Quality / API Quality Tools

This Technote describes a workaround to a bug in the first version of the Driver Loader Library in System 7.5.2 and System Update 7.5.3 (it should be fixed in later versions of the Mac OS). As of this writing, the Driver Loader Library is only available on Power Macintoshes that support PCI cards (for example: Power Mac 7200, 7500, 8500 and 9500). There is a bug in the routine `GetDriverInformation` that can possibly cause an overwriting past the end of the name string that it is passed in.

This Technote is directed primarily at writers or family experts and especially applications that get information about drivers.

Defining the Problem

In the Driver Loader Library, there is a bug in the routine `GetDriverInformation` that can possibly cause an overwriting past the end of the name string that it is passed in.

This bug surfaces only when calling `GetDriverInformation()` for a driver that has had its Device Control Entry fields zeroed when it was closed (the Chooser driver has been observed to exhibit this behavior). The bug occurs because `GetDriverInformation()` does not check for zeroed fields before using the `dCtlDriver` field to reference the driver's name; instead, it copies a garbage string from low memory into the name string passed as a parameter to `GetDriverInformation()`.

If the first byte of this garbage string is larger than the number bytes of storage allocated by the caller for the name string, then the caller's data located just past the end of name's storage will be overwritten with garbage.

You can see the zeroing out of fields for the Chooser's driver by following these steps using any version of MacsBug:

1. Open the Chooser
2. Drop into MacsBug and type:

   ```
   drvr      <return>
   ```

3. Look down the list of drivers in the Driver column of the `drvr` display and see the Chooser (on my machine it's at `dNum 0xF`).
   To see what the Chooser's DCE fields look like before the zeroing type:

   ```
   dm  xxxxx   dctlentry   <return>
   ```

   (where xxxxx is the hexidecimal number in the Chooser's row in the "DCE at"column in the drvr display)
4. Exit Macsbug by hitting Command-g
5. Close the Chooser window
6. Drop into MacsBug and type:

   ```
   drvr   <return>
   ```

   You'll see that name Chooser is replace by blanks in the row that it was in.
7. To see the zeroing of DCE fields type:

   ```
   dm  xxxxx    dctlentry  <return>
   ```

   (again, where xxxxx is the hexidecimal number in the Chooser's row in the "DCE at" column in the drvr display).

You'll see that all fields except the `RefNum` field have been zeroed

It's doubtful that any expert code will encounter this problem if the expert code executes pre-Finder; applications that execute *after* the Finder boots are the most likely victim. If your code calls `GetDriverInformation()` after a user has a chance to close one of these "zeroed-out" drivers(and the DCE fields are thus zeroed), then you will need this workaround. If you call `GetDriverInformation` only for a driver that you know doesn't have its Device Control Entry fields zeroed upon closing, then you don't need this workaround because the bug will not appear. It is only when you traverse the unit table, calling `GetDriverInformation` for unknown drivers, that you need to be aware of the workaround.

GetDriverInformation

Here is the declaration for `GetDriverInformation` as gleaned from the universal headers file, `Devices.h`:

```
extern OSErr
GetDriverInformation(DriverRefNum refNum,
                UnitNumber *unitNum,
                DriverFlags *flags,
                DriverOpenCount *count,
                StringPtr name,    // ** this is the field we are
                                                   concerned with//
                RegEntryID *device,
                CFragHFSLocator *driverLoadLocation,
                CFragConnectionID *fragmentConnID,
                DriverEntryPointPtr *fragmentMain,
                DriverDescription *driverDesc);
```

It is the `StringPtr` name parameter that we are concerned with. If you allocate, for example, a `Str31` to use to pass in as the `StringPtr`, then (if the erroneous byte `GetDriverInformation` thinks is the length byte of the garbage string is greater than 0x1f, or 31 decimal) `GetDriverInformation` will unwittingly copy all the garbage bytes to your code without regard to the actual location of the end of the name string.

Solving the Problem

The workaround is simple: allocate a String255 for the name parameter passed into GetDriverInformation(), rather than some shorter-length string. This means any garbage copied to the string will be contained in that string rather than any other data. If you're familiar with GetDriverInformation, that's all you need to know: use a Str255 for the name parameter rather than a shorter string and you're protected. If you're not familiar with GetDriverInformation and would like to see some code to traverse the unit table, sample code is provided for your information. You also have to take precautions about using the garbage string data as well (if you were going to display the driver name in an application, you would probably want to check for non-printing characters if displaying them would cause problems in your code. You might want to make sure the garbage length of the name string isn't too long for your code to handle).

Sample Code Using GetDriverInformation To Iterate Over the Driver Unit Table

To drive the point home about using a Str255, and also to alert you to another mandatory initializing of the FSSpec field of the driverLoadLocation struct, (another input of GetDriverInformation), here is some barebones sample code. Note that traversing the unit table using GetDriverInformation() is not the most efficient way to discover which units are empty and which are full. Use the Driver Loader Library routine, LookupDrivers() for that.

```
void TraverseDrivers()
{

    OSErr               err = noErr;
    DriverRefNum        refNum;
    UnitNumber          unitNum;
    DriverFlags         flags;
    DriverOpenCount     count;
    RegEntryID          device;
    CFragHFSLocator     driverLoadLocation;
    DriverDescription   driverDesc;
    // Str63            theName; // BAD, not long enough
    Str255              theName;  // GOOD: THIS IS THE WORKAROUND!
    FSSpec              loadLocSpec;
    short i;

//  this is another caveat about using GetDriverInformation();  you must
//  initialize the FSSpec ptr field of the driverLoadLocation struct to
//  point to an allocated FSSpec because GetDriverInformation assumes you
//  have.  This is done is the next line below.
      driverLoadLocation.u.onDisk.fileSpec = &loadLocSpec;

    for( i = 0;  i <= HighestUnitNumber(); ++i ){
        refNum = ~i;  // convert the unit number to a driver refNum.
        err = GetDriverInformation(     refNum,
                                        &unitNum,
                                        &flags,
                                        &count,
                                        theName,
                                        &device,
                                        &driverLoadLocation,
                                        &fragmentConnID,
                                        &fragmentMain,
                                        &driverDesc);
        if( err != noErr ){  // there’s a driver for this refNum

            // Do whatever it was you wanted to do with the information
            //  BEWARE:  If the driver is a non-native driver, that is a
            //  68k driver of pre-PCI-supporting Macintosh, the device,
            //  driverLoadLocation, fragmentConnID, fragmentMain, and
            //  driverDesc inputs above will be set to nil after the call
            //  because these fields don’t apply to 68k drivers.
        }
    }  // for
}   //  end  TraverseDrivers()
```

Reference for GetDriverInformation

GetDriverInformation

GetDriverInformation returns a number of pieces of information about an installed driver.

```
OSErr GetDriverInformation
            (DriverRefNum           refNum,
             UnitNumber             *unitNum,
             DriverFlags            *flags,
             DriverOpenCount        *count,
             StringPtr              name,
             RegEntryID             *device,
             CFragHFSLocator        *driverLoadLocation,
             CFragConnectionID      *fragmentConnID,
             DriverEntryPointPtr    *fragmentMain,
             DriverDescription      *driverDesc);
```

refNum	refNum of driver to examine
unit	resulting unit number
flags	resulting DCE flag bits
count	number of times driver has been opened
name	rresulting driver name
device	resulting Name Registry device specification
driverLocation	resulting CFM fragment locator (from which the driver was loaded)
fragmentConnID	resulting CFM connection ID
fragmentMain	resulting pointer to DoDriverIO
driverDesc	resulting pointer to DriverDescription

DESCRIPTION

GetDriverInformation is used by driver experts in PCI-bus-supporting machine, software that makes decisions about which driver to load for a particular device — or by any software that needs to get information about a driver for a device.

Given the Unit Table reference number of an installed driver, GetDriverInformation returns the driver's unit number in unit, its DCE flags in flags, the number of times it has been opened in count, its name in name, its RegEntryID value in device, its CFM fragment locator in driverLocation, its CFM connection ID in fragmentConnID, its DoDriverIO entry point in fragmentMain, and its Driver Description in driverDesc.

Note
With 68K drivers, GetDriverInformation returns meaningful information in only the unit, flags, count, and name parameters. ◆

▲ **WARNING**
You must allocate the FSSpec field of the CFragHFSLocator * driverLocation before passing it in to GetDriverInformation(). ▲

RESULT CODES

noErr	0	No error
badUnitErr	-21	Bad unit number
unitEmptyErr	-22	Empty unit number

Summary

To protect yourself against having GetDriverInformation copy garbage into the passed *StringPtr* name parameter when a driver has its Device Control Entry (DCE) fields zeroed upon closing (the Chooser, for example), allocate a large enough string (for example, String255) for the name parameter. This will assure that any garbage copied to the string will be contained in that string.

Further References

See *Designing PCI Cards and Drivers for Power Macintosh Computers* for further documentation on `GetDriverInformation` or any other Driver Loader Library calls.

Acknowledgments

Special thanks to Tom Maremaa. Thanks to Larry Chiu, Tom Mason, and Tom Saulpaugh.