# Injection as a way of life

virusbulletin.com/virusbulletin/2010/09/injection-way-life/

2010-09-01

**Raul Alvarez**

Fortinet, USA

**Editor:** Helen Martin

**Abstract**

Injecting code into a process is not a new technology, but it is still used by most prevalent malware today. Raul Alvarez dissects two examples of recent prevalent malware and shows how they inject their code into a running process.

Memory-residency is employed by malware to ensure that it is always active on the system. Techniques have been tried and tested; the good old DOS infector used Terminate and Stay Resident – TSR (using the infamous INT 21h function 31h) – and another well-known technique is code injection. Injecting code into a process is not a new technology, but it is still used by most prevalent malware today.

The main idea behind code injection is that the malware embeds itself into a running process to maintain residency. Well, of course we already know that. Behavioural analysis can tell us that a certain application has been infected; we use different tools to determine if a thread has been injected into a certain process. And lots of malware analysis online will tell us that a given piece of malware injects its code into a running process. But little has been said about the actual code-by-code steps that malware uses to inject its code.

This article will dissect two examples of recent prevalent malware and show how they inject their code into a running process. We will start with a variant of Virut, detected by *Fortinet* as W32/Virut.CE, which uses Zw*** APIs to implement code injection. Then we will explain how a variant of OnlineGames embeds its code into the Explorer.exe process.

## Part I: Virut, Virut and Virut

Virut's code injection starts by modifying the access token's privilege; the access token contains the security information for a logon session. Every time a user logs on, the system generates an access token which is also used by every process and application executed by

the current user.

Virut uses the ZwOpenProcessToken API in order to get the handle for the access token of the user. After acquiring the handle of the token, Virut resolves the address of the LookupPrivilegeValueA API by using the LoadLibrary and GetProcAddress APIs. Virut calls for the LookupPrivilegeValueA API to get the locally unique identifier (LUID) for SeDebugPrivilege, also known as SE_DEBUG_NAME; this is a privilege required for memory modification of a given process, which Virut needs to freely inject its code. This is immediately followed by a call to the ZwAdjustPrivilegesToken API, which adjusts the privilege of the access token based on the new LUID.

Setting the privilege of the access token to SeDebugPrivilege enables Virut to perform code injection with ease; the malware doesn't need to concern itself with any issue regarding the opening of a process, writing to it, hooking code in its shared memory space, creating threads and executing instructions. Once the privilege is set to the proper attributes, Virut proceeds to enumerate the running processes.

## Browsing active processes

Virut is a polymorphic virus, and after decryption and resolving the necessary APIs we can see that most variants don't go far from their intended purpose.

A typical way to enumerate the active processes in a given system starts with a call to the CreateToolhelp32Snapshot API; Virut calls the CreateToolhelp32Snapshot API to get a snapshot of the system. Using this API, a piece of malware can get a snapshot of every module, thread, heap and process, all depending on the dwFlags parameter supplied to it; Virut uses TH32CS_SNAPPROCESS to include all processes in the system. The malware enumerates the processes one by one using a single call to the Process32First API and concurrent calls to the Process32Next API. These two APIs use the PROCESSENTRY32 structure generated by the CreateToolhelp32Snapshot API which was called earlier (see Figure 1).

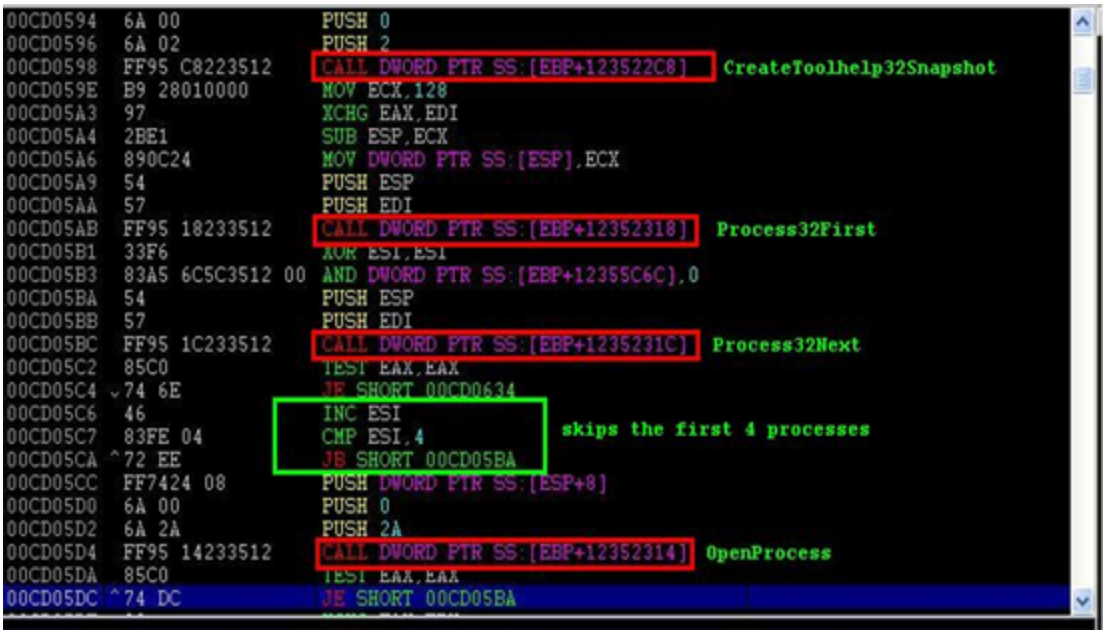



**Figure 1. Code snippets on enumerating the active processes and the skipping of the first four processes.**

While enumerating the list of processes, Virut intentionally skips the first four processes without even checking their names. Interestingly, most often, the Winlogon.exe process is the fifth on the list. Winlogon is the first process into which Virut injects its code; Winlogon is infected not by choice but for the simple reason that it is one of the first processes available for Virut infection.

The next logical step, after acquiring the handle of the process to infect, is to open it. Virut opens the process by calling the OpenProcess API with the CREATE_THREAD|VM_OPERATION|VM_WRITE access parameter; this enables the malware to create a thread in the given process and to write the codes to inject.

## Mapping a section of memory

Before the code injection stage, Virut creates a section of memory named \BaseNamedObjects\houtVt; this contains the complete code to be injected into the process. This is evident on any process that has already been injected with Virut's code. *Process Explorer* or any tool that can show the events, keys, sections and other objects of a process can be used to determine if the process is already infected.

Since the section already exists, Virut calls the ZwMapViewOfSection API to map a copy of \BaseNamedObjects\houtVt to the current process that it is working on. The actual Virut code is copied to the process's memory space by mapping the section of memory. Mapping a section of memory is like sharing a DLL in a process's memory space, thereby giving Winlogon (or other process) access rights to the section. Any viable code within the

\BaseNamedObjects\houtVt section can now be executed by any process that maps it; calling a function from within the section is just a matter of pointing it to the right memory address.

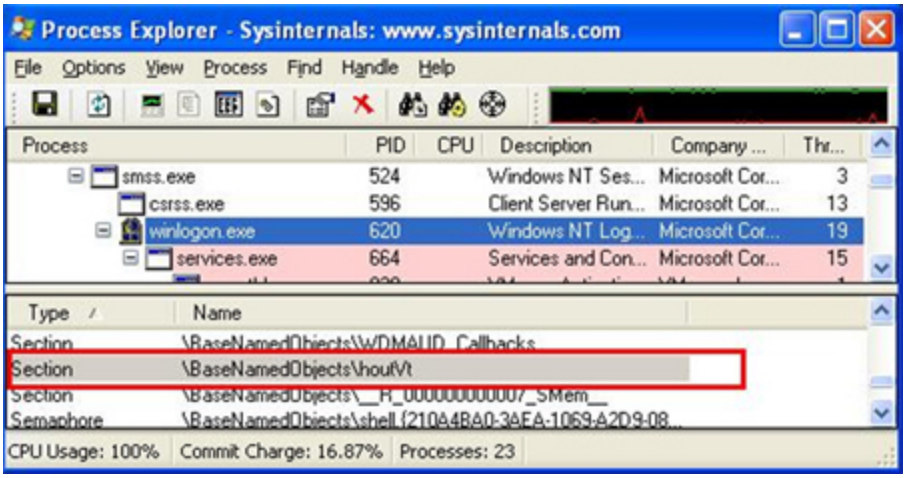



**Figure 2. The mapped section named \BaseNamedObjects\houtVt in the Winlogon.exe process.**

## Hooking NTDLL.dll

Hooking is an old technique used by malware; old DOS viruses hooked INT functions to redirect calls to their code and new malware hooks DLL functions in a similar way. When a call to the hook function is performed, execution transfers to the malware code, which is executed, and then control is transferred back to the original function routine; this is basically what happened to the hooked function.

Virut hooks some APIs from NTDLL, of a given process, simply by replacing the MOV EAX,yy instruction with a CALL xxxxxxxx, an address pointed to by the mapped \BaseNamedObjects\houtVt section. It uses the ZwProtectVirtualMemory to change the protection mode of NTDLL attached to the process to PAGE_READWRITE mode then proceeds to hook it by writing the CALL instruction using ZwWriteVirtualMemory. The PAGE_READWRITE mode ensures that the shared NTDLL can be written to by a call to ZwWriteVirtualMemory.

Virut hooks the following APIs:

- ZwCreateFile
- ZwOpenFile
- ZwCreateProcess
- ZwCreateProcessEx

- ZwQueryInformationProcess

By hooking the APIs above, Virut's code becomes available whenever a file is read, opened or created, and whenever a process is opened, created or queried.

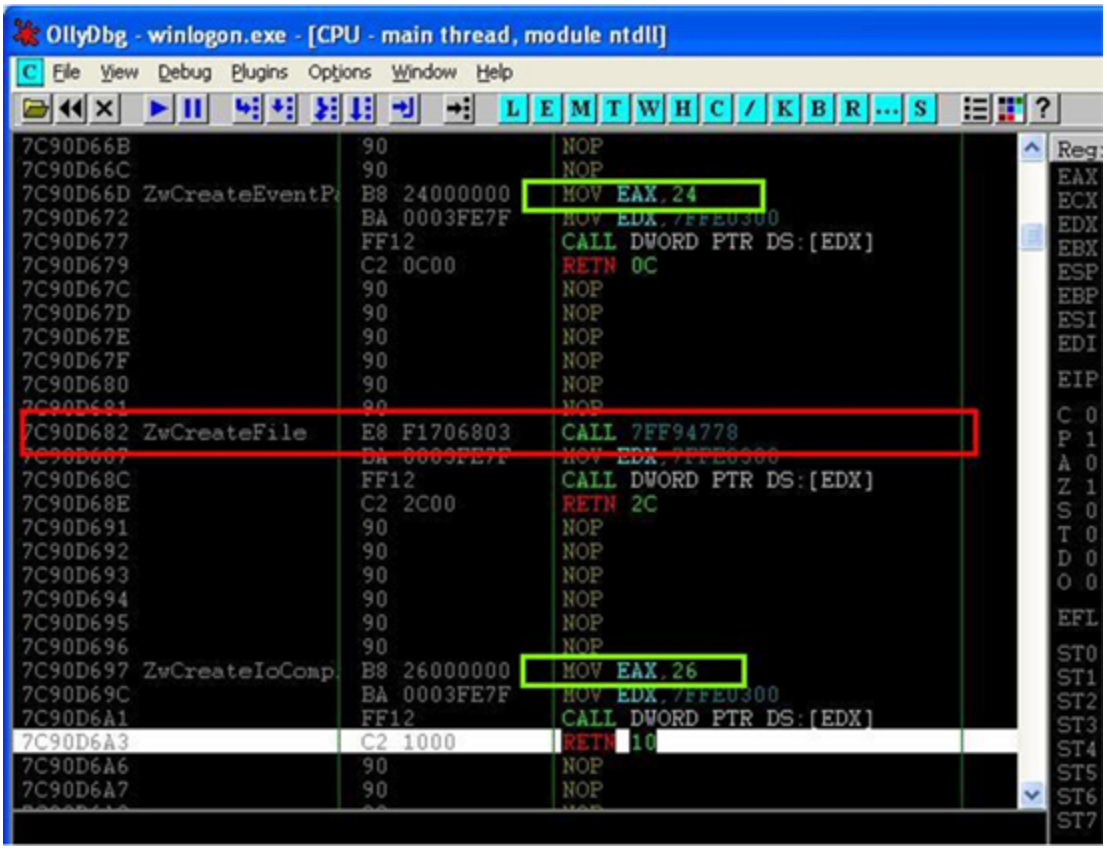



**Figure 3. The hooked NTDLL.dll; the green boxes are the normal codes and the red box is the hooked ZwCreateFile API; the MOV instruction was replaced by a call to the mapped section.**

## Running the thread

Once everything is set – privileges have been set up, a process has been selected to infect, a section of memory has been mapped, and DLL hooked – the last thing for Virut to do is to execute a thread remotely.

Virut creates a remote thread using a call to CreateRemoteThread, with dwCreationFlags equal to 0. It executes the thread immediately. When a remote thread is created, it can be suspended or, in this case, executed immediately. Virut executes the thread as soon as it is created to speed up the infection process. When all is well, Virut relinquishes its control to the process and proceeds to look for a new process to inject its code into. As we now know, Virut doesn't only infect the Winlogon.exe process; it keeps looking for more processes to inject code into.

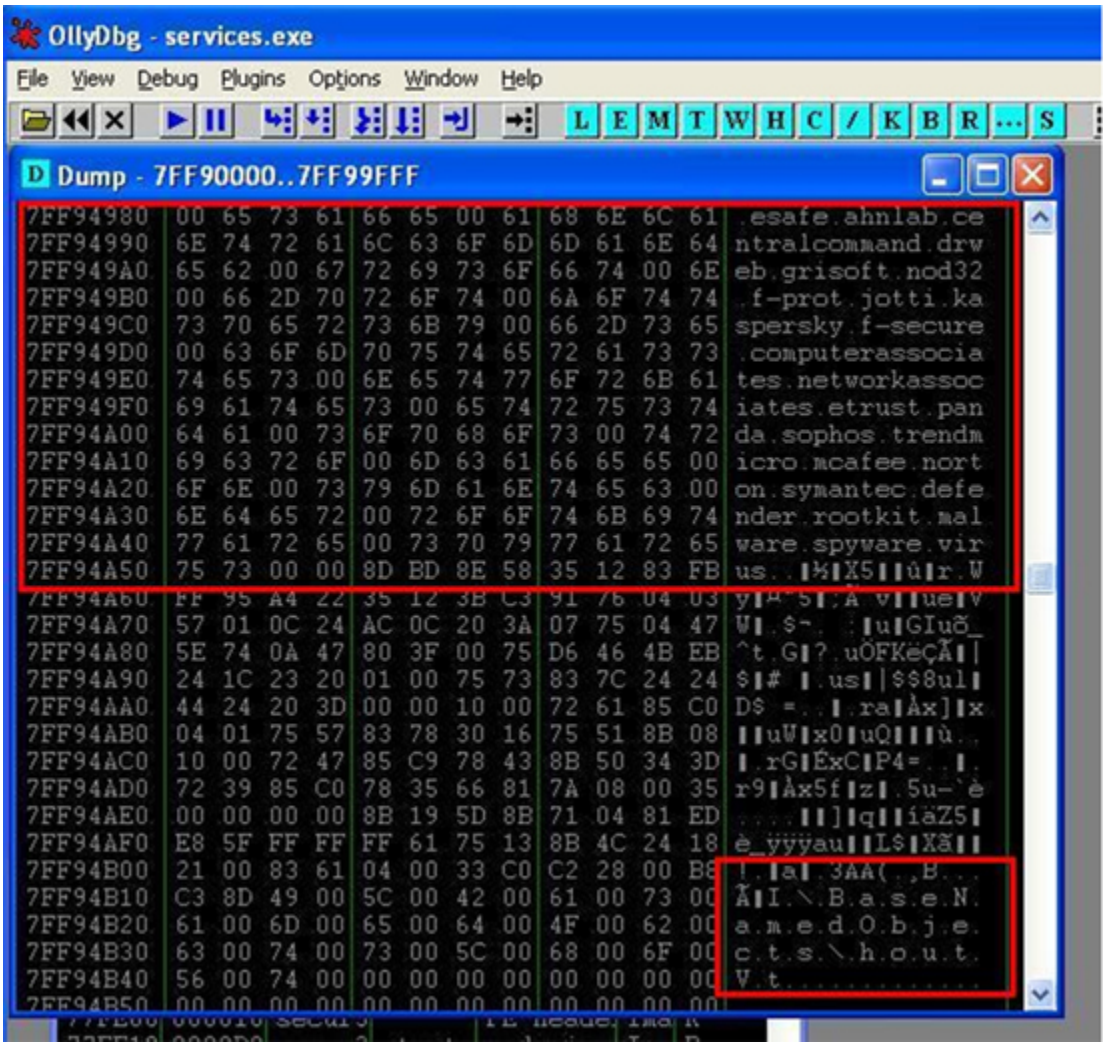


**Figure 4. Strings found in services.exe's process indicative of Virut's mapped section**

As discussed earlier, we can easily check if a process is infected by looking for the presence of the \BaseNamedObjects\houtVt section. To be certain, we can browse the process's memory and look for a sign that Virut is really there. Most often, Virut's favourite location is 7FF90000h and the size is 0A000h; however, some processes use that location, so Virut uses the next location on the block, 7FFA0000h, with the same virus size. Virut's code within the process's memory is not encrypted, thereby giving us the strings to look for. We can see strings like AV company names, the name of the section, resolved names of APIs, IRC-related strings, and registry key strings.

Virut's method of code injection is fairly common amongst malware. That being said, we will now look at another method of injecting code.

## Part II: Online Gaming

The next piece of malware we will look at is a variant of OnlineGames. Most malware families have their own style of decryption routine, and the same is true when it comes to the process of code injection. We have already noted that a variant of Virut skips the first four

processes and injects its code into Winlogon.exe and succeeding processes after that. In this variant of OnlineGames, Explorer.exe is the sole target.

We will discuss some commonalities of Virut and OnlineGames when selecting the process for injection, how codes are copied to the process's memory space and what the remote code looks like before it is executed in the process.

## Choosing Explorer.exe

Like Virut, OnlineGames uses the CreateToolhelp32Snapshot to enumerate the processes active in the system – using TH32CS_SNAPPROCESS as the dwFlags parameter. Although the malware knows what process to infect, it still uses the same pair of Process32First and Process32Next APIs to locate the pID (process ID) of Explorer.exe.

Interestingly enough, the malware has a longer code routine just to copy a string (process name) to a memory location; it also has a longer code routine comparing the process name to look for the 'Explorer.exe' string. Instead of copying the string using a single instruction, the malware copies it, character by character, to the memory. To compare the string, the malware first counts the number of characters of the name of the given process and compares it to the length of the 'Explorer.exe' string. If the size of the two strings matches, then it proceeds to check each character of both strings. After a successful attempt at getting the right process name, 'Explorer.exe', the malware captures the pID of the process.

The pID of Explorer.exe is now used by OpenProcess, with an access parameter of PROCESS_ALL_ACCESS – all possible access rights.

## Writing codes to process

Virut's method of putting its codes into memory is by mapping the entire \BaseNamedObjects\houtVt section and hooking NTDLL.dll APIs linking to the mapped section. In comparison, OnlineGames uses the WriteProcessMemory API to write codes into the Explorer.exe process. But in this respect, the code written to the process's memory space is not the whole virus code yet.

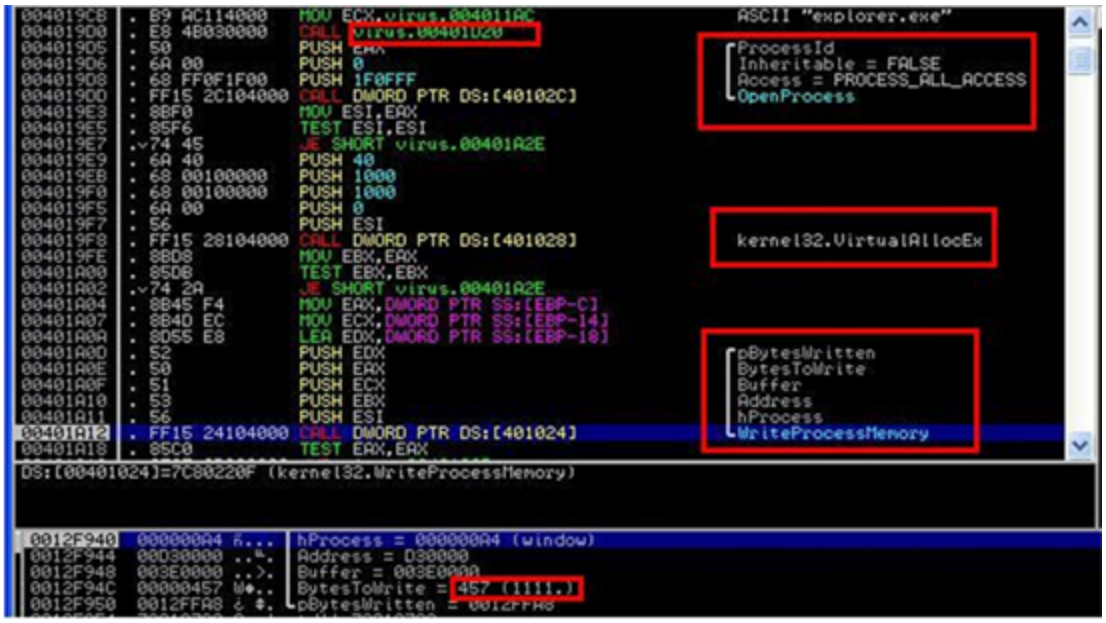



**Figure 5. Code snippet showing the call to the OpenProcess, VirtualAllocEx and WriteProcessMemory APIs. It also shows a certain call to a memory location, 00401D20, where the pID searching can be found. Lastly, it shows where the length of the codes, 457h(1111), is used.**

Before OnlineGames writes some of its code to the process, it uses the VirtualAllocEx API to reserve some memory space from the process; the resulting value is the base address where OnlineGames can write to. It then proceeds to write 457h(1111) bytes of code – which, of course, is not the whole virus code.

## Intercepting the Remote Thread

The WriteProcessMemory API is only called once within the malware body; it only writes 457h(1111) bytes of code. We can only assume that there should be more to it than just writing that small piece of code. Does OnlineGames use the same technique of mapping a section of memory to the running process as Virut? The answer is no, OnlineGames doesn't use memory mapping and it doesn't hook any functions in NTDLL, or any DLL for that matter. But how can OnlineGames copy the whole malware code to Explorer.exe? The answer lies in the 457h bytes of memory the malware wrote earlier.

The only logical way to look for the answer is to intercept the execution of the 457h mystery bytes. A remote thread is created when OnlineGames uses the CreateRemoteThread API; it points to the base address, the starting address of the 457h bytes of code taken from the call to VirtualAllocEx API earlier. Once the thread is created and Explorer.exe is within a debugger, such as OllyDbg, we will see a message box displaying 'Module "cvasds0" has entry point outside the code (as specified in the PE header). Maybe this file is self-extracting or self-modifying. Please keep it in mind when setting breakpoints!' (see Figure 6). Note that the message will only show when Explorer.exe is within a debugger context.

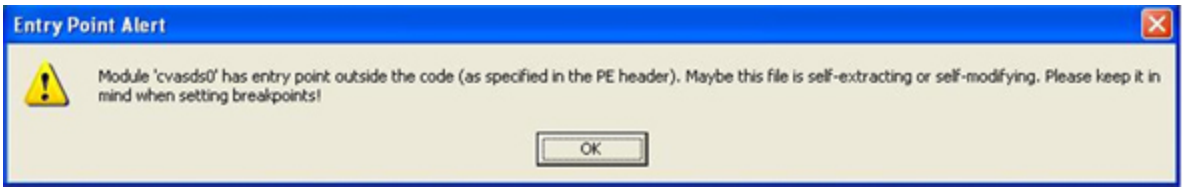



**Figure 6. Message displayed when CreateRemoteThread API from OnlineGames was executed.**

Knowing that a file named ‘cvasds0’ is being accessed by Explorer.exe, it is safe to say that it is the same malware file that we are looking for. We haven’t intercepted the code yet, so we need to go back and execute the CreateRemoteThread API; this time we are in intercept mode. Figure 8 shows a snippet of the intercepted code, the 457h bytes of code copied earlier using the WriteProcessMemory API.

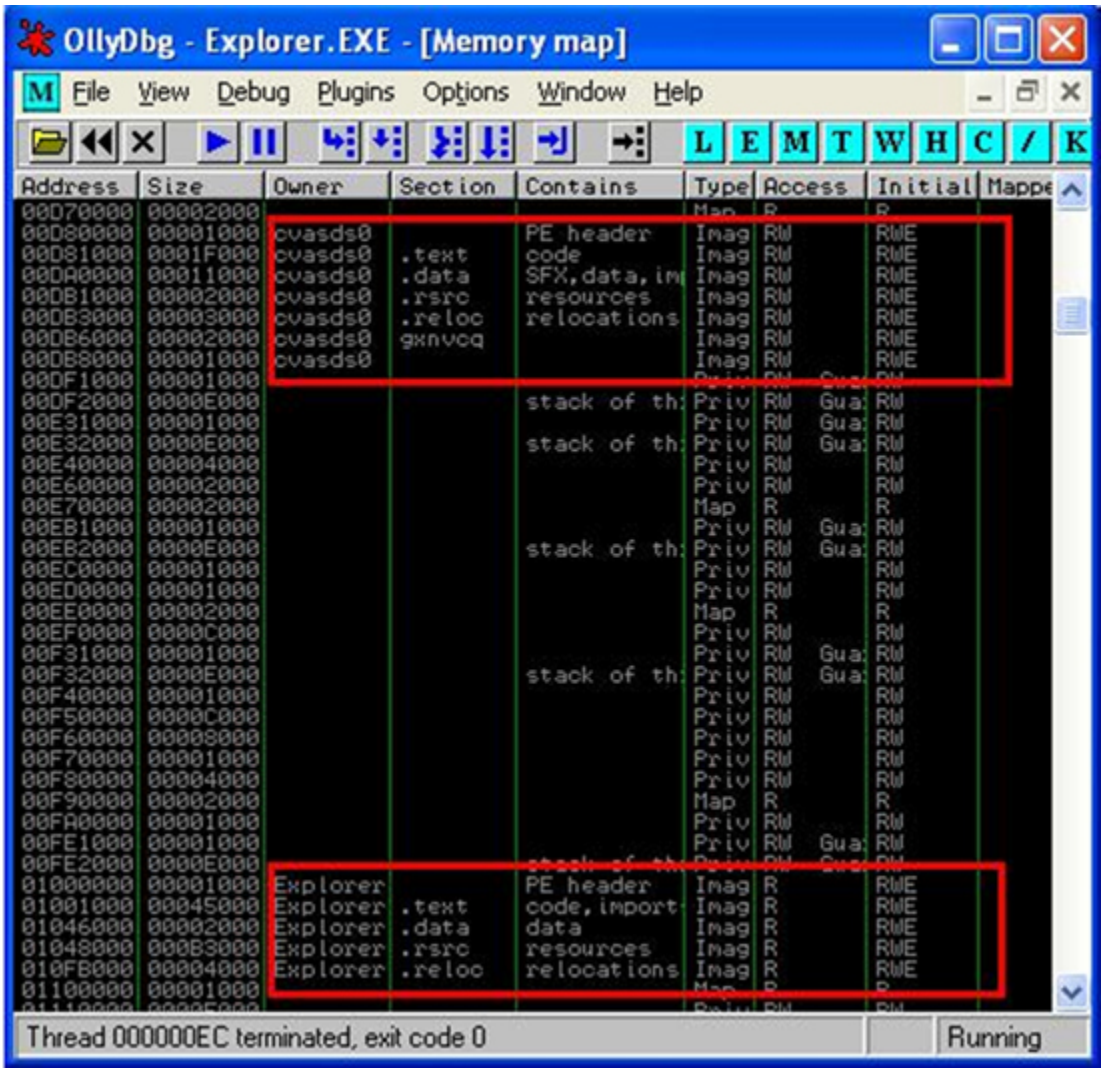
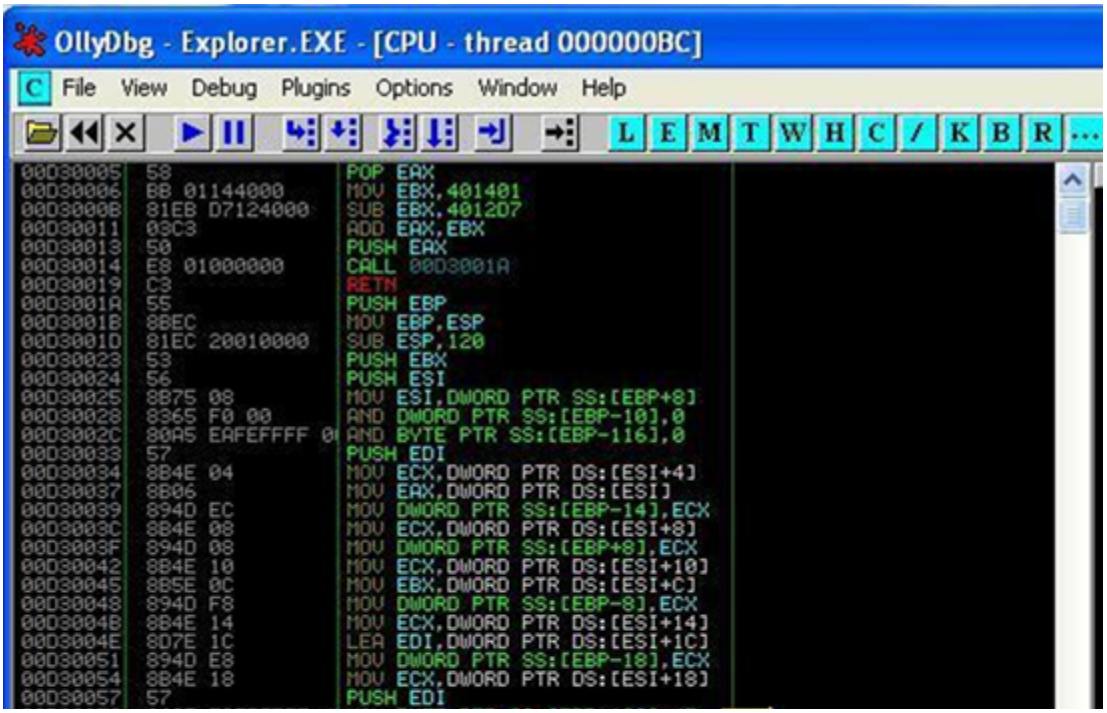



**Figure 7. Memory map of the 'Explorer.exe' process within OllyDbg. It shows the map view of 'Explorer.exe' and the new file 'cvasdds0'.**



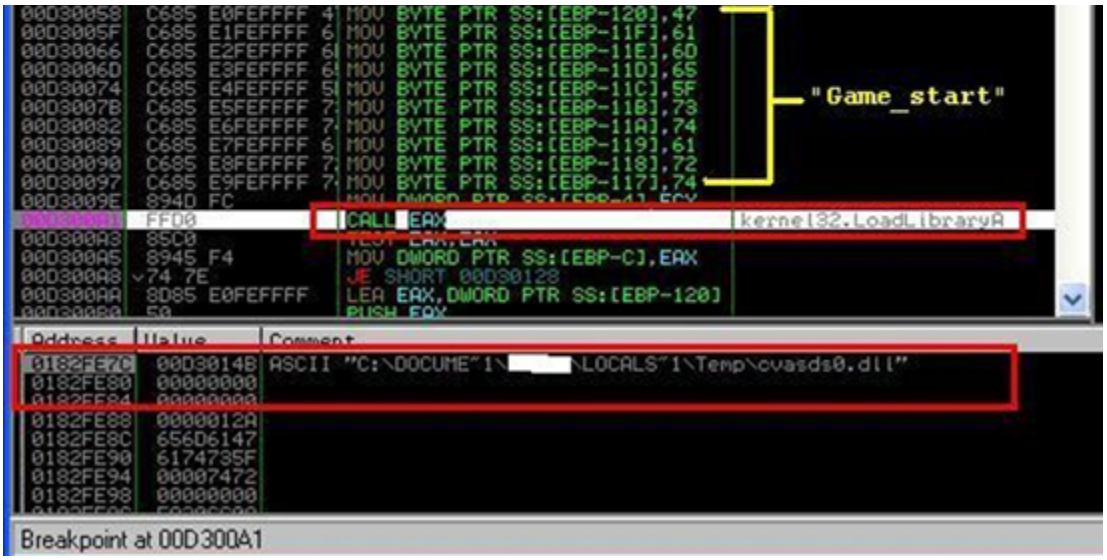



**Figure 8. Code snippet of 457h bytes of code copied to the memory space of Explorer.exe, showing the call to the LoadLibraryA API and the string 'Game_start'.**

The 457h bytes of code is responsible for loading 'cvasds0' into the Explorer.exe process; it calls the LoadLibraryA API to load the file, actually a DLL, that can be found at the 'c:\DOCUME~1\[varies]\LOCALS~1\Temp\' folder. 'cvasds0.dll' is a DLL file dropped by OnlineGames at an earlier stage of the malware's execution. The 457h bytes of code also contains the string 'Game_start', which is encoded character by character.

## Conclusion

We have seen two different pieces of malware, each demonstrating different skills in performing code injection. They both start off by using the basic techniques of enumerating, searching and opening a process. Then, they each go a different way when they start preparing the code to be injected. Virut has chosen to map its code to the process and hook NTDLL, while OnlineGames has chosen to inject a small amount of code into Explorer.exe and let it load its complete code in a library form. There are several more tricks for code injection out there; we will encounter them in one way or another, yet they will always have one thing in common - the process.

## Latest articles:

### Nexus Android banking botnet – compromising C&C panels and dissecting mobile AppInjects

Aditya Sood & Rohit Bansal provide details of a security vulnerability in the Nexus Android botnet C&C panel that was exploited to compromise the C&C panel in order to gather threat intelligence, and present a model of mobile AppInjects.

## Cryptojacking on the fly: TeamTNT using NVIDIA drivers to mine cryptocurrency

TeamTNT is known for attacking insecure and vulnerable Kubernetes deployments in order to infiltrate organizations' dedicated environments and transform them into attack launchpads. In this article Aditya Sood presents a new module introduced by…

## Collector-stealer: a Russian origin credential and information extractor

Collector-stealer, a piece of malware of Russian origin, is heavily used on the Internet to exfiltrate sensitive data from end-user systems and store it in its C&C panels. In this article, researchers Aditya K Sood and Rohit Chaturvedi present a 360…

## Fighting Fire with Fire

In 1989, Joe Wells encountered his first virus: Jerusalem. He disassembled the virus, and from that moment onward, was intrigued by the properties of these small pieces of self-replicating code. Joe Wells was an expert on computer viruses, was partly…

## Run your malicious VBA macros anywhere!

Kurt Natvig wanted to understand whether it's possible to recompile VBA macros to another language, which could then easily be 'run' on any gateway, thus revealing a sample's true nature in a safe manner. In this article he explains how he recompiled…

Bulletin Archive