Ruptura
InfoSecurity

Malware Madness: *EXCEPTION edition*

Rad Kawar
@rad98

Ruptura InfoSecurity

Whoami?

★ Offensive Security Engineer

★ Malware/Viruses

★ Most things low level

★ Making blue teamers cry

Agenda

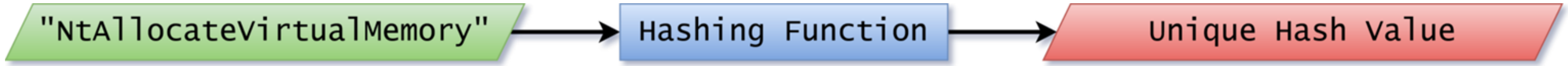
$\boxed{\text{API Hashing}}$

Syscalls

Ruptura
InfoSecurity

API Hashing

- ★ Been around for <30 years
- ★ Slow down static analysis
- ★ Avoids detection via import hashing (IAT entries)
- ★ Hashing = one way function
- ★ Used in a lot of malware (Lockbit/REvil/Trickbot/ZLoader etc.)



Ruptura InfoSecurity

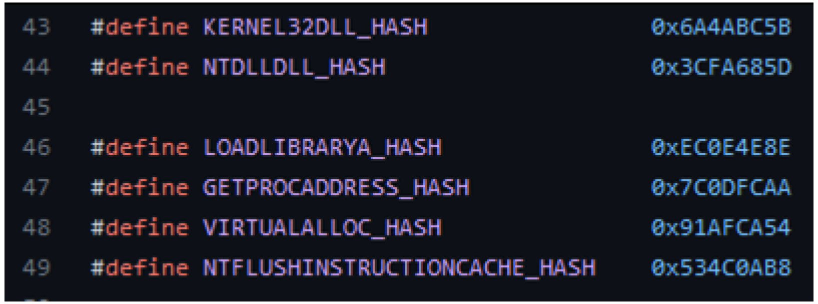
An Example

```
PVOID GetProcAddrExH( UINT funcHash, UINT moduleHash )
```

- ★ funcHash and moduleHash are pre-calculated before/during compilation
- ★ Loop through PEB_LDR_DATA to locate the correct module
- ★ Loop through the EAT of that module to locate the correct function
- ★ Return the address of the corresponding function

Reversing This

- ★ HashDB - Mostly Complete Hash Tables
- ★ Change representation hex & Google the hash
- ★ Dynamic analysis
  - ○ Open it up under a debugger
  - ○ x-ref the hashing function
- ★ Tools and guides available on both forms of analysis
- ★ Used by a lot of public projects (e.g. ReflectiveDLLInjection)



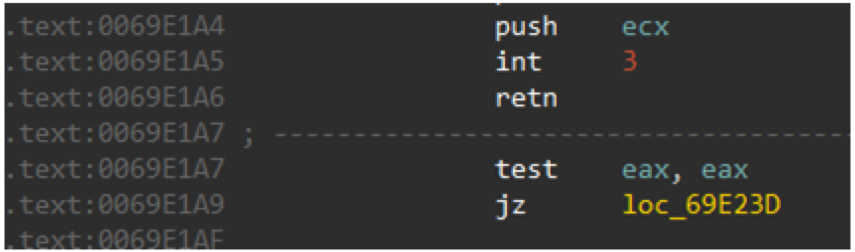
```
#define KERNEL32DLL_HASH                0x6A4ABC5B
#define NTDLLDLL_HASH                   0x3CFA685D

#define LOADLIBRARYA_HASH               0xEC0E4E8E
#define GETPROCADDRESS_HASH             0x7C0DFCAA
#define VIRTUALALLOC_HASH               0x91AFCA54
#define NTFLUSHINSTRUCTIONCACHE_HASH    0x534C0AB8
```

HashDB - https://github.com/OALabs/hashdb
Reflective DLL - https://github.com/stephenfewer/ReflectiveDLLInjection

Ruptura InfoSecurity

Dridex Banking Trojan

★ 32-bit trojan since 2014
★ Uses VEH to call APIs
★ Anti-static-analysis technique
  ○ Inserts a fake ret
  ○ Instructions after the ret for that prologue are ignored
★ Anti-dynamic-analysis
  ○ INT3 exception is swallowed by most debuggers
★ Kudos to @cPeterr for his excellent writeup



```
.text:0069E1A4                push    ecx
.text:0069E1A5                int     3
.text:0069E1A6                retn
.text:0069E1A7 ; ---------------------------------------
.text:0069E1A7                test    eax, eax
.text:0069E1A9                jz      loc_69E23D
.text:0069E1AF
```

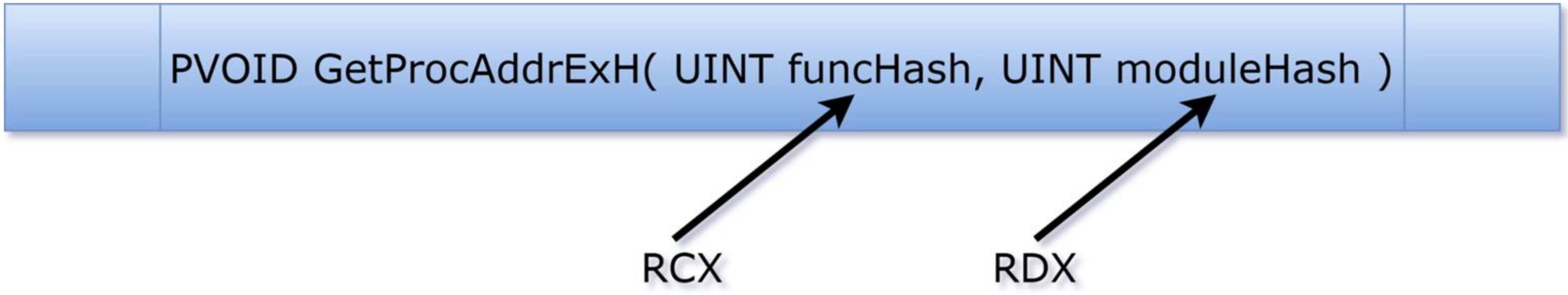
DRIDEX - https://www.0ffset.net/reverse-engineering/malware-analysis/dridex-veh-api-obfuscation/

Challenges re-implementing this

- ★ x86 calling convention different to x64
- ★ Returning a value from an exception handler
  - ○ Global value (thread safety issues & not very cool)
  - ○ Value stored in RAX
- ★ Easy to use interface as a malware developer
- ★ MSVC did **NOT** inline the assembly object
  - ○ Inling the code, provides the anti-static-analysis
  - ○ We can have a fake ret/jmp etc to break the disassembly

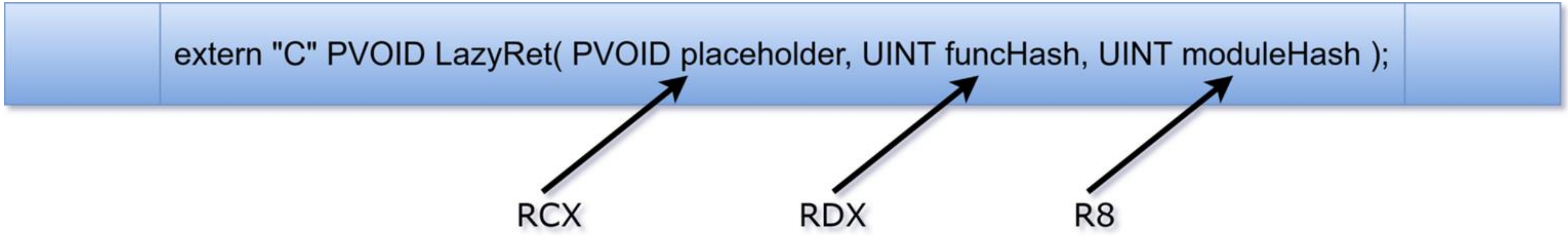
x64 calling convention

- ★ x86 arguments pushed onto stack
- ★ “four-register fastcall calling convention”
  - ○ rcx, rdx, r8, r9

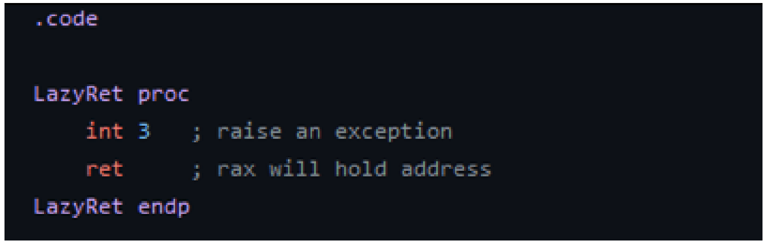


https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention

Implementing it


extern "C" PVOID LazyRet(PVOID placeholder, UINT funcHash, UINT moduleHash);
RCX
RDX
R8

- ★ Need to use assembly as tricky to return a value without actually returning a value from the function
- ★ Compiler prevents us
- ★ Compiler does the hard work for us of setting up parameters in registers etc.



```
.code

LazyRet proc
    int 3    ; raise an exception
    ret      ; rax will hold address
LazyRet endp
```

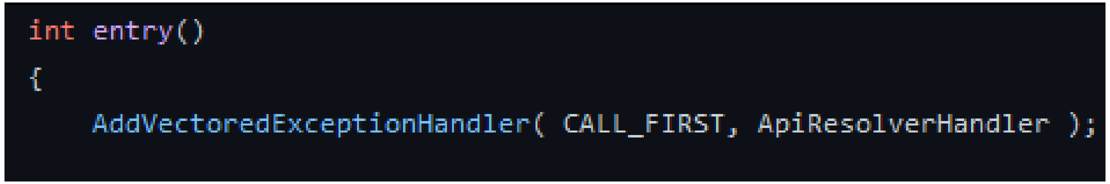
Compile and link this obj separately

64-bit Exceptions

- ★ Not catching an exception **will** in most cases lead to a crash
- ★ You can register your own VEH (AddVectoredExceptionHandler or SetUnhandledExceptionFilter)
- ★ Exceptions handlers get a copy of the CONTEXT and PEXCEPTION_RECORD
- ★ CONTEXT is a full snapshot of the current CPU state
- ★ PEXCEPTION_RECORD contains the exception information
  - ○ What exception code was thrown?
- ★ Exception handler returns EXCEPTION_CONTINUE_EXECUTION (-1) or EXCEPTION_CONTINUE_SEARCH (0)

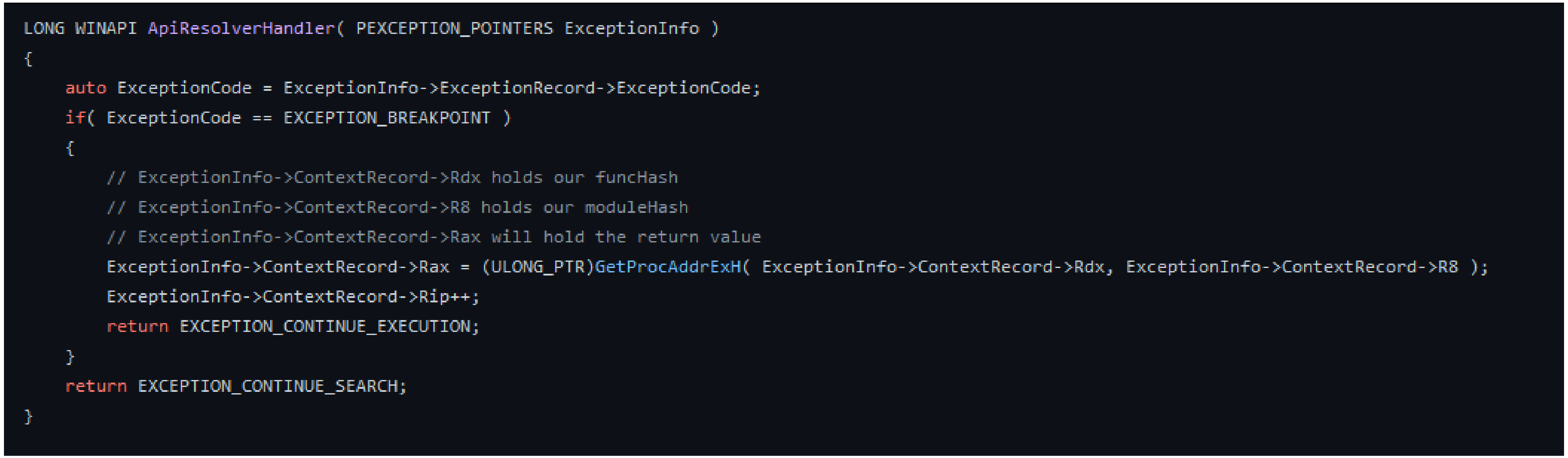
Ruptura InfoSecurity

Handling the exception



```c
int entry()
{
    AddVectoredExceptionHandler( CALL_FIRST, ApiResolverHandler );
```

Adding our Vectored Exception Handler
Our exception handler will only filter on EXCEPTION_BREAKPOINT



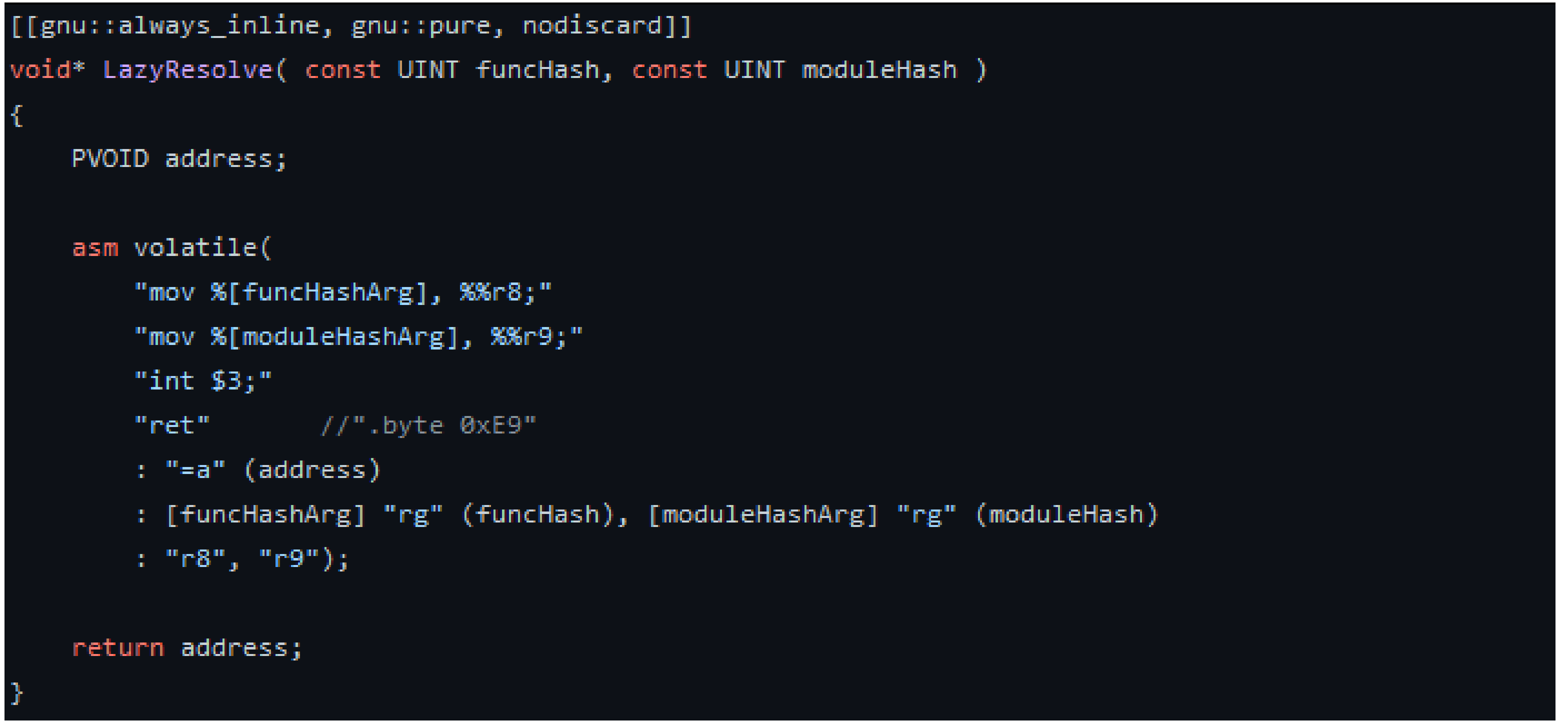
```c
LONG WINAPI ApiResolverHandler( PEXCEPTION_POINTERS ExceptionInfo )
{
    auto ExceptionCode = ExceptionInfo->ExceptionRecord->ExceptionCode;
    if( ExceptionCode == EXCEPTION_BREAKPOINT )
    {
        // ExceptionInfo->ContextRecord->Rdx holds our funcHash
        // ExceptionInfo->ContextRecord->R8 holds our moduleHash
        // ExceptionInfo->ContextRecord->Rax will hold the return value
        ExceptionInfo->ContextRecord->Rax = (ULONG_PTR)GetProcAddrExH( ExceptionInfo->ContextRecord->Rdx, ExceptionInfo->ContextRecord->R8 );
        ExceptionInfo->ContextRecord->Rip++;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

So...

- ★ We register a Vectored Exception Handler
- ★ We then define a function in assembly with the required parameters
- ★ This function throws an exception (int3), caught by the handler
- ★ The handler captures the CONTEXT and passes RDX, R8 to GetProcAddrExH (the registers storing the hashes)
- ★ We store this returned value in RAX
- ★ We set the RIP to the instruction after int3 (which is ret) and continue executing
- ★ We are then able to return the value retrieved by GetProcAddrExH

Ruptura InfoSecurity

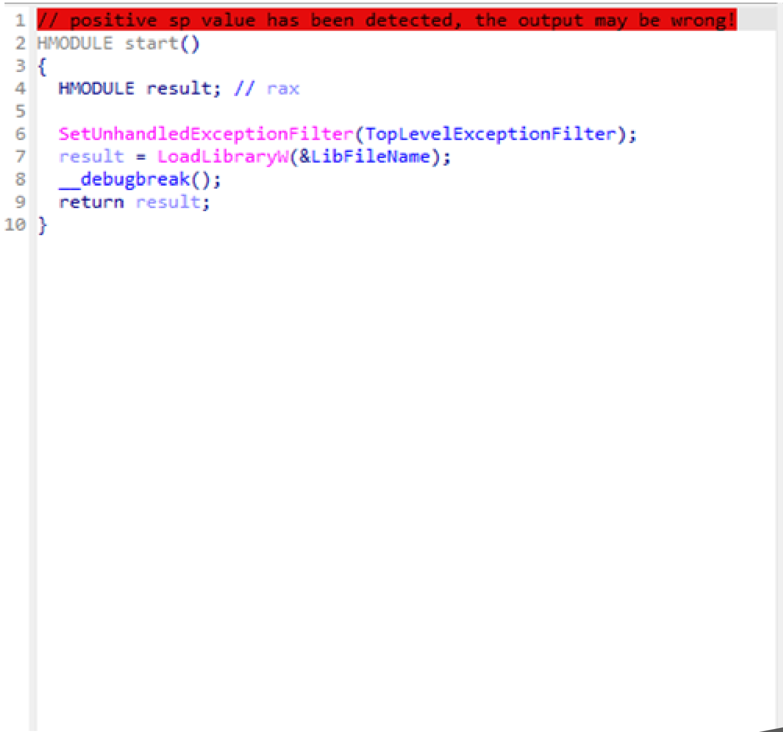
Inling using Clang’s Inline Assembly



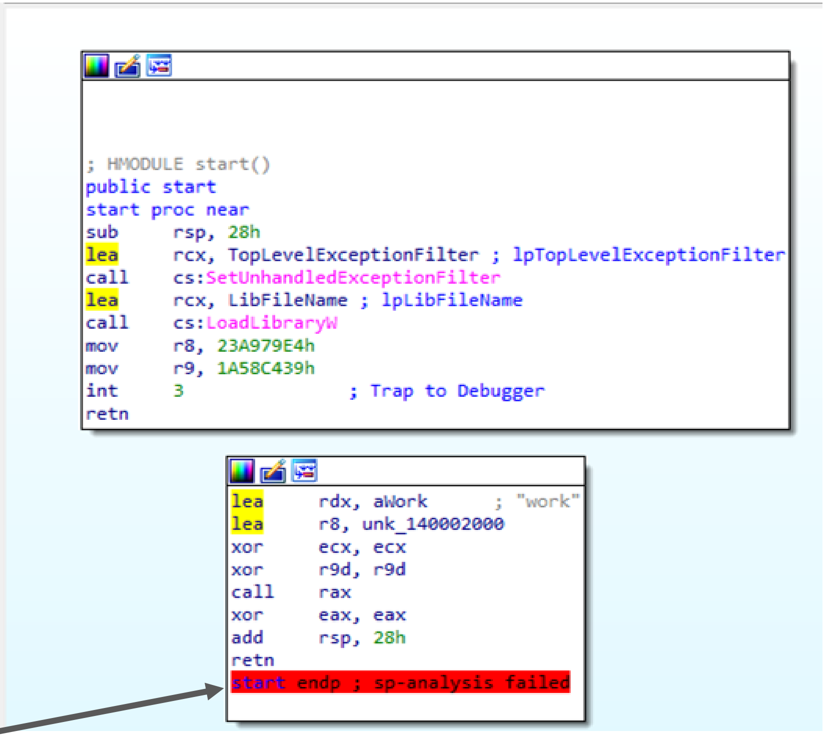
```
[[gnu::always_inline, gnu::pure, nodiscard]]
void* LazyResolve( const UINT funcHash, const UINT moduleHash )
{
    PVOID address;

    asm volatile(
        "mov %[funcHashArg], %%r8;"
        "mov %[moduleHashArg], %%r9;"
        "int $3;"
        "ret"       //".byte 0xE9"
        : "=a" (address)
        : [funcHashArg] "rg" (funcHash), [moduleHashArg] "rg" (moduleHash)
        : "r8", "r9");

    return address;
}
```

*AT&T syntax

Breaking IDA Disassembly



```
// positive sp value has been detected, the output may be wrong!
HMODULE start()
{
  HMODULE result; // rax

  SetUnhandledExceptionFilter(TopLevelExceptionFilter);
  result = LoadLibraryW(&LibFileName);
  __debugbreak();
  return result;
}
```



```
; HMODULE start()
public start
start proc near
sub     rsp, 28h
lea     rcx, TopLevelExceptionFilter ; lpTopLevelExceptionFilter
call    cs:SetUnhandledExceptionFilter
lea     rcx, LibFileName ; lpLibFileName
call    cs:LoadLibraryW
mov     r8, 23A979E4h
mov     r9, 1A58C439h
int     3               ; Trap to Debugger
retn
```

```
lea     rdx, aWork      ; "work"
lea     r8, unk_140002000
xor     ecx, ecx
xor     r9d, r9d
call    rax
xor     eax, eax
add     rsp, 28h
retn
start endp ; sp-analysis failed
```

IDA unsure of what to do

Code - https://github.com/rad9800/VehApiResolve/tree/inlined

Using VehApiResolve

- ★ Inlined version - https://github.com/rad9800/VehApiResolve/tree/inlined
- ★ Install LLVM 14.06
- ★ Copy code/use the provided project
- ★ SetUnhandleddExceptionFilter() at the start (this is what SEH uses for a global SEH filter)
- ★ Use hashFunc to define typedefs, use API(*)() to use
    - `hashFunc( MessageBoxA, int, HWND, LPCSTR, LPCSTR, UINT );`
    - `API( USER32, MessageBoxA )(NULL, "work", "plz", 0);`
- ★ Make sure required modules are loaded into your process

Evaluation

- We register a Vectored Exception Handler
- Difficult to analyze
  - Static - fake ret ruins static analysis
  - Dynamic - interrupt makes it harder to debug
- Easy to signature?
  - `mov r8, <value>`
  - `mov r9, <value>`
  - `int3`
- Exception based execution to make RE'ing hard?
- FlareOn 8 Challenge 9
  - Quite similar

Agenda

API Hashing

$\boxed{\text{Syscalls}}$

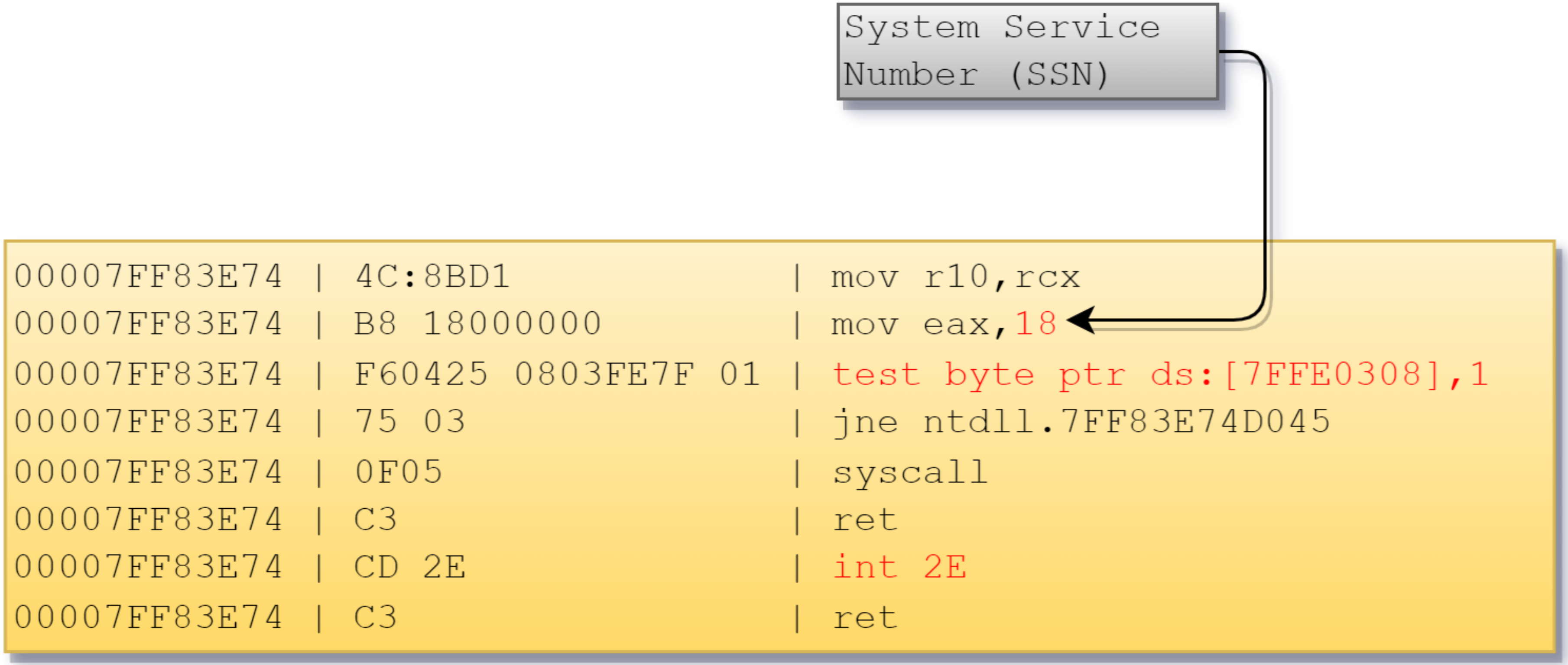
Syscalls

- ★ Interact with kernel to perform privileged actions
  - ○ Interacting with files, registry, etc.
  - ○ Memory management
- ★ Heavily monitored by endpoint security products
  - ○ Userland hooking
  - ○ Kernel callbacks
  - ○ Etw Threat Intelligence for various actions (not syscall specific)
    - ■ Kernel level ETW provider

http://redplait.blogspot.com/2019/03/windows-10-1809-kernel-sensors.html

Ruptura InfoSecurity

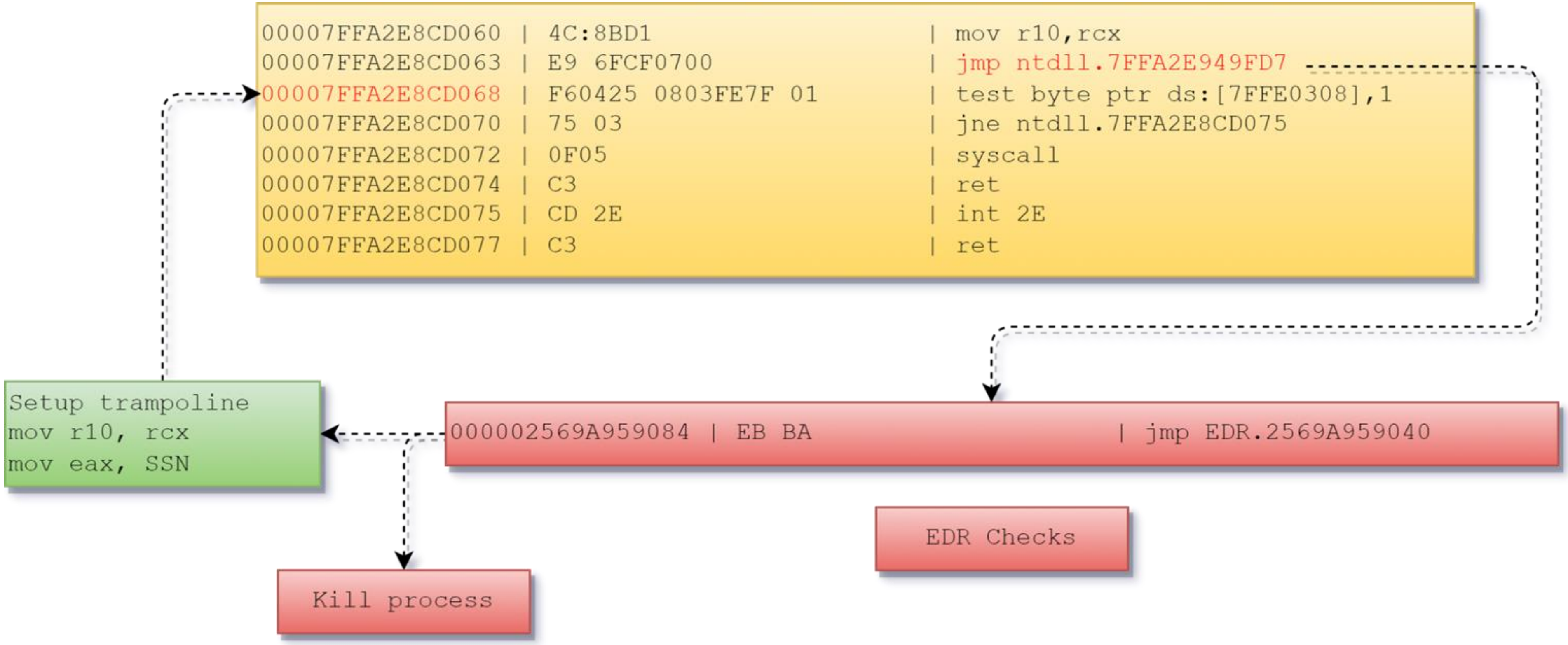
An Example Syscall Stub



```
00007FF83E74 | 4C:8BD1                  | mov r10,rcx
00007FF83E74 | B8 18000000              | mov eax,18
00007FF83E74 | F60425 0803FE7F 01       | test byte ptr ds:[7FFE0308],1
00007FF83E74 | 75 03                    | jne ntdll.7FF83E74D045
00007FF83E74 | 0F05                     | syscall
00007FF83E74 | C3                       | ret
00007FF83E74 | CD 2E                    | int 2E
00007FF83E74 | C3                       | ret
```

Ruptura InfoSecurity

An Example Hooked Syscall Stub



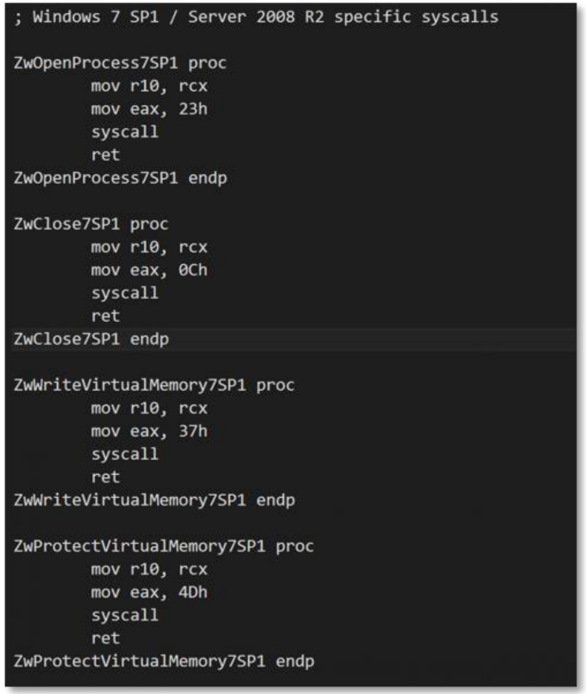
```
00007FFA2E8CD060 | 4C:8BD1                    | mov r10,rcx
00007FFA2E8CD063 | E9 6FCF0700                | jmp ntdll.7FFA2E949FD7
00007FFA2E8CD068 | F60425 0803FE7F 01         | test byte ptr ds:[7FFE0308],1
00007FFA2E8CD070 | 75 03                      | jne ntdll.7FFA2E8CD075
00007FFA2E8CD072 | 0F05                       | syscall
00007FFA2E8CD074 | C3                         | ret
00007FFA2E8CD075 | CD 2E                      | int 2E
00007FFA2E8CD077 | C3                         | ret
```

```
000002569A959084 | EB BA                | jmp EDR.2569A959040
```

```
Setup trampoline
mov r10, rcx
mov eax, SSN
```

Direct Syscalls

- ★ Hell's Gate by vx-underground
- ★ Popularized by Cn33liz @ Outflank
- ★ Store syscall stub with hardcoded SSN
  - ○ SysWhispers
- ★ SysWhispers2/3
- ★ Many ways to dynamically retrieve SSN
  - ○ No need for hardcoding SSNs
  - ○ modexp has discovered many ways

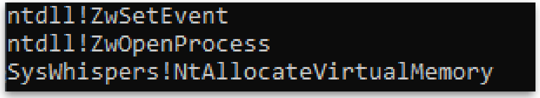


```
; Windows 7 SP1 / Server 2008 R2 specific syscalls

ZwOpenProcess7SP1 proc
        mov r10, rcx
        mov eax, 23h
        syscall
        ret
ZwOpenProcess7SP1 endp

ZwClose7SP1 proc
        mov r10, rcx
        mov eax, 0Ch
        syscall
        ret
ZwClose7SP1 endp

ZwWriteVirtualMemory7SP1 proc
        mov r10, rcx
        mov eax, 37h
        syscall
        ret
ZwWriteVirtualMemory7SP1 endp

ZwProtectVirtualMemory7SP1 proc
        mov r10, rcx
        mov eax, 4Dh
        syscall
        ret
ZwProtectVirtualMemory7SP1 endp
```

https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/
https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/

Catching Direct Syscalls

- ★ Static detection
  - ○ Trivial as the stub obj is linked
  - ○ Evasion is trivial
    - ■ Can insert nops or other garbage instructions
- ★ Dynamic detection
    - ■ Walk the call stack of the function
      - ● SymFromAddr + SymGetModuleInfo64 to get symbol information
      - ● If syscall originated outside of RX ntdll



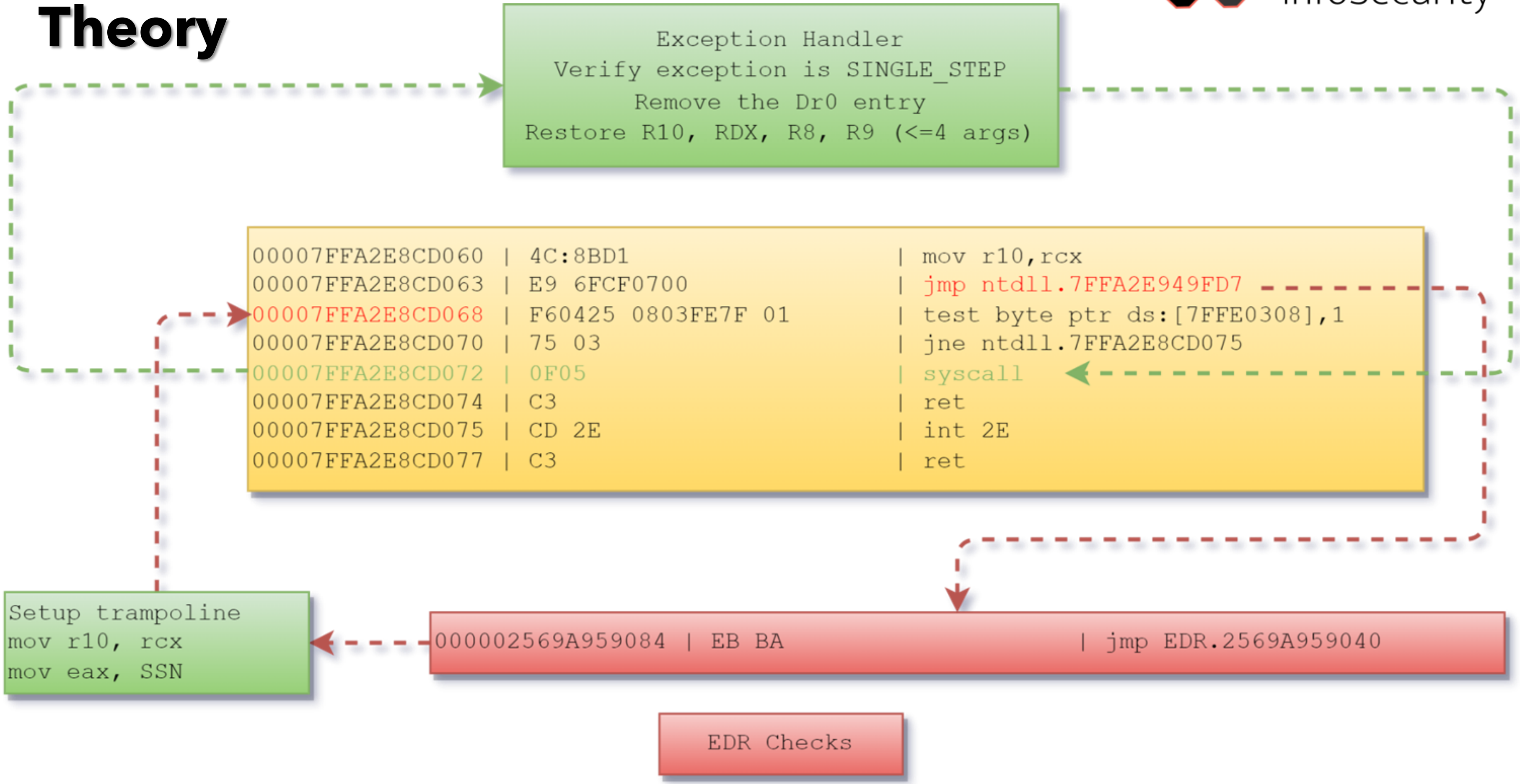
```
ntdll!ZwSetEvent
ntdll!ZwOpenProcess
SysWhispers!NtAllocateVirtualMemory
```

https://mez0.cc/posts/detecting-syscalls-with-fennec/

Hardware Breakpoints

- Dr0-Dr7
    - Dr0-Dr3 hold address
- GetThreadContext() + SetThreadContext()
- Dr7 holds conditions which need to be met for execution
    - On read/write/execution
    - Read/Write - size
    - Execution, size is 0
- GetThreadContext() / SetThreadContext()
    - “CONTEXT is a full snapshot of the current CPU state”

Theory



```
Exception Handler
Verify exception is SINGLE_STEP
Remove the Dr0 entry
Restore R10, RDX, R8, R9 (<=4 args)
```

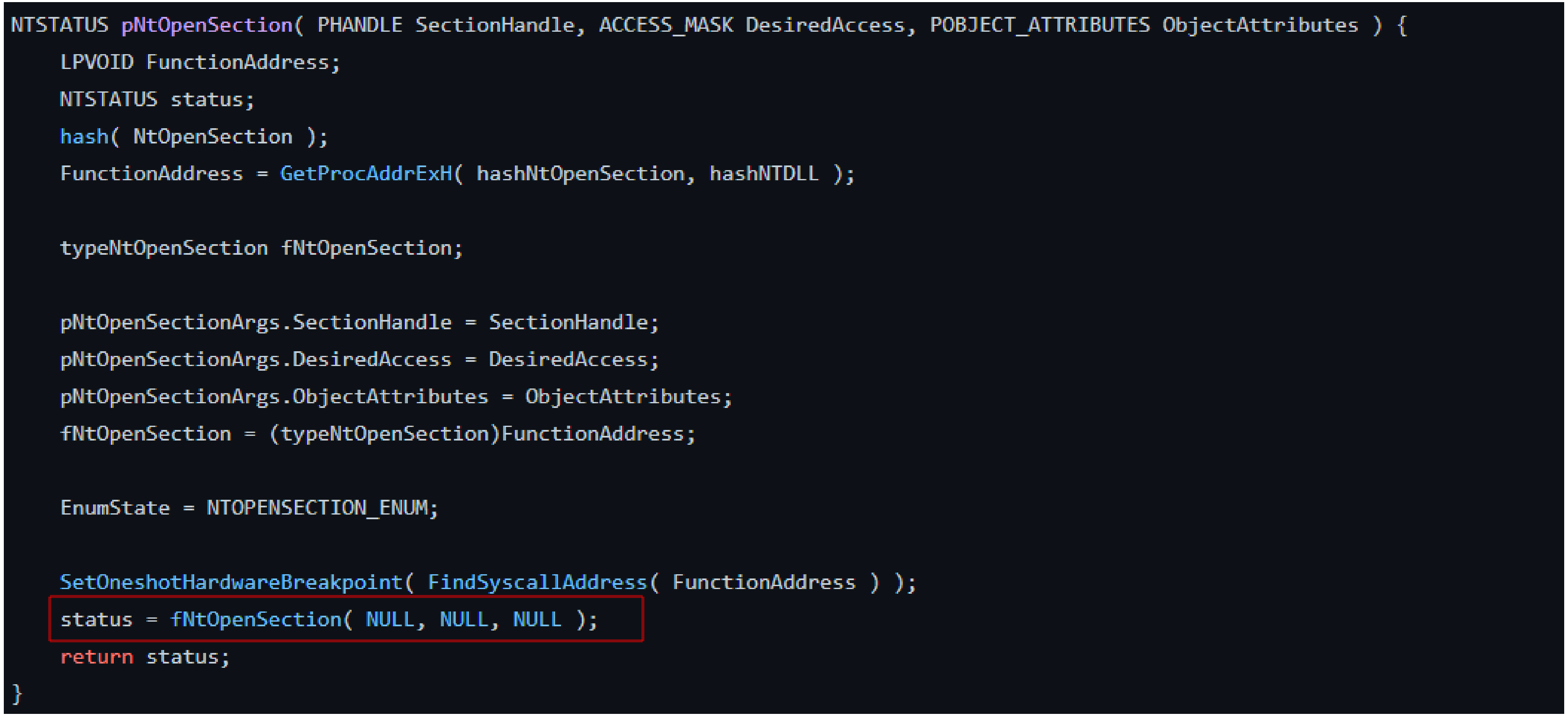
```
00007FFA2E8CD060 | 4C:8BD1                | mov r10,rcx
00007FFA2E8CD063 | E9 6FCF0700            | jmp ntdll.7FFA2E949FD7
00007FFA2E8CD068 | F60425 0803FE7F 01     | test byte ptr ds:[7FFE0308],1
00007FFA2E8CD070 | 75 03                  | jne ntdll.7FFA2E8CD075
00007FFA2E8CD072 | 0F05                   | syscall
00007FFA2E8CD074 | C3                     | ret
00007FFA2E8CD075 | CD 2E                  | int 2E
00007FFA2E8CD077 | C3                     | ret
```

```
Setup trampoline
mov r10, rcx
mov eax, SSN
```

```
000002569A959084 | EB BA     | jmp EDR.2569A959040
```

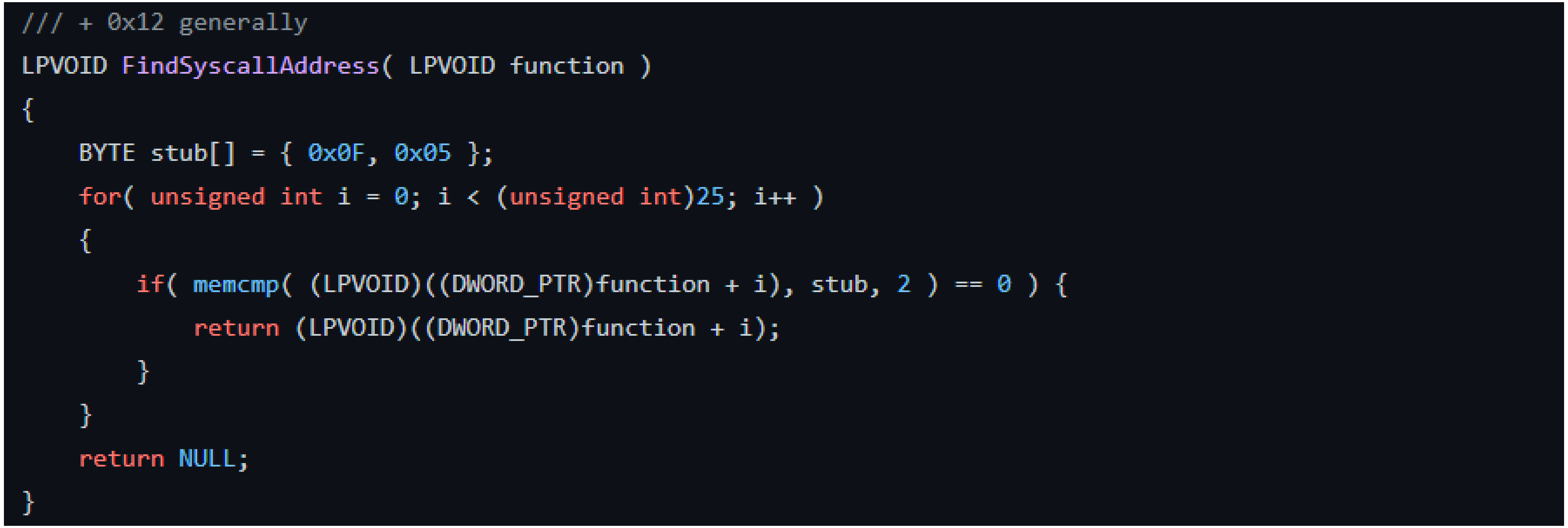
```
EDR Checks
```

An example



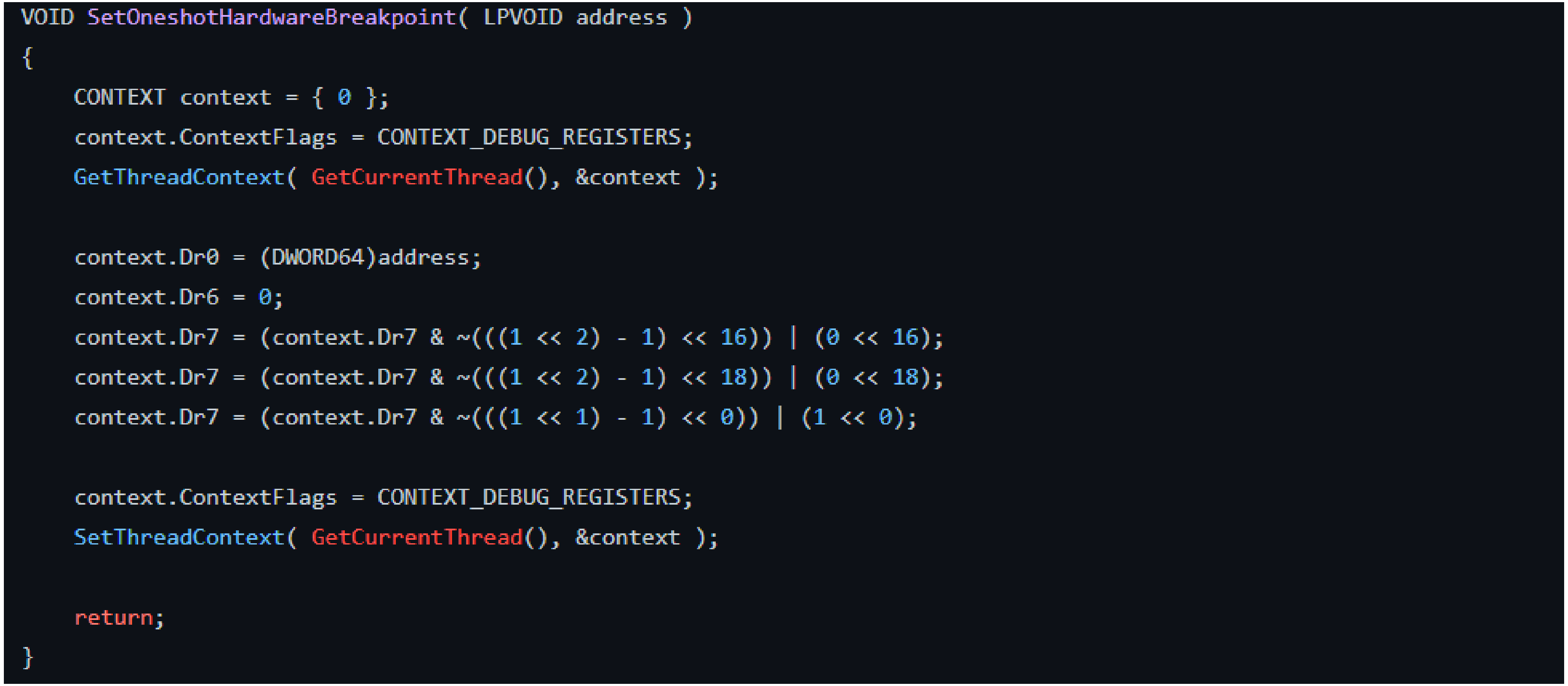
```
NTSTATUS pNtOpenSection( PHANDLE SectionHandle, ACCESS_MASK DesiredAccess, POBJECT_ATTRIBUTES ObjectAttributes ) {
    LPVOID FunctionAddress;
    NTSTATUS status;
    hash( NtOpenSection );
    FunctionAddress = GetProcAddrExH( hashNtOpenSection, hashNTDLL );

    typeNtOpenSection fNtOpenSection;

    pNtOpenSectionArgs.SectionHandle = SectionHandle;
    pNtOpenSectionArgs.DesiredAccess = DesiredAccess;
    pNtOpenSectionArgs.ObjectAttributes = ObjectAttributes;
    fNtOpenSection = (typeNtOpenSection)FunctionAddress;

    EnumState = NTOPENSECTION_ENUM;

    SetOneshotHardwareBreakpoint( FindSyscallAddress( FunctionAddress ) );
    status = fNtOpenSection( NULL, NULL, NULL );
    return status;
}
```

Locating the syscall address



```
/// + 0x12 generally
LPVOID FindSyscallAddress( LPVOID function )
{
    BYTE stub[] = { 0x0F, 0x05 };
    for( unsigned int i = 0; i < (unsigned int)25; i++ )
    {
        if( memcmp( (LPVOID)((DWORD_PTR)function + i), stub, 2 ) == 0 ) {
            return (LPVOID)((DWORD_PTR)function + i);
        }
    }
    return NULL;
}
```

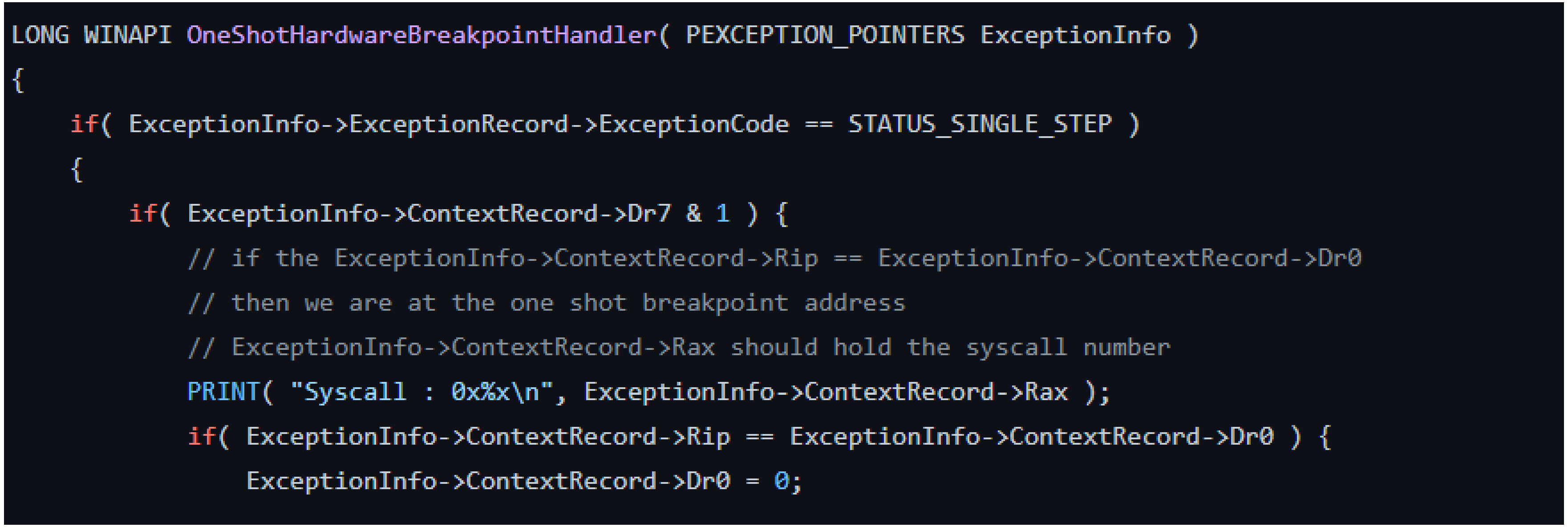
Setting a hardware breakpoint



```
VOID SetOneshotHardwareBreakpoint( LPVOID address )
{
    CONTEXT context = { 0 };
    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    GetThreadContext( GetCurrentThread(), &context );

    context.Dr0 = (DWORD64)address;
    context.Dr6 = 0;
    context.Dr7 = (context.Dr7 & ~(((1 << 2) - 1) << 16)) | (0 << 16);
    context.Dr7 = (context.Dr7 & ~(((1 << 2) - 1) << 18)) | (0 << 18);
    context.Dr7 = (context.Dr7 & ~(((1 << 1) - 1) << 0)) | (1 << 0);

    context.ContextFlags = CONTEXT_DEBUG_REGISTERS;
    SetThreadContext( GetCurrentThread(), &context );

    return;
}
```

Disabling the hardware breakpoint

- ★ The SSN is in EAX currently (returned by the trampoline if hooked)
- ★ Dr0 == RIP means this is the correct instruction we set a HWBP on



```
LONG WINAPI OneShotHardwareBreakpointHandler( PEXCEPTION_POINTERS ExceptionInfo )
{
    if( ExceptionInfo->ExceptionRecord->ExceptionCode == STATUS_SINGLE_STEP )
    {
        if( ExceptionInfo->ContextRecord->Dr7 & 1 ) {
            // if the ExceptionInfo->ContextRecord->Rip == ExceptionInfo->ContextRecord->Dr0
            // then we are at the one shot breakpoint address
            // ExceptionInfo->ContextRecord->Rax should hold the syscall number
            PRINT( "Syscall : 0x%x\n", ExceptionInfo->ContextRecord->Rax );
            if( ExceptionInfo->ContextRecord->Rip == ExceptionInfo->ContextRecord->Dr0 ) {
                ExceptionInfo->ContextRecord->Dr0 = 0;
```

Restoring the arguments

★ mov r10, rcx

```
case NTOPENSECTION_ENUM:
    ExceptionInfo->ContextRecord->R10 =
        (DWORD_PTR)((NtOpenSectionArgs*)(StateArray[EnumState].arguments))->SectionHandle;

    ExceptionInfo->ContextRecord->Rdx =
        (DWORD_PTR)((NtOpenSectionArgs*)(StateArray[EnumState].arguments))->DesiredAccess;

    ExceptionInfo->ContextRecord->R8 =
        (DWORD_PTR)((NtOpenSectionArgs*)(StateArray[EnumState].arguments))->ObjectAttributes;

    break;
```

Ruptura InfoSecurity

Using TamperingSyscalls

- https://github.com/rad9800/TamperingSyscalls
- Kudos to mez0 for collaborating to write gen.py
  - Generates the required code
  - `python gen.py NtOpenSection,NtMapViewOfSection,NtUnmapViewOfSection`
  - Switch for functions you need to use and call them as you would normally!
  - More information at fool.ish.wtf
- Dynamic syscall retrieval **only** available
  - https://github.com/rad9800/TamperingSyscalls/tree/stripped

Evaluation

- ★ Pass the userland hooks misinformation
  - https://fool.ish.wtf/2022/08/feeding-edrs-false-telemetry.html
  - TLDR; We can trivially modify the <=4 values
- ★ Goes through the userland hook (good) for highly monitored functions
- ★ Possible to spoof >4 arguments but I am simply LAZY
  - The more important arguments come first
- ★ The HWBP is cleared after it is used so capturing our thread CONTEXT would not show a Dr set
- ★ Possible detection with a hook on NtSetContextThread
  - This is the only function we can't tamper with (with HWBP)
    - ■ Would lead to a deadlock as it's used to set the HWBP

Ruptura InfoSecurity

Key Takeaways

- Exceptions are awesome
  - Helped deepen my understanding of x64 ABI
    - Inline assembly is really powerful!
  - Learnt a lot about HWBP and Drs, and the potential caveats around them
  - Took 2 days for API & 1 day for the syscalls
- Open sourcing your projects is awesome
  - Allows for better collaboration, and suggestions from far smarter people
  - Cool to see how detections could be built around your project
  - Helps shape your project and a understanding

Going forward

- ★ https://www.vx-underground.org/windows.html
- ★ https://codemachine.com/articles/x64_deep_dive.html
- ★ https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/
- ★ http://x86asm.net/articles/debugging-in-amd64-64-bit-mode-in-theory/