STrace

A DTrace on Windows Reimplementation

By Stephen Eckels

Senior Reverse Engineer - FLARE
Stephen Eckels

Senior Reverse Engineer (FLARE)

- Malware RE
- R&D - RE tooling

Experience
- Open-source development w/ a focus on hooking
- x86/x64 ASM, C++, Rust, Go, JS, C#, Java, more 😊
- System architect, embedded device software developer

Interests
- Slacklining
- Electric Mountainboarding
- Gaming

@stevemk14ebr
github.com/stevemk14ebr/
Presentation Outline

What is DTrace?
- Why is it interesting
- Usage of Microsoft’s Port
- Shortcomings

DTrace on Windows Internals
- Modern dtrace architecture
- How kernel registration works
- Documenting kernel interfaces
- Inserting ourselves into the boot process
- Accessing paged out memory safely

STrace – Steve’s Tracer
- Differences to Microsoft’s port
- Writing a driver in Rust
- Adding WASM to the kernel
- Ditching rust drivers for C++
- Loading DLLs in drivers
- DIY kernel -> user callstack trace
- DIY callstack symbolication
DTrace Introduction

What it is

- Sun Microsystems -> Oracle -> OpenDTrace (2016)
- Tracing system allowing users to register probes
- Probes ‘fire’ on specific, user defined conditions
- Multiple types of probes are supported, each offering a different source of information
  - STP – syscall hooks
  - FBT – kernel function boundary tracing
  - PID – user function boundary tracing
  - ETW – event tracing for windows (ETW) events
- When fired, probes execute user scripts in a VM inside the kernel

Examples

Print created files for a target process

```
syscall::NtCreateFile:entry
/ pid == $target && arg0 != 0 && arg2 !=0 /
{
  attr = (OBJECT_ATTRIBUTES*)copyin(arg2, sizeof(OBJECT_ATTRIBUTES));
  objectName = (UNICODE_STRING*)copyin(attr->ObjectName, sizeof(UNICODE_STRING));
  name = *((**uintptr_t)objectName->Buffer, objectName->Length);
  printf("OBJECT_NAME: %ws", name);
  ustack();
}
```

Live system dump on specific error code

```
syscall::return {
  if (arg0 != 0xc0000001UL) {
    lkd(0);
    printf(" Triggering Live dump \n ");
    exit(0);
  }
}
```
The pain begins…

**Deferring Reads**

```c
syscall::NtCreateFile:entry {
   // data behind pointer may be paged out. Don’t read until paged in.
   self->pFileHandle = args[0];
}

syscall::NtCreateFile:return {
   // system routine has paged in the memory.
   if(arg0 == STATUS_SUCCESS) {
      handle = *(HANDLE*)copyin(self->pFileHandle, sizeof(HANDLE));
      printf("%08X", handle);
   }
}
```

**Pretty print enum types.**

```c
syscall::NtCreateFile:entry
   // pid == $target && arg0 != 0 && arg2 !=0 /

   this->access = " |
   this->access = strjoin(this->access, (arg1 & FILE_READ_DATA) ? "ReadData|": " " );
   this->access = strjoin(this->access, (arg1 & FILE_WRITE_DATA) ? "WriteData|": " " );
   this->access = strjoin(this->access, (arg1 & FILE_APPEND_DATA) ? "AppendData|": " " );
   this->access = strjoin(this->access, (arg1 & FILE_READ_EA) ? "ReadEa|": " " );
   this->access = strjoin(this->access, (arg1 & FILE_WRITE_EA) ? "WriteEa|": " " );
   this->access = strjoin(this->access, (arg1 & FILE_EXECUTE) ? "Execute|": " " );
   this->access = strjoin(this->access, (arg1 & FILE_DELETE_CHILD) ? "DeleteChild|": " " );
   this->access = strjoin(this->access, (arg1 & FILE_READ_ATTRIBUTES) ? "ReadAttributes|": " " );
   this->access = strjoin(this->access, (arg1 & FILE_WRITE_ATTRIBUTES) ? "WriteAttributes|": " " );
   printf("CreateFile::ACCESS_MASK: %s", this->access);
}
```
Advanced DTrace Examples
The pain begins...

Copying UNICODE_STRINGS, C preprocessor for ‘functions’

```
// @echo off
// start /wait cmd /c "cl.exe /P %1 && dtrace -s %~dpn1.i -c %2" & del %~dpn1.i
// Use: DTCompile.bat createfile.d %windir%\system32\notepad.exe

syscall::NtCreateFile:entry
// pid == $target && arg0 != 0 && arg2 !=0
{
    copy_ustr(arg2, fname, fnamelen);
    printf("%*ws", fmt_ustr(fname, fnamelen));
}

#ifdef copy_ustr(argname, varname, lenname)\
attr = (OBJECT_ATTRIBUTES*)copyin(argname, sizeof(OBJECT_ATTRIBUTES));\
if(attr) {
    objectName = (UNICODE_STRING*)copyin((uintptr_t)attr->ObjectName, sizeof(_UNICODE_STRING));\
    if (objectName) {
        this->##varname = (struct ustr*)copyin((uintptr_t)objectName->Buffer, sizeof(struct ustr));\
        this->##lenname = objectName->Length;
    }
}
#endif

#define fmt_ustr(varname, lenname) (this->lenname / 2), this->varname->buffer
```
DTrace shortcomings

- **D language Issues**
  - No user defined functions!
  - No loops!
  - Control flow structures like if-s occasionally compile to invalid instruction sequences in some cases

- **System design issues**
  - Extremely limited side-effects. DTrace APIs like `system` only accept static strings
  - Issues accessing memory before syscalls execute
  - Features like the `dtrace` pre-processor are simply not implemented in Microsofts’ port
  - Wide string support is shaky at best. No string utilities are provided at all.
  - Developer experience tools lacking entirely, no IDE, syntax highlighting, etc.
DTrace on Windows Internals
The start of a re-implementation

- High level architecture
- Boot process
- Neat dtrace kernel tricks
- Documenting kernel interfaces

We can do better!
<403 – meme forbidden>
Architecture

• The interface changed with 20H1! \texttt{traceext.sys} is no more

• \texttt{dtrace.sys} registers a set of function pointers with the kernel for each probe type

• \texttt{dtrace.sys} executes D language scripts in a VM and collects logs and tracing data

• \texttt{dtrace.exe} communicates with \texttt{dtrace.sys} to fetch queued tracing data and provides cached symbol information to the driver.
DTrace Boot

- The boot process starts at the bootloader `winload.exe`
  - Check boot flag `dtrace` is on (`bcdedit`)
  - Add `ext-ms-win-ntos-trace-l1-1-0.dll` to schema set map

```
0slpDynamicSchemaExtensions dq_offset aExtMsWinNtosTr
    ; DATA XREF: 0slpApplyDynamicSchemaExtensions
    ; 0slpApplyDynamicSchemaExtensions+2070
    ; "ext-ms-win-ntos-trace-l1-l-0"
    ; "dtrace.sys"
    dq_offset aDtraceSys
    dq_offset 0slpDtraceExtensionEnabled
```

- `winload.exe` begins to load `ntoskrnl.exe`
- `ntoskrnl.exe` imports from this apiset, consulting the schema set map

- `ntoskrnl.exe` calls `TraceInitSystem` from `dtrace.sys`
  - Invoked during phase 1 kernel initialization, before patchguard
  - `KeInitSystem -> KilInitDynamicTraceSupport -> TraceInitSystem`
ApiSet Dlls

- ‘Pseudo’ PEs that contain a binary map pointing to implementing PEs

- HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\ApiSetSchemaExtensions\<anything>
  - FileName=<path_to_api_set.dll>
  - Name=<anything>
ApiSetExtensions are flipped
Host -> ApiSet
Signing Requirements

- dtrace.sys is loaded as an import of ntoskrnl.exe
  - Despite being a dynamic extension, it’s considered ‘part of’ the kernel
  - Must be signed with an MS root code signing cert

- Schema set DLL must also be MS signed

- So...how do we load our reimplementation of dtrace.sys?
  - We can re-use the MS signed schema set DLL to enable the dynamic schema set extension
  - Create new boot entry to disable DSE so that ntoskrnl.exe can load our unsigned driver as an import

- Signing it yourself, test signing, custom kernel signers, etc...will NOT work.
  - Microsoft: Please reconsider this design and allow dtrace.sys to be signed by non-MS parties.
DTrace Kernel Registration

**TracelnitSystem**

- **dtrace.sys** fills **KiDynamicTraceCallouts** table to handle each probes’ events

- **dtrace.sys** passes a pointer to **TraceAccessMemory** to **ntoskrnl.exe**
  - **ntoskrnl.exe** records the start and end bounds of this routine via the x64 unwind table entry

- **ntoskrnl.exe** passes a pointer to **KiDynamicTraceContext** to **dtrace.sys**
  - Provides offsets into kernel structures for the current kernel version
  - Provides APIs to set and remove trace probes
Syscall Probes!

- **KiSystemCall64**
  - Fires for all usermode syscalls
  - And Zw* calls, as they go through **KiServiceInternal**
  - Nt* calls are not traced.
  - Shadow SSDT syscalls *not* supported

- **KiTrackSystemCallEntry** -> StpCallbackEntry
- **KiTrackSystemCallExit** -> StpCallbackReturn

- Probe entry / exit callbacks must be one of the two Stp functions set in **KiDynamicTraceCallouts** during **TraceInitSystem**
  - **KiDynamicTraceCallouts** is PG protected.
DTrace Tricks

- **TraceAccessMemory** – safely read arbitrary inaccessible or paged out memory at a high IRQL w/o blue screening
  
  - copyin in DTrace scripts use this

```c
bool __fastcall KiIsTraceMemoryAccess(unsigned __int64 a1)
{
    return KiDynamicTraceEnabled && a1 >= TraceAccessMemoryStart && a1 < TraceAccessMemoryEnd && KeGetCurrentIrql() == 15;
}
```

- **MmAccessFault** -> **KeInvalidAccessAllowed**

```c
if (KeInvalidAccessAllowed (TrapInformation) == TRUE) {
    return STATUS_ACCESS_VIOLATION;
}
```
DTrace Tricks

- **self** keyword – kernel thread local storage

  ```c
  struct _KTHREAD
  {
    // ...
    ULONGLONG TracingPrivate[1]; // 0x3c0
    // ...
  }

  struct KiDynamicTraceContext
  {
    ULONG32 supported_flags;
    ULONG32 kthread_tracingprivatedata_offset;
    ULONG32 kthread_tracingprivatedata_arraysize;
    // ...
  }
  
  void setTlsValue(uint64_t value)
  {
    PKTHREAD pThread = KeGetCurrentThread();
    if (kthread_tracingprivatedata_arraysize <= 0)
    {
      __debugbreak();
      return;
    }
    uint64_t* pTlsArray = (uint64_t*)((char*)pThread + kthread_tracingprivatedata_offset);
    pTlsArray[0] = value;
  }

  uint64_t getTlsValue()
  {
    PKTHREAD pThread = KeGetCurrentThread();
    if (kthread_tracingprivatedata_arraysize <= 0)
    {
      __debugbreak();
      return 0;
    }
    uint64_t* pTlsArray = (uint64_t*)((char*)pThread + kthread_tracingprivatedata_offset);
    return pTlsArray[0];
  }
  ```

- Used by **dtrace** to cache D language intermediate data, such as thread local data
DTrace Tricks

• **dtrace_action_lkd** – live kernel dumps
  – **lkd** in DTrace scripts use this

• void *DbgWerCaptureLiveKernelDump(wchar_t* ComponentName,
   ULONG BugCheckCode,
   ULONG P1, ULONG P2, ULONG P3, ULONG P4,
   ULONG flags);
  – Generates standard .dmp files loadable in windbg
  – NtSystemDebugControl

• Flags and registry settings to control this API now officially documented!

• Generated dumps typically in C:\Windows\LiveKernelReports
DTrace Tricks

- **TraceWalkUserStack** – Generate a symbolic stack trace down to usermode.
  - `ustack()` in dtrace

- Operates differently from how ‘other’ tools generate stack traces
  - Usermode API: StackWalk64 + DIA symbol APIs
  - Small trick to start at kernel, grab Ethread.Kthread.KernelStack.RSP
  - Trace initiated from usermode, drivers can’t call this

- **TraceWalkUserStack** does what StackWalk64 does, manually
  - Trace initiated from kernelmode
  - RtlCaptureStackBackTrace doesn’t include usermode calls

- How?
  - Given some starting address, locate the module’s
    - DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION]
  - unwind data defines a series of opcodes to ‘undo’ the effects of a function on the CPU state.
    - These are ‘virtually’ executed, then the return address from the unwound stack is read, and the process repeats
    - Ex: UWOP_PUSH_NONVOL – a register pushed to stack, RSP += 8 to undo
STrace
Writing a re-implementation!

- Differences to Microsoft’s port
- Writing a driver in Rust
- Adding WASM to the kernel
- Ditching rust drivers for C++
- Loading DLLs in drivers
- DIY callstack symbolication

Who says hipster languages don’t belong in the kernel??
STrace vs. DTrace

- **dtrace** executes at IRQL == 15, **STrace** doesn’t touch IRQL – usually LOW_LEVEL
  - Memory accesses simpler
  - Enables wide range of APIs to be called in handlers

- Plugins can perform arbitrary side-effects and may call any kernel APIs

- Plugins are written in modern C++, rather than D Language
  - Faster than interpreting too!

- Symbolication of output occurs after logging, rather than within the driver.
  - For simplicity

- Only syscalls probe types are supported, FBT, PID, ETW, etc are not supported
  - Partially due to DSE being off, hypervisor features like FBT can’t be implemented
  - Open to PRs for ETW!

- Zw* APIs can be monitored, dtrace filters these out.
System Architecture Choices

• Requirements
  – Must execute scripts provided from usermode somehow
  – Must be fast enough to be called every syscall
  – The script language must be pleasant and familiar to all users
  – Must execute in the kernel – zero dependencies

• Nice to haves
  – Users shouldn’t need kernel development experience to use
  – Users should be able to call into the kernel for side-effects
  – Symbolic call stacks

• The Journey of STrace’s design
  1) Rust w/ WASM scripts
  2) C++ w/ WASM scripts
  3) C++ w/ C++ scripts
  4) C++ w/ C++ scripts & Rust symbolication
WASM in the NT Kernel!

• Just bad enough of an idea it’s great

• dtrace uses a VM for execution, it tries to be isolated. Can we do that somehow?
  – Web technologies already solved this
  – We can compile any language to WASM!

• Wasm toolchains are somewhat mature, but they all mostly focus on javascript being your build target.
  – Blockchain technologies solved this

• Enter WASMI – Parity Ethereum contracts are in WASM
  – Wasmer doesn’t support no_std 🤔
  – Not many good embedded WASM interpreters!

• Deep call stacks, exhausting the kernel stack. KeExpandKernelStackAndCalloutEx 😞
  – Massive local buffers??

• Too slow
  – Need a JIT, no hackable low dependency WASM JITs
Rust in the NT Kernel!

- Rust and the DDK together make an unpleasant experience
  - Microsoft only officially supports C
  - Complicated `.cargo/config` build profile needed to invoke VS linker properly
  - Complicated `build.rs` script needed

- Rust doesn’t properly (easily?) generate free standing binaries
  - Unwind tables aren’t reliably generated
  - Symbols like `__CxxFrameHandler3`, `__chkstk`, etc, must be stubbed – always generated 😞
  - Handful of I.C.E.’s

- Microsoft doesn’t provide Rust API projections for the kernel
  - Every single API, type, and define must be manually ported to rust 😞
  - Rust doesn’t support anonymous unions and structures; Windows uses this extensively 😞

- Rust’s `no_std` story is, complicated
  - Many libraries don’t support `no_std`
  - Providing allocator mappings isn’t exactly easy
  - At the time of writing, `std::ffi::CString` wasn’t even supported in Rust `no_std`. This has since changed.

- **UNSAFE EVERYWHERE!**
C++ > Rust

• Rust and WASM was painful

• C++ wins
  – API and structures definitions known correct
  – Toolchain supported officially

• Can C++ host our WASM scripts instead?
  – Yea, but it’s hard, no zero dependency C++ WASM interpreters – wasted weeks trying
  – Deadend.

• How else can we provide users the ability to ‘script’
  – Execute blobs of code as shellcode??
  – Position Independent compiler: https://github.com/SheLLVM/SheLLVM
  – Little too hacky...
C++ DLLs in the NT Kernel

- Windows has a mechanism for shipping blobs of dynamic code – DLLs!
  - Not at all supported in the kernel

- Manual DLL mappers aren’t that hard to write
  - Kernel isn’t wildly different to usermode really
  - Remove all dependencies like the CRT, exceptions, disable stack checks and other security mechanisms

- Re-use DLL IAT and EAT for kernel APIs

- Compiler is a bit too smart with intrinsics

- It works! And it’s fast! And Easy!
  - Toolchain is normal too!

```c
// mycrt.h
extern "C" char* strcpy(char* a, const char* b);
#pragma intrinsic(strcpy)

// mycrt.cpp
#pragma function(strcpy)
extern "C" char* strcpy(char* a, const char* b)
{
    char* saved = a;
    while (*a++ = *b++);
    return saved;
}
```
**DLLs importing kernel APIs**

- DLL imports are resolved via .lib files
  - DDK injects kernel .lib files into the msvc linker path
  - Not available when not writing drivers

- Workaround - manually link a copy of ntoskrnl.lib
An STrace plugin DLL

1. Define probe ID map
2. Set and Unset probes
3. Handle Callbacks!

```c
extern "C" __declspec(dllexport) void StcCallbackEntry(ULONG64 pService, ULONG32 probeId, MachineState& ctx, CallerInfo& callerinfo)
{
    switch (probeId) {
        case PROBE_TDS::IdSetInformationFile: {
            auto hFile = (HANDLE)ctx.read_argument(0);
            auto InformationClass = ctx.read_argument(4);
            if (InformationClass == 13) { // FileDispositionInformation
                auto pInformation = (char*)ctx.read_argument(2); // 1 == DeleteFile
                if (*pInformation == 1) {
                    auto pFilePath = getFilePathFromHandle(hFile);
                    if(pFilePath) {
                        LOG_INFO("File %s deleted\n", pFilePath);
                        backupFile((wchar_t*)backup_directory, pFilePath->Name, hFile);
                        ExFreePoolWithTag(pFilePath, POOL_TAG);
                        pFilePath = nullptr;
                        LOG_INFO("File Backup Complete\n");
                    } else {
                        LOG_INFO("File [unknown] deleted\n");
                    }
                }
            }
            PrintStackTrace(callerinfo);
        }
        break;
    }
}
```
DIY Callstack Symbolication

1. Grab callstack via RtlWalkFrameChain + RTL_WALK_USER_MODE_STACK
   ✓ Walks unwind codes just like dtrace does

2. Resolve callstack addresses to modules + offsets

3. Fetch and parse PDBs – Introducing PDBReSym!
   1. Read module PDB filename, guid, age from debug directory data
   3. Cache same as WinDbg does
   4. Parse PDB
   5. Build PC line to nearest symbol RangeMap cache (binary tree).

4. Use PC line to symbol cache to resolve module name and offset pairs to symbols

5. Re-write log lines with demangled symbolic line

NO DIA SDK USAGE!
Demo!
Thank You.