The state of advanced code injections

Archived: 2026-04-05 16:59:06 UTC

In the last few years there has been a significant interest in code injection techniques from both attackers and defenders. These techniques enable the attacker to execute arbitrary code within the address space of some target process (which is why code injections often are also called process injections), and attackers, both malware and pentesters, increasingly use these techniques to bypass anti-malware systems and endpoint protection systems in order to execute their payloads. Many of these injection techniques are already described in various blog posts, such as the excellent ones by Endgame **here** and **here**, and most recently a large survey was conducted by researchers from SafeBreach at the most recent Blackhat event with the content available **here**. However, many of these surveys are closely attached to the core programmatic aspects of the injections, whereas they leave out elements of why injections are necessarily important and when they are used. In this blog post we will cover the state of code injections from a more general setting such as their motivation, some of their technical details as well as highlight examples of attacks that have used them. Finally, we give a short view into the future.

Understanding code injections from beginner to advanced is one of the courses that we offer as part of **our software security training**. We currently have a public event scheduled for the upcoming 44CON in London 9th-11th September 2019 and you can find the necessary information **here**. In this course we will teach you the core of these techniques as well as how to develop sophisticated payloads that rely on code injections. Please consider attending the training and the conference!

Motivation for code injections, for defenders and attackers

In a general sense, attackers use code injection techniques to mitigate defensive systems. This includes everything from bypassing host-based intrusion prevention systems, evading malware sandboxes and avoiding analysis by forensic tools. Naturally, malware has used these techniques for quite a while and even in back in 2013 Palo Alto reported that 13.5% of malware samples used code injections, described in **the modern malware review report** on page 16 under "analysis avoidance ". In their report they also give an interesting insight about the motivation for code injections, namely "*Code injection was observed in 13.5 percent of samples. This technique is notable in particular because it allows malware to hide within another running process. This has the effect of the malware out of view if a user checks the task manager and can also foil some attempts at application white-listing on the host*". Wayne Low documents even before then, in 2012, the first analysis of the Gapz malware that explicility used a novel code injection technique that - due to it's novelty - bypassed host-based intrusion prevention systems **here**. Interestingly, the injection that Gapz deployed used techniques and ideas described around a decade earlier called **shatter** attacks, which was even presented at Blackhat in 2004 **here**.

In the years 2013-2019 the amount of code injections has continued to grow. There are currently many reports by anti-malware companies documenting the code injections in malware and I think it's fair to say that now, when a new malware is discovered, it's more often than not the case that it uses code injection techniques. A recent survey of techniques by SafeBreach documents 14 techniques invented in 2017-2019, and this leaves out 7 shatter-like attacks that they do not go in details with as well as a whole domain of techniques in the process hollowing space,

in which several new variants have been discovered in recent years. Furthermore, malware samples are currently not limiting their injection lifecycle to only one injection technique, but a recent report shows that **Dridex combines five different injection techniques**.

Code injections are not only used by malware. Pentesters, and red teams in general, rely on these techniques to take control of the systems once they have gotten access to the system. For example, the famous **Meterpreter** used by pentesters rely on code injection as well as related techniques, e.g. self-loading DLLs. **Shellterpro** is another well-known pentester tool that is heavily based on code injections. As defensive systems get better, understanding the design space of code injections can significantly enhance the skills of red teamers, as it allows you to manually construct payloads and write injection tools that bypass the specific defensive perimeter of your target. An example of custom tools developed for the purposes of penetration testing was given at Blackhat in 2014 **here**.

Brief overviev of code injection foundations

In this section we give a brief introduction to some of the more common-known injection techniques.

Traditional remote thread creation

This is the most well known injection technique and simply achieves execution in the target process by instantiating a remote thread. The general procedure is to get access to the target process using `OpenProcess`, allocating memory in the process using `VirtualAlloc`, writing malicious code to the allocated memory with `WriteProcessMemory` and finally having this code execute using `CreateRemoteThread`. Naturally, there are many variations of this injection technique, both in terms of getting access to the remote process, writing memory to the targets address space and also initiating execution. For example, instead of opening an existing process, the malware can create a new process with `CreateProcess` and inject its code in this new process or rely on lower-level APIs like `NtOpenProcess`. The attack can also write to memory using `NtWriteVirtualMemory` and creating the remote thread can be performed with a variety of lower-level APIs like `RtlCreateUserThread`, `NtCreateThreadEx` and `ZwCreateThreadEx`. This technique is perhaps the most commonly used by malware and example reports include **Tinba** and **Emotet**.

In general, we can construct a similar-looking attack in many ways. The primitive for writing memory to the target process need not be `WriteProcessMemory`, but can be any way of memory sharing. This includes memory mapped files by way of APIs like `ZwCreateSection` and `ZwMapViewOfSection` and also globally shared memory. Furthermore, from a programmatic perpective we can also use asynchronous procedure calls rather than explicitly starting a new thread in the remote process, which we discuss below.

```
18    if ((int)entrypointOffset != 0) {
19      lpBaseAddress = VirtualAllocEx(procHandle,(LPVOID)0x0,(ulonglong)allocSize,0x3000,0x40);
20      pvVar2 = pvVar3;
21      if (lpBaseAddress != (LPVOID)0x0) {
22        BVar1 = WriteProcessMemory(procHandle,lpBaseAddress,srcBuffer,(ulonglong)allocSize,
23                                   (SIZE_T *)0x0);
24        pvVar2 = pvVar3;
25        if (BVar1 != 0) {
26          pvVar2 = CreateRemoteThread(procHandle,(LPSECURITY_ATTRIBUTES)0x0,0x100000,
27                                      (LPTHREAD_START_ROUTINE)
28                                      ((longlong)lpBaseAddress + (entrypointOffset & 0xffffffff)),
29                                      (LPVOID)0x0,0,(LPDWORD)&local_res20);
30        }
31      }
32    }
```

Remote thread creation as decompiled by Ghidra.

Remote thread hijacking

A technique that is closely aligned with creating a remote thread is to hijack a remote thread instead of creating a new one in the target process. From a high-level point of view, the difference between this technique and the previous one is that the previous technique creates a new thread in the target whereas this technique hijacks execution of an exisiting one. One way to do this is to create a new process, by way of `CreateProcess`, in suspended mode and overwrite the entry point of the newly-started process such that it points to our attacker-controlled code instead. This effectively means the `CreateRemoteThread` call from before gets substituted with `ResumeThread`. A more aggressive approach is to simply suspend thread execution in the target process and then substitute the thread using `SuspendThread`. Code execution is then achieved by exchanging the thread context using `GetThreadContext` and `SetThreadContext` such that the registers of the thread context points to attacker-controlled memory. Naturally, since the thread execution of the target process is suspended it is in many scenarios desirable to restore faithful execution in the target process in order for the system to continue execution unnoticed.

The attacker can also initiate execution of the attacker-controlled memory in the remote process through asynchronous procedure calls such as `QueueUserAPC`, `NtQueueApcThread`, `ZwQueueApcThread` and `RtlQueueApcWow64Thread`. However, one of the drawbacks of doing this, however, is that the remote thread must be in an alertable state to trigger the APC.

Reflective DLL injection

In the previous methods we were mainly concerned with how to achieve code execution in the remote process. However, a question that comes up once you achieve code execution is *what* code to execute. In most cases just executing shellcode doesn't do the job as the attacker desires to have more comprehensive control. Reflective DLL injection is a technique that focuses on this aspect of the code-injection design space using a self-loadable DLL file. Specifically, reflective DLL injection is a technique that creates a DLL such that the DLL has a minimal Windows loader as an exported function. When this function is triggered the DLL will load itself inside the process of which it has been written, thus avoiding the need to be loaded from disk by the regular Windows loader. As such, it is not necessary to, for example, rely on calls like `LoadLibrary` to load a fully-fledged library inside the target process. Rather, the attacker can simply allocate space in the remote process, write the raw DLL content there, and then execute the exported function in the DLL itself. **Lazarus** is an example of malware that

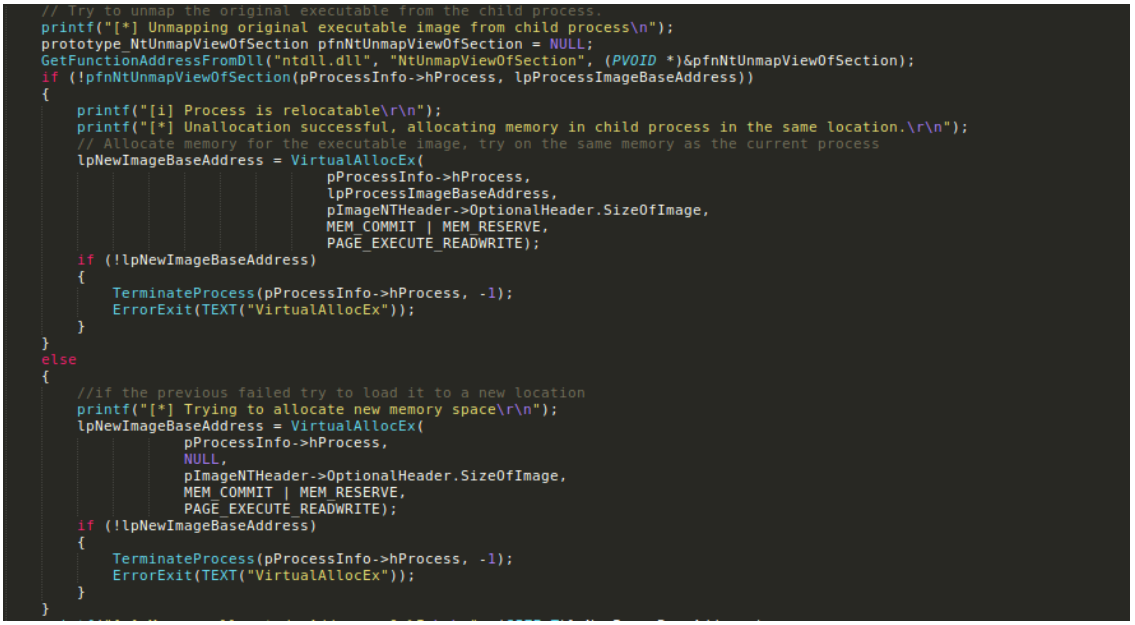
uses reflective DLL injection and pentesters published at Blackhat a white paper on a novel packer that uses reflective DLL injection **here**.

```
// itterate through all sections, loading them into memory.
uiValueE = ((PIMAGE_NT_HEADERS)uiHeaderValue)->FileHeader.NumberOfSections;
while( uiValueE-- )
{
        // uiValueB is the VA for this section
        uiValueB = ( uiBaseAddress + ((PIMAGE_SECTION_HEADER)uiValueA)->VirtualAddress );

        // uiValueC if the VA for this sections data
        uiValueC = ( uiLibraryAddress + ((PIMAGE_SECTION_HEADER)uiValueA)->PointerToRawData );

        // copy the section over
        uiValueD = ((PIMAGE_SECTION_HEADER)uiValueA)->SizeOfRawData;

        while( uiValueD-- )
                *(BYTE *)uiValueB++ = *(BYTE *)uiValueC++;

        // get the VA of the next section
        uiValueA += sizeof( IMAGE_SECTION_HEADER );
}
```

Custom loading of PE sections in reflective DLL injection. Code for the reflective loader can be found **here**.

Process hollowing

A technique closely related to hijacking a remote thread is to simply substitute the entire memory of the remote process with attacker-controlled memory. Process hollowing does this. The steps in process hollowing is to create a process in suspended mode, then deallocate the memory of the suspended process (this is where the "hollow" comes from), write an attacker-controlled image to the target process and then resume execution of the target process. Effectively, the goal is to explicitly hide execution of the malicious code in disguise of a benign process. Process hollowing is a commong techniques in malware samples, with an example report found **here**.



```
// Try to unmap the original executable from the child process.
printf("[*] Unmapping original executable image from child process\n");
prototype_NtUnmapViewOfSection pfnNtUnmapViewOfSection = NULL;
GetFunctionAddressFromDll("ntdll.dll", "NtUnmapViewOfSection", (PVOID *)&pfnNtUnmapViewOfSection);
if (!pfnNtUnmapViewOfSection(pProcessInfo->hProcess, lpProcessImageBaseAddress))
{
    printf("[i] Process is relocatable\r\n");
    printf("[*] Unallocation successful, allocating memory in child process in the same location.\r\n");
    // Allocate memory for the executable image, try on the same memory as the current process
    lpNewImageBaseAddress = VirtualAllocEx(
                            pProcessInfo->hProcess,
                            lpProcessImageBaseAddress,
                            pImageNTHeader->OptionalHeader.SizeOfImage,
                            MEM_COMMIT | MEM_RESERVE,
                            PAGE_EXECUTE_READWRITE);
    if (!lpNewImageBaseAddress)
    {
        TerminateProcess(pProcessInfo->hProcess, -1);
        ErrorExit(TEXT("VirtualAllocEx"));
    }
}
else
{
    //if the previous failed try to load it to a new location
    printf("[*] Trying to allocate new memory space\r\n");
    lpNewImageBaseAddress = VirtualAllocEx(
                pProcessInfo->hProcess,
                NULL,
                pImageNTHeader->OptionalHeader.SizeOfImage,
                MEM_COMMIT | MEM_RESERVE,
                PAGE_EXECUTE_READWRITE);
    if (!lpNewImageBaseAddress)
    {
        TerminateProcess(pProcessInfo->hProcess, -1);
        ErrorExit(TEXT("VirtualAllocEx"));
    }
}
```

Unloading of the main module and then reallocating memory for the new executable in ProcessHollowing. Source code can be found **here**.

Example of advanced techniques

The majority of injections observed in the wild are of the types described in the previous section. However, (mostly) in recent years several novel techniques have been discovered that rely on approaches outside the scope of the previous techniques. These novel techniques use different API calls to achieve their code injection, sometimes rely on exploit-like techniques such as return oriented programming and are quite often specific to certain target applications. However, on an abstract level they still remain close to our most basic techniques as they still have to (1) communicate with the target process; (2) ensure attacker-controlled memory is written to the process and (3) trigger execution of attacker-controlled code in the process. It is important to emphasize in this blog post we only highlight some examples of these techniques rather than an exhaustive list. In particular, we have prioritised selection of techniques that have been documented to be used in attacks.

Ghostwriting

The main idea of GhostWriting is to force the target process to write malicious content in it's address space and force this code to be executed without calling any of `OpenProcess`, `VirtualAlloc` or `CreateRemoteThread`, or similar. GhostWriting achieves this by selecting two atomic gadgets (ROP gadgets), one that writes the value of a register to an address given by the value of a different register (`mov [reg1], reg2`) and another gadget that simply represents an eternal loop `jmp 0x0`. The technique then makes use of `SuspendThread`, `GetThreadContext`, `SetThreadContext` and `ResumeThread` to continuously overwrite memory in the target process. Specifically, it continuously sets the registers such that they overwrite a given address with the desired content, and then jumps to the eternal loop. As SetThreadContext allows the attacker to control the registers it is easy to overwrite the stack - or any other address in the target process - with whichever content the attacker desires. After the `mov [reg1], reg2` gadget executes it returns to the eternal loop gadget, and the attacker gains control over the execution simply by calling `SuspendThread`, such that the execution does not run in the eternal loop forever. Rather, the "eternal loop" is used as a temporary safe-state for the attacker to ensure consistency in the target process. A concrete example of using this is to to create a stackframe to `NtProtectVirtualMemory` that sets the necessary permissions for attacker-written shellcode. However, the technique used for achieving write-what-where and execute-on-demand is far more general and can be used to write any type of code to the target, e.g. a fully functioning PE file.

A quite interesting aspect of GhostWriting is that the technique was first made public in 2007, yet it still remains largely one or the more sophisticated techniques. The original blog post explaining the technique is available **here**.

```
UCHAR InjectionCode[]=    {     0x6A,0x00,                    // PUSH 0
                          0xE8,0x0D,0x00,0x00,0x00,     // CALL NEXT
                          // Caption text
                          'G','h','o','s','t','W','r','i','t','i','n','g',0x00,
                          0xE8,0x1D,0x00,0x00,0x00,     // CALL NEXT
                          // Message text
                          'R','u','n','n','i','n','g',' ','i','n','t','o',' ','E','X','P','L','O','R','E','R','.','E','X','E','.','.','.',0x00,
                          0x6A,0x00,                    // PUSH 0
                          0x56,                         // PUSH ESI     ( return address where MessageBoxA should return, we will set ESI so that in points
                                                        // to a EBFE )
                          0x68,0x00,0x00,0x00,0x00,     // PUSH MessageBoxA    ( we will change those 00s to MessageBoxA address in runtime )
                          0xC3    };                    // RET
```

Shellcode written by the original GhostWriting code. The source can be found **here**.

```
// This routine will set thread's context to the values we want and wait till
// thread's EIP reaches a point we indicate. For this proof of concept, we
// will also post some GUI messages to the hijacked thread, so that thread's
// common "wait for messages" nature doesn't slow things down.
//
void WaitForThreadAutoLock(HANDLE Thread, CONTEXT* PThreadContext,HWND ThreadsWindow,DWORD AutoLockTargetEIP)
{
    SetThreadContext(Thread,PThreadContext);

    PostMessage(ThreadsWindow,WM_USER,0,0);
    PostMessage(ThreadsWindow,WM_USER,0,0);
    PostMessage(ThreadsWindow,WM_USER,0,0);

    do
    {
        ResumeThread(Thread);
        Sleep(30);    // This could also be Sleep(0) ( Yield, as NtYieldExecution would do ), but in some cases ( windows server versions ) this would
                      // lead to slowdowns or even starvation of the hijacked thread's execution. I have not done further research into this matter, but
                      // I think this is due to the fact that those server versions of windows prioritize non-GUI thread's execution over GUI thread's
                      // execution by default.

        SuspendThread(Thread);
        GetThreadContext(Thread,PThreadContext);
    }
    while(PThreadContext->Eip!=AutoLockTargetEIP);
}
```

The main function that GhostWriting uses to continuously write desired content in target process. By setting registers in the thread context so they point to selected gadgets GhostWriting maintains control of the target process.

PowerLoader and PowerLoaderEx

The idea behind PowerLoad is to abuse shared sections in Windows and overwrite several function pointers inside `explorer.exe` to point to attacker-controlled memory in the shared section. Since this shared section is non executable PowerLoader relies on a ROP chain to execute shellcode within `explorer.exe`. In more detail, PowerLoader first gets a handle to a window in `explorer.exe`. This window contains a pointer to a CTray class object, which is used for handling messages sent to the particular window. PowerLoader then uses `SetWindowLongPtr` to replace this CTray object, such that it now points to memory in a shared section. Within this shared section, PowerLoader writes its malicious code, which is a combination of a ROP chain as well as shellcode. PowerLoader then triggers the execution of the ROP chain by sending the window a message, using `SendNotifyMessage`. The ROP chain then overwrites a function within ntdll called `atan` with shellcode and transfers execution to this shellcode. This technique was first used by the Gapz malware and has since been generalised by researchers from Ensilo, that made the attack non-specific to the shared sections. You can find the source code for PowerLoaderEx **here**

```
                undefined __stdcall FUN_01001ae6(HWND param_1)
     undefined        AL:1          <RETURN>
     HWND             Stack[0x4]:4  param_1                              XREF[
                    FUN_01001ae6                                  XREF[3]:     FU
                                                                               FU
                                                                               FU
01001ae6 8b ff        MOV       EDI,EDI
01001ae8 55           PUSH      EBP
01001ae9 8b ec        MOV       EBP,ESP
01001aeb 6a 00        PUSH      0x0
01001aed ff 75 08     PUSH      dword ptr [EBP + param_1]
01001af0 ff 15 6c     CALL      dword ptr [->USER32.DLL::GetWindowLongA]  Attacker can control return value
         17 00 01
01001af6 5d           POP       EBP
01001af7 c2 04 00     RET       0x4


01001b39 e8 a8 ff     CALL      FUN_01001ae6                              undefined FUN_01001ae6(HWND para...
         ff ff
01001b3e 8b f0        MOV       ESI,EAX   EAX controlled by attacker
01001b40 85 f6        TEST      ESI,ESI
01001b42 0f 84 05     JZ        LAB_0102484d
         2d 02 00
01001b48 8b 06        MOV       EAX,dword ptr [ESI]
01001b4a 56           PUSH      ESI
01001b4b ff 10        CALL      dword ptr [EAX]   Code execution 1
01001b4d ff 75 14     PUSH      dword ptr [EBP + param_4]
01001b50 8b 06        MOV       EAX,dword ptr [ESI]
01001b52 ff 75 10     PUSH      dword ptr [EBP + param_3]
01001b55 8b ce        MOV       ECX,ESI
01001b57 57           PUSH      EDI
01001b58 53           PUSH      EBX
01001b59 ff 50 08     CALL      dword ptr [EAX + 0x8]  Code execution 2
01001b5c 81 ff 82     CMP       EDI,0x82
         00 00 00
01001b62 89 45 14     MOV       dword ptr [EBP + param_4],EAX
01001b65 0f 84 cf     JZ        LAB_0102483a
         2c 02 00

                LAB_01001b6b                                    XREF[1]:     01024848(j)
01001b6b 8b 06        MOV       EAX,dword ptr [ESI]
01001b6d 56           PUSH      ESI
01001b6e ff 50 04     CALL      dword ptr [EAX + 0x4]  Code execution 3
```

Message handler in explorer that PowerLoader hijacks.

AtomBombing

AtomBombing is another technique that uses ROP chains to get code execution in the remote process. Specifically, AtomBombing abuses the global atom table in Windows to share memory between processes and undocumented asynchronous procedure calls to force the target process into calling various functions on behalf of the injecting process. The injecting process writes a ROP chain and shellcode to the global atom table using the Windows function `GlobalAddAtom`. The injector then uses `NtQueueApcThread` to force the injected process to call `GlobalGetAtomName` to store the ROP chain and shellcode inside the target process. To invoke execution, the injector again uses `NtQueueApcThread` to force the injected process to call `SetThreadContext` to set eip and esp. Eip is set to the address of `ZwAllocateVirtualMemory` and esp is set to point to the beginning of the ROP chain. The injection is, therefore, achieved with a combination of `NtQueueApcThread` and `GlobalAddAtom`. An interesting aspect of AtomBombing is that not long after the publication of the technique an updated version of the infamous Dridex malware was **discovered**, that had adopted a modified version of the AtomBombing technique, and this is still being used in **2019**.

Process Doppelganging

The idea behind doppelganging is to improve the limitations of process hollowing, namely how the executable memory is written into the target process. Doppelganging achieves this by way of Windows Transactions. Doppelganging loads a benign executable using `CreateTransaction` and `CreateFileTransacted`, but then overwrites the content of the transacted file with malicious code, using `WriteFile`. Doppelganging then creates a section that holds the tainted transaction, i.e. the transaction that holds the malicious memory, and then performs a rollback on the transaction. The rollback will undo the changes performed by the transaction, which in Doppelganging's context is the overwriting of the benign file, so the changes to the benign file won't actually be commited to the file system. However, the caveat here is that the content of the section still contains the tainted code. Now, doppelganging proceeds to create the target process with the content of the malicious section. Doppelganging creates the process in a low-level way using `NtCreateProcess`, and, therefore, has to perform various set-ups for the process to accurately execute, such as setting up process parameters and create the process's main threads. An example of Process Doppelganging in the wild was **discovered in early 2018**.

Earlybird

Early bird refers to a technique that performs a somewhat traditional code injection via remote thread instantiation early in the process initialisation phase. Specifically, the attacker creates a new process in suspended mode and then proceeds to allocate and write memory to the process. In order to trigger execution the malware uses an asynchronous procedure call and enforces execution of the APC call using the `NtTestAlert` function. The technique was discussed by researcher from **Cyberbit** and even though it received its own name, it is closely related to earlier techniques traditional injection. This is also confirmed by the fact that the injection **dates back to at least 2012.**

```
CreateProcessA((LPCSTR)0x0,*(LPSTR *)(param_2 + 8),(LPSECURITY_ATTRIBUTES)0x0,
               (LPSECURITY_ATTRIBUTES)0x0,0,4,(LPVOID)0x0,(LPCSTR)0x0,lpStartupInfo,
               lpProcessInformation);
if (((lpProcessInformation->hProcess != (HANDLE)0x0) &&
    (pfnAPC = (PAPCFUNC)
              VirtualAllocEx(lpProcessInformation->hProcess,(LPVOID)0x0,0x1000,0x3000,0x40),
    pfnAPC != (PAPCFUNC)0x0)) &&
   (BVar1 = WriteProcessMemory(lpProcessInformation->hProcess,pfnAPC,&local_78,0x5e,
                               (SIZE_T *)0x0), BVar1 != 0)) {
  QueueUserAPC(pfnAPC,lpProcessInformation->hThread,0);
  ResumeThread(lpProcessInformation->hThread);
}
```

Code snippet of Earlybird injection.

ctrl-inject

The ctrl-c key combination is a well-known pattern for exiting and shutting down applications. The ctrl-inject injection technique exploits the underlying features that makes this hotkey possible. Specifically, when a user presses the ctrl + c keyword in a console application a system process (`csrss.exe`) invokes a function called `CtrlRoutine` in a new thread of the given console application. The `CtrlRoutine` fetches the given handler for the control signal (ctrl + c) which contains a function pointer that will be called, which effectively is used to handle the signal. In short, the technique overwrites this signal handler with a malicious function pointer, such that whenever the signal occurs the malicious handler will be called.

The strengths of ctrl-inject is that the technique does not rely on any function calls like `CreateRemoteThread`, `ResumeThread` or `SetThreadContext`, but rather triggers execution through the `csrss.exe` process. The technique that triggers the execution is a simple ctrl-c signal which in many scenarios is considered harmless and unsuspicious. The drawback of the technique is that it only works with console based applications.

```
void TriggerCtrlC(HWND hWindow)
{
        INPUT ip;
        ip.type = INPUT_KEYBOARD;
        ip.ki.wScan = 0;
        ip.ki.time = 0;
        ip.ki.dwExtraInfo = 0;
        ip.ki.wVk = VK_CONTROL;
        ip.ki.dwFlags = 0; //0 for keypress
        SendInput(1, &ip, sizeof(INPUT));
        Sleep(300);
        PostMessage(hWindow, WM_KEYDOWN, 0x43, 0);
        Sleep(300);
        ip.ki.dwFlags = 2; //2 for keyup (we want this, as we don't want to keep a system wide CTRL down)
        SendInput(1, &ip, sizeof(INPUT));

}
```

Code that triggers the injection in ctrl-inject. Source code can be found **here**

PROPagate

This technique was discovered in late 2017 and uses functionality of window subclassing to gain code execution in remote processes. Windows subclassing enables programmers to reuse functionality in existing controls by adding functionality and features to them. Whenever a window is subclassed, the messages to the original window is intercepted by the subclassing window which executes its own handlers before sending them on to the parent. You can read more about subclassing here. Whenever a window is subclassed it gets a new property called either xSubclassInfo or `CC32SubclassInfo` and stores the data structures related to these properties in its address space. This property points to a data structure inside the subclassed process which contains a function pointer that gets executed in the event of a message being sent to the window. The idea behind PROPagate is then to write a malicious data structure inside the remote process and use the `SetProp` function call to point to this handler. The attacker then uses `SendNotifyMessage` to trigger the malicious function handler. Roughly 8 months after the first documentation of PROPagate by **Adam**, FireEye **discovered** the RIG Exploit Kit delivering a dropper that used PROPagate.

```
printf("[*] Setting fake SubClass property\n");
SetProp(control, L"UxSubclassInfo", hFake);
printf("[*] Triggering shellcode....!!!\n");
SendMessage(control, WM_KEYDOWN, VK_NUMPAD1, 0);


printf("[+] Restoring subclass header.\n");
SetProp(control, L"UxSubclassInfo", hOrig);
```

The code that will trigger the PROPagate injection by setting the `UxSubclassInfo` property within the target window to a have a fake handler. Source code can be found **here**

Shatter-style attacks

The final category of code injections that we cover in this course is called Shatter attacks. The basic idea behind these injections is to misuse the message-oriented way that windows are architectured within Windows. Specifically, whenever applications use windows they control these in a message-oriented ways which is a very modular and effective way of managing windows. For example, when a key is pressed a messaged is sent to the currently active window stating this key was pressed. However, the way these messages are handled by the active window is through message handlers, i.e. data structures, and at certain times these can be overwritten with attacker controlled memory. Since a window can send messages to other windows on the desktop and we can overwrite memory using previously mentioned primitives, we can start to construct code injections by overwriting the message handlers in our target processes and then sending messages that trigger the respective handlers.

Shatter attackers were first presented by Chris Paget (Now Kristine Paget) in **2002**, however, the technique remains relevant. Recently, **Hexacorn** and **Odzhan** presented seven new ways of doing this.

Future outlooks

Code injections are continuously being used and there is a trend of increasingly more novel techniques being researched by defenders as well. For example, even in 2019 alone **modexp** has presented at least 15 novel injections (many inspired by Hexacorn), and modern malware samples use long chains of code injections to execute on the target system, such as the recent **Dridex** that uses five in total. The novel techniques are much more specific than the traditional injection attacks, and are now targeted internal structurse within the target processes rather than relying purely on common APIs. We can safely expect more exploit-like scenarios in the future and also new ways of enforcing process separation. Furthermore, even Academia is now in the field as well, both researching how we can use system-wide execution to construct highly sophisticated malware that operates across many processes (**malwash**) and **how to generically analyse novel injection techniques**. Code injections are here to stay and the complexity of them will increase. We highly recommend getting started with these techniques if you aren't already, and also predict that we will find more defensive tools and techniques being developed in the near future.

Conclusions

Code injections, also known as process injections, is an important topic in terms of post-exploitation strategies. Attackers, including both malware and pentesters, use these injections to execute code in otherwise benign processes as a way to bypass white-lists deployed by the defense products, e.g. host-based intrusion prevention and endpoint protection systems. From a defenders point of view we need to ensure our defense systems are aware of these tricks - and derivates hereof - in order to ensure our automated procedures are sound. Furthermore, from an attackers point of view, e.g. a pentester, these techniques can be of great benefit in order to secure access to a target machine. In this blog post we gave a motivation for both defenders and attackers on why studying code injections is relevant, and also highlighted technical aspects of several code injection techniques and attacks that use them.
