Когда у нас не воспроизводится

Александр Головач

О себе

Александр Головач

Системный программист в
Checkpoint Software Technologies Ltd.

Check Point
SOFTWARE TECHNOLOGIES LTD.
We Secure the Internet.

Почему отладка сложна

На практике встречаются ситуации, когда:

- Воспроизведение дефекта непредсказуемо
- Дефект проявляется в специфичном окружении, доступ к которому может быть ограничен
- Для воспроизведения дефекта необходимо время
- Для воспроизведения дефекта необходима определенная последовательность событий
- Процесс отладки ведет к нарушению работы ПО, что порой крайне не желательно

Прежде чем мы начнем

Адресное пространство процесса

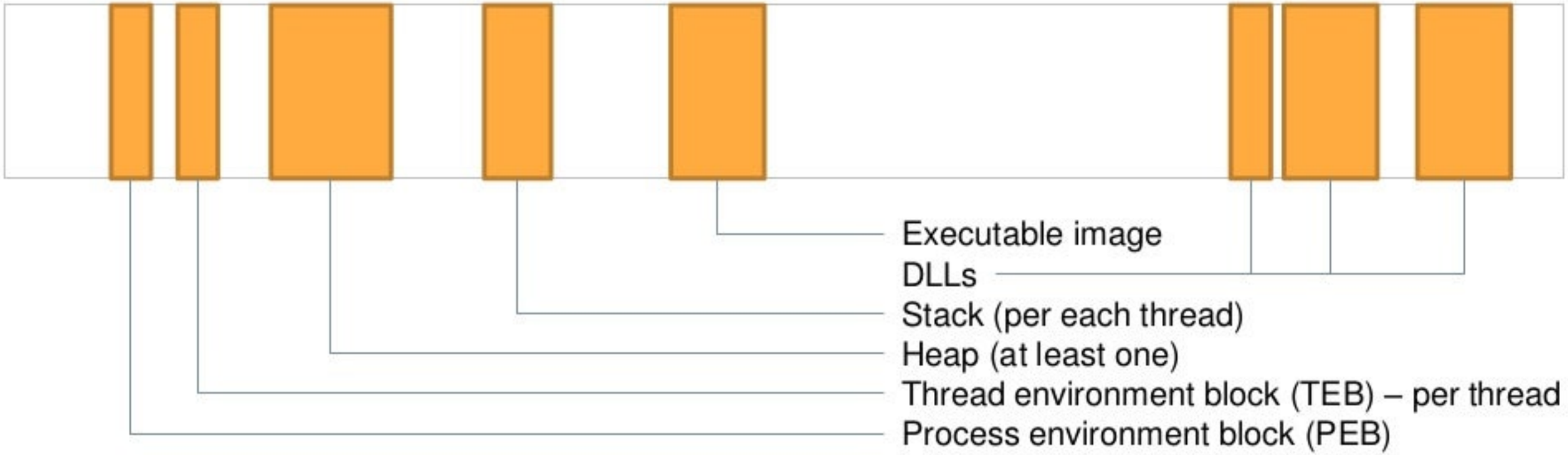
Пространство пользователя

(у каждого процесса свое изолированное)

Пространство ядра

(общее для всех процессов, недоступно из пользовательского кода)

Адресное пространство процесса - UM



Дамп памяти процесса

Дамп памяти процесса

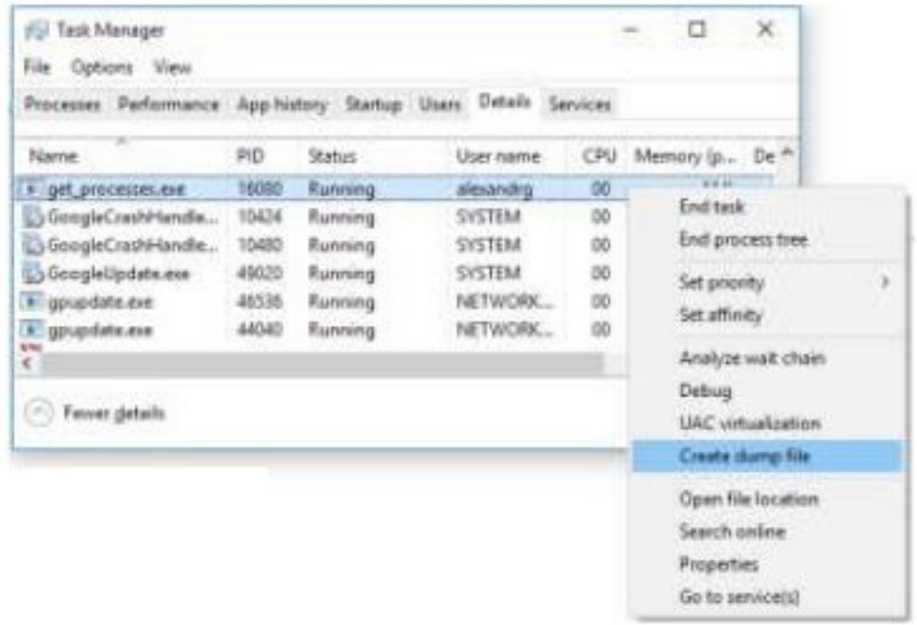
Где в дампе памяти находится . . . :

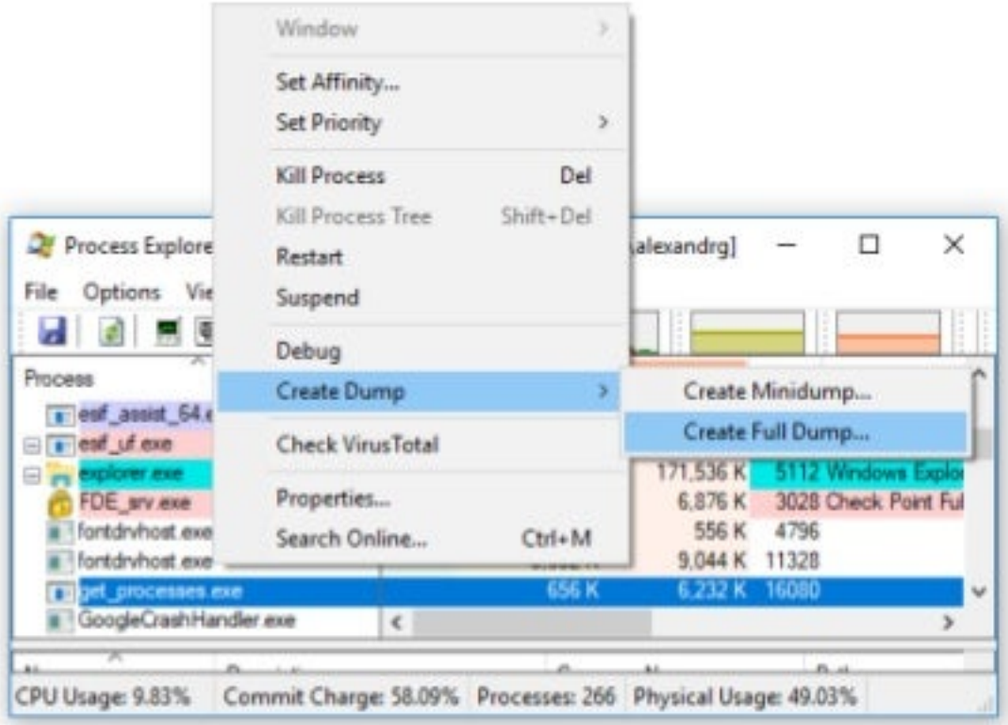
- Информация о модулях, командной строке, переменных окружения - PEB
- Глобальные и статические переменные – секция данных модуля
- Локальные переменные – стек
- Динамические данные – куча. Адреса ищи в глобальных или локальных переменных (см. выше)
- Какой именно код выполнялся в потоке – регистр EIP
- Последовательность вызовов функций – адреса возврата на стеке

.DMP

Как собрать дамп ?

- Используйте Procdump !
- Task manager
- Process Explorer
- Отладчик





Отладочные символы

- Отладочные символы (debug symbols) – информация о взаимном соответствии исходного кода и исполняемого модуля.
- Генерируются компоновщиком.
- Различают публичные и приватные отладочные символы

  публичные символы не содержат информацию о типах, номерах строк, локальных переменных и параметрах функций
- Отладочные символы для Microsoft доступны здесь: https://msdl.microsoft.com/download/symbols

Утечка дескрипторов на объекты ядра

Утечка дескрипторов на объекты ядра

- Дескриптор (HANDLE) – идентификатор объекта ядра
- Незакрытый дескриптор приводит к тому, что количество ссылок на объект ядра не станет равным нулю.
- Теряя дескрипторы на объекты ядра, процесс влияет на другие процессы и систему в целом

Примеры объектов ядра:
Process, Thread, File, GDI objects (windows, brushes, palettes), ……

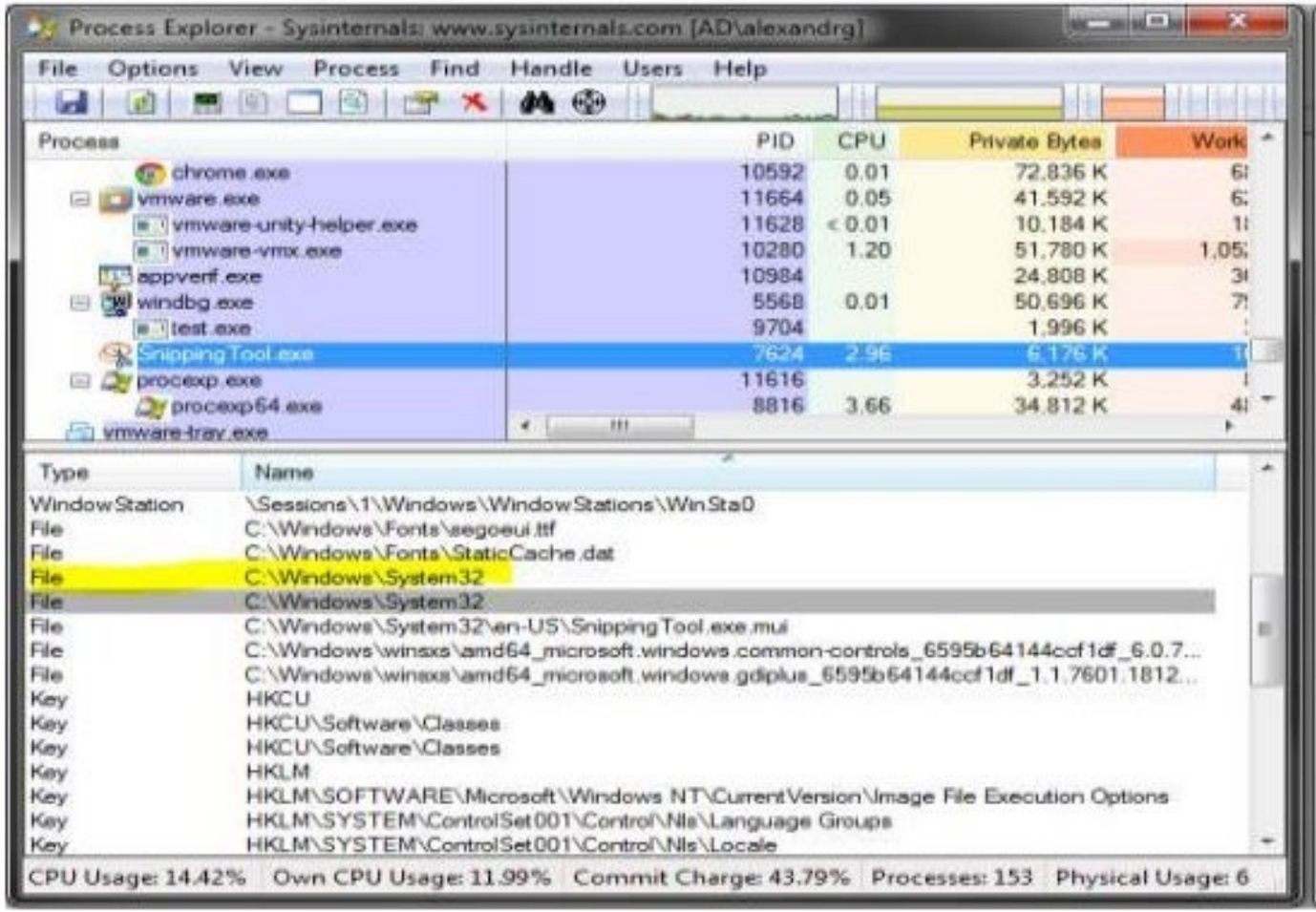
Предварительный анализ

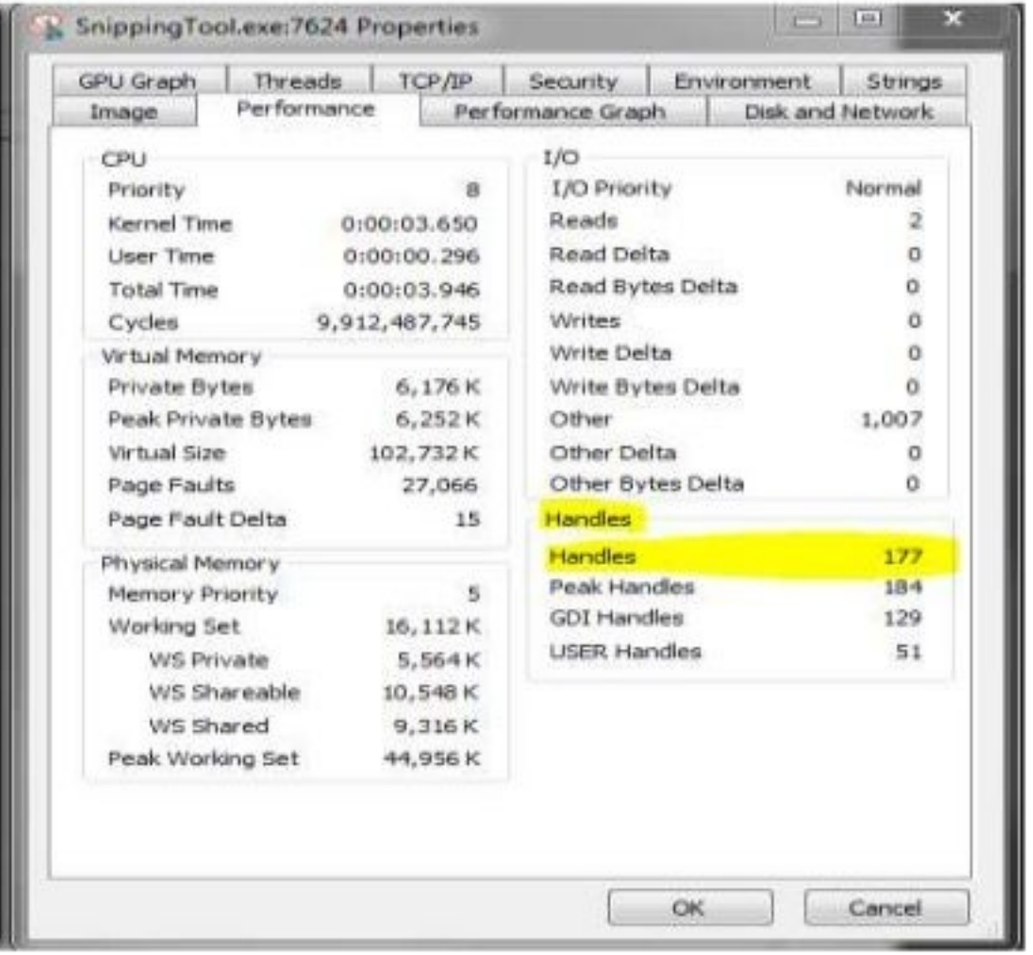
- Начальный анализ рекомендуется провести с использованием утилит
  - Handle
  - Process monitor
  - Process explorer

- Process explorer может показать дескрипторы процесса, их тип и имена соответсвующих объектов ядра (если применимо)

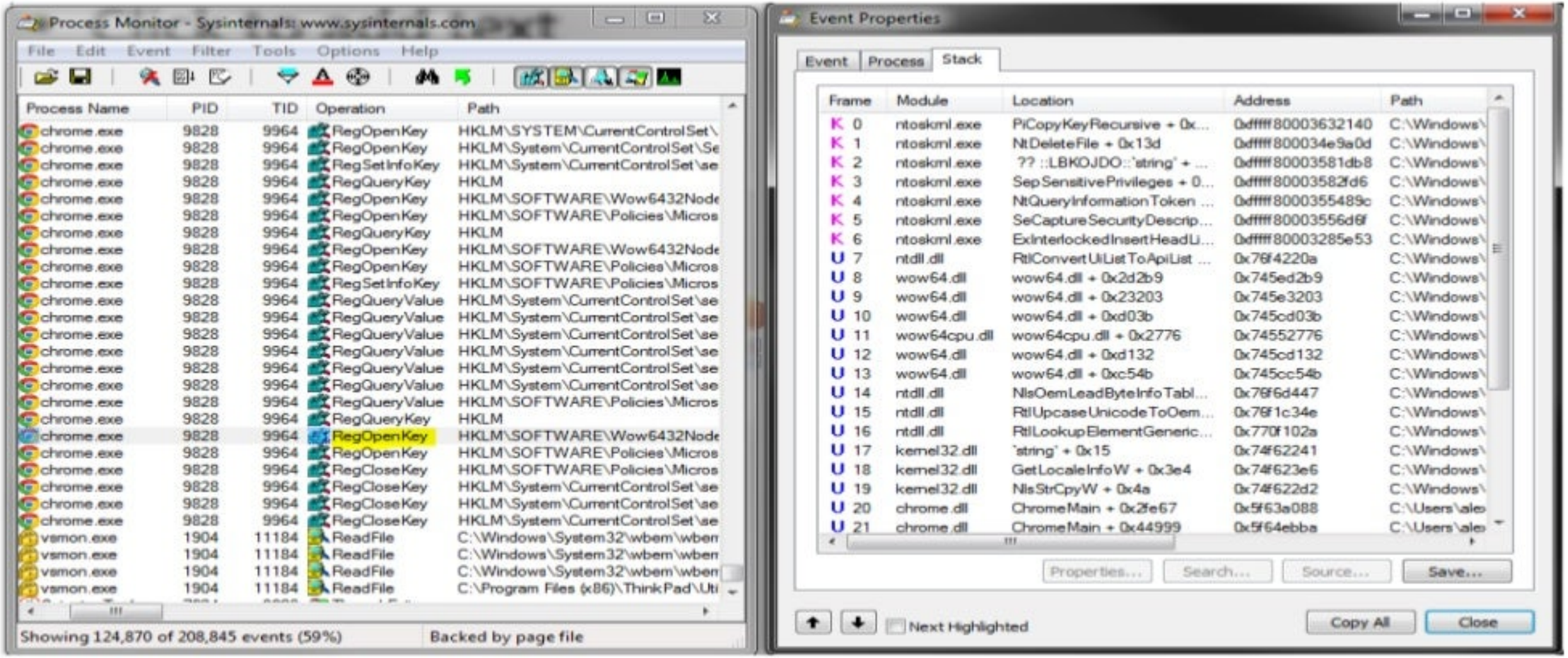
- Process monitor показывает активность процесса. Открытие ключей реестра и файлов журналируется, для операций доступен стек вызова.

Process Explorer

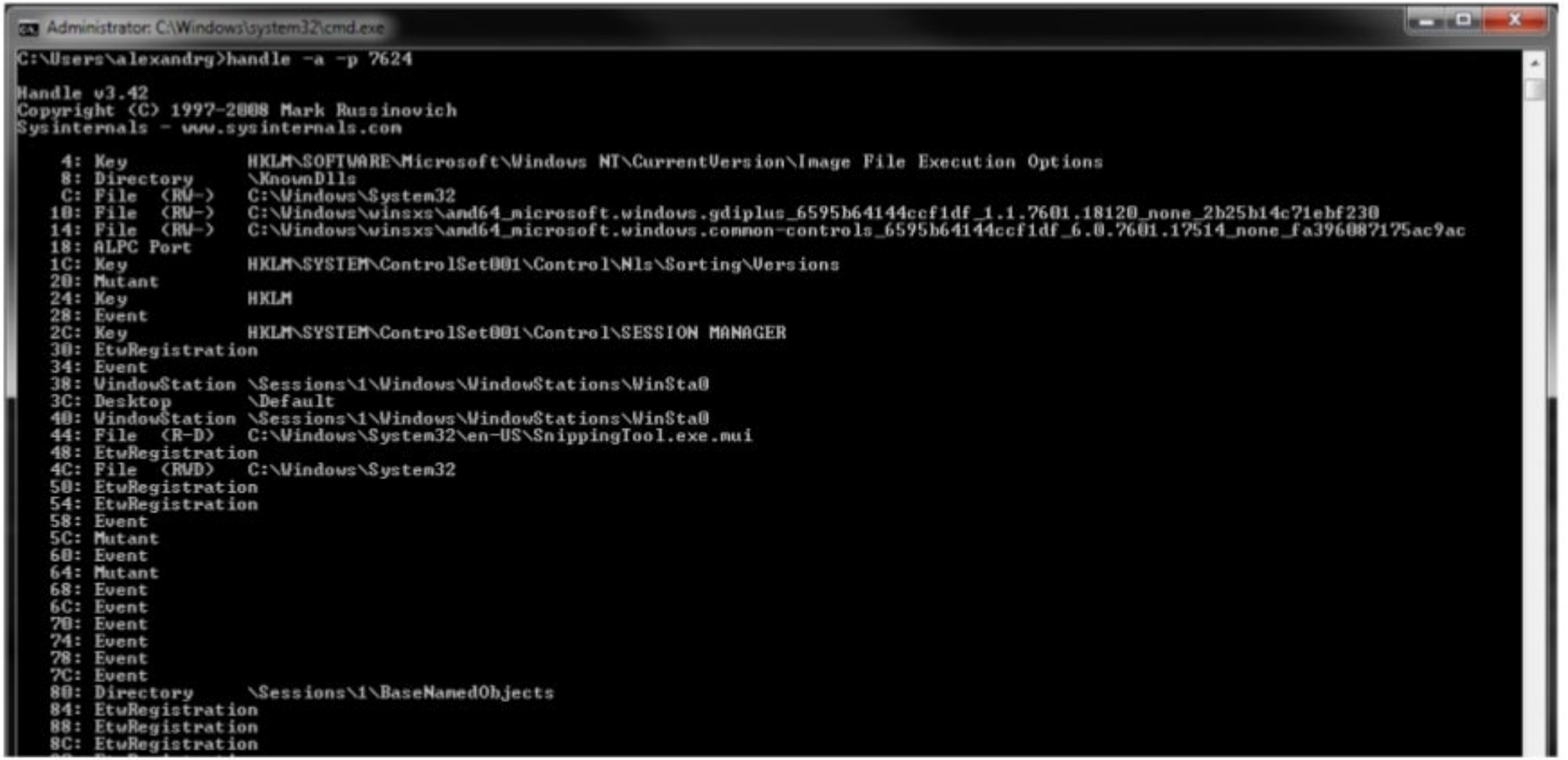




Process Monitor



HANDLE


Administrator: C:\Windows\system32\cmd.exe
C:\Users\alexandrg>handle -a -p 7624
Handle v3.42
Copyright (C) 1997-2008 Mark Russinovich
Sysinternals - www.sysinternals.com
4: Key HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
8: Directory \KnownDlls
C: File (RW-) C:\Windows\System32
10: File (RW-) C:\Windows\winsxs\amd64_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.7601.18120_none_2b25b14c71ebf230
14: File (RW-) C:\Windows\winsxs\amd64_microsoft.windows.common-controls_6595b64144ccf1df_6.0.7601.17514_none_fa396087175ac9ac
18: ALPC Port
1C: Key HKLM\SYSTEM\ControlSet001\Control\Nls\Sorting\Versions
20: Mutant
24: Key HKLM
28: Event
2C: Key HKLM\SYSTEM\ControlSet001\Control\SESSION MANAGER
30: EtwRegistration
34: Event
38: WindowStation \Sessions\1\Windows\WindowStations\WinSta0
3C: Desktop \Default
40: WindowStation \Sessions\1\Windows\WindowStations\WinSta0
44: File (R-D) C:\Windows\System32\en-US\SnippingTool.exe.mui
48: EtwRegistration
4C: File (RWD) C:\Windows\System32
50: EtwRegistration
54: EtwRegistration
58: Event
5C: Mutant
60: Event
64: Mutant
68: Event
6C: Event
70: Event
74: Event
78: Event
7C: Event
80: Directory \Sessions\1\BaseNamedObjects
84: EtwRegistration
88: EtwRegistration
8C: EtwRegistration

Application verifier. Рецепт

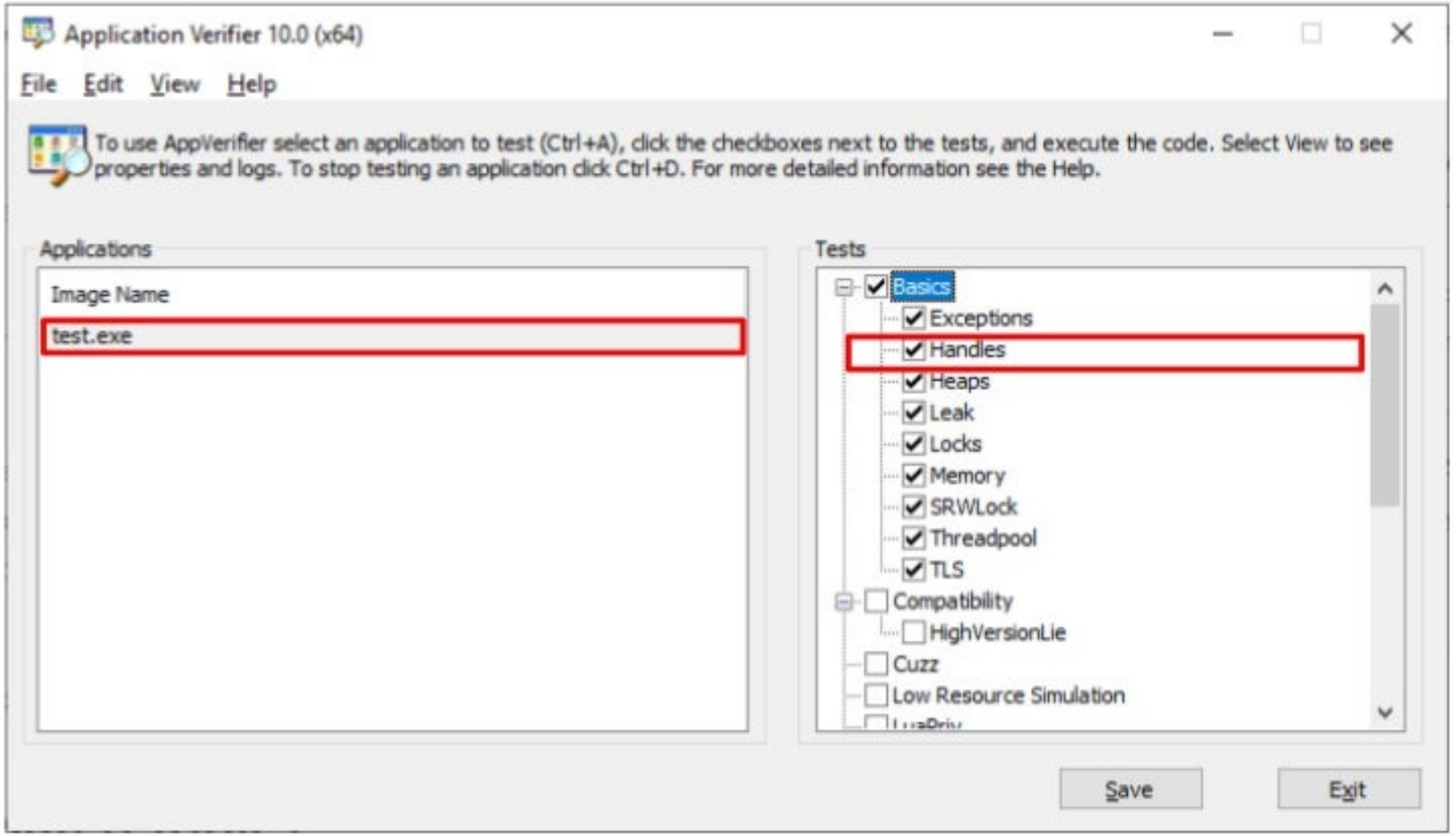
На машине с воспроизведением:

- Добавить свое приложение в список верифицируемых
- В списке Tests указать Basic->Handles
- Запустить приложение, воспроизвести ошибку
- Собрать дамп

На машине разработчика:

- Открыть дамп памяти
- Использовать команды !htrace, !handle в отладчике WinDbg

Application verifier



Windbg. Рецепт

- Запустить программу под отладчиком
- Поставить точку останова в контрольной точке [1]
- После остановки программы на точке останова выполнить команду **!htrace -enable**, продолжить выполнение
- Поставить точку останова в контрольной точке [2]
- После остановки программы на точке останова выполнить команду **!htrace –diff.**

При отладке в среде заказчика собрать дамп процесса (.dump /ma file.dmp) и выполнить команду !htrace во время анализа на стороне разработчика.

Пример

```
4   #include "stdafx.h"
5   #include <Windows.h>
6
7
8   void leak_handle()
9   {
10      HKEY key;
11      LSTATUS Status = RegOpenKey( HKE
12  }
13
14  int main()
15  {
16      for ( int i = 0; i < 20; i++ )
17          leak_handle();
18      system( "pause" );
19      return 0;
20  }
```

```
Handle = 0x00000000000000b0 - OPEN
Thread ID = 0x0000000000004e20, Process ID = 0x0000000000004678

0x00007ffd85fa7594: ntdll!NtOpenKeyEx+0x0000000000000014
0x00007ffd8298ac3c: KERNELBASE!LocalBaseRegOpenKey+0x00000000000001bc
0x00007ffd8298a3d4: KERNELBASE!RegOpenKeyExInternalW+0x0000000000000144
0x00007ffd8298a029: KERNELBASE!RegOpenKeyExW+0x0000000000000019
0x00007ffd78c862fb: vfbasics!AVrfpRegOpenKeyExW+0x00000000000000cb
0x00007ffd856053d9: advapi32!RegOpenKeyW+0x0000000000000029
0x00007ffd78c86028: vfbasics!AVrfpRegOpenKeyW+0x00000000000000a8
0x00007ff6229f1039: handle_leak!main+0x0000000000000039
0x00007ff6229f12a9: handle_leak!__scrt_common_main_seh+0x000000000000011d
0x00007ffd83be2784: kernel32!BaseThreadInitThunk+0x0000000000000014
0x00007ffd85f50c31: ntdll!RtlUserThreadStart+0x0000000000000021
0:004> !handle 0x00000000000000b0 f
Handle 00000000000000b0
  Type          Key
  Attributes    0
  GrantedAccess 0x20019:
        ReadControl
        QueryValue,EnumSubKey,Notify
  HandleCount   2
  PointerCount  32770
  Name          \REGISTRY\MACHINE\SOFTWARE
  No object specific information available
```

Подведем итог

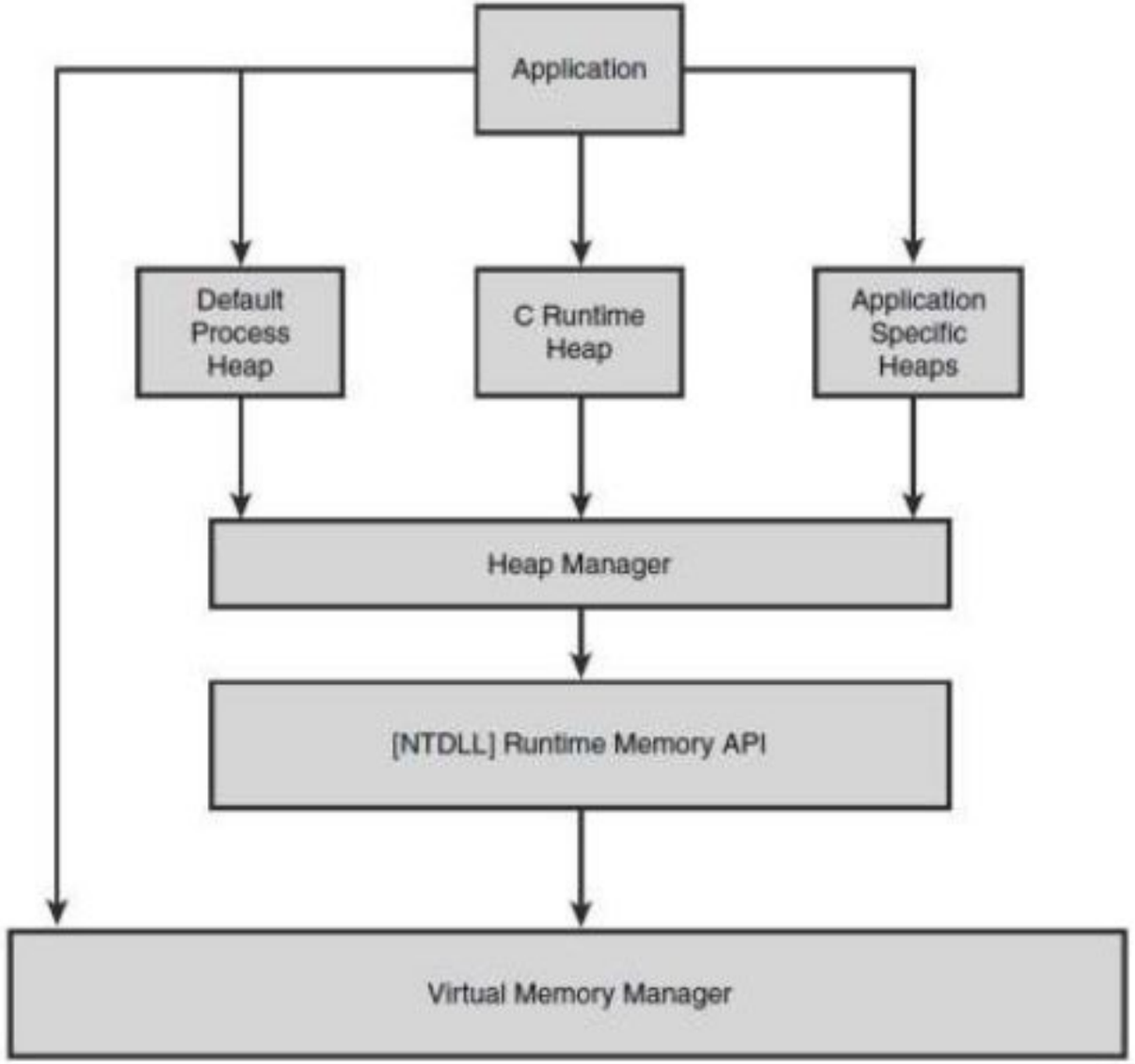
- Предварительный анализ выполняем утилитами Sysinternals
- Система может собирать стеки вызовов (AppVerifier). Требуется перезапуск процесса
- Если есть возможность подключить отладчик – пользуемся! Проще, перезапуск приложение не требуется

Утечка памяти

Утечка памяти

- Основным признаком утечки является постоянное увеличение объема памяти, используемой приложением.
- Исследование данной категории проблем предполагает отслеживание выделения и освобождение памяти, сравнение состояния памяти в разные моменты времени (так называемых memory snapshots)
- Два основных подхода для получения информации о работе с динамической памятью: перехват системных вызовов и использование функционала системы по сбору стеков вызовов (конфигурируется GFlags/AppVerifier/UMDH)
- В некоторых случаях не требуется предварительная конфигурация системы

Виды динамической памяти



Утилита UMDH

- Входит в состав Debugging Tools for Windows
- Работает на уровне Heap Manager’a
- Основная идея следующая:
  - Сделать снимок использования памяти перед воспроизведением (контрольная точка А)
  - Сделать снимок использования памяти после воспроизведения (контрольная точка Б)
  - Сравнить полученные снимки.

Утилита UMDH – перед использованием

- Перед использованием утилиты необходимо сконфигурировать пути поиска отладочных символов в переменной окружения _NT_SYMBOL_PATH (на стороне разработчика)

- Также необходимо сконфигурировать систему сохранять стеки вызовов при выделении памяти (Application Verifier или gflags).
  В случае отсутствия конфигурации утилита выполнит конфигурацию самостоятельно.

Утилита UMDH. Рецепт

На машине с воспроизведением:

- `UMDH -p:PID  > first_snapshot_filename`
- Воспроизвести утечку памяти
- `UMDH -p:PID  > second_snapshot_filename`

На машине разработчика:

- `UMDH first_snapshot_filename second_snapshot_filename > diff`
- Проанализировать файл diff

Подведем итог

- Поиск причины утечки памяти может быть сложным из-за обилия API / уровней работы с динамической памятью
- Рекомендуется использовать UMDH для быстрого исследования: не требует предварительной конфигурации и перезапуска процесса

Взаимоблокировка потоков

Взаимоблокировки

Выделим следующие ситуации:

- Взаимоблокировки при использовании CRITICAL_SECTION, в рамках одного процесса
- Взаимоблокировки при использовании объектов ядра, межпроцессные взаимоблокировки
- Взаимоблокировки с использованием примитивов синхронизации C++

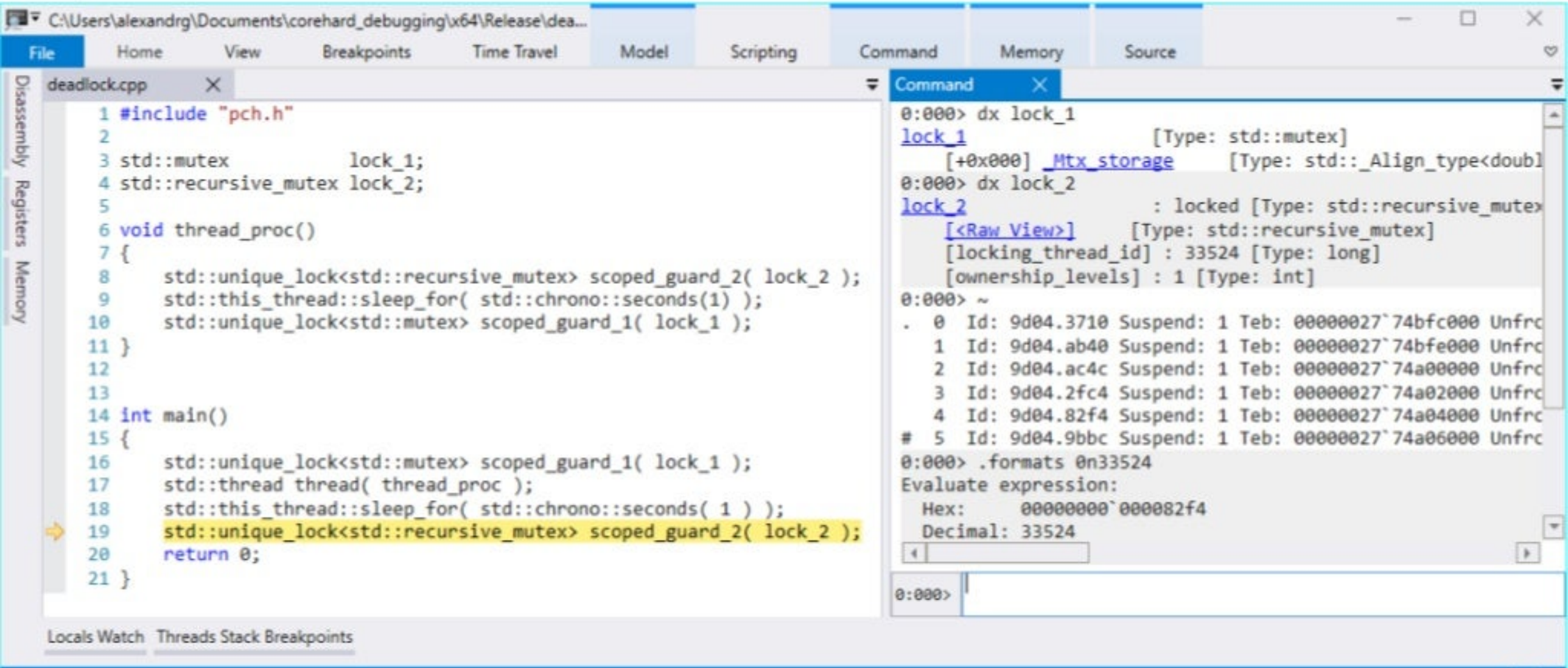
Взаимоблокировки

- В любом случае сделать дамп памяти процесса
- При анализе дампа распечатать все потоки. Выделить заблокированные.
- В случае с CRITICAL_SECTION использовать команды !cs, !locks. Данный примитив синхронизации содержит информацию о текущем владельце. Раскручивание таких цепочек наименее трудоемкое
- В случае использования объектов ядра дампа памяти процесса может быть недостаточно, необходимо знание логики приложения для анализа цепочки блокировки.
- При наличии межпроцессной блокировки могут понадобиться дампы всех задействованных процессов и полный дамп памяти системы.

std::mutex и std::recursive_mutex

- Объекты синхронизации c++ активно используют «родные» для Windows примитивы
- Иногда могут содержать информацию о потоке владельце (зависит от реализации)
- Из коробки, windbg может найти владельца для std::recursive_mutex (проверено для VS2017 + WinDbg Preview)
- Немного смекалки – и ту же информацию можно получить и о std::mutex. Подсказка в файле *%windbg_root%\visualizers\stl.natvis* и немного позже в докладе

std::*mutex


C:\Users\alexandrg\Documents\corehard_debugging\x64\Release\dea...
File
Home
View
Breakpoints
Time Travel
Model
Scripting
Command
Memory
Source
deadlock.cpp
1 #include "pch.h"
2
3 std::mutex lock_1;
4 std::recursive_mutex lock_2;
5
6 void thread_proc()
7 {
8 std::unique_lock<std::recursive_mutex> scoped_guard_2(lock_2);
9 std::this_thread::sleep_for(std::chrono::seconds(1));
10 std::unique_lock<std::mutex> scoped_guard_1(lock_1);
11 }
12
13
14 int main()
15 {
16 std::unique_lock<std::mutex> scoped_guard_1(lock_1);
17 std::thread thread(thread_proc);
18 std::this_thread::sleep_for(std::chrono::seconds(1));
19 std::unique_lock<std::recursive_mutex> scoped_guard_2(lock_2);
20 return 0;
21 }
Disassembly
Registers
Memory
Locals Watch
Threads Stack Breakpoints
Command
0:000> dx lock_1
lock_1 [Type: std::mutex]
[+0x000] _Mtx_storage [Type: std::_Align_type<doubl
0:000> dx lock_2
lock_2 : locked [Type: std::recursive_mute
[<Raw View>] [Type: std::recursive_mutex]
[locking_thread_id] : 33524 [Type: long]
[ownership_levels] : 1 [Type: int]
0:000> ~
. 0 Id: 9d04.3710 Suspend: 1 Teb: 00000027`74bfc000 Unfrc
1 Id: 9d04.ab40 Suspend: 1 Teb: 00000027`74bfe000 Unfrc
2 Id: 9d04.ac4c Suspend: 1 Teb: 00000027`74a00000 Unfrc
3 Id: 9d04.2fc4 Suspend: 1 Teb: 00000027`74a02000 Unfrc
4 Id: 9d04.82f4 Suspend: 1 Teb: 00000027`74a04000 Unfrc
5 Id: 9d04.9bbc Suspend: 1 Teb: 00000027`74a06000 Unfrc
0:000> .formats 0n33524
Evaluate expression:
Hex: 00000000`000082f4
Decimal: 33524
0:000>

Подведем итог

- При блокировке потоков получить дамп
- В большинстве случаев необходимо знание кода для восстановления цепочки ожидания
- Объекты синхронизации C++ могут содержать информации о владельце

Аварийное завершение

Кабы знал, где упасть, соломки бы подостлал
Народная поговорка

Аварийное завершение

- При возникновении в потоке исключительной ситуации система начинает раскрутку стека в поисках обработчика исключения (англ. exception handler).
- Если ни один из фильтров не обработает исключение, будет вызван фильтр необработанных исключений
- По умолчанию фильтр необработанных исключений запускает процесс werfault.exe, который создает отчет об ошибке.
- Процесс принудительно завершается

При аварийном завершении приложения во многих случаях достаточно дампа памяти, содержащего информацию о необработанном исключении.

Аварийное завершение

Получить дамп при аварийном завершении можно следующими способами:

- Сконфигурировать Windows error reporting
- Сконфигурировать Just in time debugger
- Реализовать свой фильтр необработанных исключений

Windows error reporting

Конфигурация создания дампов памяти при аварийном завершении находится: **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\ Windows Error Reporting\LocalDumps [\ImageName.exe]**

DumpFolder	Путь к каталогу, в котором будут сохраняться дампы	REG_EXPAND_SZ
DumpCount	Максимальное количество дампов.	REG_DWORD
DumpType	Тип дампа памяти: • 0: Custom dump • 1: Mini dump • 2: Full dump	REG_DWORD

Just in time debugger

Функция в ОС Windows позволяет автоматически запустить отладчик и подключить его к приложению при возникновении необработанного исключения

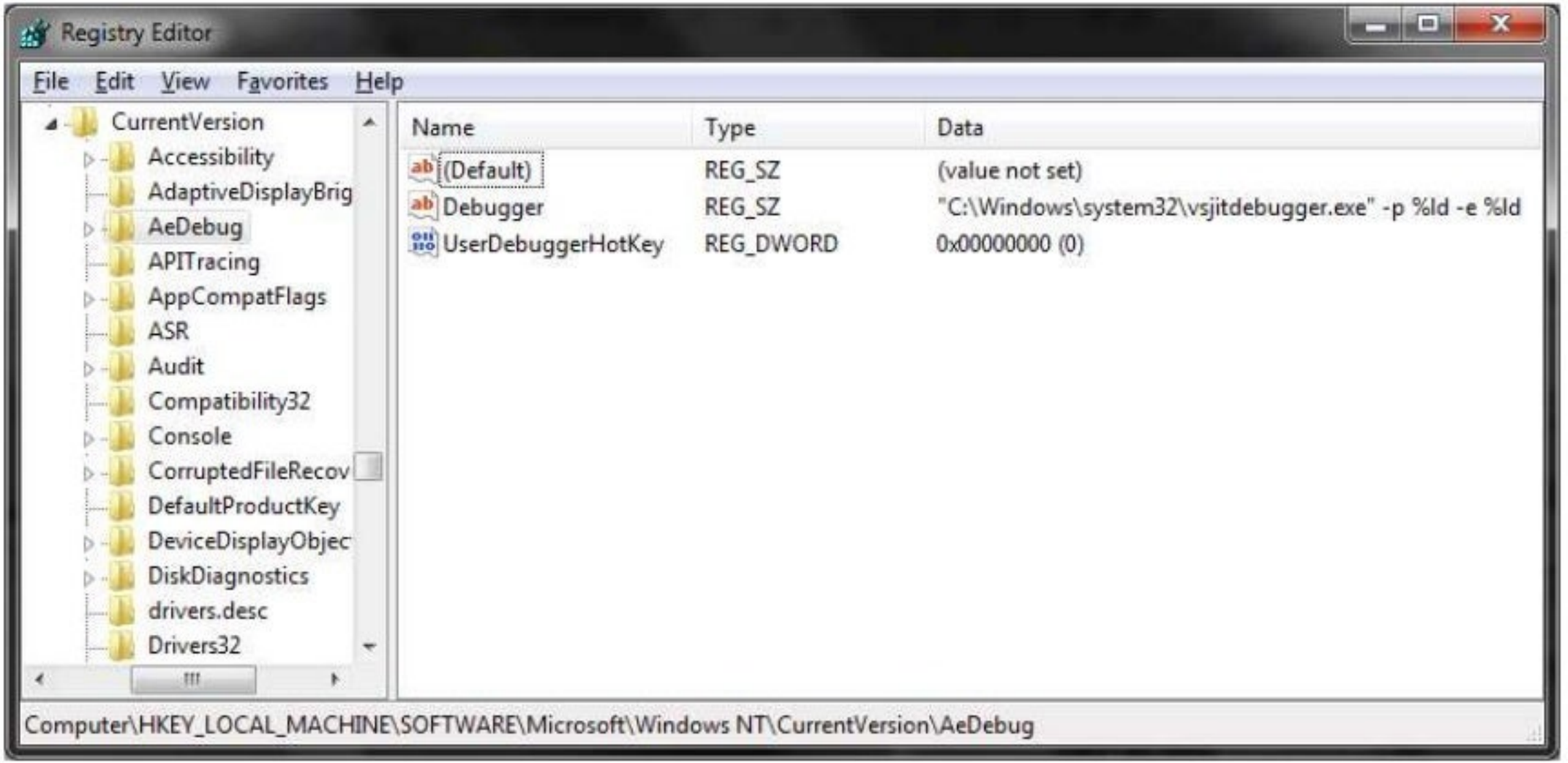
Отладчики имеют возможность зарегестрировать себя самостоятельно. (для windbg это параметр командной строки `-I`)

В реестре конфигурация сохраняется в

HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug

HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\AeDebug

Just in time debugger. Visual Studio



OK, у нас есть дамп...

Начинаем с **!analyze –v**. Данная команда проведет первичный анализ дампа. В случае с аварийным завершением постарается восстановить контекст процессора на момент возникновения исключительной ситуации.

Дальнейшие действия зависят от конкретной ситуации.

Memory corruption: возможные причины

- Нарушение соглашения о вызовах (поврежден стек)
- Неинициализированные переменные
- Переполнение буфера
- Использование памяти после освобождения
- Возврат из функции ссылки / указателя на локальную переменную
- «Забытые» указатели на переменные, которые могут измениться другими потоками (незавершенный аснхронный ввод-вывод + отсутствие вызова CancelIO) – также источник heap corruption

Stack corruption

- Многие нарушения обнаруживаются компилятором
- В дампе памяти повреждение стека может быть легко определено командой **k**.
- При выходе из функции содержимое стека не стирается, меняется только значение регистра ESP. Как следствие, можно попытаться восстановить стек

  !teb – показать thread environment block. Это даст нам границы стека

  dps – print pointer resolve symbol.

  k – распечатать стек. Поддерживает переопределение ESP/EBP

  d* - распечатать память

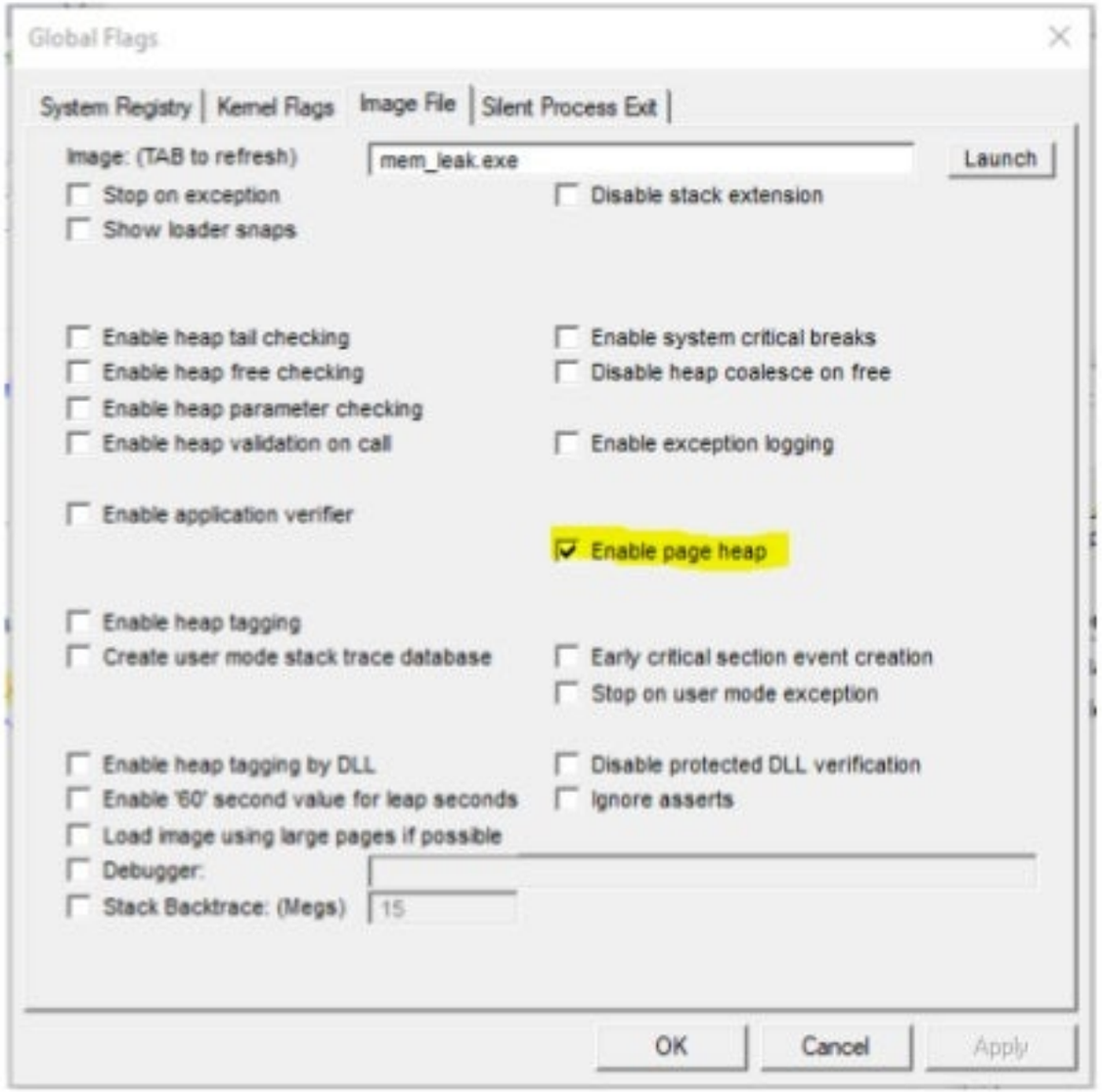
Heap corruption

Пожалуй, самый тяжелый для отладки класс проблем

При относительно стабильном воспроизведении рекомендуется использовать **application verifier** / **gflags** для конфигурирования отладочной кучи

Использование отладочной кучи несколько влияет на производительность приложения, однако позволяет получить намного больше информации о блоках динамической памяти, включая стек вызовов. Также предполагается заполнение блоков памяти специальными паттернами, упрощающую идентификацию их начала и конца

Конфигурирование отладочной кучи (gflags)



Стек вызовов при выделении блока (дамп)


```
0:000> !heap -p -a 000001ad0a9feff0
    address 000001ad0a9feff0 found in
    _DPH_HEAP_ROOT @ 1ad0a7b1000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -         VirtAddr
                            1ad0a7b8ea0:     1ad0a9feff0                c -      1ad0a9fe000
    00007ffd85fc578b ntdll!RtlDebugAllocateHeap+0x0000000000034b23
    00007ffd85fba5c2 ntdll!RtlpAllocateHeap+0x000000000008d9c2
    00007ffd85f2a9ca ntdll!RtlpAllocateHeapInternal+0x000000000000a0a
    00007ffd82421346 ucrtbase!_malloc_base+0x0000000000000036
    00007ff7f281100f mem_leak!main+0x000000000000000f [c:\users\alexandrg\documents\corehard_debugging\
    00007ff7f2811259 mem_leak!__scrt_common_main_seh+0x000000000000011d [f:\dd\vctools\crt\vcstartup\sr
    00007ffd83be2784 KERNEL32!BaseThreadInitThunk+0x0000000000000014
    00007ffd85f50c31 ntdll!RtlUserThreadStart+0x0000000000000021
```

Недостатки дампов памяти

- Дамп файл содержит снимок состояния программы в определенный момент времени
- Невозможно проследить развитие дефекта во времени

Time travel debugging

- Во время отладки Windbg записывает выполнение программы в лог файл, который позже можно повторно воспроизвести.
- Позволяет выполнять инструкции в обратном порядке. Легко понять причины, которые привели к ошибке.
- Включает модель данных, позволяющую выполнять LINQ запросы к сохраненной трассировке
- Значительно влияет на производительность (!) . Отлично подходит для отладки предсказуемых дефектов

Dump file vs TTD trace


C:\>TTD_DEMO

Автоматизация анализа дампов памяти

- Экономия времени на поиск источника проблемы: быстрое определение модуля с ошибкой и ответственного разработчика
- Позволяет приоритезировать задачи: определение дампов с одинаковой проблемой и автоматическая группировка
- Экономия времени на анализ аварийных завершений: разработчики даже не открывают дампы с уже известными проблемами

Для автоматизации используется встроенный скриптовый язык. Можно также применить расширение PyKd и язык Python

Напоследок

Пару практических советов, как сделать отладку приятнее

- Cmdtree
- Произвольные визуализации


C:\>DEMO

Спасибо за внимание!

golovach@checkpoint.com