VIRUS ANALYSIS

LEAPS AND BOUNDS

Peter Ferrie
Symantec Security Response, USA

Imagine you're a virus writer, someone who specialises in one-of-a-kind viruses, and you want to do something really new and different. You want it to be entrypoint-obscuring, using a technique that no one has used before. You want a polymorphic decryptor, one that appears to be deceptively simple. Of course, you also want a 32-bit and a 64-bit version. What would it look like? The answer is W32/Bounds and W64/Bounds!AMD64.

THE IMPORT/EXPORT BUSINESS

Bounds uses an entrypoint-obscuring technique that no one has used before. The secret lies in the Bound Import Table (hence the name of the virus), but we need to start with the Import Table.

The Import Table begins with an array of Import Directory Tables, which describe the rest of the import information. Each Import Directory Table contains the name of the DLL from which functions will be imported, the time/date stamp of the DLL, an array of function names to import, and an array of host memory locations in which to store the function addresses.

BOUND IMPORT TABLE

The Bound Import Table works in conjunction with the Import Table, and can decrease loading time for some applications.

The idea is that the array of host memory locations can be filled in advance, given the knowledge of the DLL from which functions will be imported. The assumption is that for any given DLL, the combination of its name and its time/date stamp is unique. Thus, the functions inside that DLL will always have the same addresses, and any application that uses those functions can have those addresses stored in the Import Table.

However, not all DLLs are suitable for this kind of manipulation, which brings us to the Bound Import Table. The Bound Import Table is an array of DLL names and time/date stamps for the DLLs for which the addresses are considered permanent. When the operating system loads an application, it checks whether the application contains a Bound Import Table. If it does, then the operating system checks that each time/date stamp in the Bound Import Table matches the time/date stamp for each DLL that is named in the Import Table.

If the time/date stamp matches, then the addresses that correspond to the Import Table entry for that DLL are assumed to be correct, and are not updated. If a time/date stamp does not match, or there is no entry for it in the Bound Import Table when compared to the Import Table, the address will be fetched from the DLL that is named in the Import Table in the usual way.

BOUNDARY CONDITIONS

Now let us return to Bounds. The virus appears to be based on a member of the Chiton family. Indeed, we can see from a text string in the virus body that the author is the same.

The virus behaves in much the same way as several viruses we have seen previously. It begins by retrieving the address of kernel32.dll, using the address of the ExitProcess() API as a hint to where in memory to begin looking. After gaining access to kernel32.dll, the virus will retrieve the addresses of the API functions that it requires, using the CRC method to match the names, so no strings are visible in the code. The virus then searches for files in the current directory and all subdirectories.

Files are examined for their potential to be infected, regardless of their suffix, and will be infected if they conform to a very strict set of conditions.

The first of these is that the file is not protected by the System File Checker. The remaining filters include the condition that the file being examined must be a character mode or GUI application, that the file must have no digital certificates, and that it must have no bytes outside the image. The virus also requires a particular CPU, depending on the variant of the virus. For the W32 version, the required CPU is an *Intel 386+*; for the W64 version, the required CPU is an *AMD64* or *Intel EM64T*.

ENTRYPOINT OBSCURING

The virus's entrypoint-obscuring technique works by checking first if a file has a Bound Import Table. The virus does not create its own Bound Import Table, so if a file does not have one, it will not be a candidate for infection.

If the file does have a Bound Import Table, then the virus checks whether it contains an entry for kernel32.dll. The reason is that the virus wants to hook the ExitProcess() API within the Import Table, which is exported by kernel32.dll. Thus, if kernel32.dll is not referenced by the Bound Import Table, then even if ExitProcess appears in the Import Table, its address will be replaced by the operating system whenever the application loads.

If the Bound Import Table does have an entry for kernel32.dll, then the virus searches the Import Table for the Import Directory Table that corresponds to kernel32.dll. The virus examines only the first entry that refers to kernel32.dll, since this covers the most common case. (There may be more than one entry for any given DLL, and compilers such as *Borland Delphi* produce such files, but these are exceptions.)

Once the Import Directory Table that corresponds to kernel32.dll has been found, the virus searches within the array of host memory locations for a reference to the address of the ExitProcess() API. If the address is found, it is replaced within the array by the entrypoint of the virus.

When a file that meets the infection criteria is found, it will be infected. If relocation data exists at the end of the file, the virus will move the data to a larger offset in the file, placing its own code in the gap that has been created. If there is no relocation data at the end of the file, the virus code will simply be placed here.

POLYMORPHISM

The polymorphic decryptor in Bounds is perhaps the most interesting thing about the virus. In a typical decryptor, the CPU registers are initialized to fixed values, using any combination of MOV/XOR/PUSH+POP, after which the values might be altered in obscure ways to other values.

Bounds, on the other hand, uses no such instructions to initialize the registers. Instead, only two operations are used: AND and OR. These operations are used repeatedly to initialize the individual bits within each register.

In addition to these operations, the decryptor uses the rest of the set – ADC/ADD/SBB/SUB/XOR/CMP – to obfuscate the values temporarily. Once the registers have been initialized completely, these other operations are used to alter the values permanently. The use of ADC and SBB is not random – the virus keeps track of the carry flag status, so the effects of the ADC and SBB are known.

The result is something that looks like this (W32 version):

```
81 E5 59 E6 5A ED    and  ebp, 0ED5AE659h
81 D4 0A A1 DA F9    adc  esp, 0F9DAA10Ah
81 F1 D8 AF FF 07    xor  ecx, 007FFAFD8h
81 CE A2 46 3E CB    or   esi, 0CB3E46A2h
```

or this (W64 version):

```
48 81 CE 0E EB 43 23  or   rsi, 2343EB0Eh
48 81 F0 3D DD 81 52  xor  rax, 5281DD3Dh
48 81 D4 F4 BE 9A 43  adc  rsp, 439ABEF4h
48 81 CB 36 F7 90 42  or   rbx, 4290F736h
```

An impenetrable list of instructions, all the same length.

The virus generates a random number of these instructions before it generates the real decryptor instructions. Since the two are indistinguishable, the problem is knowing where to start.

The reason this is a problem is because the ESP register is similarly transformed. Since the register values are not known to the virus prior to initializing them in the decryptor, an anti-virus CPU emulator could simply start emulating from the first instruction and eventually reach the real entrypoint of the virus. At that point, the decryptor would start to initialize the registers in the usual manner, and it would work regardless of the initial values.

Normally, this would defeat the entrypoint obscuring technique. However, the use of the ESP register means that emulating from the first instruction will result in a value of the ESP register which has been transformed in an unpredictable way. This appears to be intentional, since the decryptor then writes decrypted values to the stack prior to placing them into executable memory.

If the ESP register has been randomized, then when the decryptor starts to write to the stack as shown below, the memory location that will be touched is no longer known to be the stack.

W32 version:

```
89 84 B4 D7 7C 94 1B  mov  [esp+esi*4+1B947CD7h], eax
```

W64 version:

```
89 B4 54 7C E9 AA 84  mov  [rsp+rdx*2-7B551684h], esi
```

If the memory location happens to point instead to the decryptor code, then the decryptor will be damaged, and the virus will not work in the emulated environment.

Even without that complication, a typical decryptor will write to memory in a linear manner, so an emulator could simply find the first memory reference, then start emulating from there, knowing what value will come next in memory, and eventually recovering all of the registers to decrypt the entire code. The author of Bounds was probably aware of this. While the writes to the stack memory are linear, the values that are written there do not correspond to linear addresses within the virus code. Instead, the virus writes a random number of values to the stack, then begins to pop some of them into the virus body, as shown below.

W32 version:

```
8F 84 FD D7 29 AF 2D  pop dword ptr
[ebp+edi*8+2DAF29D7h]
```

W64 version:

```
66 8F 05 4A B4 FF FF  pop word  ptr [rip-00004BB6h]
```

The 32-bit version uses the registers to decode to a random address located earlier than the current position, not exactly at the start of the decryptor.

The 64-bit version uses RIP-relative addressing to overwrite the decryptor from the initial address. The reason for the RIP-relative addressing has to do with a limitation of register assignment: 64-bit CPUs do not support 64-bit immediate values. Therefore, the virus cannot perform 64-bit arithmetic to set the 64-bit CPU registers to point to the memory address of the decryptor.

The entire virus is never stored on the stack all at once – some values are placed onto the stack, and some values are then removed from the stack. Sometimes, more values can be written to the stack before all values are removed from the stack; sometimes all values are removed from the stack before more values are written to the stack.

OOPS

Every value in the virus is decoded individually using this method, resulting in very large decryptors. Since the size of the decryptor is hard to guess, it is easy to understand how a miscalculation could creep into the virus code.

Sure enough, while the virus always allocates enough bytes to hold the decryptor, a bug sometimes results in not all of the bytes being copied into the host. Both the 32-bit and 64-bit versions are affected, but in the case of the 64-bit version, the decryptor almost always ends before the cutoff point, so the bug is not so obvious.

CONCLUSION

So imagine that you're a virus writer, someone who specialises in one-of-a-kind viruses, and you want to do something that's really new and different. What should it be? How about quitting?

W32/Bounds, W64/Bounds!AMD64

Type:	Direct-action parasitic appender/inserter.
Size:	246kb (W32), 583kb (W64).
Infects:	*Windows* PE files (32-bit for W32, 64-bit AMD64 for W64).
Payload:	None.
Removal:	Delete infected files and restore them from backup.