DevWarsztaty - Plan

Przygotowania

- ściągamy:
  - dumps: http://bit.ly/24nqdXL
  - tools: http://bit.ly/25Fkq5z
  - sources: http://bit.ly/1r64syu oraz http://bit.ly/1ZkNvw9
- rozpakowujemy do `c:\Program Files (x86)\Windows Kits\8.1\Debuggers\`

Przystanek 1 - .NET Memory Internals

Wprowadzenie do WinDbg, pierwsze użycia sos i sosex. Obejrzenie struktury pamięci w .NET.

Kroki:

- ustawienie zmiennej środowiskowej _NT_SYMBOL_PATH - z panelu sterowania, na `symsrv*symsrv.dll*c:\localsymbols*http://msdl.microsoft.com/download/symbols`
- odpalenie WinDbg (wspomnieć o wersji x86 vs amd64/x64)
- ładujemy **Dump1**
- podstawowe zabawy z WinDbg:
  - metakomendy: `.chain`, `.cls`, `.hh`, `?128/4`
  - `vertarget` - czas zrobienia dumpa
  - `lmf` - załadowane moduły - m.in. czy .net 2.0/4.0/32/64bit
  - `?` - pomoc
  - `~` - wątki, `~1s` - przełączenie na 1-wszy itd.
  - `k` - stack trace, zaczną się sciągać symbole `clr.pdb` - **zajrzeć do c:\localsymbols**
  - `~* k` - wszystkie stack trace
- clr/mscordacwks oraz sos i sosex [4]
  - SoS (Son Of Strike) - część code base .NET, doskonale rozumiejąca wszystkie jego struktury, dzięki czemu może je interpretować. Dostępna w każdej wersji .NET: np `c:\Windows\Microsoft.NET\Framework64\v4.0.30319` itd.
  - `.loadby sos clr`
  - wątki `!threads`, `~0s`, `!clrstack`
  - `!eeversion`, `!eeheap -gc`, `!heapstat`
  - `!dumpheap -stat` - zwróćmy uwagę na StackOverflowException, OutOfMemoryException
  - `!dumpheap -type System.String`, `!do 000000e400002068`, `!gcwhere 000000e400002068`
  - `!dumpheap -mt 00007ffc86b0b7b8` - żeby pokazać tylko konkretny typ (String)
  - `!help`

- `.load sosex`
- `!bhi` - może potrwać na dużych plikach, `!sosex.help`
- `!dumpgen 3 -stat`, `!dumpgen 3` - tutaj w LOH internalsowe elementy CLR sobie wyklikujemy - tej największej tablicy

Przystanek 2 - Object Memory Model

Kilka kroków dalej - wykonanie własnego memory dumpa za pomocą ProcDumpa procesu SimpleConsoleApplication. Wspomnienie o różnicy w rodzajach dumpów [1]. Pokazanie VMMap - szerszy kontekst pamięci procesu niż CLR Heap. Obejrzenie struktur obiektów wartościowych i referencyjnych.

Kroki:

- skompilowanie SimpleConsoleApplication, uruchomienie
- uruchomienie VMMap - pokazanie różnych obszarów pamięci procesu
- `procdump -ma <PID>` - wspomnieć o różnych rodzajach dumpów [1]
- uruchamiamy WinDbg, ładujemy powstałego dumpa, ładujemy `sos` i `sosex`
- szukamy typu - `!dumpheap -type SomeClass`
- `!dumpobj <address>` (`!do`) - dostajemy coś w stylu:

```
Name:        SimpleConsoleApplication.SomeClass
MethodTable: 00007ffcffe95bf0
EEClass:     00007ffcfffe11a0
Size:        56(0x38) bytes
File:        C:\DevWarsztaty2\Programs-Sources\MemoryLeakGUI\MemoryLeakGUI\SimpleC
Fields:
              MT    Field   Offset                 Type VT     Attr            Val
00007ffd5e784b08  4000001        8        System.String  0 instance 00000048bf0745
00007ffcffe95a60  4000002       18 ...pplication.Vector  1 instance 00000048bf0746
00007ffcffe95cb8  4000003       10 ...cation.OtherClass  0 instance 00000048bf0746
```

- pola oznaczone VT to są value-types, sięgamy poprzez: `!dumpvc 00007ffcffe95a60 00000048bf074660`
- `.prefer_dml 1` - doskonały moment by o nim wspomnieć
- rozmiar `!objsize` vs rozmiar samego obiektu
- "szukamy" Empty class, **jak?** `!dumpheap -type EmptyClass`, `!do <address>` - size 24B!
- `db <address> L24` - błąd! to hex! Poprawnie: `db <address> L0n24` - tutaj widać MT

Przystanek 3 - Fragmentacja LOH

Pokazanie problemu fragmentacji LOH [2] na przykładzie procesu .NET 4.5 oraz .NET 4.0 (32bit).

- WinDbg - kogo zirytowało ciągłe ustawianie czcionki? *"ustawienie fonta "Open WinDbg. View-> Options. Change Workspace Prompts to Always Ask. File -> Delete Workspaces. Close WinDbg. You will be asked "Save Information for workspace". Check the "Don't ask again in this session", and choose Yes."*

- ładujemy **Dump4-Net40-32bit** (na podstawie MemoryLeakGUI\MemoryLeakGUI) - pokazanie problemu z brakującym clr.dll:

    - `.loadby sos clr` pokazuje: *The call to LoadLibrary(C:\Windows\Microsoft.NET\Framework\v4.0.30319\sos) failed, Win32 error 0n193. "%1 nie jest prawidłową aplikacją systemu Win32."*
    - `lmf` - w ścieżkach nie widzimy *64*
    - ładujemy w WinDbg x86,
    - `.loadby sos clr`, sprawdzamy `!eeversion`
    - Infamous bug: *Failed to load data access DLL, 0x80004005*, ładujemy sos podając ścieżkę do wersji leżącej obok dumpa
    - `.cordll -ve -u -l` - `mscordacwks` niby się ładuje ale i tak nie działa
    - `!heapstat` - ~1.2 GB z czego połowa wolna (LOH: 54%)
    - `.load sosex` - upewniamy się, że rozpakowaliśmy do wersji x86
    - `!dumpgen 3 -stat` - prawdopodobnie się wywali, `.unload sosex`
    - `!eeheap -gc` - pobieramy dane jednego z segmentów (begin i allocated)
    - `!dumpheap -stat 27e31000 29623050`
    - `!heaptraverse plik.log` - potem otwieramy w CLRProfiler

- ładujemy **Dump4-Net451-64bit**

    - `.loadby sos clr`
    - `!heapstat` - też około ~0.6GB danych, ale fragmentacja tylko 5%

Przystanek 4 - memory hog (kontatenacja)

Pokazanie problemu memory hog - dużego narzutu CPU, przy właściwie braku wycieku. Wynik load testu. Bardzo duzy narzut, bo wiele obiektow trafia do LOH lub Gen2.

- Dump2 - przy zmiennej Count od 100 do 500 (bardzo dużo trafia do LOH, ogromny narzut CPU)
- Dump2 - omówienie zebranych Performance Counterów
    - *Czas procesora (%)* - ogromny więc mamy problem
    - *Rozmiar sterty pokolenia 0/1/2 i obiektów wielkich* - ze skalą 0.000001 (pięc zer), max 200, teoretycznie nic strasznego się nie dzieje. LOH taki poszarpany ale ~17MB, czy to problem?
    - *Liczba operacji gromadzenia obiektów...* - zdecydowanie złamana złota zasada - praktycznie każdy GC obejmuje wszystkie generacje
- Dump2 - pokazujemy VMMap

- WinDbg - ładujemy **Dump2**, `.loadby sos clr`
- `!heapstat` oraz `!dumpgen 3 -stat` - widać fragmentację LOH
- `.load netext` - i jedziemy z `!windex -tree`
- pokazujemy: `!whttp`, klikamy przykładowy HTTP Context, `!wconfig`,
- `!dumpheap -type LeakWebApi.Controllers.ValuesController -stat` - jest 10, klikamy MT, patrzymy w dół klikając `_controllerContext._request.requestUri.m_String`
- `!whelp wfrom` - wyczes
- `!wfrom -type *.ValuesController select _controllerContext._request.requestUri.m_String`
- `!wfrom -nofield -type *.ValuesController select _controllerContext._request.requestUri.m_String`
- Dump3 - Count od 10 do 200. Bardzo duzy narzut, bo wiele obiektow trafia do Gen 2, nic nie trafia do LOH. Nie jest to oczywisty dump...
- Dump3 - omówienie zebranych Performance Counterów - te same jak powyżej, ale tu nie widać złamania złotej zasady, nadal jednak duży % CPU in GC
- WinDbg - ładujemy **dump3**
  - `.loadby sos clr`, `.load sosex`, `.load netext`
  - `!heapstat` - nie widać fragmentacji specjalnie, `!wheap -detailsonly` - tu też
  - `!dumpgen 2 -stat` - tu też nie widać, ale sporo gen2 to StringBuilder
  - `!dumpgen 1 -type System.String`, albo 0 - tutaj po zawartości stringów

Przystanek 5 - Pinned

Pokazujemy na szybko jak znaleźć obiekty typu pinned. Przykład z życia wzięty: https://ayende.com/blog/170243/long-running-async-and-memory-fragmentation

- ładujemy **Dump1** lub zostawiamy poprzedni
- `.loadby sos clr`
- `!gchandles`
- `.shell -ci "!gchandles" findstr "Pinned"` albo `!gchandles -type Pinned`
- `!gcwhere <address>` - będą z LOH (internalsy)

Przystanek 6 - Finalizers i Windsor.Castle

Demonstracja live ciekawego wycieku - problem z konfiguracją Windsor.Castle [3].

- podpięcie aplikacji SampleWebApi do IIS pod DefaultWebSite ...
- powinien zwracać xmla pod adresem: http://localhost/SampleWebApi/api/Values
- zapięcie PC na procesie w3wp

- uruchomienie load testu za pomocą sb: `run.bat`, treść `sb -u http://localhost/SampleWebApi/api/Values -c 10 -n 100 -y 100`
- obserwujemy, np. w Task Manager użycie pamięci, jak się zachowują generacje
- `procdump -ma <PID>` - po około ~10000
- otwieramy dumpa
- patrzymy na `!fq`, `!finq <0|1|2> -stat`, `!frq` - widzimy, że strasznie dużo `SampleService`
- `!name2ee * SampleWebApi.Services.SampleService` - bierzemy MT
- potem `!refs -target` na jednym z serwisów, albo `!gcroot`
- track: `var track = WebApiApplication.container.Kernel.ReleasePolicy.HasTrack(this.service);`
- poprawka:

  ```
  request.RegisterForDispose(new Release(() => _container.Release(controller)));
  ```

  przed `returnem` w `WindsorCompositionRoot.Create`

- [1] https://msdn.microsoft.com/en-us/library/windows/hardware/ff560251(v=vs.85).aspx
- [2] https://www.simple-talk.com/dotnet/.net-framework/the-dangers-of-the-large-object-heap/
- [3] https://beniaminzaborski.wordpress.com/2014/12/19/castle-windsor-transient-memory-leak/
- [4] https://www.microsoftpressstore.com/articles/article.aspx?p=2201303&seqNum=3