MALWARE ANALYSIS 1

OGEE WHIZ

Peter Ferrie
Microsoft, USA

The programming of General-Purpose Graphics Processing Units (GPGPU) has become a common way to take advantage of the great power available on video cards. The programs, known as 'shaders', have a language that has evolved over the years to become something so high-level that it resembles a dialect of the C programming language. Many things can be implemented using shader programs, including the decryption of arbitrary data, and now we have a virus that does exactly that. We call it W32/Ogee.

STACKING THE DECK

The virus begins by pushing the RVA of the host entry point onto the stack, along with a pointer to the Process Environment Block. Both of these values are used in the final stage. The virus then retrieves the base address of kernel32.dll. It does this by walking the InLoadOrderModuleList from the PEB_LDR_DATA structure in the Process Environment Block (the address of kernel32.dll is always the second entry on the list). If the virus finds the PE header for kernel32.dll, it resolves the addresses of the required APIs.

The virus uses hashes instead of names, but the hashes are sorted alphabetically according to the strings they represent. This means that the export table needs to be parsed only once for all of the APIs instead of once for each API, as is common in some other viruses. Each API address is placed on the stack for easy access, but because stacks move downwards in memory, the addresses end up in reverse order in memory. This becomes important later on.

The virus resolves the addresses of just four APIs from kernel32.dll: GetModuleHandleA(), GetProcAddress(), LoadLibraryA() and VirtualAlloc(), but then uses only three of them (GetProcAddress() is not used). It uses the LoadLibrary() API to load glu32.dll. The address of only one API is resolved from here: gluOrtho2D(). The virus uses the GetModuleHandle() API to access the copy of gdi32.dll that is loaded implicitly by glu32.dll. It is not clear why the virus doesn't use the LoadLibrary() API instead, to avoid the need to import the GetModuleHandle() API. The virus resolves the addresses of two APIs from gdi32.dll: ChoosePixelFormat() and SetPixelFormat(). It uses the GetModuleHandle() API again to access the copy of user32.dll that is also loaded implicitly by glu32.dll. Once again, it is not clear why the LoadLibrary() API was not used instead. The virus never frees the DLLs, so the increased reference count should not affect anything. In fact, even if the virus attempted to free the DLLs, the behaviour of the *nVidia* video drivers, for example, would prevent the action from succeeding – the drivers intercept calls to the ChoosePixelFormat() API, and create a thread which does not terminate until the process does.

The virus resolves the addresses of five APIs from user32: CreateWindowExA(), DefWindowProcA(), DestroyWindow(), GetDC() and ReleaseDC(). The virus uses the GetModuleHandle() API to access the copy of opengl32.dll that is loaded implicitly by glu32.dll, and resolves the addresses of 21 APIs, including wglGetProcAddress(). All of the resolved API addresses from all of the loaded DLLs are placed on the stack.

The virus caches the value of the stack pointer in a register in order to access the existing APIs as well as the APIs that are subsequently loaded. This allows the virus to access stack elements, such as the APIs, without having to keep track of the value of the stack pointer. It also has a benefit in terms of the size of the code, provided that no more than 32 DWORD elements exist above or below the cached pointer value. Of course, the value of the cached pointer can be biased at the time it is calculated, such that it points into the middle of the block of values to access, in order to maximize the number of elements that remain within the +/-32 DWORD range. There is also a secondary benefit here, but it is minor in comparison: it provides a neater way to free an accumulation of stack parameters below the cached value, simply by assigning the cached value back to the stack pointer (biased by whatever value was applied when it was cached in the first place).

WINDOW OF OPPORTUNITY

The virus creates a window with a width and height of zero pixels, using the class-name 'EDIT'. The window is made as small as possible because it cannot be made invisible during the creation stage – it can only be hidden by the use of an additional API call, which presumably the virus writer wanted to avoid. 'EDIT' is the smallest built-in class-name, and conveniently fits within a single DWORD. The virus chooses a pixel format for the window which is intended to be 32 bits of 8/8/8/0 in RGBA format, but there is a bug in the structure layout. The bug probably results from a miscounting of the zero bytes during the dynamic structure construction, so the green shift is assigned eight bits, and the blue channel is assigned zero bits instead. Fortunately for the virus writer, this has no practical effect on the behaviour of the virus code, because the virus does not write anything to the window. In fact, none of the channels needed to be specified at all, and even the colour bit-count could have been zero. The virus sets the returned

pixel format for the window, and uses it to create the GL context.

It resolves the addresses of 18 GL APIs by name. Since opengl.dll does not export these functions in a table that the virus can parse, it must use the wglGetProcAddress() API. Interestingly, the list of names does not contain an explicit sentinel. Instead, the virus relies on the fact that a double zero appears later in the code, with no single zero in between. This makes the code extremely fragile – and could cause some trouble for any wannabe virus writers who try to alter it. The reliance on the double zero allows the virus to fetch a 'fake' API address which is placed on the stack automatically, and which is used as a placeholder for the next API call. All this to save a single byte of code. The virus creates a new framebuffer object and binds to it, and then proceeds to use old-style rendering initialization, via the MatrixMode() and LoadIdentity() APIs. These APIs have been deprecated since OpenGL 3.0, but are needed to maintain compatibility with older software. This is probably another example of the extreme legacy support that the virus exhibits later.

The virus creates a square orthographic projection space that is equal to the size of the texture. This is used to hold the texture data during projection mode. The size of the texture is calculated by the first-generation sample, and never changes. To calculate the size of the texture, the virus takes the size of its code, doubles it, takes its square root to derive the size of the square that would hold the code, divides the square root by four to produce the number of DWORDs in that square, and then rounds up the result to avoid truncation. It then reloads the model view identity. Presumably, the virus avoided using the PushMatrix() and PopMatrix() APIs because it would have increased the number of APIs in use, and thus the number of elements on the stack. Increasing the number of elements on the stack could result in some elements being outside of the +/- 32 DWORD range from above.

The virus creates a viewport which is used to specify the affine transformation between the internal representation and the window that it has created. The virus requires a one-to-one mapping between the two representations, to avoid scaling or wrapping of the texture. If the virus had created the window with the proper dimensions, then the viewport would have been assigned the proper dimensions too, when the context was created – in that case, there would be no need to create the viewport explicitly. Furthermore, the preceding initialization code could have been replaced by just three API calls from glut32.dll. However, despite that DLL being present on many *Windows* systems, it is not installed by default – which, presumably, is the reason the virus writer did not attempt to make use of it.

A QUESTION OF TEXTURE

The virus creates three texture arrays: one to hold the encrypted code, one to hold the decryption keys, and one to hold the decrypted code. During the infection phase, the roles of the first and third texture arrays are reversed. The virus binds the three texture arrays, and then sets parameters for each texture: min filter, mag filter, wrap s and wrap t. These parameters correspond to the filters for minifying, magnifying and wrapping of a texture. Interestingly, none of these parameters is needed, since the texture will never need to be scaled, and it will always fit within the co-ordinate space. It seems likely that this code was copied blindly from a tutorial. The virus defines the parameters for a texture image, but does not point it to any data. Instead, in non-*ATI* mode, the virus defines the parameters for a texture sub-image which overlaps the parent image entirely. This sub-image points to the texture data, but since the sub-image completely covers the parent image, it is not needed at all, and the data could have been supplied by the parent image alone. It seems likely that this code was also copied blindly from a tutorial. There is an indication that the virus supports an alternative method for the texture generation for *ATI* cards, but this has not been verified. Finally, the texture environment is set to copy the values exactly, so that no blending or interpolation occurs.

FIFTY SHADES OF CODE

The virus creates a new program and shader object, then binds the shader source to the shader object. The shader source is very simple, and implements the formula 'x=a+b*c', where 'a' is the texture array that holds the encrypted code, 'b' is the texture array that holds the decryption keys, and 'c' is a randomly chosen modifier value. The shader source is compiled and attached to the program object, then the program is linked. The virus determines the locations of the three variables within the compiled program in order to assign them the appropriate values. It is unclear why the virus does this at this time, given that they will still be available later – one possible reason is that the register that is used to locate the variables is used for a different purpose later. However, it seems that the virus author overlooked the fact that there was a spare register that could have been used instead. By using the spare register, the virus would also have avoided the need to cache the variable locations, and that would have saved three stack elements.

PRIMITIVE GEOMETRY

The virus attaches the input and output textures to the framebuffer object, and adds the program to the rendering

pipeline. It activates a texture unit, binds the decryption key texture array to it, and then assigns the texture index to the 'b' variable. The virus assigns the modifier value to the 'c' variable. This value is selected during the infection phase. The virus selects the colour buffer for the drawing target, activates a texture unit, binds the encrypted code texture array to it, and then assigns the texture index to the 'a' variable. The virus defines the vertices of the primitive as a quadrilateral, defines the vertices for the square, and then initiates the rendering (decryption). It selects the colour buffer for the output, reads the decrypted code into the buffer, and then decodes the decrypted code. The need to decode the decrypted code is because the encoded form uses 32 bits to store each value, but the virus requires only the low 16 bits. After decoding the code, the virus runs it. The code begins by detaching and deleting the shader object, and deleting the program and framebuffer objects, the texture arrays and the context, and the window. At this point, the main virus body is reached.

SHARE AND ENJOY

The main virus body begins by allocating some memory and copying itself to that memory. It registers a Structured Exception Handler in order to intercept any errors that occur during infection. It also initializes the random number generator by reading a value directly from the KUSER_SHARED_DATA structure in memory, instead of using an API such as GetTickCount(). The reason for this behaviour is because no APIs were resolved earlier that could be used as the seed for the random number generator, and the virus has not resolved any additional APIs at this point. The virus chooses a random number and assigns it to the modifier value. It then generates the table of decryption keys and encrypts the code at the same time.

The Random Number Generator (RNG) is interesting in itself, since it is neither the usual GetTickCount()-based randomizer nor the Knuth-inspired algorithm. Instead, the virus uses a complex RNG known as the 'Mersenne Twister', named after the kind of prime number at its heart. The virus author has used this RNG in almost all of his viruses for which he requires a source of random numbers.

hAPI hAPI, JOY JOY

The virus uses the LoadLibrary() API to load kernel32.dll, then it resolves the addresses of the required APIs. The virus uses hashes instead of names here, too. After retrieving the API addresses from kernel32.dll, the virus attempts to load 'sfc_os.dll'. If this fails, then it attempts to load 'sfc.dll'.

If either of these attempts succeed, then the virus resolves the SfcIsFileProtected() API. The reason the virus attempts to load both DLLs is that the API resolver in the virus code does not support import forwarding. The problem with import forwarding is that while the API name exists in the DLL, the corresponding API address does not. If a resolver is not aware of import forwarding, then it will retrieve the address of a string instead of the address of the code. In the case of the SfcIsFileProtected() API, the API is forwarded in *Windows XP* and later from sfc.dll to sfc_os.dll.

CULTURAL AWARENESS

The virus retrieves both the ASCII and Unicode versions of the required APIs. Due to the way in which the virus uses the APIs, it must swap the address of the CreateFileW() API and the CreateFileMappingA() API on the stack, even though this goes against the alphabetical ordering. The reason for the swap is that the virus requires the ASCII and Unicode versions of any given API to be sequential on the stack. This allows for transparent use of the appropriate API.

Specifically, the virus calls the GetVersion() API to determine the current *Windows* platform, and uses the result to select the appropriate API set (ASCII for *Windows 9x/Me*, and Unicode for *Windows NT* and later). Despite some of the virus author's more recent creations that support only *Windows NT* and later, this virus still supports *Windows 95*! This is because the infection engine used here is the same as the one we first saw the virus author use in 2002. In fact, the only updates to the code are the addition of support for Data Execution Prevention for *Windows XP* and later (by setting the executable bit in the section characteristics), and the DLL imagebase resolution for *Windows 7* and later (by walking the InLoadOrderModuleList list instead of the Structured Exception Handler list).

The GetVersion() API returns a bit that specifies whether the platform is *Windows 9x*-based (1) or *Windows NT*-based (0). The virus multiplies this value by four, adds the stack pointer value to it, and places the result in a register. Now, whenever the virus wishes to use an API which exists in the two forms, it simply calls the function relative to the register. As such, there is no need ever to check for the platform again. For example, the virus can call '[ebp+CreateFile]', where ebp contains the platform-specific value. If ebp is zero, the CreateFileW() API is called, and if ebp is four, the CreateFileA() API is called. This is why the reverse alphabetical order is important for the API addresses on the stack, and why the CreateFileW() and the CreateFileMappingA() API addresses had to be swapped.

FILTRATION SYSTEM

After finishing with the API trickiness, the virus searches for files. The virus searches for files in the current directory and all subdirectories, using a linked list instead of a recursive function. This is important from the point of view of the virus author, because the virus infects DLLs, whose stack size can be very small. The virus avoids any directory that begins with a '.'. This is intended to skip the '.' and '..' directories, but in *Windows NT* and later, directories can legitimately begin with this character if other characters follow. As a result, those directories will also be skipped.

Files are examined for their potential to be infected, regardless of their suffix, and will be infected if they pass a very strict set of filters. The first of these filters is that the file must not be protected by the System File Checker that exists in *Windows 98/Me*, and *Windows 2000* and later. Since directory searching on the *Windows 9x/Me* platforms uses ANSI paths, and since the SfcIsFileProtected() API requires a Unicode path, the virus converts the path from ANSI to Unicode, if appropriate, before calling the API.

The remaining filters include the condition that the file being examined must be a *Windows* Portable Executable file, a character mode or GUI application for the *Intel* 386+ CPU, that the file must have no digital certificates, and that it must have no bytes outside of the image. Additionally, if the file is a DLL, then it must have an entry point.

TOUCH AND GO

When a file is found that meets the infection criteria, it will be infected. The virus resizes the file by a random amount in the range of 4KB to 6KB in addition to the size of the virus. This data will exist outside of the image, and serve as the infection marker. Interestingly, despite its reliance on exceptions during the infection process, the virus does not check that exceptions are allowed by the host – the NO_SEH (No Structured Exception Handling) flag is not cleared in the header. If the flag is not cleared, *Windows* will terminate the application at the moment an exception occurs.

If relocation data is present at the end of the file, the virus will move the data to a larger offset in the file and place its own code in the gap that has been created. If no relocation data is present at the end of the file, the virus code will be placed there. The virus checks for the presence of relocation data by checking a flag in the PE header. However, this method is unreliable because even though the flag causes Address Space Layout Randomization to be disabled if it is set, *Windows* will ignore it and use the base relocation table directly if the image must be relocated due to address conflict.

The virus increases the physical size of the last section by the size of the virus code, then aligns the result. If the virtual size of the last section is smaller than its new physical size, then the virus sets the virtual size to be equal to the physical size, and increases and aligns the size of the image to compensate for the change. The virus also changes the attributes of the last section to include the executable and writable bits. The executable bit is set in order to allow the program to run if Data Execution Prevention is enabled, and the writable bit is set to allow the decryptor to write directly to the image.

The virus alters the host entry point to point to the last section, and saves the original entry point RVA in the virus body. This allows the virus to support both Address Space Layout Randomization and the proper infection of DLLs.

Once the infection is complete, the virus calculates a new file checksum, if one existed previously, before continuing to search for more files. Once the file searching has finished, the virus will allow the host code to execute by forcing an exception to occur, which transfers control to the handler that the virus registered. This technique appears a number of times in the virus code and is an elegant way to reduce the code size, in addition to functioning as an effective anti-debugging method. Since the virus has protected itself against errors by installing a Structured Exception Handler, the simulation of an error condition results in the execution of a common block of code to exit a routine. This avoids the need for separate handlers for successful and unsuccessful code completion. The handler unregisters itself, converts the original entry point from an RVA to a VA by adding the value from the ImageBaseAddress field in the Process Environment Block, and then transfers control to it.

CONCLUSION

The use of the GPU presents unimaginable challenges for anti-malware emulators, especially given that there are two major execution environments which have quite different behaviours, and there is no easy way to determine which one is intended to be used. Fortunately, the requirement for shaders to be stored in plain text in order to be compiled means that they can be extracted by anti-malware engines and treated like scripts. When combined with the data that uses the shader, an acceptable detection becomes reasonably straightforward, even in the absence of a complete decryption.