How To Write A Virus For The Macintosh

*or 'How I Learned To Stop Worrying And Love The Resource Manager'
document version 1.0

Due to numerous requests for this type of information, I will delve myself into the dark side and release that information by which people can be arrested. Please note that this file contains no source code and no information about destructive code, but simply gives the basic ideas and principles behind writing a reproducing code resource and how it can be used to better society.

Chapter 1: Basic Principles

A computer virus, by definition, is a piece of processor-executable code that can reproduce itself within it's environment. In the Macintosh system, an object called a resource can contain executable code. Most common executable resources are of type 'CODE', along with others such as 'DRVR', 'CDEF', 'WDEF', 'INIT', and so on. These resources are loaded into memory and 'jmp'ed to to be executed (an assembly language term for jump). Note that not only these types listed above can contain code. The trick is to get the virus code loaded and called by 'accident'.
There are many places where code resources are loaded and executed. For one example, at the launch of an application, the CODE resource with ID=0 is loaded (the jump table), and then jumps to the first listing in it's table. As another example, a 'CDEF' resource is called to draw certain controls (ID=0 for buttons, checkboxes, and radio buttons; ID=16 for scroll bars, etc.). Another example: an 'INIT' resource is called at startup time if the file which contains it is in one of the special system folders. There are numerous other places within applications, and even within system software itself, where code is loaded and called. Each virus uses a trick with one of these methods to get itself called. Some of these methods are described in more detail below.
Once the virus code is executed, it's main responsibility is to duplicate itself. This in itself is a fairly easy process. Since your executable code resource is already loaded into memory, you can use a few popular toolbox calls to place it into any other file or application that would suit your needs (where it would also have the chance of being executed). After the duplication is complete, the virus may do any other task it deems necessary.
One of the reasons why viruses crash is that their reproduction or startup code is not compatible with other systems and/or machines, not that their damage system actually did any damage. If you write code following the Inside Macintosh rules and code defensively, you should be able to write a clean piece of code that travels without problems. Always code defensively: it's your work out there... you want to be proud of it. Read below to find some tips on doing just that.
Virus testing is a very difficult process, in that your own system is constantly infected, possibly numerous times with older versions of the virus. There are methods to the madness, so again, read on. A few of the catches to writing a virus is being aware of the methods used by virus-protection software. If simply written, a virus could be caught very quickly and not have much effect beyond your own system. If the methods are thought out and the patches made by the protection software are understood, then a virus could at least require software

companies to update their existing detection methods. Every virus to date has been able to be detected and destroyed, so don't feel bad.
Is everybody happy? Then let's go!

Chapter 2: Writing Executable Code

An executable code resource is easy to create with a good software-development application such as THINK C (or C++) or THINK Pascal or MPW. There are slight differences between the environments, but nothing major. I will be giving examples for code written in THINK C, for that is the system I use.

An executable code resource usually starts with a

```
void main(void)
```

and within such, your executable code exists. Note, as always, that executable code cannot handle global variables (variables defined above the definition of the main code, accessible by the whole file/project). THINK C handles ways around this, and MPW uses the methods in Tech Note #256, but in most cases, you won't really need global variables, unless the code is complex enough to require separate procedures and/or object-oriented code. In any case, you can usually define your variables inside the main procedure itself. There aren't too many rules as far as writing code resources, so long as you know under which circumstances your code will be called. If you are patching a Toolbox trap, for example, you must take the same form as the patch you are trapping:

```
void ModalDialogPatch(ProcPtr procFilter,int *itemHit)
```

If you are patching an operating system trap, you need to do some register playing, but you need to take an empty procedure form:

```
void OpenPatch(void)
```

even though the FSOpen, PBOpen, etc. take paramBlocks. Note: they are stored in registers A0, D0, and A1 usually. Check the trap for specifics. You need to save these before you execute your code, and then restore them upon return.
If you are executing code that is to be run as a VBL or Time Manager task, always remember that you cannot use code that even **thinks** about using the Memory Manager (i.e. moves or purges memory). Make sure all the toolbox calls you use are not in the 'dreaded list' in the appendix of each volume of Inside Macintosh.

The type of the code resource is very dependent upon which method you wish to use to get your code executed. Read the next section for details on such execution theories.

After you're done writing the code, check it over for simple things you might have forgotten (original Quickdraw compatibility, system versions, etc.), and compile the sucker. For right now, you can throw the code resource into some sort of test file (or stagnant file, where the code will not be executed and/or reproduce). Note that you should NOT have any external resource files to compile along with it. Code resources such as this should preferably be self-contained, and not have to 'carry around the extra luggage', so to speak. There are methods to carry along bitmaps (as 'unknown data') and use them as graphics. But you should never rely on things like GetNewDialog, because that requires the existence of a DITL resource. Instead, use calls like NewDialog, where the code builds the relevant information in. It might make things harder to read and a bit harder to edit, but it's what you have to do in order to make everything self-contained.

Most of the compilers create some sort of header at the beginning of each executable code resource. This header could give away some vital information about the resource which would make it easy for a virus-detector to find. Double check it after compilation to make sure it's clean and doesn't look suspicious.

Chapter 3: Getting Your Code Executed

This technique you use here defines how your virus spreads. Some earlier viruses were more virulent than others; nVir needed an infected application to boot for it to execute; WDEF required only that a disk be inserted. There are lots of places for code to be "patched" so that your code can be executed. The trick is finding them and recovering from them gracefully. Not every method can be discussed in this note, but I will give some general examples and how to find your own 'hooks'.

One of the most popular methods amongst virii is infecting applications, since they, by definition, have executable code built right into it. If you can get your code executed along with the many other little segments in the application, the code could recover undetected. When an application starts up, the resource CODE with ID=0 is loaded into memory, and it's popularly known as the jump table. It keeps track of all of the procedures or segments in an application. If part of an application needs to call another procedure inside the application, it checks with the jump table for it's location. If it sees that the procedure is not in memory, it will load it first, then execute it. This is all taken care of by the compiler and the system software, so it's invisible to the programmer (in most cases). The system loads the jump table and immediately executes the first entry in the list when an application begins.
You can patch yourself into this list of procedures by modifying the jump table itself. You can modify the first entry of the jump table to be your code, but save the original entry so that you can call the actual application when you're done (destroying the first entry in any CODE 0 resource renders the application totally useless). So, instead of the system executing what it thinks is the application, it will run your code first, and **then** run the application.

Another method by which virii get executed is by utilizing a wonderful feature of the Resource Manager. As given in Inside Macintosh volume 1, the Resource Manager will look for resources in the top-most resource file that is open (the one most recently opened in most cases, unless UseResFile has been used). These searches also include searches for basic system code resources, such as window definitions, control definitions, and even international transliteration code. If you have a resource with the same type and ID as one in the system, and this resource file is open, the system will execute your resource instead of the system's. The catch again with this is that you should call the original resource as well to make your code invisible to the naked eye. This, apparently, is how the WDEF virus worked. When a disk was inserted, the desktop file was automatically opened and put in the list of resource files. Within this time that the file was open, the WDEF file which existed within was executed (the system needed to draw the window itself using the standard WDEF resource). This method requires no patching of other code and makes it very elegant. The thing that makes them easy to detect is that you find code resources in very odd places. A WDEF resource is not usually found in the desktop file.

To find other hooks for code execution, look at all the executable code inside the system. These pieces are executed at one time or another for certain calls. Things that might not seem

obvious right away may make good places for patching. Lots of applications use (maybe indirectly) the International Utilities Package, for it has many good string manipulation routines. A patch there might be possible. 'ptch' resources in the system are loaded automatically at startup time to patch bits of ROM. A system could be infected there and be loaded before all other extensions. Poke around the resources and find out which ones are executable, and then find out **when** they are executed. You might be able to find a great patch to live off of.

Chapter 4: Reproduction

Reproduction of any code resource requires the help of the File Manager and/or the Resource Manager. The concept is not very simple, but the execution is very easy. Since the virus itself is simply one code resource (preferably not more than one), then it can be loaded, added, modified, changed, and saved just like any other resource. And the fun part of it is, you can do all this to the code you are currently executing. This is apparently dangerous (Apple warns us about self-modifying code), but we're not modifying anything about our code; simply our placement. With a few simple calls we can duplicate ourselves anywhere we wish.

When an executable code resource is called, the pointer to the resource is placed in register A0. You can use this pointer to reference yourself. A simple line of assembly can place A0 in any variable you choose. Once you have this variable, you must translate it into a handle with the RecoverHandle call. Now you have a handle to your own loaded resource, but you still cannot duplicate it. As a handle to a resource, you cannot use it to be copied into other files. You 'belong' to your owning file, and are not expected to go elsewhere. Use the DetachResource call to remove your reference to the file you came from. After this call, you are simply an executable block of memory floating around with a handle on yourself (phallic, isn't it?). All you need to have to have total freedom with a block of memory is a handle to it. You've got this free handle now. Now comes the time to find the file you have to duplicate yourself to. The file you find depends on how your virus is designed to work. You can copy yourself into applications, into desktop files, or into the system. Again, dependent on how your executing mechanism requires it. Once you have found your file (usually with use of the File Manager), open up it's resource fork with OpenResFile or any other similar procreation of it (FSpOpenResFile, etc.). Call AddResource with the required parameters, then call WriteResource to forcefully write the resource to the file or simply close the file itself (it will automatically be saved). Your code has now copied itself into another file. Reproduction! Now just let it sit and wait for it to be called!

Chapter 5: Defensive, Clean Coding

As always, if you want your code to be run cleanly on all systems, you have to be prepared for any type of situation. Apple warns us of this all the time, so I don't have to go into too much detail, but there are a few things I would like to stress so that your code doesn't simply crash when it gets executed. You goal is then not found. Here are some tips and things to watch out for.

1. *ResError*. Who knows? Maybe you've been purged. Check it after every Resource Manager call you can, taking efficiency into account, of course.
2. *Nil pointers*. Who knows? Maybe a virus detector caught part of your set of resources (if you're using a set, which I highly discourage) and deleted them. Find an alternate route, or exit gracefully.
3. *Patch well*. If you are going to modify something like a jump table, be sure you keep the originals somewhere for your own use so you can call the code (pass-through coding). If you don't, and you just destroy it and call the next code resource down the line, who knows what you might be calling. A bezier-curve calculation routine does nothing if the caller knows not what he's doing.
4. *File Manager*. Don't depend on each hard drive being called "Macintosh HD". Don't depend on an "Applications" folder. Don't depend on anything. Read the directory and see what you find interesting. System 7's File Manager is great, but watch out for:
5. *System 6 or before*. You wouldn't want your code to execute only on one system version now, would you? By known figures, only 50% of Mac users use System 7. Sad, eh? But why exclude them from the pleasures of your code?
6. *Error Check, Error Check, Error Check*. The thought police are on you again. Never forget that nothing is permanent.

Chapter 6A: Virus Protection Software: How It Works

Virus protection software was a good idea. It worked for a while. Then it became a commercial product. Virex, SAM, etc. The best one out in the world today is freeware: Disinfectant. A beautifully-written piece by John Norstad. I personally am against commercially-written virus protection. However, I am not here to give praise to independent software authors. I am here to tell you how some of their mechanisms work.

Patching toolbox traps is a popular method of modifying the system's own code. Before it calls the real thing, it calls the patch (your code). Virus detectors use this method to keep an eye out for parameters passed through certain toolbox calls to check to see if they are virus-related.
One popular patch is AddResource. If a virus detector sees that the type of resource that is being added is of type 'nVir', then it'll catch you. If it sees 'WDEF' with ID=0 and the open resource file is the desktop, then it'll catch you. Since AddResource is a very dependent call used for replications, it's almost certain to work every time. Other less-popular but more efficient patches are those at the base level of the operating system, not even documented by Apple. Traps such as `_vBasicIO`, `_VInstall`, `_NewHandle`, `_vMRdAddr`, and even `_ADBReInit` get trapped by the Disinfectant extension. Because these are very basic calls (used by nearly anything that does input and output, in the case of `_vBasicIO`), it can catch nearly anything coming toward it. It's nearly foolproof. After knowing what type of virus it is, the software can delete the virus quickly and easily.

Good applications also use their own version of virus protection. At the startup of their application, the number of resources in the file is counted, and the more important executable resources in the file are checked for their size. This way, if an application has had a resource added, it will be able to alert the user and stop execution.

Chapter 6B: Virus Protection Software: How To Bypass It

Though virus protection is great in most cases, there are still 'back doors' which haven't been explored at the time of this writing. Here are some ideas for getting around the checks that most virus protection software uses.

A trap is still a trap. It is not the real code; it is a dispatcher. It stops you on the way there, but it **doesn't** stop you from doing what those basic calls do on your own. This does require a fair amount of assembly language and ROM copying, but it gets you around the catch of using operating system traps at all. Simply copy the code that is contained in the trap itself and use that code. To never get caught, never use traps. But we know how nearly impossible that is. However, things are gained and lost in good code writing.

A drawback of virus protection in general is that the software has to be continually updated for each new virus and identified by name in most cases. One ingenious idea (mine? I don't know) is to make the name and/or type of the virus variable. It doesn't always have to be called 'nVir' or 'WDEF' (unless the mechanism depends on the name or type). Make the type change from permutation to permutation. This makes it much more difficult to catch.

One feature of the File Manager is it's automatic updating of the modification dates. Every time a file is updated or modified in any way, the modification date is changed. You can find and modify the date with a fairly simple low-level File Manager call. This is really a frivolous precaution, but it makes it easy to find the source of a virus attack. Changing the dates to something fairly feasible (NOT Jan. 1, 1904) may bypass such checks.

The application checks can be overridden with good code-writing as well. If the virus is to add a resource to a file (as it usually has to), why not delete one in it's place? You've got to be sure that the type is the same as another type (this is where the variable types come in handy), and you may even want to vary the size of it to make it match the one it replaced (hopefully a larger size). Simply modifying a resource (like CODE 0) with the same amount of bytes will usually not be detectable. This way, the applications still counts and finds nothing unusual. However, in the process, the application is permanently damaged in some way.

Chapter 7: Testing

In testing a virus on your own system, you subject it to many continuous attacks - maybe even ones that are unintended. There are some rules to follow to be sure that you can keep track of it's location and make sure it doesn't destroy your work in the process.

1. *SysBeep debugging*. I'm sure most of us a pretty familiar with this technique. It's compatible on all systems, and it's an aural identification. No visuals to set up, no extra resources. Simple `SysBeep(0);` is sometimes enough to know that everything's all right. When testing your duplication code to find out when it actually happens, use SysBeep after each one and then check to see where it went.
2. *Modification dates and times*. If you use random selection of files to infect, it becomes rather difficult to find which one got infected. If you know when an infection happened, you can

immediately check the modification dates of all files - simply by using the Find… command in System 7's Finder.
3. *Text Files*. This could be known as a common-file technique. For testing purposes, use a mechanism that whenever an infection takes place, the virus writes the process and the file names and such into a common file in a common folder somewhere. This way, you can check the text file afterwards and know exactly what your code has done. You need not make the mechanism too elaborate, as it will only be for use in testing.
4. *Backup, Backup, Backup*. The thought police are at it again. In these cases, it's all too familiar. A trashed project is no fun.

Chapter 8: Conclusion

In short, the devices behind writing a virus are not all that complicated. There are many checks to counterattack, and part of the puzzle itself is no find new ways to get around them. Find back doors. Give the code a personality. Make it try to find the best way around a counterattack if it is able to detect one. Size is no longer a constraint in today's memory-hog world. A virus of near 50k would probably go undetected in modern-day storage, so don't feel constrained in that way. Time should be a consideration, however.
Make code that is efficient, so that users don't notice a slowdown when it is executed. All in all, your code is your work. Don't let it out of the bag until it works well and clean, and don't forget to leave no trace.

Appendix: Questions and Answers

Q: How do you get it to randomly choose an application on the HD to infect?

A: Any file on the hard drive is stored in the directory. This includes documents, applications, system files, and so on. Files are found by using the directory (via the File Manager). If you wanted to choose an application that appeared as though it was a random choice, you can still move through the directory. Files are stored in the directory by an index, just like resources, and you can pick a random index number to check a random file. You can use Quickdraw's Random routine to pull out a number, check that index, and see if that file's type is 'APPL'. If not, simply choose another file. If you've found one, then you've got your random application. Granted that not all hard drives have 65536 files on them, so you may have to tone down the returns from Random, but that's simply mathematics. Note also that there are many small applications on the hard drive as well. TeachText, CompactPro, PrintMonitor, etc. This will also come up in the list of applications along with larger applications like Microsoft Word, Aldus Freehand, and the Finder itself. This method will **not** choose the System file (it has a type of 'zsys'), and documents, any desk accessories, or extensions/control panels. You can modify the routine to work with other file types as well.

Q: How can I include a bitmap or other separate resource data into my code?

A: You've got to use a little assembly (and disassembly) for this method, but it works wonders (the system's scroll bar CDEF use to use it). Create the resource you want to include in your favorite resource editor to suit your taste. Close the resource and re-open the sucker in hex. Copy all of the hex codes into the clipboard. Go into your development

environment and at the very end of your code add a bit of assembly. Use the DC.B or DC.W operands to define a bytes or words (respectively) and re-define the complete resource for each byte or word in the resource (it might take a little bit of re-typing). Give this procedure (which never gets 'called') a name. Now, whenever you want to use this 'resource', simply replace the handle or pointer to the resource with this procedure pointer. There's no need to load anything (the entire code resource is already loaded), and the pointer is always valid. This method also saves you from using the Memory Manager for anything.