MI-PB-15

Debuggery a debugging. Body přerušení. Obrana proti debuggingu.

Debugger: nástroj používaný k hledání chyb vznikajících během běhu aplikace.

Dovoluje zkoumat aplikační kód, buď na úrovni zdrojového kódu nebo na úrovni assembleru

Použití v reverzním inženýrství:

- Sledování toku kódu
- Lepší porozumění aplikaci
- Zjištění používaných argumentů API, místa pádu aplikace, principu použitého algoritmu
- Odstranění obfuskace/šifrování
- Ověření hypotéz o chování aplikace

Typy debuggerů:

- **Debugger v uživatelském režimu:** ladění běžných aplikací
- **Jádrový debugger:** ladění jádra OS a ovladačů
- **Debugger zdrojového kódu:** nastavování breakpointů na řádky zdrojového kódu, typicky integrován do IDE
- **Nízkoúrovňový debugger:** pracuje na úrovni assembleru dané architektury

Činnosti debuggerů:

- Spuštění aplikace
- Zobrazení aktuálního stavu proměnných/registrů
- Zastavení, krokování aplikace
- Vyhodnocení jmen symbolů
- Zpětný debugging (vracení zpět v toku kódu)
- Čtení/zápis paměti programu
- Vzdálený debugging -- po síti

Debuggery:

- Komerční:

    - IDA Pro
    - Hopper
    - Visual DuxDebugger
- Zdarma:
    - WinDBG (i kernel-level)
    - OllyDBG
    - x64dbg
    - GDB

Jak funguje debugování

Debugger vytvoří nového debuggee, nebo se připojí k existujícímu.

Typy připojení debuggeru:

- **Neinvazivní:** debugger se nepřipojí k aplikaci, všechna vlákna aplikace pozastavena, aby bylo možné přečíst stav programu (registry, paměť,...)
- **Invazivní:** Pouze jeden připojený debugger, tomu jsou zasílány debugovací události aplikace

Připojení debuggeru:

- Při vytváření procesu:
    - Debugger spustí debuggeeho voláním `CreateProcess` s flagem `DEBUG_ONLY_THIS_PROCESS`
    - Debugger obdrží `CREATE_PROCESS_DEBUG_EVENT`
- Připojení k existujícímu procesu
    - `DebugActiveProcess(dwProcessId)`
    - Nutná příslušná oprávnění: uživatel může debugovat vlastní procesy, pro debugování cizích nutné oprávnění `SeDebugPrivilege`
    - Všechna vlákna pozastavena
    - Debuger obdrží `LOAD_DLL_DEBUG_EVENT` pro každé DLL
    - Debugger obdrží `CREATE_PROCESS_DEBUG_EVENT` z prvního vlákna debuggeeho

Debugger zpracovává debugovací události:

- `CREATE_PROCESS_DEBUG_EVENT:` v debugovaném procesu vytvořen nový proces (vyvoláno systémem tešně před tím, než je program spuštěn v user-mode)
- `CREATE_THREAD_DEBUG_EVENT:` v debugee vytvořeno nové vlákno
- `EXCEPTION_DEBUG_EVENT:` v debugee došlo k výjimce (přístup k nedostupné paměti, spuštění

instrukce breakpointu, dělní nulou, ...)

- `EXIT_PROCESS_DEBUG_EVENT:` debuggee končí
- `EXIT_THERAD_DEBUG_EVENT:` končí vlákno, které je součástí debuggee
- `LOAD_DLL_DEBUG_EVENT:` debugge nahraje do paměti nové DLL (počátek programu nebo funkce `LoadLibrary`)
- `UNLOAD_DLL_DEBUG_EVENT:` uvolnění DLL debuggeem (funkce `FreeLibrary`)
- `OUTPUT_DEBUG_STRING_EVENT:` debuggee použil funkci `OutputDebugString`
- `RIP_EVENT:` došlo k systémové chybě (interní chyba debuggeru, ne debuggeeho)

Trasování (Trace)

Sledování přesného toku řízení procesu

Trasování instrukcí: zaznamenávání prováděných instrukcí

Trasování funkcí: zaznamenávání prováděných volání

Komunikace debuggeru a debuggeeho

Práce s pamětí:

- `ReadProcessMemory` / `WriteProcessMemory` : Čtení/zápis paměti procesu, vkládání softwarových breakpointů
- Debugger vždy pracuje s obrazem procesu v paměti -- nemění spustitelný kód na disku

Práce s vlákny:

- `GetThreadContext` / `SetThreadContext`
- Struktura `CONTEXT` : aktuální stav všech registrů, závislá na architektuře procesoru
- Změna hodnot registrů, práce s EIP

Breakpointy

Softwarové breakpointy:

- **Softwarové přerušení:** událost vyvolaná SW, která informuje jádro OS o tom, že normální tok

instrukcí musí být změněn

- Ukazatel na nejvyšší funkci pro obsluhu výjimek uložen v první proměnné **Thread Information Blocku (TIB),** lze zjistit z FS:[0]
- **Výjimka** ⇒ procházení řetězce výjimek, nalezení vhodného handleru (neexistuje ⇒ standardní způsob OS)
- **Většina debuggerů:** breakpoint instrukcí `int3` (`0xCC`)
- **Vytvoření breakpointu:**
  - Přečtení 1 bytu z adresy `Breakpoint_Address` , zapamatování si ho
  - Přepis prvního bajtu instrukce hodnotou `0xCC`
  - Vyprázdnění instrukční cache
  - Pokračování v debugování
- **Zpracování interního breakpointu debuggeru:**
  - Načtení struktury `CONTEXT` vlákna
  - Snížení EIP o 1 (návrat o 1 byte zpět)
  - Nastavení nového kontextu
  - Obnovení původní instrukce (odstranění breakpointu z paměti)
  - Pokračování v debugování
- **Zpracování uživatelského breakpointu:**
  - Načtení struktury `CONTEXT` vlákna
  - Snížení EIP o 1 (návrat o 1 byte zpět)
  - Nastavení příznaku Trap v EFL na single-step výjimku
  - Nastavení nového kontextu pomocí `SetThreadContext`
  - Obnovení původní instrukce
  - Pokračování v debugování
  - Až dojde k `STATUS_BREAKPOINT` , obnovení breakpointu
- **Vlastnosti:**
  - Výchozí ve většině debuggerů
  - Neomezené množství breakpointů v programu
  - Detekce spuštění instrukce
  - Změna obsahu paměti -- lze detekovat

Hardwarové breakpointy:

- Intel x86 -- 6 debugovacích registrů
  - `DR0` - `DR3` : každý jedna lineární adresa HW breakpointu
  - `DR6` : sdělení aplikaci, jaká debug situace nastala
  - `DR7` : příznaky -- lokálně povolený HW brakpoint, globálně povolený HW breakpoint, přerušení při spuštění, přerušení při zápisu, přerušení při přístupu (zápisu nebo čtení),

velikosti sledovaného paměťového místa (1,2,4 nebo 8 B)

- **Vlastnosti:**
  - Omezený počet
  - Detekce spuštění instrukce nebo přístupu do paměti
  - Nemění obsah paměti -- obtížnější detekce
  - Čtení a zápis debugovacích registrů -- privilegovaná operace
  - Podporován většinou debuggerů

Debugování jádra: nutný 2. počítač, který řídí debugování

Antidebugging

Obrana aplikace proti debugování. Snaha detekovat připojený debugger, ukončit debugger nebo z něj uniknout

Detekce debuggeru:

- **API funkce Windows:**
  - `IsDebugerPresent` : vrací pole `BeingDebugged` ze struktury Process Environment Block
    - lze provést i manuálně -- těžké na detekci
    - anti-antidebugging -- vynulování `BeingDebugged`
  - `CheckRemoteDebuggerPresent` (totéž co `IsDebuggerPresent` , ale lze použít i na jiný proces)
  - `NtQueryInformationProcess`
  - `OutputDebugString` , následně volání `GetLastError`
- **Příznak `ProcessHeap`**
  - nedokumentovaný
  - Struktura v PEB -- na offsetu `0x10` pole `ForceFlags` , pokud halda vytvořena debuggerem, je pole nenulové
  - anti-antidebug: přepis hodnoty `ForceFlags` nebo spuštění debuggeru s vypnutou haldou
- **Příznak `NTGlobalFlag`**
  - nezdokumentovaný
  - proces vytvořen debuggerem $\Rightarrow$ nastaveny bity `FLG_HEAP_ENABLE_TAIL_CHECK` , `FLG_HEAP_ENABLE_FREE_CHECK` , `FLG_HEAP_VALIDATE_PARAMETERS`
- **Skenování procesů:**
  - Získání seznamu procesů, hledání jména debuggeru v něm
- **Časování:**
  - Ověření, zda funkce prováděny v normálním čase (milisekundy), nebo zda jsou zdržovány

    - Odhalení krokování, pozastavování
    - Anti-antidebug: zaháčkování časovacích funkcí
- **Hledání instrukcí `int3` ve vlastním kódu:** kontrolní součty sama sebe

Vměšování se do debugování:

- Výjimky:
    - Vytvoření stovek výjimek ⇒ nezvýší výrazně dobu běhu, ale otráví analytika ⇒ vypnnutí upozornění na výjimky, v některém z handlerů je skryt důležitý kód
    - Instrukce `int3` přímo v programu
        - zmatení debuggeru -- bez něj se vykoná následující instrukce, s ním výjimka `STATUS_BREAKPOINT`
        - odlišný tok kódu pro debugouvanou aplikaci
    - Využití zranitelností debuggerů, jejich shození

Únik z debuggeru:

- Pouze jeden debugger připojen v jeden okamžik k jednomu procesu -- program sám sebe připojí jako debugger
- Injekce důležitého kódu do jiného procesu
- Uložení důležitého kódu do handleru výjimky
- Uložení důležitého kódu za/před `main`