Debugging in Windows

Crash Dump Analysis 2014/2015

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

Department of Distributed and Dependable Systems

Windows vs. UNIX

- **Many things very similar (in principle)**
- **Many things slightly different**
  - Terminology
  - Tools and file formats
    - Visual C++, PE, PDB
  - Methods and techniques
    - No strict line between kernel syscalls and user space library functions (heap allocation, resources, etc.)
  - Conventions and habits

Calling conventions

- **cdecl (C calling convention)**
  - Almost identical to System V ABI on IA-32
  - Arguments passed on stack in reverse order
    - Support for variadic functions
    - Caller cleans the stack (pops the arguments)
  - Usual prologue, epilogue and stack frames (using the frame pointer)

```
push %ebp              leave
movl %esp, %ebp        ret
sub $imm, %esp
```

Calling conventions (2)

- **stdcall (standard calling convention)**
  - Used for all Win32 API calls (WIN32API macro)
  - Arguments passed on stack in reverse order
    - **No support** for variadic functions
      - Arity encoded in the mangled function name
    - Callee cleans the stack

```
leave
ret $imm
```

      - Slightly shorter code (less duplicity)
  - Different prologue

```
enter $imm, 0
```

Calling conventions (3)

- **fastcall**
  - Almost identical to stdcall
    - But first two arguments passed in ECX, EDX
- **thiscall**
  - For C++
  - Almost identical to stdcall
    - Implicit object argument (*this) passed in ECX

Calling conventions (4)

- **64bit cdecl**
  - Similar to System V ABI on AMD64
    - Used as the universal calling convention on AMD64
  - First four arguments passed in RCX, RDX, R8, R9
    - Space on stack is reserved for possible spill
  - Other arguments passed on stack in reverse order
    - Caller cleans the arguments (support for variadic functions)
    - 16B stack alignment, 16B red zone
    - Scratch registers: RAX, RCX, RDX, R8, R9, R10, R11

Debugging facilities

- **User space debuggers**
  - Common debugging API (dbghelp.dll)
    - Standard debuggers
      - Visual Studio Debugger
      - CDB, NTSD
    - 3rd party debuggers
      - OllyDbg, etc.
- **Kernel debugger**
  - Part of Windows NT kernel
    - KD
    - Remote debugging (serial line, FireWire, USB 2.0, VMware extension)

Debugging facilities (2)

- **WinDbg**
  - GUI front-end for CDB, NTSD and KD
  - Both instruction-level and source-level debugging
  - Extensible via DLL plugins
    - Support for debugging .NET binaries, etc.
- **3rd party kernel debuggers**
  - SoftICE, Syser, Rasta Ring 0 Debugger
    - Kernel-only instruction-level debugging
    - Run-time kernel patching to gain control over the Windows NT kernel
    - Can make some use of virtualization environments

Resources

- Debugging tools for Windows
  - WinDbg and related tools

    http://www.microsoft.com/whdc/devtools/debugging/
  - Documentation in MSDN

    https://msdn.microsoft.com/en-us/library/ff551063.aspx
  - Tutorials

    http://www.codeproject.com/Articles/6084/Windows-Debuggers-Part-A-WinDbg-Tutorial
    - WinDbg from A to Z

      http://windbg.info/

Debugging API

- Common methods for writing debuggers
  - Parsing binaries (`ImageNtHeader`)
  - Dumping core (`MiniDumpWriteDump`)
  - Generating stack trace (`StackWalk`)
  - Symbol handling (`SymFromAddr`)
    - Original symbol information format: COFF
    - Current symbol information format: PDB file

Symbols

- **Symbols location**
  - _NT_SYMBOL_PATH environment variable
    - Binaries and symbols matched according to compilation timestamp and/or GUID
    - Symbols for Windows components (all public builds)
      - Available from Microsoft public symbol server
      - Can be also downloaded by hand (hundreds of MBs)
    - It is possible to provide a custom symbol server
      - For debugging of release binaries at the customer's site

```
_NT_SYMBOL_PATH=srv*c:\sym_cache*http://msdl.microsoft.com/download/symbols
```

http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx

CDB

- Command line user space debugger
  - NTSD is almost identical, but it is not a console application
  - Debugging modes
    - Invasive debugging
      - A break-in thread in target process
      - Full-featured debugging (but only one debugging session)
    - Non-invasive debugging
      - Only freezing threads
      - Memory analysis possible, but no flow control (breakpoints etc.)

KD

- Command line kernel debugger
    - Local kernel debugging very limited
    - Remote debugging
        - Serial line
            - Limited to 115 kbaud
            - VMware virtual serial line can be much faster
        - FireWire (IEEE 1394)
            - Fast, but the generic FireWire driver has to be deactivated
        - USB 2.0
            - Fast, but a special debugging cable is required

WinDbg

- **Universal GUI front-end**
  - Both for CDB and KB
    - Running processes
    - Attaching to existing processes
    - Opening core and crash dumps
    - Remote debugging
  - Basically still the same command line interface
    - More windows, special views for easier navigation
      - Watches, breakpoints, disassembly, source code, registers, etc.

Remote debugging

- **For user space applications**
  - Debugging target

    ```
    dbgsrv.exe -t tcp:port=1025
    ```
  - Debugging client

    ```
    windbg.exe -premote tcp:server=hostname,port=1025
    ```
    - Useful commands

      ```
      .tlist
      ```
      – List processes running on the target

WinDbg commands

- **Regular commands**
  - No prefix, but possible suffixes (variants)
  - Controlling the debugging session

`? <cmd>`
- Help on *cmd*

`g`
- Continue execution

`p`
- Step over (instruction or source line)

`t`
- Step into

`pt`
- Step over until next return

`tt`
- Step into until next return (skipping nested returns)

WinDbg commands (2)

`pc`

- Step over until next call (if the current instruction is a call, then it is ignored)

`tc`

- Step into until next call

`pa` *`<addr>`*

- Step over until address *addr* is reached

`r`

- Dump all registers

`u` *`[addr]`*

- Disassemble

`lm`

- List loaded modules (DLLs)

`k`

- Print the stack trace

WinDbg commands (3)

`~`

- Get information from all threads

`~.`

- Get the current thread information

`~[tid]`

- Get information from the thread *tid*

`~* k`

- Print the stack trace of all threads

`kP`

- Print the stack trace with full function arguments

`kv`

- Print the stack trace with the information about calling conventions

WinDbg commands (4)

`dd <addr>`

`da <addr>`

`du <addr>`

- Display doubleword, ASCII, Unicode at *addr*

`f <addr> <value> ...`

- Fill the memory at *addr* with the *values*

`bl`

- List breakpoints

`bp <addr>`

- Set execution breakpoint at *addr*

`ba <addr>`

- Set memory access breakpoint at *addr*

`bc <addr>`

- Clear breakpoint at *addr*

`be <addr>`

`bd <addr>`

- Enable/disable breakpoint at *addr*

WinDbg expressions

```
?? <expr>
@@c++(<expr>)
```

- Return the value of any C++ expression which does not have any side effects (i.e. no function calls)
  - Compound types, arrays, pointer arithmetics, etc.
- Implicitly used in watch and locals windows for watches and displaying local variables
  - Display an integer variable value

    ```
    ?? local_var
    int 42
    ```
  - Display the memory location of an integer variable

    ```
    ?? &local_var
    int * 0x00123456
    ```

Advanced breakpoints

```
bp module!my_func_*
```

- Breakpoints on multiple functions (wildcards)

```
bp @@c++(MyClass:MyMethod)
```

- Breakpoint on a member function of all instances of a class

```
~1 bu kernel32!LoadLibraryExW
```

- Breakpoint on a function which hits only in a given thread
- Lazy symbol resolving

WinDbg commands (5)

- **Dot commands**
  - Slightly more advanved

    `.help <cmd>`
    - Help on dot-*cmd*

    `.lastevent`
    - Information about last event/exception

    `.dump`
    - Create a core dump

    `.attach <pid>`
    - Attach to a process *pid*

    `.detach`
    - Detach from the attached process

    `.restart`
    - Restart the attached process

WinDbg commands (6)

`.if <expr>`

`.else`

`.elseif <expr>`

- Optional command execution
- C++ expressions as conditions
- Multiple commands can be enclosed in {} blocks

`.for ...`

`.while <expr>`

`.Break`

`.continue`

- Advanced scripting

`.foreach <cmd>`

- The output of *cmd* is fed to a other commands
- Usually line-by-line
  - The semantics differs for each *cmd*

WinDbg commands (7)

- **Extension commands**
  - Supplied by add-on modules (DLLs)
  - Advanced functionality

    `!runaway`
    - Display timing information of all threads
    - Can be used to detect hangs

    `!locks`
    - Display information about locked critical sections

    `!address <addr>`
    - Display information (protection status, owner) of the given page

`!analyze`

`!analyze -hang`

- Various heuristics for analyzing the root cause of the previous event/exception
- Runs various consistency checks on kernel structures
- Stack analysis, heap analysis
- Corrupted code stream analysis (bad RAM)
- Invalid call sequences (bad CPU)

WinDbg pseudoregisters

- **Various values useful for debugging**
  - Can be used in expressions or directly as command arguments
    - `$ra`
      - Current stack frame return address
    - `$ip`
      - Current instruction address (EIP, RIP)
    - `$retreg`
      - Current value of the return register (EAX, RAX)
    - `$csp`
      - Current stack pointer (ESP, RSP)
    - `$tpid`
      - Current process ID
    - `$tid`
      - Current thread ID