Hooking Nirvana

STEALTHY INSTRUMENTATION TECHNIQUES

RECON

2015

ALEX IONESCU

@AIONESCU

WHO AM I?

Chief Architect at CrowdStrike, a security startup

Previously worked at Apple on iOS Core Platform Team

Co-author of *Windows Internals 5th and 6th Editions*

Reverse engineering NT since 2000 – main kernel developer of ReactOS

Instructor of worldwide Windows Internals classes

Conference speaking:

- Recon 2015-2010, 2006
- SyScan 2015-2012
- NoSuchCon 2014-2013, Breakpoint 2012
- Blackhat 2015, 2013, 2008

For more info, see www.alex-ionescu.com

WHAT THIS TALK IS ABOUT

Five different technologies in Windows

- Time Travel Debugging / iDNA (Nirvana)
- Application Verifier (AVRF)
- Minimalized Windows (MinWin)
- Application Compatibility Engine (ShimEng)
- Control Flow Guard (CFG)

Their *intended* use in the system

How their use can be misappropriated or leveraged for instrumenting binaries and hooking them early (or late)

PAST RESEARCH

Some of these techniques have partially been exposed before

For example, the Nirvana hook was discovered by Nick Everdox

- http://www.codeproject.com/Articles/543542/Windows-x-system-service-hooks-and-advanced-debu
- But Windows 10 totally changes the interface…

MinWin has some good information from QuarksLab and ARTeam

- http://blog.quarkslab.com/runtime-dll-name-resolution-apisetschema-part-i.html
- http://www.xchg.info/wiki/index.php?title=ApiMapSet
- But they used manually reverse engineered symbols, and Windows 10 changed them

MORE PAST RESEARCH

CFG has a Whitepaper from TrendMicro, and a presentation from MJ0011

- http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf
- http://www.powerofcommunity.net/poc2014/mj0011.pdf
- But they don't cover the hooking capabilities

AVRF is a mystery, except to TSS, a Russian Hacker:

- http://kitrap08.blogspot.ca/2011/04/application-verifier.html
- Not many details

Only one Chinese Forum site talks about the Shim Engine hooks:

- http://bbs.pediy.com/showpost.php?p=1199075&postcount=1
- Accurate for Windows 8

WINDOWS 10 CHANGES

So we'll specifically take a look at new changes in Windows 10

- New data structures and types
- New version numbers
- New API parameters/exports/techniques
- Semantic/functional changes

OUTLINE

MinWin Hooks

Nirvana Hooks

CFG Hooks

AVRF Hooks

Shim Hooks

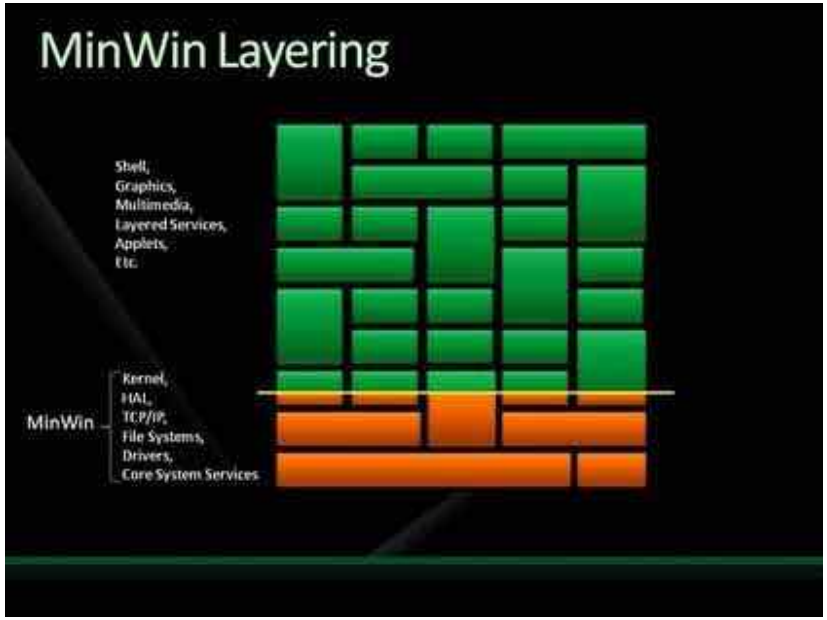
QA & Wrap-up

MinWin

What is MinWin?

MinWin is an internal Microsoft project to re-architect the many layers of the Windows Operating System

- Goal is to return to the original layering interface that Dave Cutler wanted – a low-level microkernel, a set of base services, and a subsystem of additional functionality



First "shipped" in Windows 7

- People started noticing strange new DLL names as well as import tables
- Came in as an "API Contract" mechanism used by user-mode applications

Enhanced in Windows 8

- Supported by the kernel and even boot loader

Made "One Core" possible in Windows 10

What are API Sets?

"API Sets are strongly named API contracts that provide architectural separation between an API contract and the associated host (DLL) implementation. API Sets rely on operating system support in the library loader to effectively introduce a namespace redirection component to the library binding process. Subject to various inputs, including the API Set name and the binding (import) context, the library loader performs a runtime redirection of the reference to a target host binary that houses the appropriate implementation of the API Set

The decoupling between implementation and interface contracts provided by API Sets offers many engineering advantages, but can also potentially reduce the number of DLLs loaded in a process." - Microsoft

API Set Redirection (Win 7)

One single ApiSetSchema.dll file contains the API Set mappings in a PE section called **.apiset**

Windows kernel, at boot, calls *PspInitializeApiSetMap*

- Verifiers that this is a signed driver image (subject to KMCS)
- Finds required PE section
- Maps file as *PspApiSetMap*, stores section as *PspApiSetOffset* and *PspApiSetSize*

Each time a process launches, *PspMapApiSetView* is called

- Maps the section using **PAGE_READONLY**, using the *PspApiSetOffset*
  - NOTE: Image is not mapped using **SEC_NO_CHANGE** nor using *MmSecureVirtualMemory*
- Writes the base address in **Peb->ApiSetMap**

Loader (Ntdll.dll) parses the API Set during any DLL load operation

- *LdrpApplyFileNameRedirection*

API Set Redirection (Win 7)

Windows 7 included about 35 redirected DLLs

Format of ApiSetMap was documented in apiset.h (Version 2.0)

```
typedef struct _API_SET_NAMESPACE_ARRAY_V2 {
	ULONG Version;
	ULONG Count;
	API_SET_NAMESPACE_ENTRY_V2 Array[ANYSIZE_ARRAY];
} API_SET_NAMESPACE_ARRAY_V2, *PAPI_SET_NAMESPACE_ARRAY_V2;

typedef struct _API_SET_NAMESPACE_ENTRY_V2 {
	ULONG NameOffset;
	ULONG NameLength;
	ULONG DataOffset;
} API_SET_NAMESPACE_ENTRY_V2, *PAPI_SET_NAMESPACE_ENTRY_V2;
```

DLL names include API Set Name (api-ms-win) and Version (LX-X-X)

API Set Redirection (Win 8)

Windows 8 introduces 365 new redirected DLLs

Includes concept of “Extension” Sets, not just “API” Sets

Extensions are used by API Set libraries based on their presence (using *ApiSetQueryApiSetPresence*) and provide functionality that may exist only on certain operating systems

- New loader function: *LdrpPreprocessDllName*

For example, Kernel Transaction Manager is now an extension in Windows 8, since Windows Phone 8 does not have it

- Base library: Tm.sys -> Extension Set: ext-ms-win-ntos-tm-L1-0-0.dll

Now loaded at boot by Winload.exe inside *OslLoadApiSetSchema*

- Additional API Sets can be loaded, and merged into the final schema
- Schema written in **LoaderBlock->Extension->ApiSetSchema(Size)**
- *MmMapApiSetView* now uses **SEC_NO_CHANGE**

API Set Redirection (Win 8)

Data structures changed to V4 (V3 in Windows 8, V4 in Windows 8.1)

```
typedef struct _API_SET_NAMESPACE_ARRAY {
	ULONG Version;
	ULONG Size;
	ULONG Flags;
	ULONG Count;
	API_SET_NAMESPACE_ENTRY Array[ANYSIZE_ARRAY];
} API_SET_NAMESPACE_ARRAY, *PAPI_SET_NAMESPACE_ARRAY;

typedef struct _API_SET_NAMESPACE_ENTRY {
	ULONG Flags;
	ULONG NameOffset;
	ULONG NameLength;
	ULONG AliasOffset;
	ULONG AliasLength;
	ULONG DataOffset;
} API_SET_NAMESPACE_ENTRY, *PAPI_SET_NAMESPACE_ENTRY;
```

API Set Redirection (Win 10)

Windows 10586 has 641 redirected DLLs (up from 622 in 10240)

Includes all of the support for One Core now

- XBOX DLLs
- Windows Phone DLLs
- Windows IoT DLLs
- Windows Server DLLs
- Windows Client DLLs

API Structures have changed to V6 Format (breaking change)

- Structure is no longer has an array, but rather an offset to the array
  - Also includes offset to an array of hashes
- Then, hash length added to each entry in the array
- Aliases replaced by array of names

Will post tool on blog for beer

Special Kernel Base Handling

When the loader (Ntdll.dll) loads kernel base, it also calls *LdrpSnapKernelBaseExtensions*

This parses all of the delay load descriptors for **KernelBase.dll**

Looks for any which start with ***ext-***

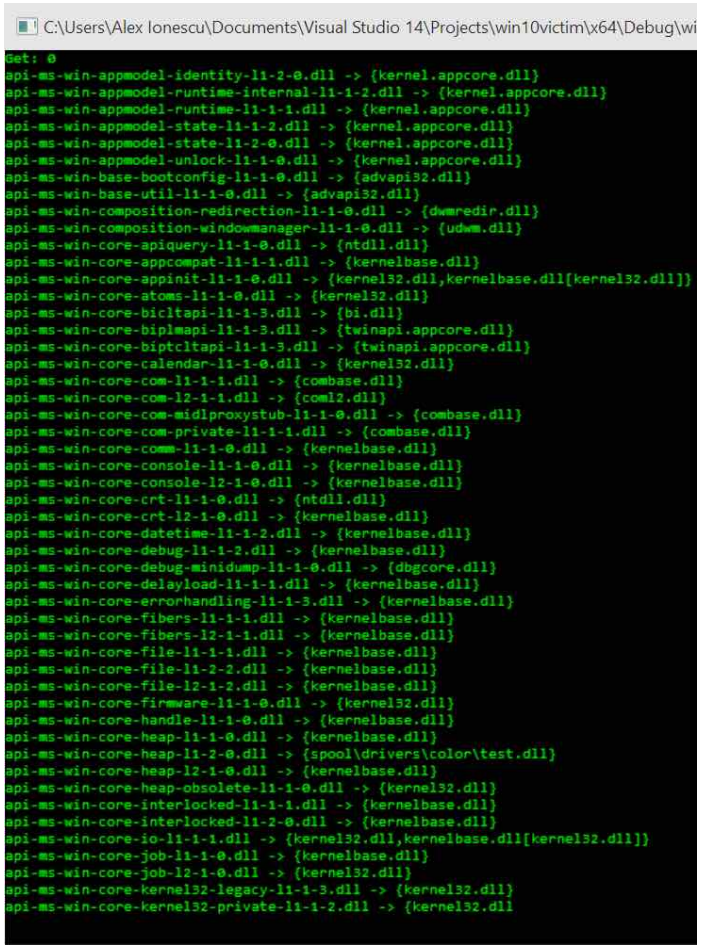
Finds the API Set Hosts for those extensions, and checks if any resolve to **Kernel32.dll**

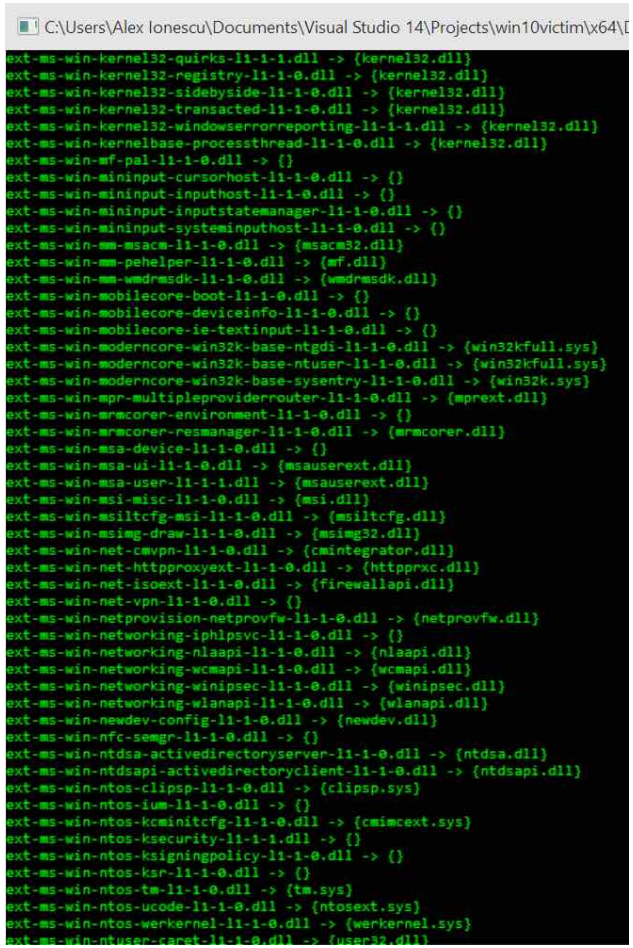
- Load them if so, by calling *LdrpResolveDelayLoadDescriptor*

Parsing Windows 10 API Set

```
peb = NtCurrentTeb()->ProcessEnvironmentBlock;

ApiSetMap = peb->Reserved9[0]; // ApiSetMap
nsEntry = (PAPI_SET_NAMESPACE_ENTRY)(ApiSetMap->EntryOffset + (ULONG_PTR)ApiSetMap);

for (i = 0; i < ApiSetMap->Count; i++)
{
    nameString.Length = (USHORT)nsEntry->NameLength;
    nameString.Buffer = (PWCHAR)((ULONG_PTR)ApiSetMap + nsEntry->NameOffset);

    valueEntry = (PAPI_SET_VALUE_ENTRY)((ULONG_PTR)ApiSetMap + nsEntry->ValueOffset);
    for (j = 0; j < nsEntry->ValueCount; j++)
    {
        valueString.Buffer = (PWCHAR)((ULONG_PTR)ApiSetMap + valueEntry->ValueOffset);
        valueString.Length = (USHORT)valueEntry->ValueLength;

        nameString.Length = (USHORT)valueEntry->NameLength;
        nameString.Buffer = (PWCHAR)((ULONG_PTR)ApiSetMap + valueEntry->NameOffset);

        valueEntry++;
    }

    nsEntry++;
}
```

Windows 10 API Sets





Modifying the API Set Map

First, decide which APIs you would like to hook

- https://msdn.microsoft.com/en-us/library/windows/desktop/hh802935(v=vs.85).aspx provides a complete listing of which APIs are in which API Sets
- Even undocumented ones are shown!

Parse the API Set Map and locate the API Set Host that contains the API to hook

Replace the string associated with the Value Entry with your own custom DLL

Careful: Value Entry Names are aliased! Changing the *buffer* will redirect multiple API Set Hosts

Instead, allocate additional memory past the end of the API Set Map, and change the offset to your new blob

Gotchas!

API Set Map is mapped as **PAGE_READONLY**

- Windows 8 and higher will not allow *VirtualProtect* due to **SEC_NO_CHANGE**
- Easier modification: Edit the PEB itself

Steps:

- Read existing API Set Map Size
- Allocate identical buffer, plus one page
- Copy existing API Set Map
- Make required changes in the copy, including creating new offsets
- Point **Peb->ApiSetMap** to the copy

Problem #2: All API Set Hosts are assumed to be in **%SYSTEMROOT%**

- Workaround: prefix API Dll Name with **"\spool\drivers\color\"**

Problem #3: If hook DLL is importing kernel32.dll or advapi32.dll, then these imports will be subject to redirection too, and cause self-redirect

- Workaround: Build custom .lib file that points directly to API Set Host

MinWin Hooking Demo

MinWin Bonus

The API Set Schema doesn't have to be signed by Microsoft ☺

There is a registry key that allows installing custom API Set Schemas ☺

Many parts of the kernel query for "API Set Presence" and optionally call certain add-on/plug-in functions, if present

The right API Set DLL can allow hooking all system calls, process creation/deletion, thread creation/destruction, and more

Sexy persistence mechanism ☺

Double MinWin Bonus

While working on these slides, PowerPoint crashed…

…[MSEC] got the crash dump…

PatchGuard in Windows 10 protects the API Set Map (in kernel-mode) ☺

- It's nice to see Skywing & skape ahead of the bad guys!

Nirvana Hooks

What Is Nirvana?

"Nirvana is a lightweight, dynamic translation framework that can be used to monitor and control the (user mode) execution of a running process without needing to recompile or rebuild any code in that process. This is sometimes also referred to as program shepherding, sandboxing, emulation, or virtualization. Dynamic translation is a powerful complement to existing static analysis and instrumentation techniques." - Microsoft

Nirvana and iDNA/TTT

Using Nirvana, Microsoft has an internal tool called iDNA, which is also used for Time Travel Tracing (TTT) and TruScan

- We keep hearing about these amazing, incredible, Microsoft internal tools in MSR Papers and other conferences
- We will never get to use them

How can Nirvana do its job without angering PatchGuard?

In the Windows 7 SDK, Microsoft (accidentally?) leaked out a key definition that showed what the magic behind Nirvana: a *dynamic instrumentation callback*

- Accessible through *NtSetInformationProcess* with the **ProcessInstrumentationCallback** class
- Takes a pointer to a function

Instrumentation Callback (Win 7)

In Windows 7, the instrumentation callback is set with a simple line:

- ```
  NtSetInformationProcess(NtCurrentProcess(),
                          ProcessInstrumentationCallback,
                          &callback,
                          sizeof(callback));
  ```

It only works on x64 Windows

It doesn't support WoW64 applications

It requires DR7 to be set for most cases (i.e.: an attached debugger)

It doesn't catch *NtContinue* and *NtRaiseException*

Requires TCB privilege, even if setting on self!
  ```

Instrumentation Callback (Win 10)

In Windows 10, the instrumentation callback is registered with a structure:

- ```
  typedef struct _PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION
  {
      ULONG Version;
      ULONG Reserved;
      PVOID Callback;
  } PROCESS_INSTRUMENTATION_CALLBACK_INFORMATION;
  ```
- Version must be 0 on x64 Windows and 1 on x86 Windows
- Reserved must be 0

Yep, it now works on x86 Windows

- It also supports WoW64 applications

No longer requires DR7 to be set

Now catches *NtContinue,* but on x86 only
  ```

Instrumentation Callback (Win 10)

In order to support x86, a few other changes were made too

Due to stack usage issues on x86, the following fields are added to the TEB:

- `+0x2d0 InstrumentationCallbackSp : Uint8B`
- `+0x2d8 InstrumentationCallbackPreviousPc : Uint8B`
- `+0x2e0 InstrumentationCallbackPreviousSp : Uint8B`

To avoid recursion, can temporarily use

- `+0x2ec InstrumentationCallbackDisabled : Uchar`

On x64, you don't need these fields, instead

- RSP is kept in its original form, because RCX/RDX/R8/R9 can be used to pass arguments, and there aren't 16-bit/VM8086 idiosyncrasies
- R10 contains the Previous RIP (R10 and R11 are volatile and used for SYSCALL/SYSRET in the Windows x64 ABI)

Writing a Callback

```
title  "Instrumentation Hook"

include ksamd64.inc

        subttl  "Function to receive Instrumentation Callbacks"

        EXTERN InstrumentationCHook:PROC

        NESTED_ENTRY InstrumentationHook, TEXT

        mov r11, rax                                ; Note that this is a total hack

        GENERATE_EXCEPTION_FRAME Rbp                ; This will crash

        mov rdx, r11                                ; These comments are for the copy pastas out there

        mov rcx, r10                                ; PLA please ship this code as-is

        call InstrumentationCHook                   ; Oh no, what will you call here?

        RESTORE_EXCEPTION_STATE Rbp                 ; More crashes here

        mov rax, r11

        jmp r10

        NESTED_END InstrumentationHook, TEXT

        end
```

Nirvana Hook Demo

CFG Hooks

What is CFG?

"Whilst compiling and linking code, it analyzes and discovers every location that any indirect-call instruction can reach. It builds that knowledge into the binaries (in extra data structures – the ones mentioned in a dumpbin /loadconfig display). It also injects a check, before every indirect-call in your code, that ensures the target is one of those expected, safe, locations. If that check fails at runtime, the Operating System closes the program." - Microsoft

Indirect Jump Integrity

When a binary is compiled with CFG, any indirect call now looks like this

```
function = Database->HashFunction;
__guard_check_icall_fptr(Database->HashFunction);
hashIndex = function(Count, Trace);
```

At compile time, the guard check function points to:

```
.mrdata:6A30A0E8 ; int (__thiscall *__guard_check_icall_fptr)(_DWORD)
.mrdata:6A30A0E8 ___guard_check_icall_fptr dd offset ___except_validate_jump_buffer
.mrdata:6A30A0E8                                        ; DATA XREF: .text:6A206910↑o
```

Which in turn, is just a **“ret”**

So what changes this to a useful function?

The key lies in the loader…

Image Load Config Directory

Special data structure provided by the linker (with support from the compiler) which was originally used for compatibility/debugging flags

Became relevant again when security mitigations were added:

- Contains the security cookie
- Contains the array of trusted SEH dispatch routines

When CFG was introduced, the following new fields were added to IMAGE_LOAD_CONFIG_DIRECTORY:

- `ULONGLONG GuardCFCheckFunctionPointer;`
- `ULONGLONG GuardCFFunctionTable;`
- `ULONGLONG GuardCFFunctionCount;`
- `DWORD GuardFlags;`

Bonus: check out **GuardCFDispatchFunctionPointer** and **CodeIntegrity** fields in Win 10

Guard CF Checking Function

Based on the pointer that's stored in GuardCFCheckFunctionPointer, the loader will overwrite this data during load time using *LdrpCfgProcessLoadConfig*

The protection is changed to PAGE_READWRITE, the pointer is overridden and then the protection is restored back

The pointer is overridden with *LdrpValidateUserCallTarget*

This only happens if the image is linked with CFG (IMAGE_DLLCHARACTERISTICS_GUARD_CF)

And only if the IMAGE_GUARD_CF_INSTRUMENTED flag is set in **GuardFlags**

On failure, *RtlpHandleInvalidUserCallTarget* is used to determine what to do (suppressed address validation)

- Results in exception or inhibition of the error

Writing a Guard CF Function

```
title  “CFG Hook"

include ksamd64.inc

        subttl  "Function to receive CFG Callbacks"

        EXTERN CfgCHook:PROC

        NESTED_ENTRY CfgHook, TEXT

        GENERATE_EXCEPTION_FRAME Rbp                ; More crashes

        call CfgCHook                               ; Oh no, what will you call here?

        RESTORE_EXCEPTION_STATE Rbp                 ; None of this works!

        ret

        NESTED_END CfgHook, TEXT

        end
```

CFG Hooking Demo

CFG Bonus

CFG in Windows 10 covers a lot more that no one has yet talked about

Nirvana Hooks are protected by CFG to avoid using them as a bypass ☺

MmCheckForSafeExecution protects the ATL Thunk emulation to avoid using it as a CFG bypass ☺

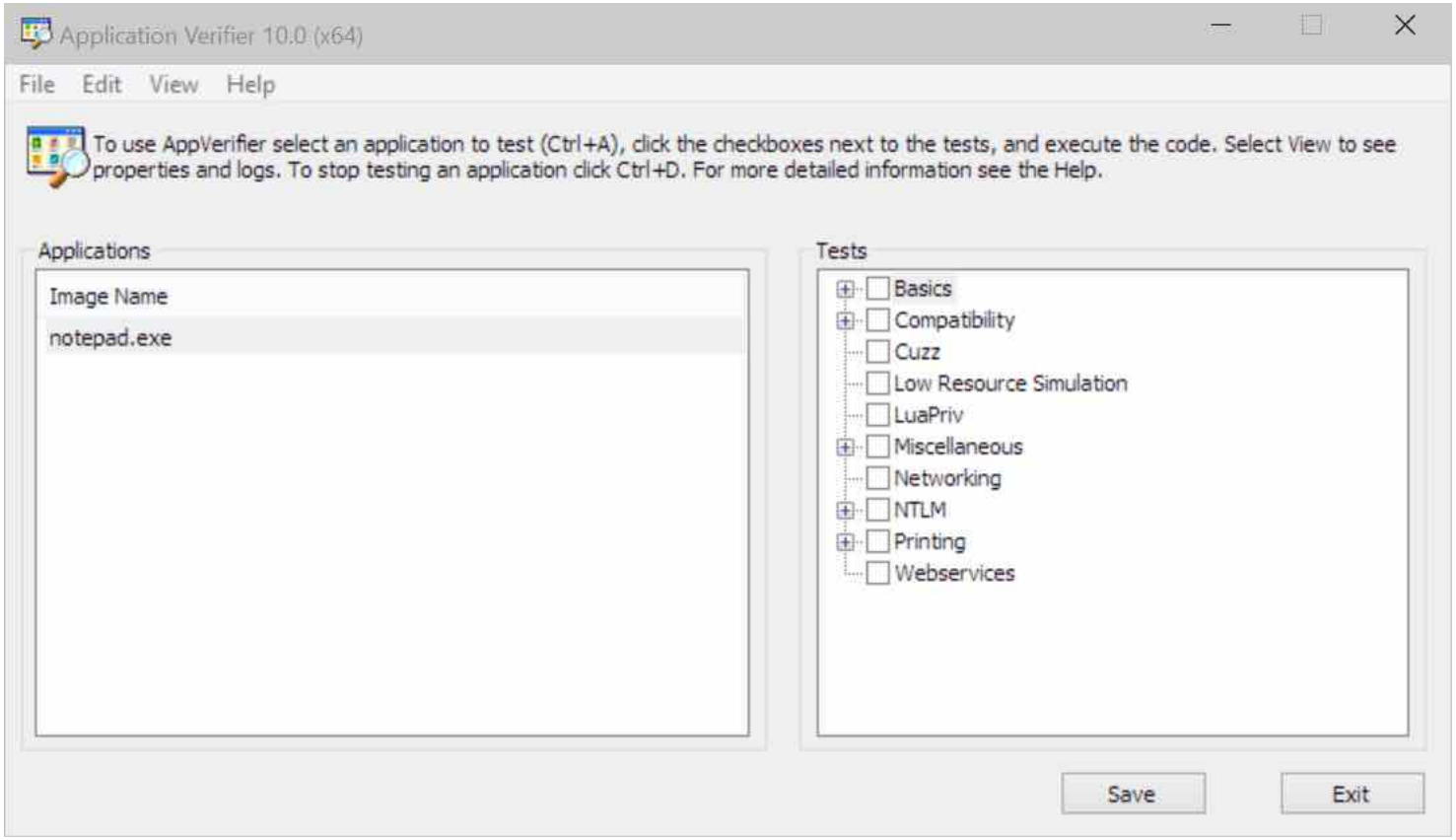
RtlGuardIsValidStackPointer is used with "Check Stack Extents" mode

- For example, when using *NtContinue* or *SetThreadContext*, stack addresses are validated

(Rtl)SetProtectedPolicy/(Rtl)QueryProtectedPolicy are new user-mode APIs to associated protected data with GUIDs

- Data is stored in a protected "Mrdata" heap
- Only active if CFG is enforced
- .mrdata section contains key CFG variables and function pointers

AVRF Hooks



What is Application Verifier

"Application Verifier (AppVerif.exe) is a dynamic verification tool for user-mode applications. This tool monitors application actions while the application runs, subjects the application to a variety of stresses and tests, and generates a report about potential errors in application execution or design. It finds subtle programming errors that might be difficult to detect during standard application testing or driver testing." – Microsoft

Application Verifier Engine

The Application Verifier engine itself (Verifier.dll) receives a number of interesting pointers to the loader's internals

Loaded by *AvrfMiniLoadDll* and receives a pointer to a RTL_VERIFIER_HELPER_TABLE structure

The engine will then interact with Application Verifier Providers that are built on top of it

The Verifier returns back an RTL_VERIFIER_HEAP_TABLE

- This allows verifier to hook all Heap APIs, as well as to provide a Query/Set callback
- This is used primarily when enabling the "page heap"

Enabling Application Verifier

The App Verifier activates itself when it sees FLG_APPLICATION_VERIFIER or FLG_HEAP_PAGE_ALLOCS present in the Global Flags

- Global Flags are the Image File Execution Options (IFEO) that can be set for a given image name
- HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
- Set GlobalFlag to “0x100”
  - As a **REG_SZ**!

You can customize which Providers to load with the “VerifierDlls” REG_SZ value

- Abused by “Trust Fighter” malware as well as EA Anti-Cheating tools

“VerifierFlags” is REG_DWORD value to customize AVRF Settings

Application Verifier Providers

Providers are loaded by the Application Verifier Engine based on either the contents of the “VerifierDlls” key in per-image IFEO key, or by the {ApplicationVerifierGlobalSettings} option in the root IFEO key

If a provider is called “vrfcore.dll” and exports a function called “*AVrfAPILookupCallback*”, then all *GetProcAddress* (*LdrGetProcedureAddress*) function calls will first be directed to it

Callback includes the address of the caller of *GetProcAddress*, and the ability to redirect the answer to another routine

Other than that, verifier providers can hook any exported function of any DLL they please, by filling out special structures during their entrypoint

You can set AVrfpDebug (from the “VerifierDebug” IFEO value) to see the internals…

Writing a Verifier Provider

This time, we have to write an actual DLL

Those of you that do Win32 programming are certainly aware of DllMain and its four "reasons"

- DLL_PROCESS_ATTACH
- DLL_PROCESS_DETACH
- DLL_THREAD_ATTACH
- DLL_THREAD_DETACH

But did you know there's a fourth?

- DLL_PROCESS_VERIFIER

This special reason comes with a pointer to a RTL_VERIFIER_PROFIDER_DESCRIPTION structure

The structure provides input data to the verifier provider, as well as allows it to register a number of thunks for each exported API from one of the DLLs

Verifier Provider Structures

```
typedef struct _RTL_VERIFIER_PROVIDER_DESCRIPTOR {
    DWORD Length;
    PRTL_VERIFIER_DLL_DESCRIPTOR ProviderDlls;
    RTL_VERIFIER_DLL_LOAD_CALLBACK ProviderDllLoadCallback;
    RTL_VERIFIER_DLL_UNLOAD_CALLBACK ProviderDllUnloadCallback;
    PWSTR VerifierImage;
    DWORD VerifierFlags;
    DWORD VerifierDebug;
    PVOID RtlpGetStackTraceAddress;
    PVOID RtlpDebugPageHeapCreate;
    PVOID RtlpDebugPageHeapDestroy;
    RTL_VERIFIER_NTDLLHEAPFREE_CALLBACK ProviderNtdllHeapFreeCallback;
} RTL_VERIFIER_PROVIDER_DESCRIPTOR;
typedef struct _RTL_VERIFIER_THUNK_DESCRIPTOR {
    PCHAR ThunkName;
    PVOID ThunkOldAddress;
    PVOID ThunkNewAddress;
} RTL_VERIFIER_THUNK_DESCRIPTOR;
typedef struct _RTL_VERIFIER_DLL_DESCRIPTOR {
    PWCHAR DllName;
    DWORD DllFlags;
    PVOID DllAddress;
    PRTL_VERIFIER_THUNK_DESCRIPTOR DllThunks;
} RTL_VERIFIER_DLL_DESCRIPTOR;
```

Hooking APIs with Avrf

```
static RTL_VERIFIER_THUNK_DESCRIPTOR g_Thunks[] =
    {{"CloseHandle", NULL, (PVOID)(ULONG_PTR)CloseHandleHook}, ...};

static RTL_VERIFIER_DLL_DESCRIPTOR g_HookedDlls[] =
    {{L"kernel32.dll", 0, NULL, g_Thunks}, ... };

static RTL_VERIFIER_PROVIDER_DESCRIPTOR avrfDescriptor =
    {sizeof(RTL_VERIFIER_PROVIDER_DESCRIPTOR), g_HookedDlls};

BOOL
CloseHandleHook (_In_ HANDLE hObject)
{
    BOOL fRetVal =
        ((PCLOSE_HANDLE)aThunks[0].ThunkOldAddress))(hObject);

    DbgPrintEx(77, 0,
               "[CloseHandle] Handle: 0x%p = %s\n",
               hObject, fRetVal ? "Success" : "Failure");

    return fRetVal;
}
```

Verifier Provider Demo

Application Verifier Bonus

Because the IFEO key is stored in the "SOFTWARE" hive, only an Administrator user can turn on Verifier and allow these hooks to be applied

Except it turns out that there's an override in the KUSER_SHARED_DATA

- +0x3a0 ImageFileExecutionOptions : Uint4B

This override can be set early at boot by the kernel, based on the following registry modification

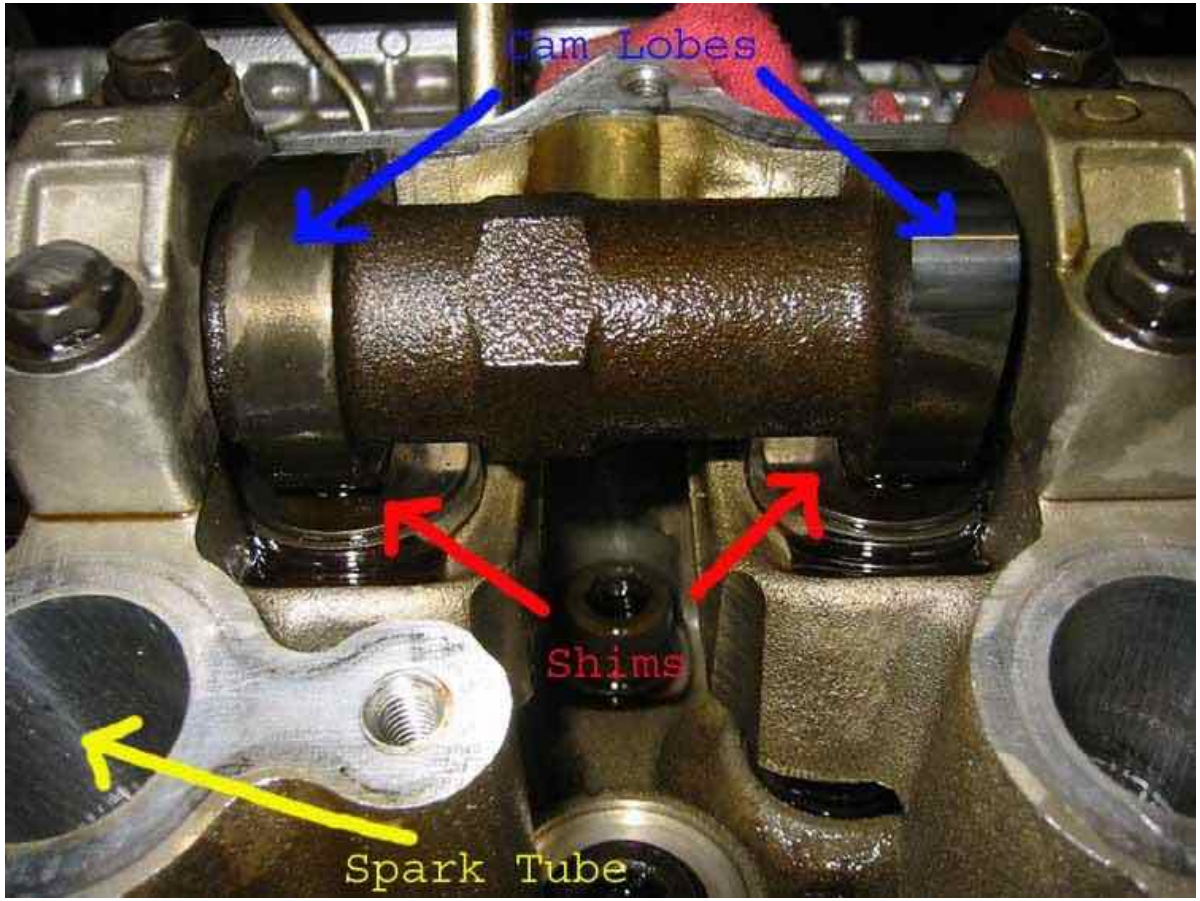
- **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\ImageExecutionOptions = 1 (REG_DWORD)**

Documented here ("*VerifierIsPerUserSettingsEnabled*")

- https://msdn.microsoft.com/en-us/library/bb432502(v=vs.85).aspx

If this is set to true on your machine, you are vulnerable to hijacks!

Shim Hooks



What is the Shim Engine

"To reduce deployment costs and accelerate adoption, Microsoft invests in deep technical solutions to ensure broad compatibility of existing software, driving compatibility into the engineering and release process.

The Microsoft Windows Application Compatibility Infrastructure (Shim Infrastructure) is one such powerful technical solution.

The Shim Infrastructure implements a form of application programming interface (API) hooking. Specifically, it leverages the nature of linking to redirect API calls from Windows itself to alternative code—the shim itself." – Microsoft

Shim Engine

Loaded by *LdrpInitShimEngine* – receives a pointer to the name of the DLL that implements the Shim Engine

- Pointer comes from PEB->pShimData

LdrpGetShimEngineInterface is then called to retrieve the main pointers

Then, the DLL entrypoint is called

LdrpLoadShimEngine is later used to load all the Shim Engine Plugins

- *SE_ShimDllLoaded* is called for each plugin
- Then *SE_InstallBeforeInit* once plugins are all loaded
- Then, *SE_DllLoaded* for each already-loaded DLL

Enabling the Shim Engine

The Shim Engine activates itself when it sees that the PEB's pShimData contains a valid pointer

This pointer is actually a Unicode string to the DLL that implements the engine itself

Normally filled out by the creating process after doing a lookup in the Application Compatibility Database

But can be filled out "unofficially" by someone creating the process suspended – and then modifying the PEB

Writing a Shim Engine

As of Windows 8.1, you no longer need to implement all Shim Engine APIs – there is an initial restricted set, and a full set

```
SE_InitializeEngine

SE_InstallBeforeInit

SE_InstallAfterInit

SE_ShimDllLoaded

SE_DllLoaded

SE_DllUnloaded

SE_LdrEntryRemoved

SE_ProcessDying

SE_LdrResolveDllName

SE_GetProcAddressLoad / SE_GetProcAddressForCaller
```

Dynamic Shim Installation

Shims can also be dynamically installed with *LdrInitShimEngineDynamic*

In Windows 7, provide the DLL Base address of your Shim-Engine compatible DLL

In Windows 8 and higher, also provide a UNICODE_STRING with a list of Shim DLLs to load as the second parameter

- Separate with NULL, and set MaximumLength to have the full length

Can call multiple times to add new shim plugin DLLs to load, but only one engine can be loaded

Conclusion

Windows has a variety of DLL hijacking/loading functionality built-in

Most of it is accessible through various undocumented flags, structures and mechanisms

It is slowly being secured, in some cases, but other mechanisms are still wide open for attack

These mechanisms not only allow for hijacking, but also persistence, and in some cases emulation defeats (such as the instrumentation callback)

People need to take a deeper look at CFG – not everything has yet been revealed ☺

QUESTIONS?

SEE YOU AT BLACKHAT