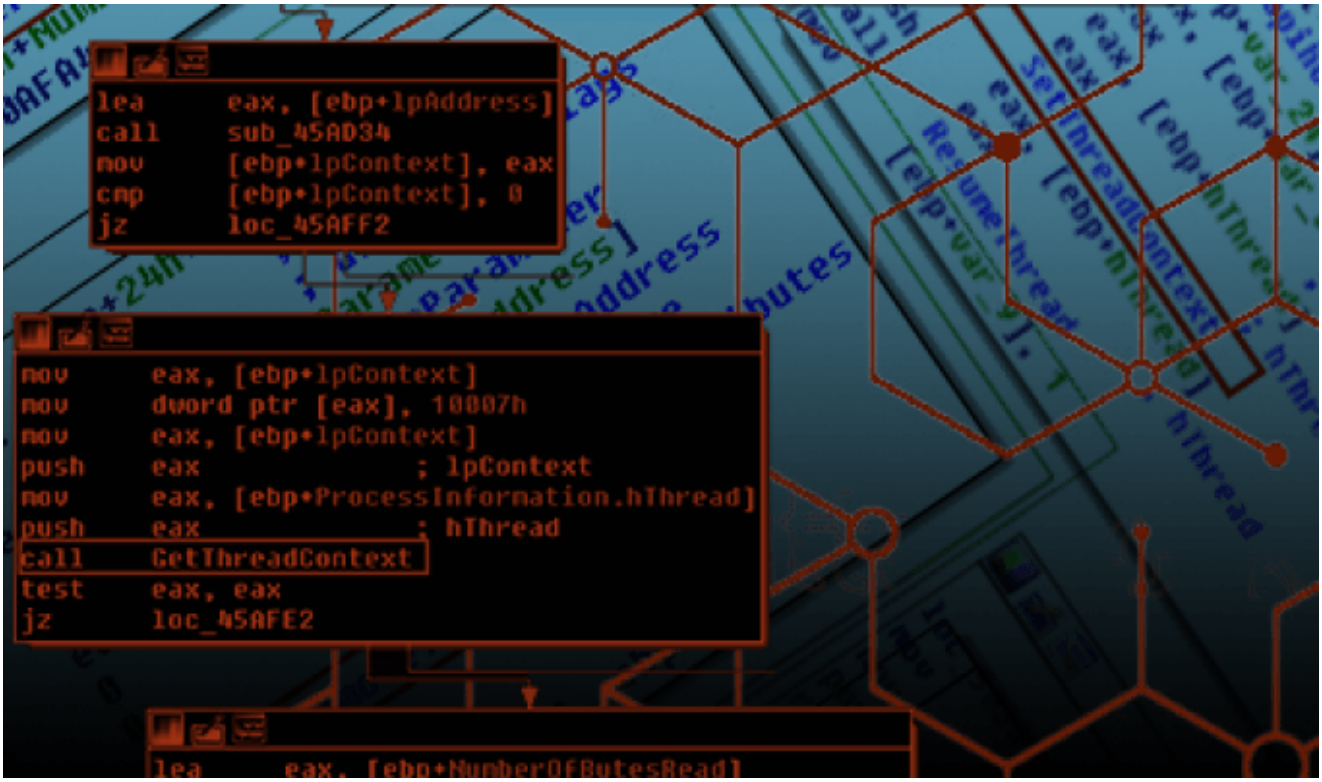
Ten process injection techniques: A technical survey of common and trending process injection techniques

endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process

July 18, 2017



18 July 2017Tech Topics

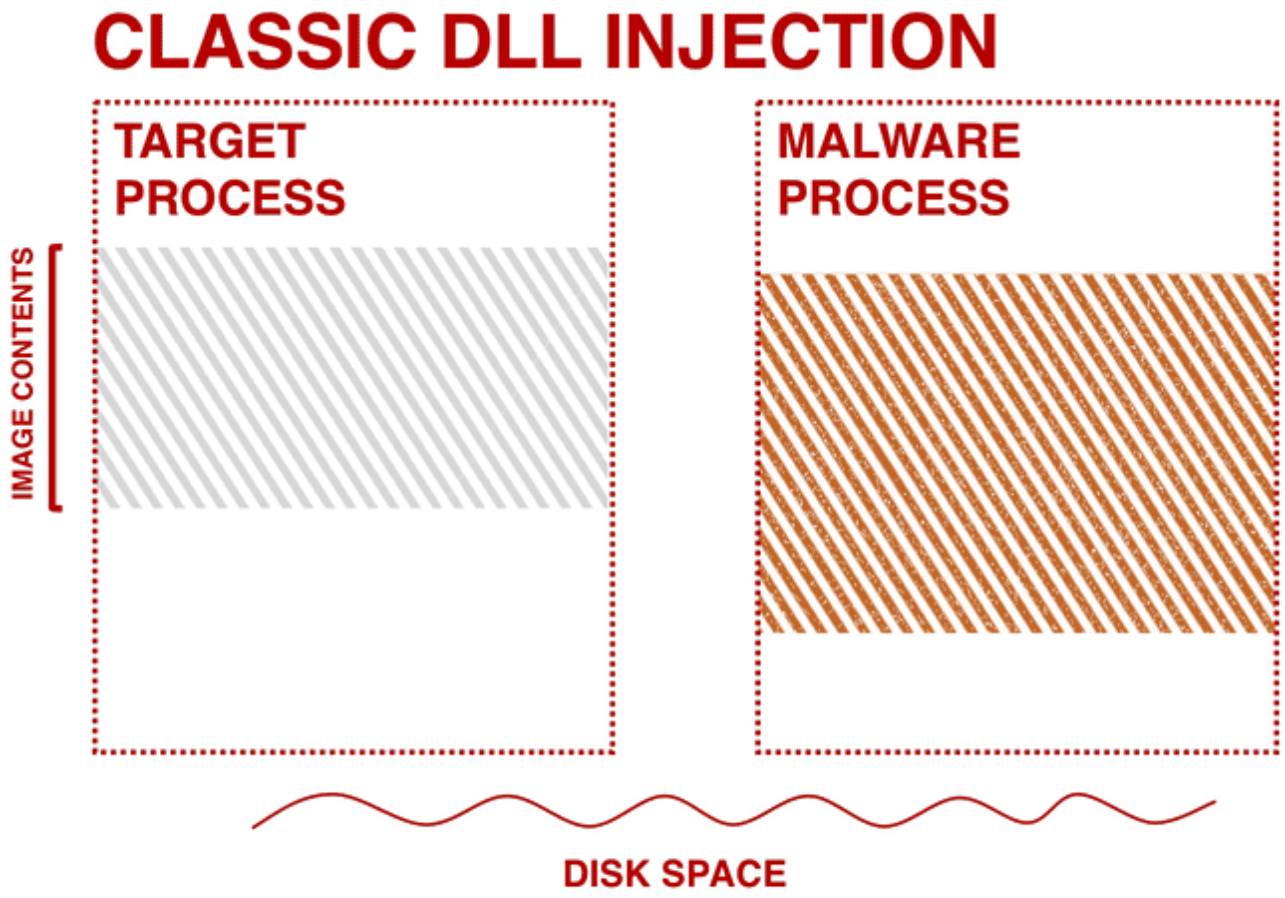
By
Ashkan Hosseini
Share

Editor's Note: Elastic joined forces with Endgame in October 2019, and has migrated some of the Endgame blog content to elastic.co. See Elastic Security to learn more about our integrated security solutions.

Process injection is a widespread defense evasion technique employed often within malware and fileless adversary tradecraft, and entails running custom code within the address space of another process. Process injection improves stealth, and some techniques also achieve persistence. Although there are numerous process injection techniques, in this blog I present ten techniques seen in the wild that run malware code on behalf of another process. I additionally provide screenshots for many of these techniques to facilitate reverse engineering and malware analysis, assisting detection and defense against these common techniques.

1. CLASSIC DLL INJECTION VIA CREATEREMOTETHREAD AND LOADLIBRARY

This technique is one of the most common techniques used to inject malware into another process. The malware writes the path to its malicious dynamic-link library (DLL) in the virtual address space of another process, and ensures the remote process loads it by creating a remote thread in the target process.



The malware first needs to target a process for injection (e.g. svchost.exe). This is usually done by searching through processes by calling a trio of Application Program Interfaces (APIs): CreateToolhelp32Snapshot, Process32First, and Process32Next. CreateToolhelp32Snapshot is an API used for enumerating heap or module states of a specified process or all processes, and it returns a snapshot. Process32First retrieves

information about the first process in the snapshot, and then Process32Next is used in a loop to iterate through them. After finding the target process, the malware gets the handle of the target process by calling OpenProcess.

As shown in Figure 1, the malware calls VirtualAllocEx to have a space to write the path to its DLL. The malware then calls WriteProcessMemory to write the path in the allocated memory. Finally, to have the code executed in another process, the malware calls APIs such as CreateRemoteThread, NtCreateThreadEx, or RtlCreateUserThread. The latter two are undocumented. However, the general idea is to pass the address of LoadLibrary to one of these APIs so that a remote process has to execute the DLL on behalf of the malware.

CreateRemoteThread is tracked and flagged by many security products. Further, it requires a malicious DLL on disk which could be detected. Considering that attackers are most commonly injecting code to evade defenses, sophisticated attackers probably will not use this approach. The screenshot below displays a malware named Rebhip performing this technique.

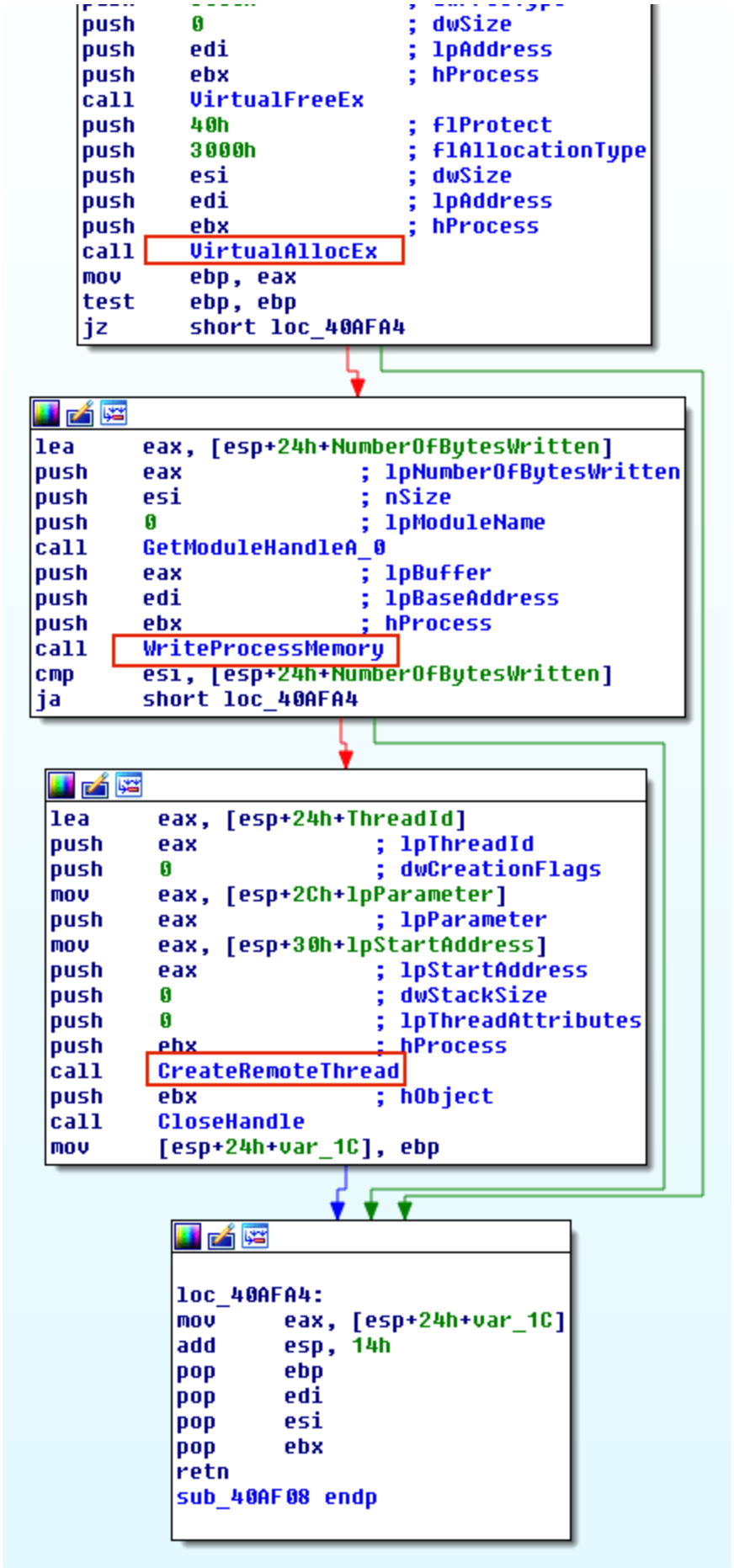
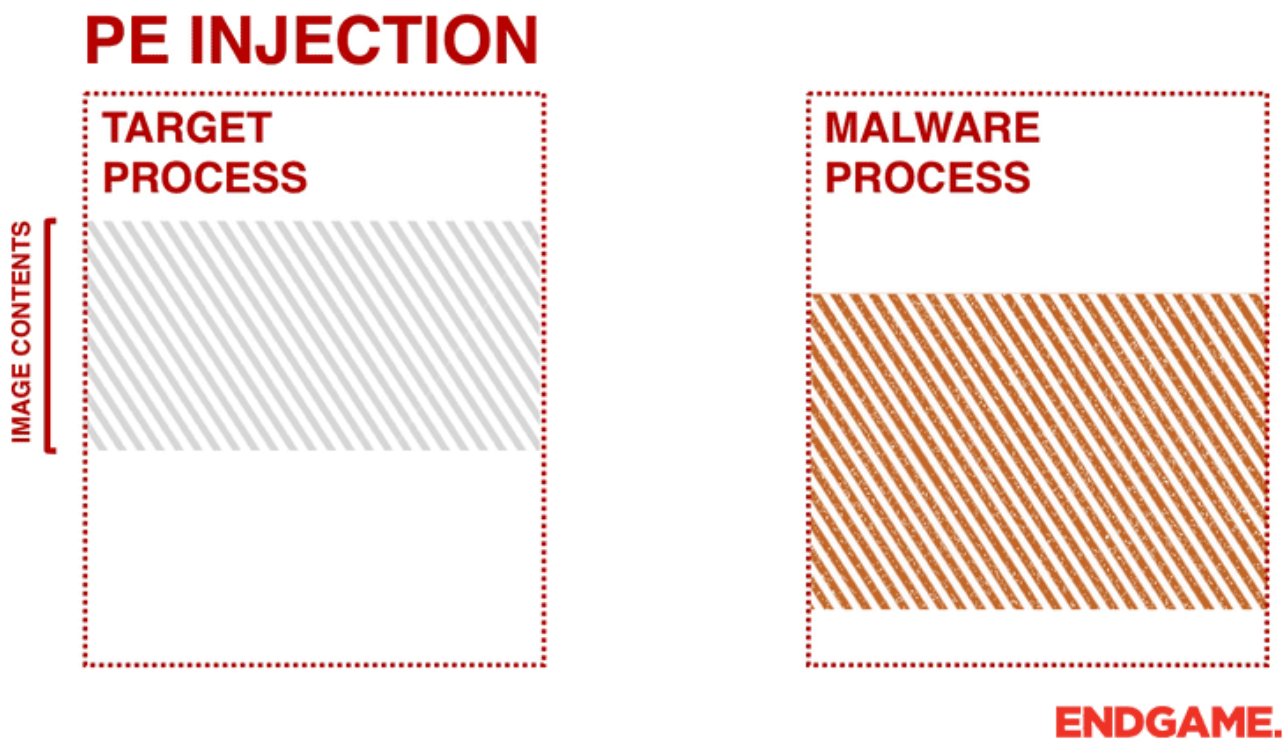

push 0 ; dwSize
push edi ; lpAddress
push ebx ; hProcess
call VirtualFreeEx
push 40h ; flProtect
push 3000h ; flAllocationType
push esi ; dwSize
push edi ; lpAddress
push ebx ; hProcess
call VirtualAllocEx
mov ebp, eax
test ebp, ebp
jz short loc_40AFA4
lea eax, [esp+24h+NumberOfBytesWritten]
push eax ; lpNumberOfBytesWritten
push esi ; nSize
push 0 ; lpModuleName
call GetModuleHandleA_0
push eax ; lpBuffer
push edi ; lpBaseAddress
push ebx ; hProcess
call WriteProcessMemory
cmp esi, [esp+24h+NumberOfBytesWritten]
ja short loc_40AFA4
lea eax, [esp+24h+ThreadId]
push eax ; lpThreadId
push 0 ; dwCreationFlags
mov eax, [esp+2Ch+lpParameter]
push eax ; lpParameter
mov eax, [esp+30h+lpStartAddress]
push eax ; lpStartAddress
push 0 ; dwStackSize
push 0 ; lpThreadAttributes
push ebx ; hProcess
call CreateRemoteThread
push ebx ; hObject
call CloseHandle
mov [esp+24h+var_1C], ebp
loc_40AFA4:
mov eax, [esp+24h+var_1C]
add esp, 14h
pop ebp
pop edi
pop esi
pop ebx
retn
sub_40AF08 endp

Figure 1: Rebhip worm performing a typical DLL injection
Sha256: 07b8f25e7b536f5b6f686c12d04edc37e11347c8acd5c53f98a174723078c365

2. PORTABLE EXECUTABLE INJECTION (PE INJECTION)

Instead of passing the address of the LoadLibrary, malware can copy its malicious code into an existing open process and cause it to execute (either via a small shellcode, or by calling CreateRemoteThread). One advantage of PE injection over the LoadLibrary technique is that the malware does not have to drop a malicious DLL on the disk. Similar to the first technique, the malware allocates memory in a host process (e.g. VirtualAllocEx), and instead of writing a “DLL path” it writes its malicious code by calling WriteProcessMemory. However, the obstacle with this approach is the change of the base address of the copied image. When a malware injects its PE into another process it will have a new base address which is unpredictable, requiring it to dynamically recompute the fixed addresses of its PE. To overcome this, the malware needs to find its relocation table address in the host process, and resolve the absolute addresses of the copied image by looping through its relocation descriptors.



This technique is similar to other techniques, such as reflective DLL injection and memory module, since they do not drop any files to the disk. However, memory module and reflective DLL injection approaches are even stealthier. They do not rely on any extra Windows APIs (e.g., CreateRemoteThread or LoadLibrary), because they load and execute themselves in the memory. Reflective DLL injection works by creating a DLL that maps itself into memory when executed, instead of relying on the Window’s loader. Memory Module is

similar to Reflective DLL injection except the injector or loader is responsible for mapping the target DLL into memory instead of the DLL mapping itself. In a previous blog post, these two in memory approaches were discussed extensively.

When analyzing PE injection, it is very common to see loops (usually two “for” loops, one nested in the other), before a call to CreateRemoteThread. This technique is quite popular among crypters (softwares that encrypt and obfuscate malware). In Figure 2, the sample unit test is taking advantage of this technique. The code has two nested loops to adjust its relocation table that can be seen before the calls to WriteProcessMemory and CreateRemoteThread. The “and 0x0fff” instruction is also another good indicator, showing that the first 12 bits are used to get the offset into the virtual address of the containing relocation block. Now that the malware has recomputed all the necessary addresses, all it needs to do is pass its starting address to CreateRemoteThread and have it executed.

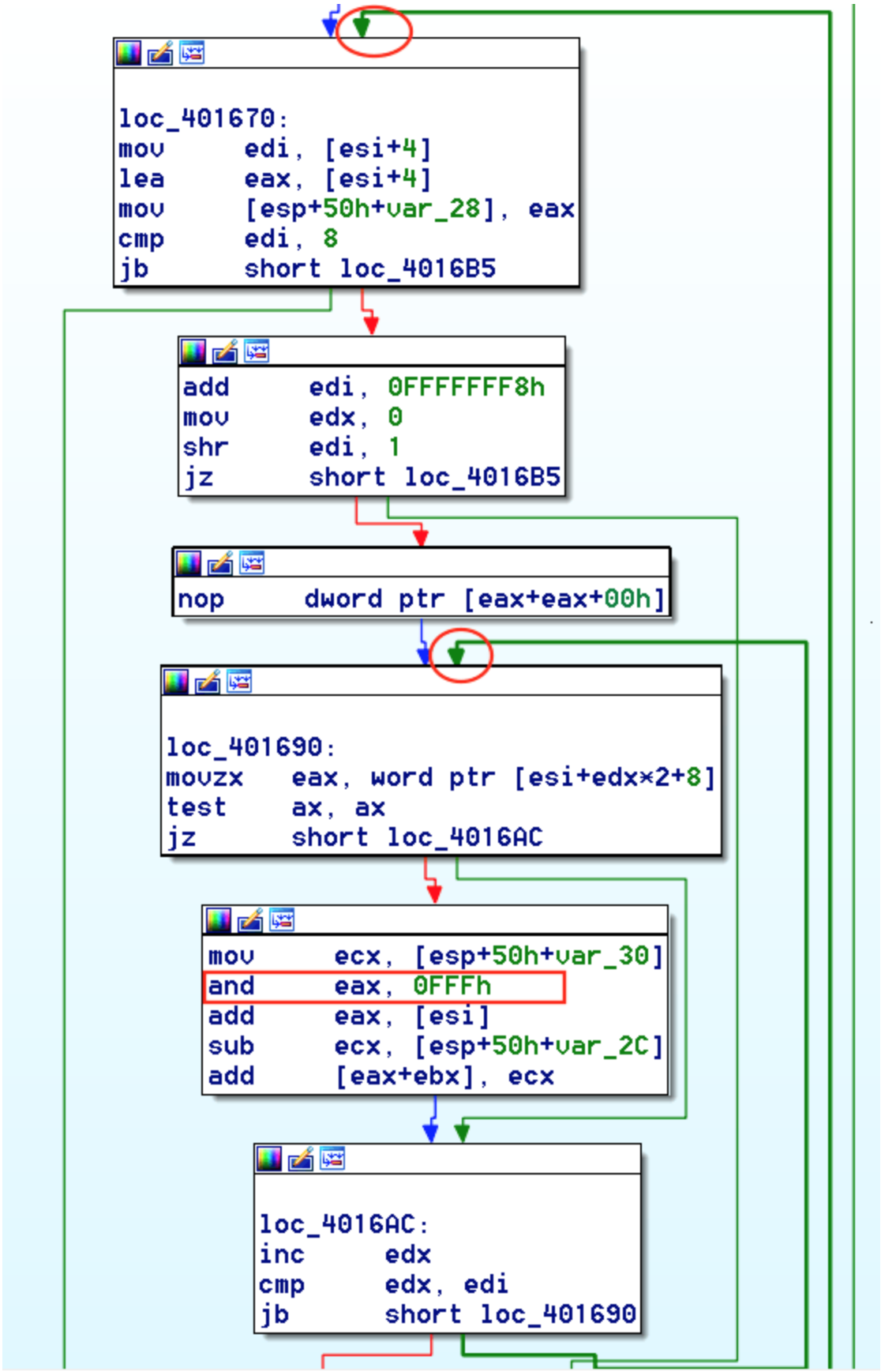
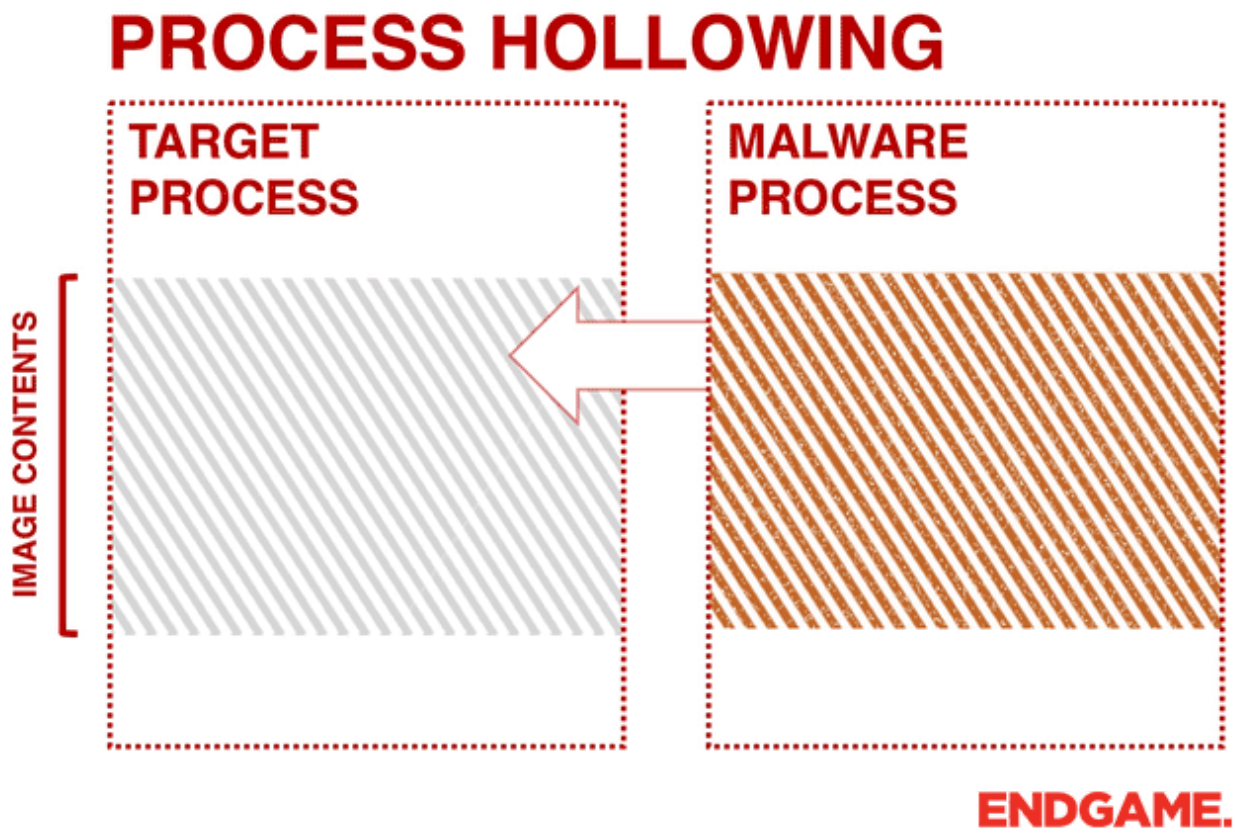


Figure 2: Example structure of the loops for PE injection prior to calls to CreateRemoteThread

Sha256: ce8d7590182db2e51372a4a04d6a0927a65b2640739f9ec01cfd6c143b1110da

3. PROCESS HOLLOWING (A.K.A PROCESS REPLACEMENT AND RUNPE)

Instead of injecting code into a host program (e.g., DLL injection), malware can perform a technique known as process hollowing. Process hollowing occurs when a malware unmaps (hollows out) the legitimate code from memory of the target process, and overwrites the memory space of the target process (e.g., svchost.exe) with a malicious executable.



The malware first creates a new process to host the malicious code in suspended mode. As shown in Figure 3, this is done by calling CreateProcess and setting the Process Creation Flag to CREATE_SUSPENDED (0x00000004). The primary thread of the new process is created in a suspended state, and does not run until the ResumeThread function is called. Next, the malware needs to swap out the contents of the legitimate file with its malicious payload. This is done by unmapping the memory of the target process by calling either ZwUnmapViewOfSection or NtUnmapViewOfSection. These two APIs basically release all memory pointed to by a section. Now that the memory is unmapped, the loader performs VirtualAllocEx to allocate new memory for the malware, and uses WriteProcessMemory to write each of the malware's sections to the target process space. The malware calls

SetThreadContext to point the entrypoint to a new code section that it has written. At the end, the malware resumes the suspended thread by calling ResumeThread to take the process out of suspended state.

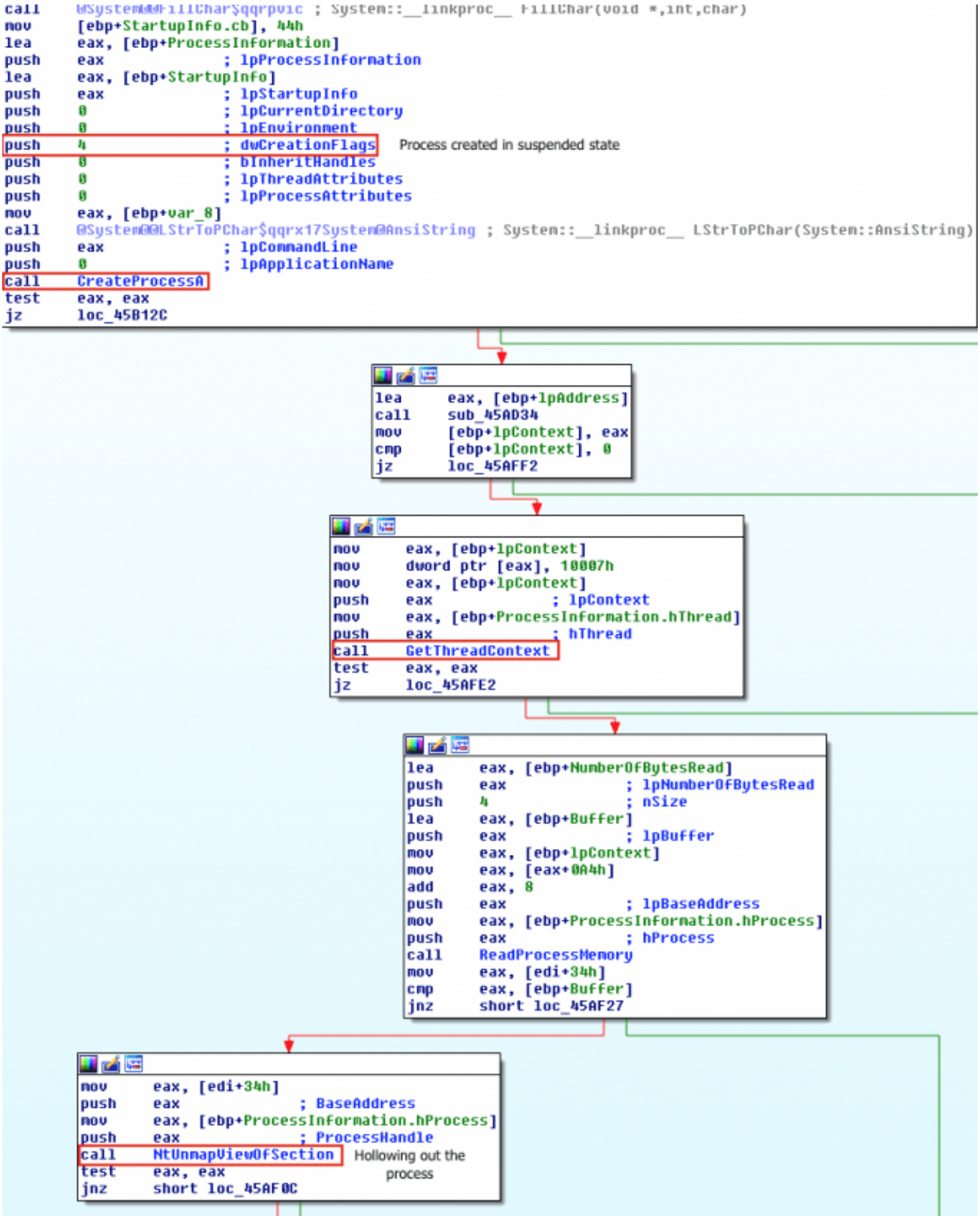
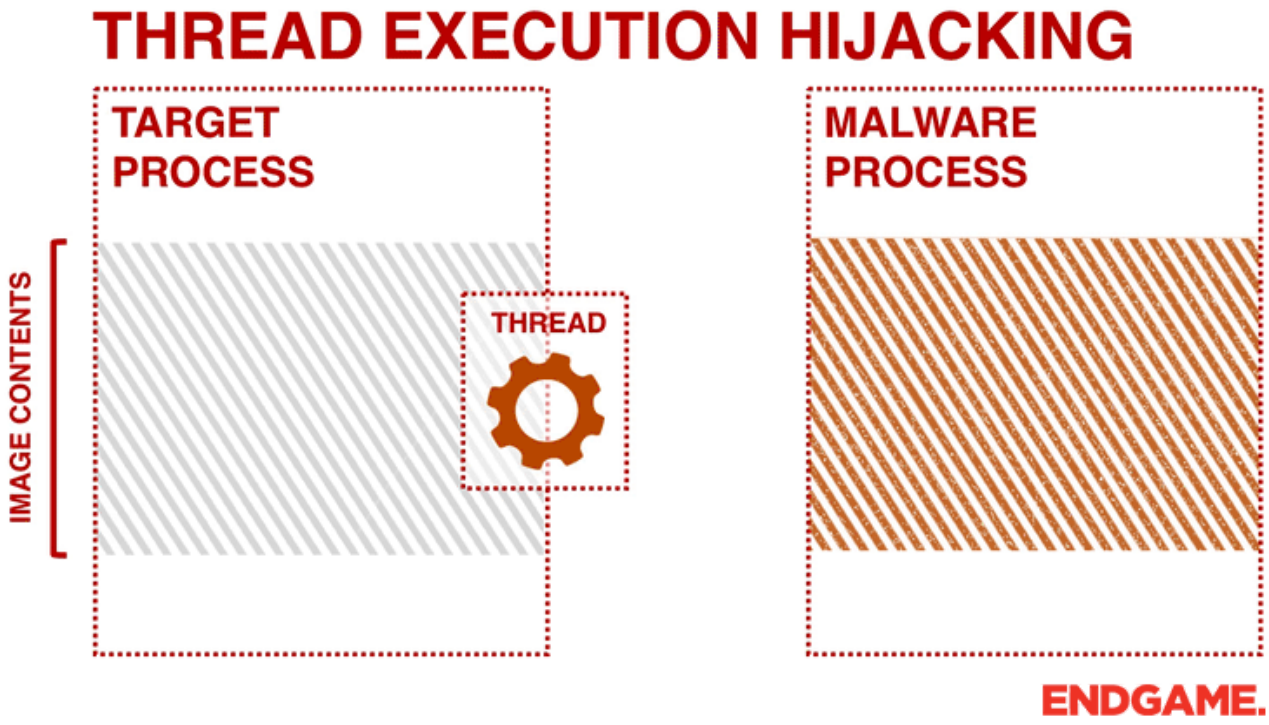


Figure 3: Ransom.Cryak performing process hollowing
Sha256: eae72d803bf67df22526f50fc7ab84d838efb2865c27aef1a61592b1c520d144

4. THREAD EXECUTION HIJACKING (A.K.A SUSPEND, INJECT, AND RESUME (SIR))

This technique has some similarities to the process hollowing technique previously discussed. In thread execution hijacking, malware targets an existing thread of a process and avoids any noisy process or thread creations operations. Therefore, during analysis you will probably see calls to CreateToolhelp32Snapshot and Thread32First followed by OpenThread.



After getting a handle to the target thread, the malware puts the thread into suspended mode by calling SuspendThread to perform its injection. The malware calls VirtualAllocEx and WriteProcessMemory to allocate memory and perform the code injection. The code can contain shellcode, the path to the malicious DLL, and the address of LoadLibrary.

Figure 4 illustrates a generic trojan using this technique. In order to hijack the execution of the thread, the malware modifies the EIP register (a register that contains the address of the next instruction) of the targeted thread by calling SetThreadContext. Afterwards, malware resumes the thread to execute the shellcode that it has written to the host process. From the attacker's perspective, the SIR approach can be problematic because suspending and resuming a thread in the middle of a system call can cause the system to crash. To avoid this, a more sophisticated malware would resume and retry later if the EIP register is within the range of NTDLL.dll.

Figure 4: A generic trojan is performing thread execution hijacking
Sha256: 787cbc8a6d1bc58ea169e51e1ad029a637f22560660cc129ab8a099a745bd50e

5. HOOK INJECTION VIA SETWINDOWSHOOKEX

Hooking is a technique used to intercept function calls. Malware can leverage hooking functionality to have their malicious DLL loaded upon an event getting triggered in a specific thread. This is usually done by calling SetWindowsHookEx to install a hook routine into the hook chain. The SetWindowsHookEx function takes four arguments. The first argument is the type of event. The events reflect the range of hook types, and vary from pressing keys on the keyboard (WH_KEYBOARD) to inputs to the mouse (WH_MOUSE), CBT, etc. The second argument is a pointer to the function the malware wants to invoke upon the event execution.The third argument is a module that contains the function. Thus, it is very common to see calls to LoadLibrary and GetProcAddress before calling SetWindowsHookEx. The last argument to this function is the thread with which the hook procedure is to be associated. If this value is set to zero all threads perform the action when the event is triggered. However, malware usually targets one thread for less noise, thus it is also possible to see calls CreateToolhelp32Snapshot and Thread32Next before SetWindowsHookEx to find and target a single thread. Once the DLL is injected, the malware executes its malicious code on behalf of the process that its threadId was passed to SetWindowsHookEx function. In Figure 5, Locky Ransomware implements this technique.

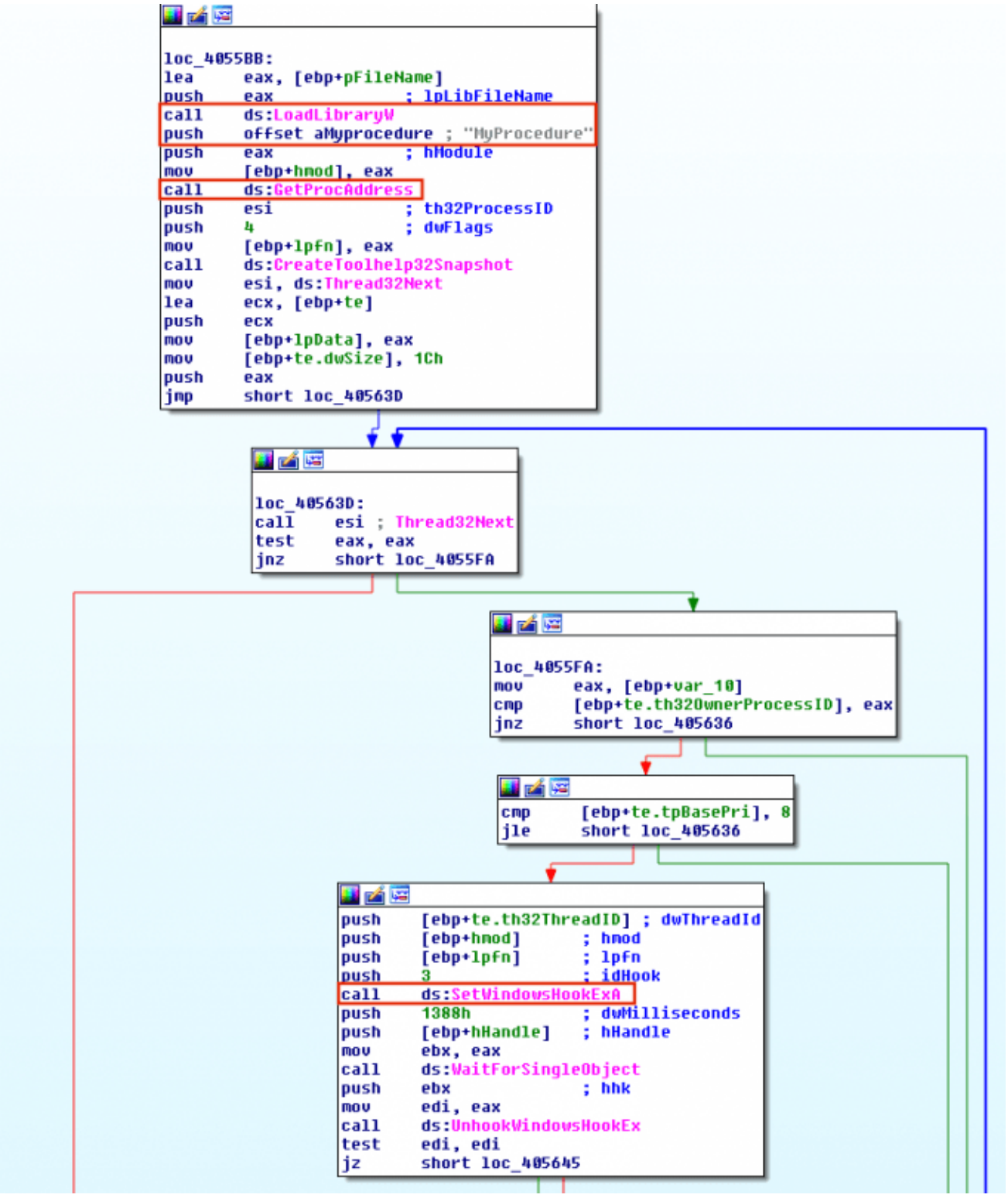


Figure 5: Locky Ransomware using hook injection
Sha256: 5d6ddb8458ee5ab99f3e7d9a21490ff4e5bc9808e18b9e20b6dc2c5b27927ba1

6. INJECTION AND PERSISTENCE VIA REGISTRY MODIFICATION (E.G. APPINIT_DLLS, APPCERTDLLS, IFEO)

Appinit_DLL, AppCertDlls, and IFEO (Image File Execution Options) are all registry keys that malware uses for both injection and persistence. The entries are located at the following locations:

HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\Appinit_Dlls HKLM\System\CurrentControlSet\Control\Session Manager\AppCertDlls HKLM\Software\Microsoft\Windows NT\currentversion\image file execution options

AppInit_DLLs

Malware can insert the location of their malicious library under the Appinit_Dlls registry key to have another process load their library. Every library under this registry key is loaded into every process that loads User32.dll. User32.dll is a very common library used for storing graphical elements such as dialog boxes. Thus, when a malware modifies this subkey, the majority of processes will load the malicious library. Figure 6 demonstrates the trojan Ginwui relying on this approach for injection and persistence. It simply opens the Appinit_Dlls registry key by calling RegCreateKeyEx, and modifies its values by calling RegSetValueEx.

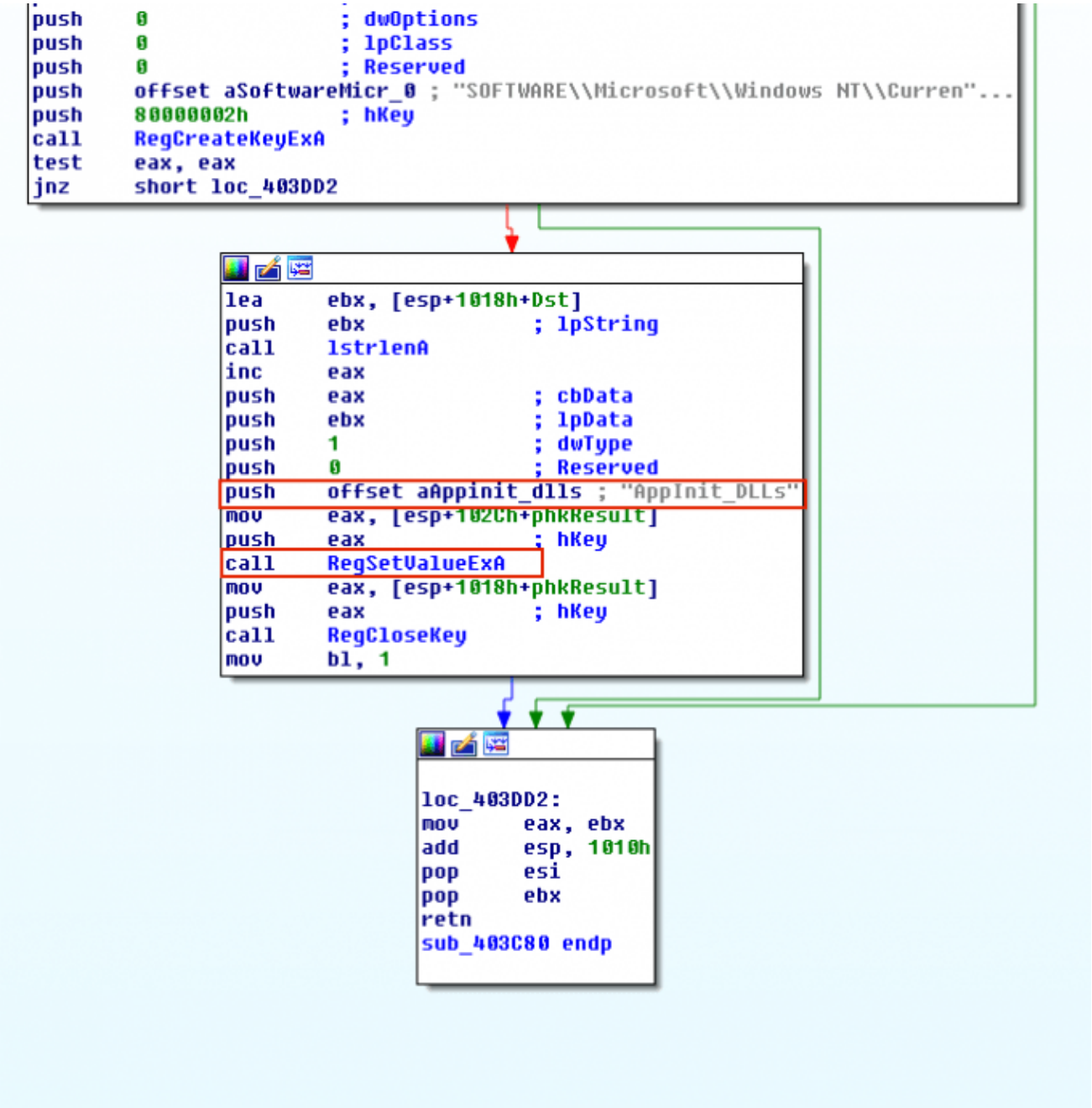


Figure 6: Ginwui modifying the AppIniti_DLLs registry key
Sha256: 9f10ec2786a10971eddc919a5e87a927c652e1655ddbbae72d376856d30fa27c

AppCertDlls

This approach is very similar to the AppInit_DLLs approach, except that DLLs under this registry key are loaded into every process that calls the Win32 API functions CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, CreateProcessWithTokenW, and WinExec.

Image File Execution Options (IFEO)

IFEO is typically used for debugging purposes. Developers can set the “Debugger Value” under this registry key to attach a program to another executable for debugging. Therefore, whenever the executable is launched the program that is attached to it will be launched. To use this feature you can simply give the path to the debugger, and attach it to the executable that you want to analyze. Malware can modify this registry key to inject itself into the target executable. In Figure 7, Diztakun trojan implements this technique by modifying the debugger value of Task Manager.

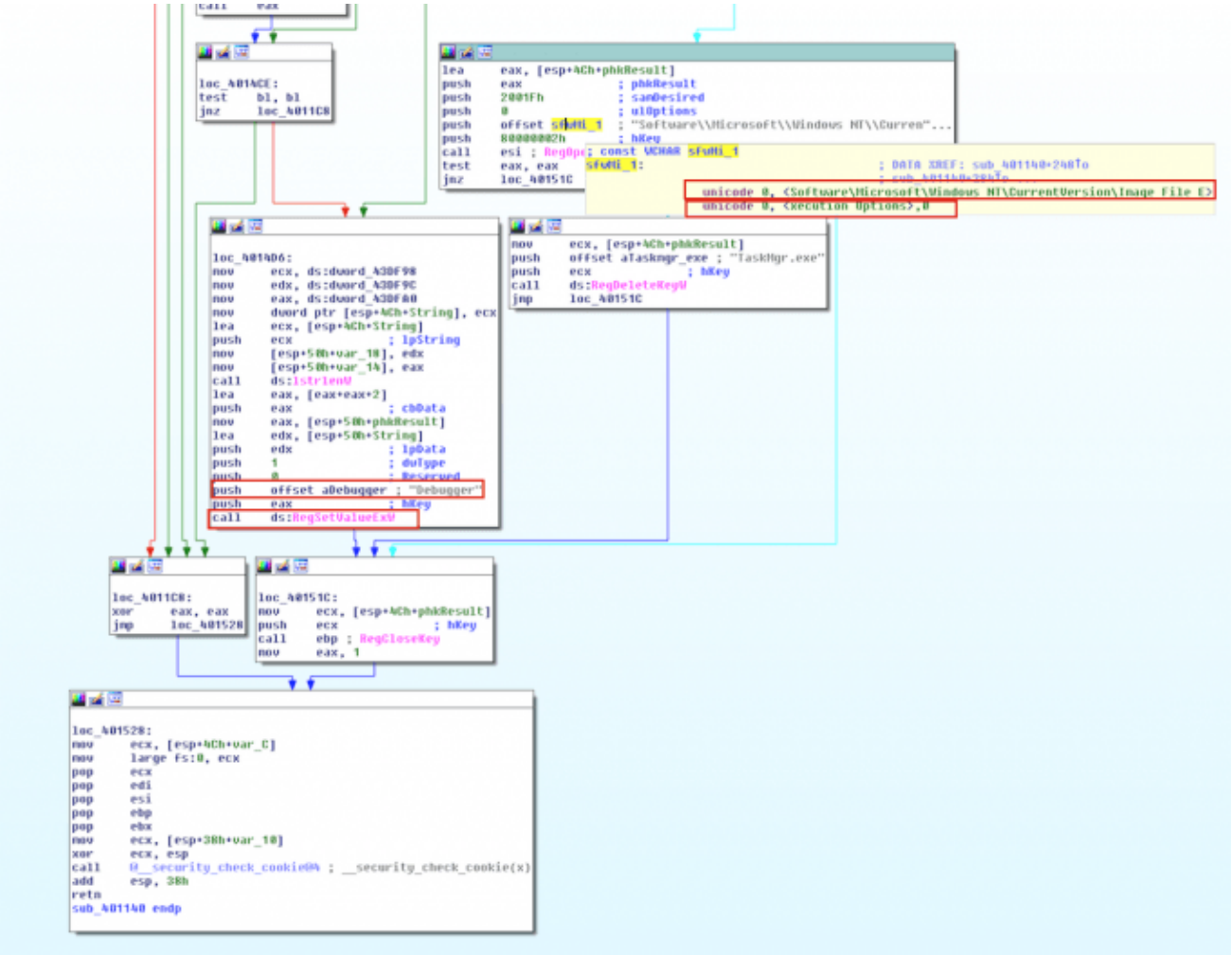


Figure 7: Diztakun trojan modifying IFEO registry key
Sha256: f0089056fc6a314713077273c5910f878813fa750f801dfca4ae7e9d7578a148

7. APC INJECTION AND ATOMBOMBING

Malware can take advantage of Asynchronous Procedure Calls (APC) to force another thread to execute their custom code by attaching it to the APC Queue of the target thread. Each thread has a queue of APCs which are waiting for execution upon the target thread entering alterable state. A thread enters an alertable state if it calls SleepEx, SignalObjectAndWait, MsgWaitForMultipleObjectsEx, WaitForMultipleObjectsEx, or WaitForSingleObjectEx functions. The malware usually looks for any thread that is in an

alterable state, and then calls OpenThread and QueueUserAPC to queue an APC to a thread. QueueUserAPC takes three arguments: 1) a handle to the target thread; 2) a pointer to the function that the malware wants to run; 3) and the parameter that is passed to the function pointer. In Figure 8, Amanahe malware first calls OpenThread to acquire a handle of another thread, and then calls QueueUserAPC with LoadLibraryA as the function pointer to inject its malicious DLL into another thread.

AtomBombing is a technique that was first introduced by enSilo research, and then used in Dridex V4. As we discussed in detail in a previous post, the technique also relies on APC injection. However, it uses atom tables for writing into memory of another process.

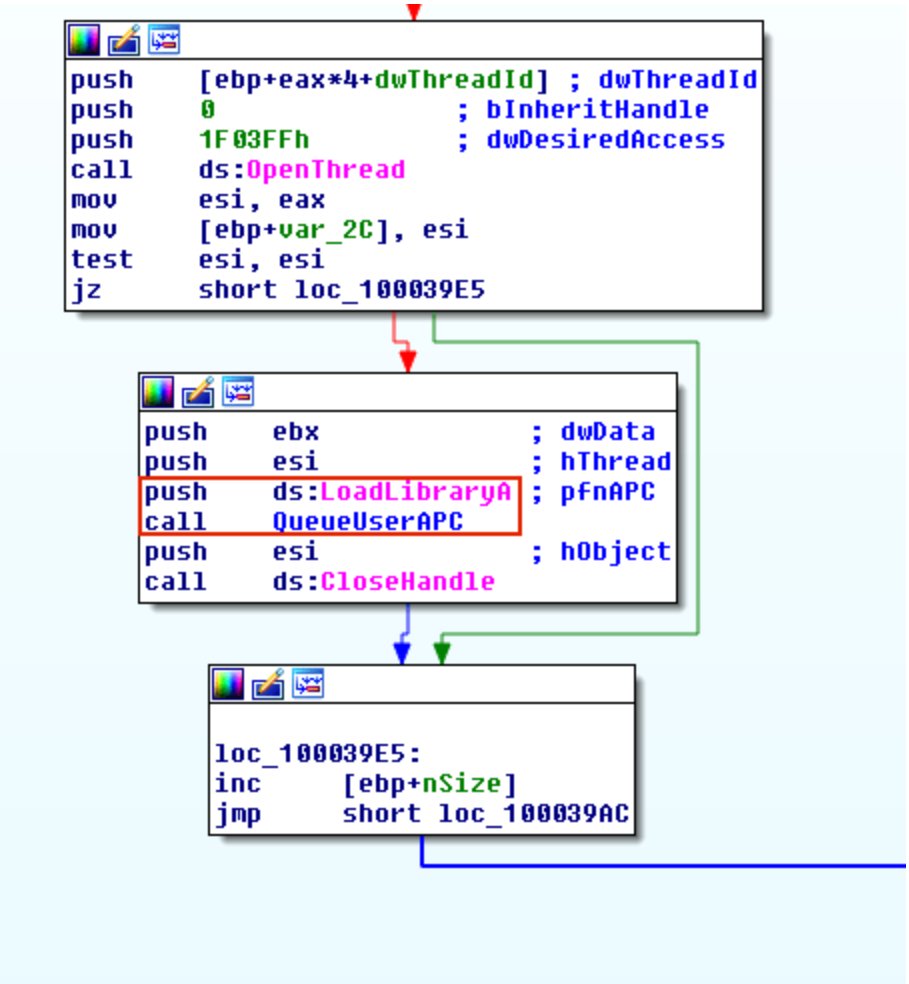


Figure 8: Almanahe performing APC injection
Sha256: f74399cc0be275376dad23151e3d0c2e2a1c966e6db6a695a05ec1a30551c0ad

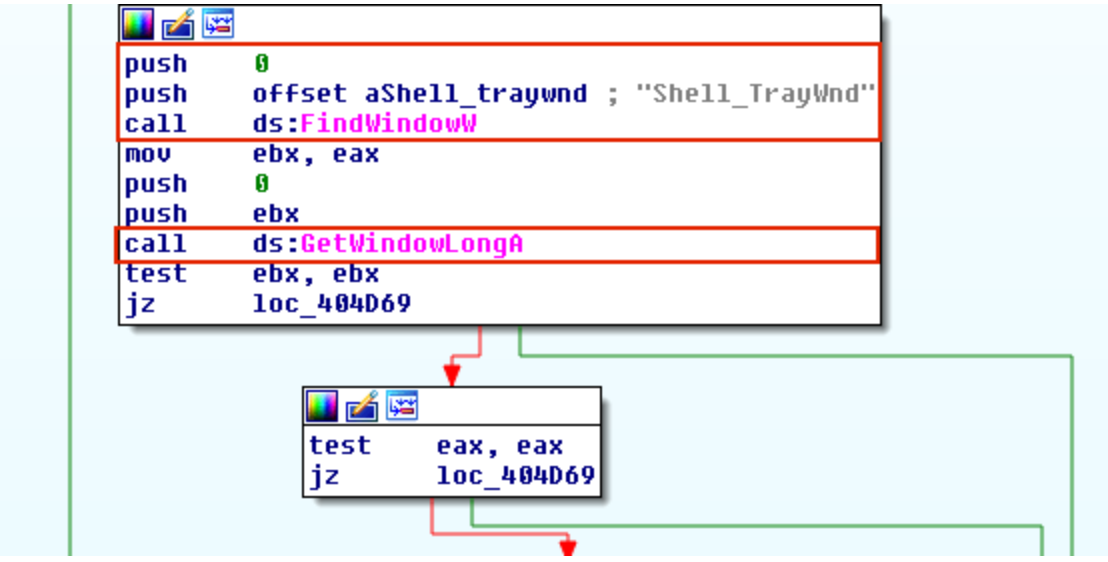
8. EXTRA WINDOW MEMORY INJECTION (EWMI) VIA SETWINDOWLONG

EWMI relies on injecting into Explorer tray window's extra window memory, and has been used a few times among malware families such as Gapz and PowerLoader. When registering a window class, an application can specify a number of additional bytes of

memory, called extra window memory (EWM). However, there is not much room in EWM. To circumvent this limitation, the malware writes code into a shared section of explorer.exe, and uses SetWindowLong and SendNotifyMessage to have a function pointer to point to the shellcode, and then execute it.

The malware has two options when it comes to writing into a shared section. It can either create a shared section and have it mapped both to itself and to another process (e.g., explorer.exe), or it can simply open a shared section that already exists. The former has the overhead of allocating heap space and calling NTMapViewOfSection in addition to a few other API calls, so the latter approach is used more often. After malware writes its shellcode in a shared section, it uses GetWindowLong and SetWindowLong to access and modify the extra window memory of “Shell_TrayWnd”. GetWindowLong is an API used to retrieve the 32-bit value at the specified offset into the extra window memory of a window class object, and SetWindowLong is used to change values at the specified offset. By doing this, the malware can simply change the offset of a function pointer in the window class, and point it to the shellcode written to the shared section.

Like most other techniques mentioned above, the malware needs to trigger the code that it has written. In previously discussed techniques, malware achieved this by calling APIs such as CreateRemoteThread, QueueUserAPC, or SetThreadContext. With this approach, the malware instead triggers the injected code by calling SendNotifyMessage. Upon execution of SendNotifyMessage, Shell_TrayWnd receives and transfers control to the address pointed to by the value previously set by SetWindowLong. In Figure 9, a malware named PowerLoader uses this technique.



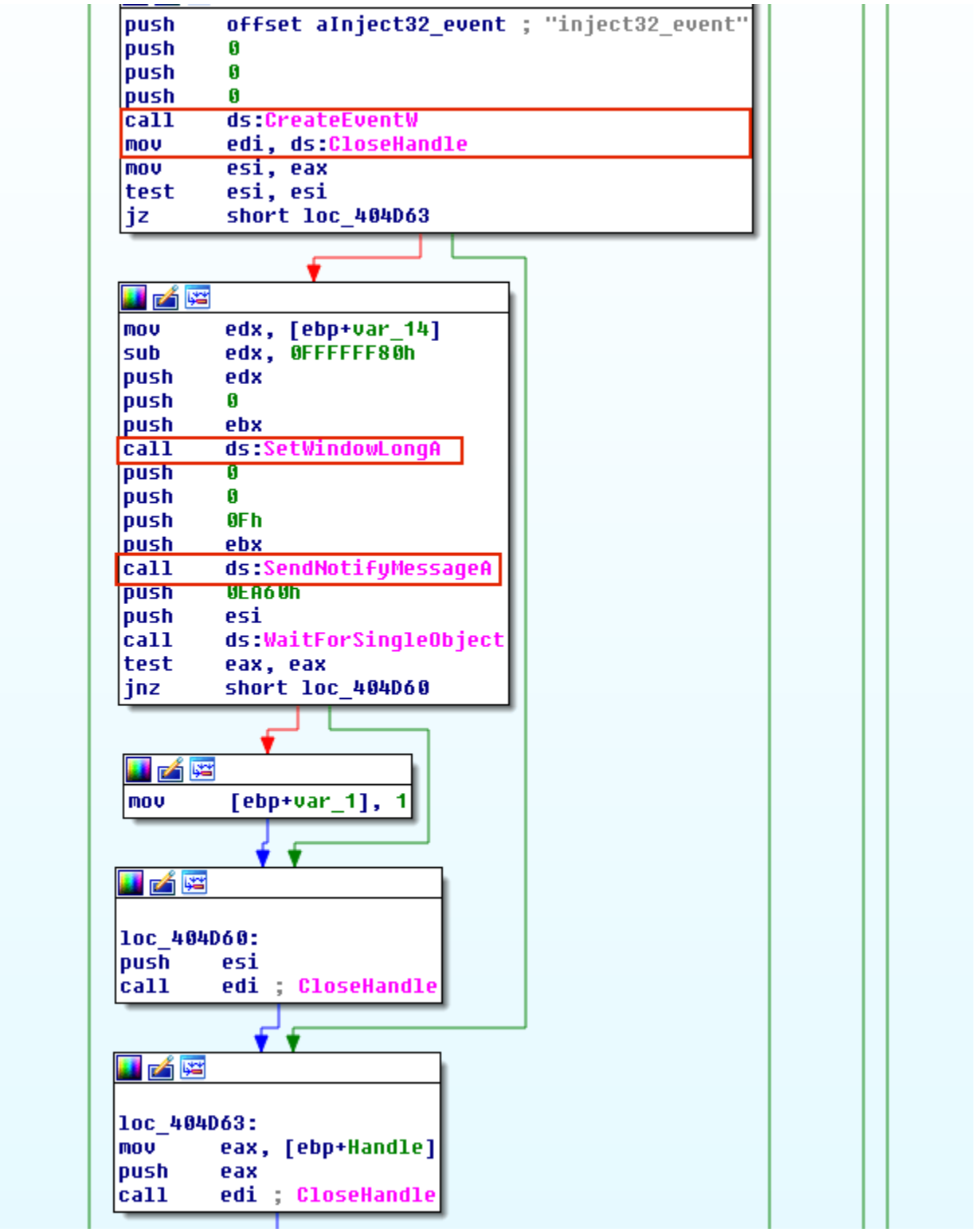


Figure 9: PowerLoader injecting into extra window memory of shell tray window
Sha256: 5e56a3c4d4c304ee6278df0b32afb62bd0dd01e2a9894ad007f4cc5f873ab5cf

9. INJECTION USING SHIMS

Microsoft provides Shims to developers mainly for backward compatibility. Shims allow developers to apply fixes to their programs without the need of rewriting code. By leveraging shims, developers can tell the operating system how to handle their application. Shims are essentially a way of hooking into APIs and targeting specific executables. Malware can take advantage of shims to target an executable for both persistence and injection. Windows runs the Shim Engine when it loads a binary to check for shimming databases in order to apply the appropriate fixes.

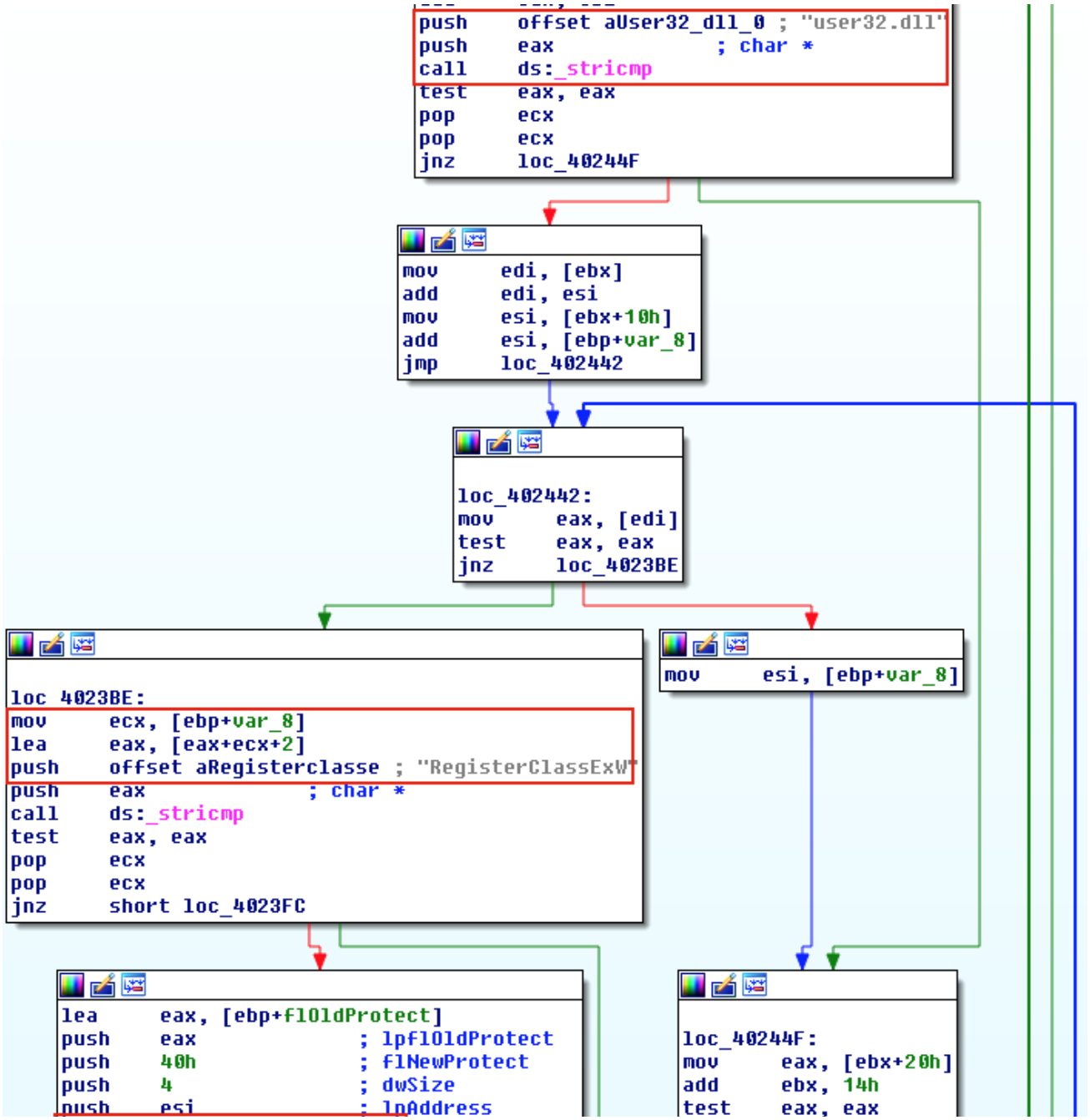
There are many fixes that can be applied, but malware's favorites are the ones that are somewhat security related (e.g., DisableNX, DisableSEH, InjectDLL, etc). To install a shimming database, malware can deploy various approaches. For example, one common approach is to simply execute sdbinst.exe, and point it to the malicious sdb file. In Figure 10, an adware, "Search Protect by Conduit", uses a shim for persistence and injection. It performs an "InjectDLL" shim into Google Chrome to load vc32loader.dll. There are a few existing tools for analyzing sdb files, but for the analysis of the sdb listed below, I used python-sdb.

```
    <NAME type='stringref'>0x1ac</NAME>
    <APP_NAME type='stringref'>0x1e6</APP_NAME>
    <VENDOR type='stringref'>0x106</VENDOR>
    <EXE_ID type='hex'>ce8affb6-1e8f-41f3-a1a3-cafd2e996ab8</EXE_ID>
    <MATCHING_FILE>
      <NAME type='stringref'>0x120</NAME>
    </MATCHING_FILE>
    <LAYER>
      <NAME type='stringref'>0x30</NAME>
      <LAYER_TAGID type='integer'>0x19a</LAYER_TAGID>
    </LAYER>
  </EXE>
  <EXE>
    <NAME type='stringref'>0x1f2</NAME>
    <APP_NAME type='stringref'>0x22e</APP_NAME>
    <VENDOR type='stringref'>0x106</VENDOR>
    <EXE_ID type='hex'>2b4c4b81-d5b5-4cb2-9436-ef2799a4630c</EXE_ID>
    <MATCHING_FILE>
      <NAME type='stringref'>0x120</NAME>
    </MATCHING_FILE>
    <LAYER>
      <NAME type='stringref'>0x30</NAME>
      <LAYER_TAGID type='integer'>0x19a</LAYER_TAGID>
    </LAYER>
  </EXE>
</DATABASE>
<STRINGTABLE>
  <STRINGTABLE_ITEM type='string'>2.1.0.3</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>Apps32</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>VC32Ldr</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>InjectDll</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>\\.\globalroot\systemroot\apppatch\nbin\vc32loader.dll</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>chrome.exe</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>ch</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>&lt;Unknown&gt;</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>*</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>explorer.xxx</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>ex</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>firefox.exe</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>ff</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>iexplore.exe</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>ie</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>software_removal_tool.exe</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>sr</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>software_reporter_tool.exe</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM type='string'>sr2</STRINGTABLE_ITEM>
</STRINGTABLE>
```

Figure 10: SDB used by Search Protect for injection purposes
Sha256: 6d5048baf2c3bba85adc9ac5ffd96b21c9a27d76003c4aa657157978d7437a20

10. IAT HOOKING AND INLINE HOOKING (A.K.A USERLAND ROOTKITS)

IAT hooking and inline hooking are generally known as userland rootkits. IAT hooking is a technique that malware uses to change the import address table. When a legitimate application calls an API located in a DLL, the replaced function is executed instead of the original one. In contrast, with inline hooking, malware modifies the API function itself. In Figure 11, the malware FinFisher, performs IAT hooking by modifying where the CreateWindowEx points.



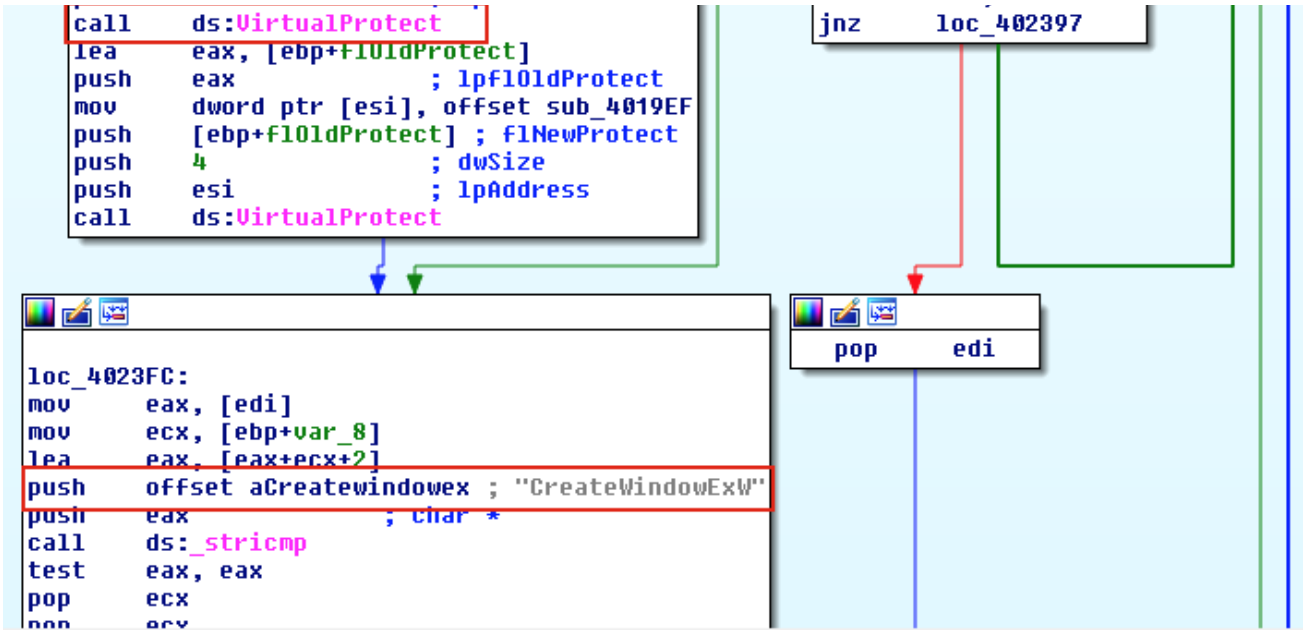


Figure 11: FinFisher performing IAT hooking by changing where CreateWindowEx points to
Sha256: f827c92fbe832db3f09f47fe0dcaafd89b40c7064ab90833a1f418f2d1e75e8e

CONCLUSION

In this post, I covered ten different techniques that malware uses to hide its activity in another process. In general, malware either directly injects its shellcode into another process or it forces another process to load its malicious library. In Table 1, I have classified the various techniques and provided samples to serve as a reference for observing each injection technique covered in this post. The figures included throughout the post will help the researcher recognize the various techniques when reversing malware.

	Shellcode Injection	Forcing A DLL To Be Loaded	Sha256
1. DLL Injection		X	07b8f25e7b536f5b6f686c12d04edc37e11347c8acd5c53f98a174723078c365
2. PE Injection	X		ce8d7590182db2e51372a4a04d6a0927a65b2640739f9ec01cfd6c143b1110da
3. Process Hollowing	X		eae72d803bf67df22526f50fc7ab84d838efb2865c27aef1a61592b1c520d144
4. Thread Execution Hijacking	X		787cbc8a6d1bc58ea169e51e1ad029a637f22560660cc129ab8a099a745bd50e
5. Hook Injection		X	5d6ddb8458ee5ab99f3e7d9a21490ff4e5bc9808e18b9e20b6dc2c5b27927ba1
6. Registry Modification		X	9f10ec2786a10971eddc919a5e87a927c652e1655ddbbae72d376856d30fa27c
7. APC Injection		X	f74399cc0be275376dad23151e3d0c2e2a1c966e6db6a695a05ec1a30551c0ad
8. Shell Tray Window Injection	X		5e56a3c4d4c304ee6278df0b32afb62bd0dd01e2a9894ad007f4cc5f873ab5cf
9. Shim Injection		X	6d5048baf2c3bba85adc9ac5ffd96b21c9a27d76003c4aa657157978d7437a20
10. IAT and Inline Hooking	X	X	f827c92fbe832db3f09f47fe0dcaafd89b40c7064ab90833a1f418f2d1e75e8e

Table1: Process injection can be done by directly injecting code into another process, or by forcing a DLL to be loaded into another process

Attackers and researchers regularly discover new techniques to achieve injection and provide stealth. This post detailed ten common and emerging techniques, but there are others, such as COM hijacking. Defenders will never be “done” in their mission to detect and prevent stealthy process injection because adversaries will never stop innovating.

At Endgame, we constantly research advanced stealth techniques and bring protections into our product. We layer capabilities which detect malicious DLLs that load on some persistence (like AppInit DLLs, COM Hijacks, and more), prevent many forms of code injection in real-time via our patented shellcode injection protection, and detect malicious injected payloads running in memory delivered through any of the above techniques through our patent-pending fileless attack detection techniques. This approach allows our platform to be more effective than any other product on the market in protecting against code injection, while also maximizing resiliency against bypass due to emerging code injection techniques.

We're hiring

Work for a global, distributed team where finding someone like you is just a Zoom meeting away. Flexible work with impact? Development opportunities from the start?