Related Posts

malwaretech.com/2013/08/powerloader-injection-something-truly.html

MalwareTech August 13, 2013

I'm not dead

It has been a while since i wrote an article (I've been pretty busy in real life), so I decided to get writing. This article will probably only make sense to people from a malware research / programming background, but to compensate i will be posting a fairly non technical article in the near future.
I will be talking about the infamous injection method from PowerLoader 2.0, which has been seen in many different malware families such as: Carberp, Redyms and Gapz. Recently, after looking at the difference between 0vercl0ck's proof of concept and the real deal, a friend asked me "Why does PowerLoader go to all the trouble of using ROP chains instead of just executing the shellcode like 0vercl0ck does.", I already had a perfect idea of why, but decided to do some digging and answer the question "How?", this digging resulted in me finding something that truly impressed me, (I try not to admire the work of criminals as i don't want to seem like a psychopath 😉). I would have written this article sooner, but i was totally unaware that no blogs had really gone into depth on this method, i like to be unique!

The Purpose

Most antiviruses don't treat all processes the same, a known "trusted" process is usually far less likely to flag up any warnings from the antivirus. In this case, the goal of malware is to inject code into one of these "trusted" processes in order to run with less risk of detection. Of course antiviruses will attempt to catch injection too, so the challenge is for malware to find a way into the trusted process without being detected.
In order to give a better idea of the stealthiness of PowerLoader I have listed below some common telltale signs of a malicious process attempting to inject.
(The following only apply to a process trying to perform any of these actions on another process)

- Allocating heap space
- Creating threads
- Overwriting process/module memory
- Manipulating thread context
- Queuing asynchronous procedure calls (APCs)

Proactive antiviruses will check for processes trying to perform these actions and could likely result in the user being alerted to a malicious process. The aim of PowerLoader is to subvert this, (which seems to be a success as it is not picked up by antiviruses, and does not cross off anything on the list).

Writing the code to explorer

In the case of PowerLoader, the trusted process targeted is explorer. I won't be putting any images/reversed code for this part as it has already been well documented by ESET. PowerLoader gets the malicious code into the process by opening an existing, shared section already mapped into explorer, removing the need to allocate heap space or overwrite process memory. PowerLoader then proceeds to map the shellcode onto the end of the chosen section. Below is a list of targeted shared sections.

- BaseNamedObjectsShimSharedMemory
- BaseNamedObjectswindows_shell_global_counters
- BaseNamedObjectsMSCTF.Shared.SFM.MIH
- BaseNamedObjectsMSCTF.Shared.SFM.AMF
- BaseNamedObjectsUrlZonesSM_Administrator
- BaseNamedObjectsUrlZonesSM_SYSTEM

Executing the code

In order to execute the remote code without creating a thread, PowerLoader uses a little trick with the explorer tray window procedure. By opening "Shell_TrayWnd" and calling SetWindowLong, PowerLoader is able to set a variable used by the window procedure to point to a specific address in its shellcode. Here PowerLoader sets the address to a pointer to a pointer to KiUserApcDispatcher, whereas 0vercl0ck's code will just set it to a pointer to a pointer to the payload (which resides in a shared section).
When SendNotifyMessage is called by the malware, the window procedure inside explorer is triggered and this is what happens.

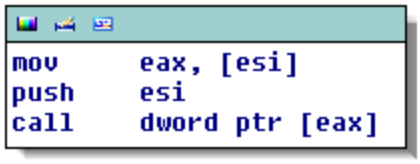

Figure 1: A snippet from the Window Procedure

Now this code is simple, it will perform a double indirection that will result in the address pointed to by the pointer that was set using SetWindowLong, being executed.

This is where PowerLoader differs from 0vercl0ck's version. The instruction "call dword ptr eax" will read the value pointed to by EAX and then call it. The read part won't trigger DEP (Data Execution Prevention), if the section is not executable (in later versions of windows it is execute-protected), however if EAX points to an address inside the section, DEP will be triggered. Because the sections protection is only set to Read/Write in later versions of windows, 0vercl0ck's code will likely trigger DEP and crash explorer, however, because

PowerLoader's pointer points to KiUserApcDispatcher (resides in ntdll), DEP is not triggered.
Well how does one get from KiUserApcDispatcher to code execution, without executing the non-executable shellcode, I hear you ask?

ROP Chains, Unicorns, and Rainbows

This part greatly interested me, partly because I have never seen a ROP chain in the wild before but mainly because it is the most advanced injection method I have ever come across. In order to understand how PowerLoader gets from KiUserApcDispatcher, to shellcode execution, we need to do some disassembling.
In Figure 1, we see the Window Procedure pushing ESI onto the stack, then calling KiUserApcDispatcher. It is important to remember ESI contains the address (held in the shellcode) of the pointer to the KiUserApcDispatcher pointer.
So let's see dissasemble KiUserApcDispatcher.

```
; Exported entry  45. KiUserApcDispatcher
; __stdcall KiUserApcDispatcher(x, x, x, x, x)
public _KiUserApcDispatcher@20
_KiUserApcDispatcher@20 proc near

Context= byte ptr  10h

lea     edi, [esp+10h]  ; Address in stack (Sets EDI for REP MOVSB)
pop     eax             ; Pop return address into EAX
call    eax             ; Call what would have been the return address
push    1
push    edi
call    _ZwContinue@8   ; ZwContinue(x,x)
nop
```

Figure 2: KiUserApcDispatcher

Pay attention to the first 3 instructions. "lea edi, [esp+10h]" is loading the last parameter into the EDI register. If you remember in Figure 1, the last parameter pushed to the stack was ESI, which contains an address within the shellcode. Next it pops the return address into the EAX and then calls it, this results in execution being transferred back to the Window Procedure.
So really nothing has happened here, We've just set the EDI to an address inside the shellcode and then gone back to where we came from. So in order to see what happens next, we are going to have to dig deeper. Here is some more of the Window Procedure.

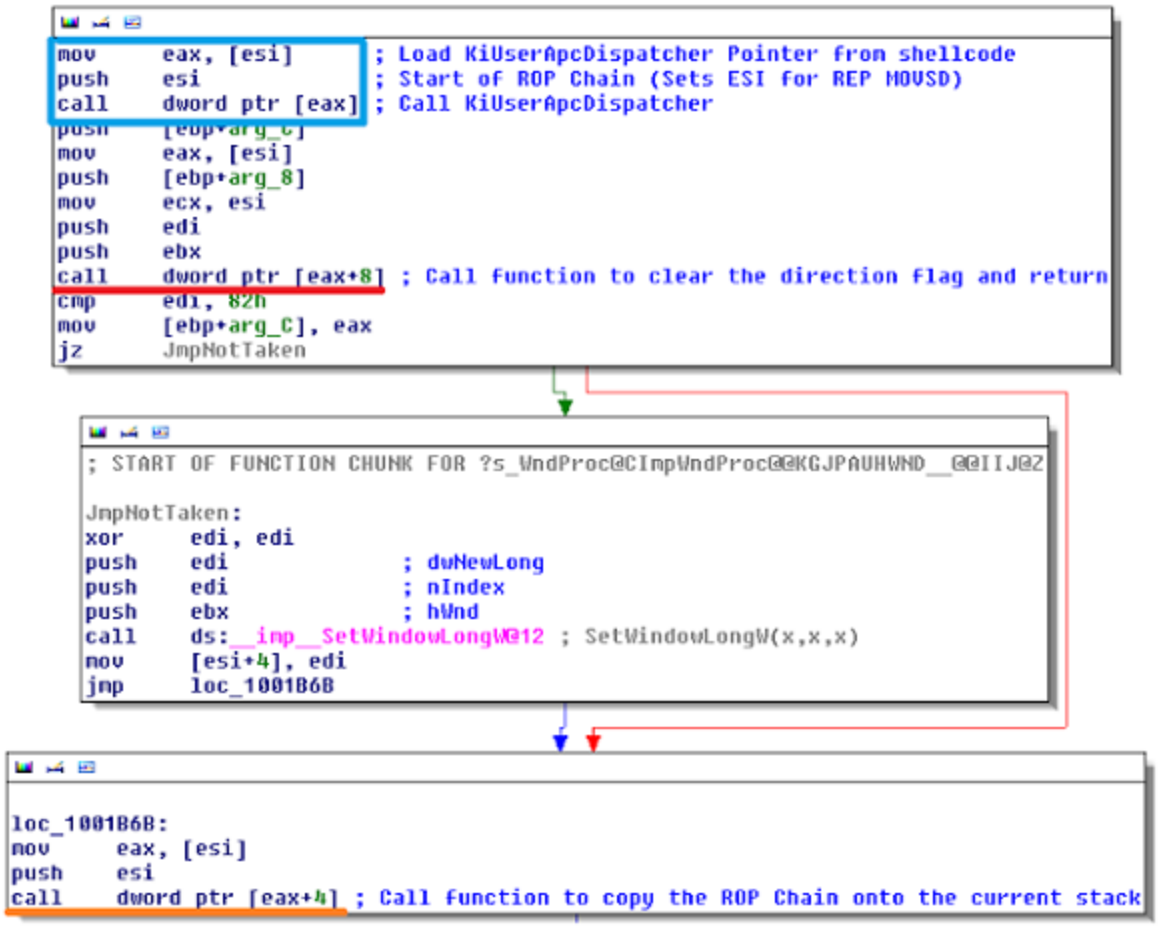

Figure 3: More of the Window Procedure shown in Figure 1

Now in this disassembly we need to pay attention to the instructions underlined in red and orange, the blue box is the code we already discussed (executes KiUserApcDispatcher and sets EDI to ESI), the rest of the code can be ignored. As you can see, the function makes 2 more calls (EAX+8, followed by EAX+4), if you remember earlier, EAX is an address in the shellcode, so the next call is to the address 8 Bytes below.

Let's take a look at the shellcode shall we?

```
00100E0C dd offset KiUserApcDispatcherPtr
00100E10 dd 0
00100E14 dd 0
00100E18 dd 0
00100E1C dd 0
00100E20 KiUserApcDispatcherPtr dd offset ntdll_KiUserApcDispatcher
00100E24 ROPGadget_MOVSB dd 0
00100E28 ROPGadget_SetDF dd 0
```

Figure 4: A small snippet from the shellcode

When SetWindowLong was called by PowerLoader it set the ESI (Blue Box Figure 3) to 00100E0C (Which holds the address 00100E20), The code then performs and indirection

and EAX ends up pointing to KiUserApcDispatchPtr (00100E20). Using some very basic maths, EAX+8 points to 00100E28 and EAX+4 to 00100E24.
What are 00100E28 & 00100E24? When the shellcode was made during runtime, PowerLoader searched for some byte sequences in explorer using ReadProcessMemory, then stored the addresses of those sequences in the shellcode. The sequences are instruction within the executable regions of explorer's memory, their purpose is to perform certain operations as PowerLoader can't execute any of its own code yet, due to the section being execute-protected.
00100E28 points to some code in explorer that executes the instruction "STD" followed by "RET", As a result the instruction underlined in red will result in the direction flag being set and execution being returned to the Window Procedure.

Until now, nothing makes any sense at all. We've set the ESI to an address in the shellcode (Figure 1), we've set the EDI to an address on the stack (Figure2), and we've set the direction flag. What happens next makes sense of it all. EAX+4 is called from the window procedure, as we established EAX+4 is a pointer in our shellcode, but what does it point to? Again, we need to do some disassembling.

```
; public: virtual long __stdcall CFileSysBindData::SetFindData(struct _WIN32_FIND_DATAW const *)
?SetFindData@CFileSysBindData@@UAGJPBU_WIN32_FIND_DATAW@@@Z proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
push    esi
mov     esi, [ebp+arg_4]
push    edi
mov     edi, [ebp+arg_0]
add     edi, 20h
mov     ecx, 94h
rep movsd               ; Copy ROP Chain over stack
pop     edi
xor     eax, eax
pop     esi
pop     ebp
retn    8
```

Figure 4: A random function in shell32.dll

Remember i said PowerLoader scanned some byte sequences in explorer? Well these bytes were found, in this case inside some random shell32 function (it doesn't matter). Now the pointer doesn't point to the start of the function, it points somewhere in the middle, as a result, only the bytes in the red box are executed. It should become apparent what is happening. The instruction "REP MOVSD" will move ECX (0x94) bytes from the address in ESI to the address in EDI. Earlier the code managed to use code within explorer to set the ESI to the shellcode address, the EDI to an address on the stack, then Set the direction flag to 1. Because of this, the shellcode starting at address 00100E0C will be copied to the stack

backwards (The copying will start at the address in ESI, copy a DWORD, then subtract the address by 4 and repeat.
(Remember: because all addresses points to executable code within explorer address space, and they are called using a pointer, no code in the shellcode is actually executed, thus resulting in no nasty DEP errors.)
This is where things start to heat up, PowerLoader has just used code located within explorer to overwrite some of the stack with some shellcode, which means although still incapable of directly executing its own code, PowerLoader has control over return addresses and stack based parameters.
Let's have a look at the code that was copied.

```
00090D8C dd 0
00090D90 dd 1
00090D94 dd 2
00090D98 dd 3
00090D9C dd offset ROPGadget_PopEax
00090DA0 CurrentProcHandle dd 0FFFFFFFFh
00090DA4 dd offset ntdll_atan
00090DA8 dd offset ShellcodeStart
00090DAC ShellcodeLength dd 80h
00090DB0 dd offset EmptyLocation
00090DB4 dd offset ntdll_atan
00090DB8 ROPGadget_JmpEax dd 1173CEFh
00090DBC EmptyLocation dd 0Ch
00090DC0 dd 0Dh
00090DC4 ShellCodeIAT dd offset unk_90E2C
00090DC8 dd 0Fh
00090DCC dd 10h
00090DD0 dd 11h
00090DD4 dd 12h
00090DD8 dd 13h
00090DDC dd 14h
00090DE0 dd 15h
00090DE4 dd 16h
00090DE8 dd 17h
00090DEC dd 18h
00090DF0 dd offset ROPGadget_ClearDF
00090DF4 dd 1Ah
00090DF8 dd 1Bh
00090DFC dd offset ROPGadget_PopEax
00090E00 dd 70h
00090E04 dd offset ntdll__alloca_probe
00090E08 dd offset kernel32_WriteProcessMemory
```

Figure 5: The ROP Shellcode that is written to the stack

Once the code copying the ROP Shellcode to the stack is done, it hits the ret instruction, but because the stack has been overwritten, it instead ends up executing code pointed to by the ROP Shellcode, Each bit of code has a ret instruction which causes the next ROP gadget to be executed. I stepped through in a debugger, below i have made a list of the ROP Gadgets in order of execution, each line is a different gadget.

1. Direction Flag Clear

2. Pop 0x70 into EAX
3. Call _alloca_probe
4. WriteProcessMemory
5. Pop the address of ntdll!atan into EAX
6. Jmp to EAX

Some things to note:

- The _alloca_probe function is undocumented but I believe it takes the value in EAX and check that the stack can hold that many items, if not it triggers the guard page to allocate more stack space (0x70 is in EAX)
- The parameters for WriteProcessMemory are at address 00090DA0, these parameters cause WriteProcessMemory to read the shellcode from the shared section, then write it over ntdll!atan which we can assume isn't used by explorer.
- Finally the last instruction jumps to ntdll!atan and the code begins execution.

TLDR / Recap

PowerLoader bypasses the execution protection on the shared sections, by using code found inside explorer to copy a ROP Chain to the stack, then uses the ROP Chain to manipulate the call stack into causing Explorer to call WriteProcessMemory and overwrite an unused function in ntdll with some shellcode to complete the injection.

Conclusion

So there we have it, from non-executable section to shellcode execution by using explorer's own code against itself. I'll try and get a new article up soon, sorry for the inactivity <3