

# Reverzní inženýrství

## 6. Debugging a Anti-debugging

Ing. Tomáš Zahradnický, EUR ING, Ph.D.

Ing. Martin Jirkal

Ing. Josef Kokeš



**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

České vysoké učení technické v Praze  
Fakulta informačních technologií  
Katedra informační bezpečnosti

Verze 2022-09-09

# Obsah

## 1 Debugging

- Debuggery
- Debugging
- Breakpointy
- Trasování

## 2 Debugování jádra

- Setup
- Nalezení vstupního bodu

## 3 Anti-debugging

# Debugger

## Debugger

Debugger je nástroj, který používáme k hledání chyb vznikajících během běhu aplikace. Dovoluje zkoumat aplikační kód, a to buď na úrovni zdrojového kódu, nebo na úrovni assembleru.

- Reverzní inženýr používá debugger:
  - Ke sledování toku kódu.
  - Pro získání lepšího porozumění určitým částem aplikace.
  - K nalezení odpovědí na otázky:
    - Jaké argumenty API používá?
    - Kde přesně dochází k pádu aplikace?
    - Pracuje algoritmus tak, jak očekáváme?
  - K odstranění obfuskace/šifrování.
  - K ověření hypotéz o chování aplikace.

# Typy debuggerů

- **Debugger v uživatelském režimu** (user-mode) dokáže ladit běžné aplikace.
- **Jádrový debugger** (kernel) může debugovat jádro OS a ovladače jádra.
- **Debugger zdrojového kódu** (source-level) může nastavovat breakpointy na řádky zdrojového kódu aplikace. Typicky je integrován do IDE vývojového nástroje.
- **Nízkoúrovňový debugger** (low-level, assembly-level) pracuje na úrovni assembleru dané architektury.

# Vlastnosti debuggerů

## Debugger:

- Spouští aplikaci.
- Zobrazuje aktuální stav proměnných/registrů.
- Na požádání aplikaci zastaví a umožní krokovat po jednotlivých řádcích.
- Vyhodnocuje jména symbolů.
- Umožňuje zpětný debugging (vracení zpět v toku kódu).
- Čte/zapisuje paměť programu.
- Poskytuje podporu pro vzdálený debugging (po sériové lince nebo přes síť).

# Debuggery

- Komerční
  - Hex-Rays Interactive Disassembler Pro (IDA Pro)
  - Hopper
  - Visual DuxDebugger
  - † Syser, † SoftICE
- Zdarma (user-level)
  - WinDBG
  - OllyDBG
  - Immunity Debugger
  - x64dbg
  - Eclipse
  - GDB
  - LLDB
  - Microsoft Visual Studio Debugger
- Zdarma (kernel-level)
  - WinDBG

# OllyDBG

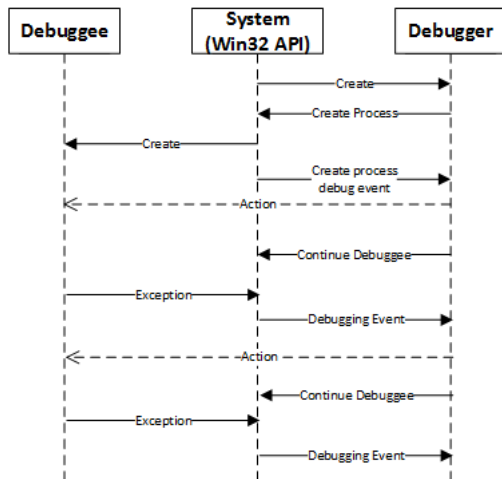
- Free.
- Běžně používaný pro dynamickou analýzu v rámci reverzního inženýrství.
- Vlastnosti:
  - Analýza kódu — sleduje registry, rozpoznává procedury, smyčky, API volání, příkazy switch, tabulky, konstanty, řetězce.
  - Dokáže přímo načítat a debugovat DLL.
  - Skenování objektových souborů — dohledává rutiny z objektových souborů a knihoven.
  - Podporuje uživatelsky definovaná návěští, komentáře, popisy funkcí.
  - Zpracovává debug info ve formátu Borland.
  - Umí uložit změny mezi sezeními do souboru a propsat je do exe souboru či do procesu.
  - Podporuje pluginy třetích stran, zejména ve verzi 1.x. Pro verzi 2.x existuje pouze zlomek jejich množství.

# Jak debugování funguje

## Debuggee

Proces, který je debugován.

- 1 Debugger buď vytvoří nového debuggee, nebo se připojí k existujícímu.
- 2 Debugger zpracovává debugovací události v debugovací smyčce.





# Připojení debuggeru

- Neinvazivní

- Debugger se nepřipojí k aplikaci.
- Všechna vlákna aplikace jsou pozastavena, aby bylo možné přechíst stav programu (registry, paměť, atd.).
- Dává jen omezenou kontrolu nad cílovou aplikací.

- Invazivní

- Aplikace může mít pouze jeden připojený debugger.
- Připojenému debuggeru jsou zasílány všechny debugovací události aplikace. Ty zahrnují:
  - CREATE\_PROCESS\_DEBUG\_EVENT
  - CREATE\_THREAD\_DEBUG\_EVENT
  - EXCEPTION\_DEBUG\_EVENT
  - EXIT\_PROCESS\_DEBUG\_EVENT
  - EXIT\_THREAD\_DEBUG\_EVENT
  - LOAD\_DLL\_DEBUG\_EVENT
  - OUTPUT\_DEBUG\_STRING\_EVENT
  - UNLOAD\_DLL\_DEBUG\_EVENT
  - RIP\_EVENT

# Debugovací události I

## CREATE\_PROCESS\_DEBUG\_EVENT

Událost je generována pokaždé, když je v debugovaném procesu vytvořen nový proces, nebo v okamžiku, kdy se debugger připojí k už aktivnímu procesu. Systém vyvolá tuto událost těsně před tím, než je program v user-modu spuštěn, a to před všemi ostatními debugovacími událostmi tohoto procesu.

## EXIT\_PROCESS\_DEBUG\_EVENT

Událost je generována v okamžiku, kdy debugovaný proces končí, a to těsně poté, co systém odpojil knihovny používané procesem a aktualizoval výstupní kód (exit code) procesu.

## Debugovací události II

### CREATE\_THREAD\_DEBUG\_EVENT

Událost je generována pokaždé, když je v debugovaném procesu vytvořeno nové vlákno, nebo v okamžiku, kdy se debugger připojí k už aktivnímu procesu. Událost je vyvolána těsně před zahájením běhu vlákna v user-mode.

### EXIT\_THREAD\_DEBUG\_EVENT

Událost je generována v okamžiku, kdy končí vlákno, které je součástí debugovaného procesu, a to těsně poté, co byl aktualizován výstupní kód vlákna.

## Debugovací události III

### LOAD\_DLL\_DEBUG\_EVENT

Událost je generována, když debugovaný proces nahraje do paměti nové DLL. Příčinou může být jak vyhodnocování odkazů na DLL při spouštění procesu, tak použití funkce `LoadLibrary` debugovaným procesem. Událost je pro každé DLL vyvolána pouze jednou, a to v okamžiku, kdy bylo DLL načteno do paměťového prostoru procesu.

### UNLOAD\_DLL\_DEBUG\_EVENT

Událost je generována, když debugovaný proces uvolnil DLL funkcí `FreeLibrary`. Pro každé DLL je událost vyvolána jen jednou, a to v okamžiku, kdy bylo DLL skutečně odstraněno z paměťového prostoru procesu (tzn. v okamžiku, kdy počet použití DLL v procesu klesl na nulu).

## Debugovací události IV

### EXCEPTION\_DEBUG\_EVENT

Událost je generována pokaždé, když v debugovaném procesu dojde k výjimce. Příčinou výjimky může být pokus o přístup k nedostupné paměti, spuštění instrukce breakpointu, pokus o dělení nulou nebo jakákoliv jiná výjimka sledovatelná strukturovanou obsluhou výjimek.

### OUTPUT\_DEBUG\_STRING\_EVENT

Událost je generována v okamžiku, kdy debugovaný proces použil funkci `OutputDebugString`.

# Debugovací události V

## RIP\_EVENT

Událost je generována v okamžiku, kdy došlo k systémové chybě během debugování. Prakticky jde o interní chybu debuggeru, ne debugovaného procesu. Debugovaná aplikace může a nemusí přežít, podle závažnosti chyby.

# Připojení debuggeru při vytváření procesu

## Kód pro spuštění debugovaného programu

```
DWORD dwError = ERROR_SUCCESS;
STARTUPINFO si;
PROCESS_INFORMATION pi;

ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );

dwError = CreateProcess( DebuggeePath, NULL, NULL, NULL, FALSE,
                        DEBUG_ONLY_THIS_PROCESS, NULL, NULL, &si, &pi );
```

- Proces bude vytvořen v pozastaveném (suspended) stavu.
- Debuggeru bude ke zpracování zaslána událost `CREATE_PROCESS_DEBUG_EVENT`.

# Připojení debuggeru k existujícímu procesu

## API pro připojení k procesu

```
BOOL WINAPI DebugActiveProcess( DWORD dwProcessId )
```

- Debugger musí mít příslušná oprávnění.
  - Každý uživatel může i bez dodatečných oprávnění debugovat své vlastní procesy.
  - Pro debugování **cizích** procesů je vyžadováno oprávnění SeDebugPrivilege! To platí i pro API Read/WriteProcessMemory.
- Debuguje aktivní proces, jako kdyby byl vytvořen pomocí API CreateProcess s příznakem DEBUG\_ONLY\_THIS\_PROCESS.
- Všechna vlákna debugovaného procesu jsou systémem pozastavena.
- Debugger obdrží událost LOAD\_DLL\_DEBUG\_EVENT pro všechny moduly načtené do debugovaného procesu.
- Debugger obdrží událost CREATE\_PROCESS\_DEBUG\_EVENT z prvního vlákna debugovaného procesu.



## Debugovací smyčka

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;

DEBUG_EVENT debug_event = {0,};

for(;;)
{
    if(!WaitForDebugEvent(&debug_event, INFINITE))
        return;

    // zpracování debugovací události

    ContinueDebugEvent(debug_event.dwProcessId, debug_event.dwThreadId, DBG_CONTINUE);
}
```

# Komunikace debuggeru s debugovaným procesem I

## Funkce pro práci s pamětí

```
BOOL WINAPI WriteProcessMemory(  
    _In_ HANDLE hProcess,  
    _In_ LPVOID lpBaseAddress,  
    _In_ LPCVOID lpBuffer,  
    _In_ SIZE_T nSize,  
    _Out_ SIZE_T *lpNumberOfBytesWritten  
);  
  
BOOL WINAPI ReadProcessMemory(  
    _In_ HANDLE hProcess,  
    _In_ LPCVOID lpBaseAddress,  
    _Out_ LPVOID lpBuffer,  
    _In_ SIZE_T nSize,  
    _Out_ SIZE_T *lpNumberOfBytesRead  
);
```

- Umožňují čtení/zápis paměti procesu.
- Debuggery používají tyto funkce pro čtení/zápis paměti a pro vkládání softwarových breakpointů do kódu.
- Debuggery vždy pracují s obrazem aplikace v paměti, **nemění spustitelný kód na disku**.

# Komunikace debuggeru s debugovaným procesem II

## Funkce pro práci s vlákny

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

- Struktura CONTEXT obsahuje aktuální stav všech registrů.
- Struktura CONTEXT je závislá na architektuře procesu, který je debugován (t.j. liší se pro i386 a x86\_64).
- Debuggery používají tyto funkce ke čtení a změně hodnot registrů a k práci s EIP.

# Softwarové breakpointy

- Softwarové přerušení (nebo výjimka) je událost vyvolaná softwarem, která informuje jádro OS o tom, že normální tok instrukcí programu musí být změněn (program se dostal do nenormálního stavu).
- Ukazatel na (nejvyšší) funkci pro obsluhu výjimky je uložen v první proměnné Thread Information Blocku (TIB) a můžeme ho zjistit z FS:[0].
- Pokud dojde k výjimce, prochází se řetězec výjimek a hledá se vhodný handler, který bude spuštěn.
- Pokud pro danou výjimku neexistuje handler, použije se standardní mechanismus OS pro zpracování výjimky.
- Většina debuggerů používá pro breakpointy instrukci `int3` (operační kód `0xCC`). Stejného výsledku by se dalo dosáhnout také instrukcí `int 3` (`0xCD03`): obě instrukce vyvolají výjimku Breakpoint.

# Vytvoření breakpointu

```
BOOL bSuccess;
BYTE cInstruction;
BYTE OriginalInstruction;
DWORD dwReadBytes;

bSuccess = ReadProcessMemory( hProcess, (void*)Breakpoint_Address,
                             &cInstruction, 1, &dwReadBytes );

OriginalInstruction = cInstruction;
cInstruction = 0xCC;

// Přepsat původní instrukci hodnotou 0xCC
bSuccess = WriteProcessMemory( hProcess, (void*)Breakpoint_Address,
                              &cInstruction, 1, &dwReadBytes );

FlushInstructionCache( hProcess, (void*)Breakpoint_Address, 1);
```

- 1 Přečteme 1 bajt z adresy Breakpoint\_Address a zapamatujeme si ho.
- 2 Přepíšeme první bajt instrukce hodnotou 0xCC.
- 3 Vyprázdníme instrukční cache.
- 4 Pokračujeme v debugování.

# Zpracování jednorázového breakpointu

## Zpracování jednorázového breakpointu

```
CONTEXT lcContext;  
DWORD dwWriteSize;  
  
lcContext.ContextFlags = CONTEXT_ALL;  
GetThreadContext( hThread, &lcContext );  
lcContext.Eip --;  
SetThreadContext( hThread, &lcContext );  
WriteProcessMemory( hProcess, Breakpoint_Address,  
                    &OriginalInstruction, 1, &dwWriteSize );  
FlushInstructionCache( hProcess, StartAddress, 1 );
```

- 1 Načteme strukturu CONTEXT vlákna.
- 2 Snížením EIP o 1 se vrátíme o 1 bajt zpět (na instrukci breakpointu).
- 3 Nastavíme nový kontext.
- 4 Obnovíme původní instrukci.
- 5 Pokračujeme v debugování.

Všimněte si, že breakpoint byl odstraněn!

# Zpracování trvalého breakpointu

## Provedení jedné instrukce a zastavení

```
CONTEXT lcContext;  
DWORD dwWriteSize;  
  
lcContext.ContextFlags = CONTEXT_ALL;  
  
GetThreadContext( hThread, &lcContext );  
lcContext.EFlags |= 0x100; // Nastavit příznak Trap na EXCEPTION_SINGLE_STEP  
SetThreadContext( m_cProcessInfo.hThread, &lcContext );
```

- 1 Načteme CONTEXT vlákna a uložíme ho.
- 2 Obnovíme původní instrukci (zrušíme breakpoint).
- 3 Snížením EIP o 1 se vrátíme k této instrukci.
- 4 Nastavíme příznak "Trap" v EFL na single-step výjimku.
- 5 Nastavíme nový CONTEXT pomocí SetThreadContext.
- 6 Pokračujeme v debugování.
- 7 Po první provedené instrukci dostaneme opět STATUS\_BREAKPOINT.
- 8 Obnovíme breakpoint.

# Hardwarové breakpointy

- Architektura Intel x86 obsahuje 6 debugovacích registrů.
- DR0–DR3 mohou každý obsahovat jednu lineární adresu hardwarového breakpointu.
- DR6 (Debug Status) sděluje aplikaci, jaká debugovací situace nastala.
- DR7 (Debug Control) obsahuje příznaky:
  - Lokálně povolený hardwarový breakpoint.
  - Globálně povolený hardwarový breakpoint.
  - Přerušení při spuštění.
  - Přerušení při zápisu.
  - Přerušení při přístupu (zápisu nebo čtení).
  - Velikost sledovaného paměťového místa (1 B, 2 B, 4 B, nebo 8 B).



# Hardwarové vs. softwarové breakpointy

- Softwarový breakpoint

- Výchozí typ breakpointů ve většině debuggerů.
- Neomezené množství breakpointů v programu.
- Může detekovat spuštění instrukce.
- Mění obsah paměti → lze snadno detekovat.

- Hardwarový breakpoint

- Podporován je jen omezený počet breakpointů (závisí na procesoru, Intel x86 podporuje 4).
- Může detekovat spuštění instrukce i přístup k paměti.
- Nemění obsah paměti → obtížnější detekce.
- Čtení i zápis debugovacích registrů je privilegovaná operace.
- Podporován většinou debuggerů.

# Trasování (Trace)

- Většina debuggerů podporuje trasování instrukcí a trasování funkcí. Využívá k tomu příznak Trap v registru příznaků.
- Tato funkcionality umožňuje sledovat přesný tok řízení procesu.
- Při **trasování instrukcí** se zaznamenávají prováděné instrukce.
- Při **trasování funkcí** se zaznamenávají prováděná volání.
- Trasování významně zpomaluje provádění vlákna, protože debugger musí po každé instrukci zastavit a aktualizovat svůj log.

# Úvod do debugování jádra

- “Vytvářím ovladač pro jádro a počítač mi neustále padá.”
  - “Přirozeně. Jádro není přátelské prostředí jako uživatelský režim. Chyba typicky vyústí v BSOD. Přečti si BSOD zprávu, analyzuj dumpy vytvořené při BSOD, nebo prostě debuguj svůj ovladač.”
- “Co když zastavím program, abych mohl prozkoumat stav počítače?”
  - “Zastavíš jádro → počítač zamrzne.”
- “A jak to tedy mám debugovat? Vytvořit aplikaci bez debugování je hrozně těžké!”
  - “Potřebuješ druhý počítač, který bude debugování řídit.”
- “Jakou aplikaci mám použít?”
  - “Nejlepší je WinDBG.”

# Setup pro debugování jádra – lokální debugování

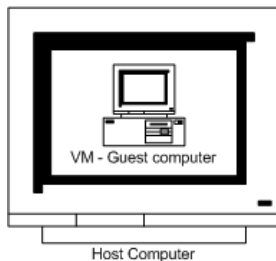
- Spustíte WinDBG, stiskněte CTRL+K a zvolte Local.
- Nyní můžete debugovat počítač, na kterém debugger běží.
- Nemáte k dispozici všechny příkazy. Nedostupné jsou příkazy pro řízení běhu (go, step, atd.), nelze použít dump, breakpointy, sledovat registry ani zásobník.
- Můžete číst a psát paměť.
- Kernel stále běží → všechny informace se vám mohou měnit pod rukama.
- MŮŽETE shodit celý systém.

# Setup pro debugování jádra – vzdálený počítač



- Debugujeme počítač přes TCP/IP, IEEE 1394, COM, nebo USB.
- Nejspolehlivější a nejpřesnější řešení.
- Dlouhá doba obnovy po pádu.

# Setup pro debugování jádra – virtuální počítač



- Debugujeme virtuální počítač. Pro “vzdálené” debugování používáme pojmenované roury nebo virtuální síť. Závisí na použitém virtualizačním software.
- Může být dost náročné to zprovoznit.
- Virtuální počítač se chová skoro stejně jako opravdový. V 99.9% případů nám dá stejné výsledky.
- Stav stroje se dá ukládat a obnovovat → velmi rychlý návrat do původního stavu.

# Debugování jádra - Trik s breakpointem I

- Zjistěte souborovou adresu vstupního bodu (Raw Entrypoint).  
$$EP_{Raw} = EP_{RVA} - Segment_{RVA} + Segment_{Raw}$$
- Změňte bajt na  $EP_{Raw}$  na 0xCC (breakpoint). Uložte si původní hodnotu!
- Nezapomeňte aktualizovat CRC a digitální podpis změněného souboru, jinak ho Windows odmítnou spustit.
- Spusťte upravený ovladač. Jádro by mělo zastavit na vašem breakpointu.
- Pomocí debuggeru změňte bajt ve vstupním bodu na původní hodnotu.  
Ve WinDBG: `eb [ENTRYPOINT_RVA] [PUVODNI_HODNOTA]`
- Další možností je použít instrukci 0xEB 0xFE (JMP -2), která vytvoří nekonečnou smyčku skokem sama na sebe.

# Debugování jádra - Trik s breakpointem II

- Nalezněte nedokumentovanou funkci `IopLoadDriver()`.
- Najděte instrukci `call [edi + xx]`. Na Windows XP SP3 by měla ležet na offsetu `0x66A` od začátku funkce.
- Zapište tam breakpoint a spusťte debugger. Na debugovaném stroji spusťte ovladač. Provádění se zastaví jeden `call` před vstupním bodem.

## Nastavení breakpointu ve WinDBG

```
bp nt!IopLoadDriver + 0x66a
```



# Anti-debugging

## Anti-debugging

Anti-debuggingem rozumíme obranu aplikace proti debugování. Aplikace se snaží detekovat připojený debugger, vměšovat se do jeho činnosti nebo z debuggeru uniknout.

- Přítomnost debuggeru může být detekována rozličnými metodami, od prostého volání API až po detekování chování typického pro debugger.
- Debugger můžeme ukončit zneužitím jeho specifických zranitelností.
- Aplikace může debuggeru uniknout přesunutím do jiného vlákna nebo procesu.

# Použití Windows API k detekci debuggeru

- Základní detekce debuggeru může být provedena voláním API funkcí Windows.
  - **IsDebuggerPresent** — vrací pole BeingDebugged ze struktury Process Environment Block (PEB).
  - **CheckRemoteDebuggerPresent** — totéž co IsDebuggerPresent, ale může být použita i na jiný proces.
  - **NtQueryInformationProcess** — nativní API funkce z ntdll.dll. Může vracet různé informace, např. ProcessDebugPort, který bude nastaven, pokud je aplikace debugována.
  - **OutputDebugString** — pak voláním GetLastError ověříme, zda je proces debugován.
- Anti-anti-debuggovací techniky:
  - Ovlivňování toku programu.
  - Napojení se na API a vracení správných hodnot.

# Příznak BeingDebugged

- IsDebuggerPresent načítá z PEB příznak BeingDebugged.  
Manuální provedení téhož je těžké na detekci i na zablokování:

## i386 ASM

```
// Přečti lin. adr. PEB z TIB
mov eax, dword ptr fs:[30h]

// Přečti PEB.BeingDebugged
movzx ebx, byte ptr [eax+2]

// Test na nenulu
test ebx, ebx
jz DebuggerNebylDetekovan
```

## x86\_64 ASM

```
// Přečti lin. adr. PEB z TIB
mov rax, qword ptr gs:[60h]

// Přečti PEB.BeingDebugged
movzx rbx, byte ptr [rax+2]

// Test na nenulu
test rbx, rbx
jz DebuggerNebylDetekovan
```

## PEB struktura v C

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PPEB_LDR_DATA Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    BYTE Reserved4[104];
    PVOID Reserved5[52];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE Reserved6[128];
    PVOID Reserved7[1];
    ULONG SessionId;
} PEB, *PPEB;
```

- Anti-anti-debuggingová technika: Vynulování BeingDebugged.

# Příznak ProcessHeap

- Microsoft řadu interních struktur nedokumentuje, ale mnoho z nich je na Internetu analyzováno.
- Pole `PEB.ProcessHeap` obsahuje strukturu, která může být použita pro detekci debuggeru.
- Pole na offsetu `0x10` od začátku `ProcessHeap` se nazývá `ForceFlags`. Pokud byla halda vytvořena debuggerem, je nenulové.

## ASM kód pro Windows XP

```
mov eax, dword ptr fs:[30h]
mov eax, byte ptr [eax+18h]
cmp dword ptr ds:[eax+10h], 0
jne DebuggerDetekovan
```

- Anti-anti-debuggingová technika:
  - Přepíšeme hodnotu `ProcessHeap` nebo spustíme debugger s vypnutou haldou (`windbg -hd`).

# Příznak NTGlobalFlag

- Další dobře známý ale nezdokumentovaný příznak je NTGlobalFlag. Proces vytvořený debuggerem v něm má nastavené bity FLG\_HEAP\_ENABLE\_TAIL\_CHECK, FLG\_HEAP\_ENABLE\_FREE\_CHECK, a FLG\_HEAP\_VALIDATE\_PARAMETERS.
- Obvyklý přístup je, porovnat NTGlobalFlag na offsetu 0x68 s hodnotou 0x70. Pozor, není to spolehlivé, protože mohou být zapnuty i další bity.

## ASM kód

```
mov eax, dword ptr fs:[30h]  
cmp dword ptr ds:[eax+68h], 70h  
jz DebuggerDetekovan
```

- Anti-anti-debuggingová technika:
  - Přepsat NTGlobalFlag.

# Detekování debuggeru skenováním procesů

- Tento test na debugger můžeme obejít změnou jména souboru debuggeru.

## Detekce WinDBG

```
#include <windows.h>
#include <tlhelp32.h>

BOOL isDebugged(void)
{
    HANDLE hProcessSnapshot;
    HANDLE hProcess;
    PROCESSENTRY32 pe32;
    DWORD dwPriorityClass;

    hProcessSnapshot = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
    pe32.dwSize = sizeof( PROCESSENTRY32 );

    if( !Process32First( hProcessSnapshot, &pe32 ) )
        return( FALSE );

    do {
        if (wcsicmp(pe32.szExeFile, L"windbg.exe") == 0 )
            return ( TRUE );
    } while( Process32Next( hProcessSnapshot, &pe32 ) );

    CloseHandle( hProcessSnap );
    return( FALSE );
}
```

# Detekce debuggeru pomocí časování

- Ověříme, zda jsou funkce prováděny v normálním čase (milisekundy) nebo zda provádění něco zdržuje.
- Tato metoda odhalí, zda je aplikace krokovaná nebo došlo k jejímu pozastavení mezi měřícími body.
- Alternativou je vyvolání výjimky. Ta je normálně zpracována rychle, ale debugger obvykle počká na reakci uživatele.

## Příklad s rdtsc

```
rdtsc
xor ecx,ecx
add ecx,eax
rdtsc
sub eax,ecx
cmp eax,0xFFFF
jb NeniDebugger
```

## GetTickCount

```
ULONGLONG a = GetTickCount64();
DulezitaFunkce();
ULONGLONG b = GetTickCount64();
delta = b - a;
if ( delta > 30 ) {
    /*Debugger detekován*/
} else {
    /*Nenalezen debugger*/
}
```

- Anti-anti-debuggingová technika:
  - Zaháčkování časovacích funkcí.

## Další metody pro detekci debuggeru

- Aplikace může prohledat svůj vlastní kód na instrukce `int3` (0xCC) a tím odhalit softwarové breakpointy.
- Pokročilé aplikace provádějí kontrolní součty sebe sama a porovnávají dosažené hodnoty s očekávanými. Změna v kódu (softwarový breakpoint) je tak detekována.



# Vměšování se do debugování

- Výjimky vyžadují vstup od uživatele. Vytvoření stovek výjimek s prázdným handlerem nezvýší viditelně dobu běhu, ale otráví analytika natolik, že vypne upozornění na výjimky. A to je přesně ten moment, kdy můžeme schovat důležitý kód do některého handleru.
- Program může obsahovat instrukce `int3`, kterými zmate debugger, aby si myslel, že jde o breakpoint. Bez debuggeru by byla vyvolána výjimka `STATUS_BREAKPOINT`, s debuggerem k výjimce nedojde a bude spuštěna následující instrukce. Debugovaná aplikace tak má odlišný tok kódu!
  - Některé debuggery dokonce na takovém neznámém breakpointu spadnou!
- Každý debugger má známou sadu zranitelností, které ho dokáží sestřelit. Například OllyDBG v1.1 je známý tím, že spadne, když zavoláme `OutputDebugString("%s%s%s%s%s%s%s%s%s%s")`.
- K procesu může být v jeden okamžik připojen jen jeden debugger. Připojte se sami k sobě jako debugger!

# Únik z debuggeru

- Injektujte důležitý kód do jiného procesu.
- Uložte důležitý kód do handleru výjimky.
- Uložte důležitý kód před/za main (např. využijte funkce `initterm` vysvětlené ve 2. přednášce).
- Uložte důležitý kód před start: Ještě před entry-pointem se mohou spouštět tzv. TLS<sup>1</sup> callbacky, které slouží pro inicializaci TLS proměnných. Pokud celý škodlivý kód proběhne už zde, analytik – a někdy ani debugger – se o něm vůbec nedozví!

---

<sup>1</sup>thread-local storage

## Další anti-debuggingové techniky

- Existuje mnoho dalších anti-debuggingových metod. Pěknou kompilaci naleznete v <http://pferrie.host22.com/papers/antidebug.pdf>.
- Pozn.: Pro základní a některé pokročilé anti-debuggingové triky existují anti-anti-debuggingové pluginy, jako např. IDA Stealth pro IDA Pro nebo Phant0m pro OllyDBG.

# Literatura



Ajay Vijayvargiya: *Writing a basic Windows debugger*, January 2015, <http://www.codeproject.com/Articles/43682/Writing-a-basic-Windows-debugger>.



Microsoft: *MSDN*, January 2015, <https://msdn.microsoft.com/>.



Peter Ferrie: *The “Ultimate” Anti-Debugging Reference*, April 2011, <http://pferrie.host22.com/papers/antidebug.pdf>.