

PE continued

Xeno Kovah – 2012
xkovah at gmail

See notes for citation

1

Image from: <http://upload.moldova.org/movie/2007/dec/bee.jpg>

All materials are licensed under a Creative Commons “Share Alike” license.

- <http://creativecommons.org/licenses/by-sa/3.0/>

You are free:



to **Share** — to copy, distribute and transmit the work



to **Remix** — to adapt the work

Under the following conditions:



Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

2

Attribution conditions: Just state author name and where the slides were obtained from

Bound Imports

- Import binding is a speed optimization. The addresses of the functions are resolved at link time, and then placed into the IAT.
- The binding is done under the assumption of specific versions of the DLL. If the DLL changes, then all the IAT entries will be invalid. But that just means you have to resolve them, so you're not much worse off than if you had not used binding in the first place.
- notepad.exe and a bunch of other stuff in C:\WINDOWS\system32 are examples

Import Descriptor

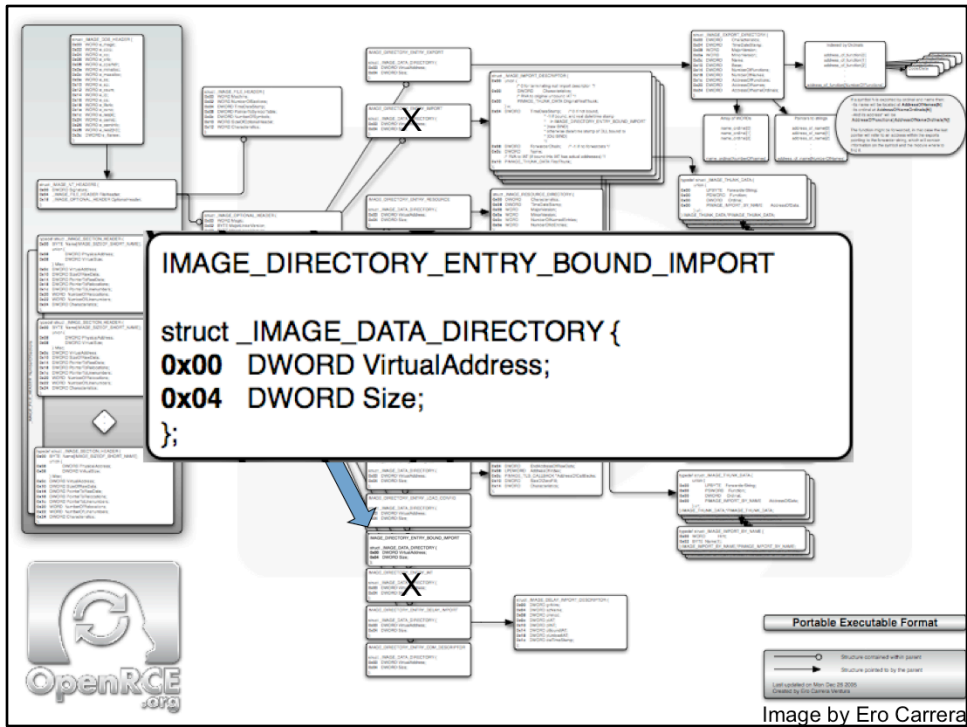
(from winnt.h)

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics;           // 0 for terminating null import descriptor
        DWORD OriginalFirstThunk;       // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
    DWORD TimeDateStamp;                 // 0 if not bound,
                                        // -1 if bound, and real date\time stamp
                                        // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                                        // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD ForwarderChain;                 // -1 if no forwarders
    DWORD Name;
    DWORD FirstThunk;                 // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
```

I think they meant "INT" →

- While the things in blue are the fields filled in for the most common case, we will actually have to understand everything for this structure, because you could run into all the variations.



Missing from the picture

- The bound import data directory entry points at an array of IMAGE_BOUND_IMPORT_DESCRIPTORs, ending with an all-zeros IMAGE_BOUND_IMPORT_DESCRIPTOR (like what was done with IMAGE_IMPORT_DESCRIPTOR)

```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD TimeDateStamp;
    WORD  OffsetModuleName;
    WORD  NumberOfModuleForwarderRefs;
    // Array of zero or more IMAGE_BOUND_FORWARDER_REF follows
} IMAGE_BOUND_IMPORT_DESCRIPTOR, *PIMAGE_BOUND_IMPORT_DESCRIPTOR;

typedef struct _IMAGE_BOUND_FORWARDER_REF {
    DWORD TimeDateStamp;
    WORD  OffsetModuleName;
    WORD  Reserved;
} IMAGE_BOUND_FORWARDER_REF, *PIMAGE_BOUND_FORWARDER_REF;
```

IMAGE_BOUND_IMPORT_DESCRIPTOR

- **TimeStamp** is just the value from the Exports information of the DLL which is being imported from, as we will see later
- **OffsetModuleName** is *not* a base-relative RVA, it's the offset from the beginning of the first IMAGE_BOUND_IMPORT_DESCRIPTOR
- We are going to return to NumberOfModuleForwarderRefs and IMAGE_BOUND_FORWARDER_REF after we learn about forwarded functions.

7

Notepad.exe's IMAGE_BOUND_IMPORT_DESCRIPTOR array

	VA	Data	Description	Value
notepad.exe	01000250	4802A0C9	Time Date Stamp	2008/04/14 Mon 00:09:45 UTC
IMAGE_DOS_HEADER	01000254	0058	Offset to Module Name	comdlg32.dll
MS-DOS Stub Program	01000256	0000	Number of Module Forwarder Refs	
IMAGE_NT_HEADERS	01000258	4802A111	Time Date Stamp	2008/04/14 Mon 00:10:57 UTC
Signature	0100025C	0065	Offset to Module Name	SHELL32.dll
IMAGE_FILE_HEADER	0100025E	0000	Number of Module Forwarder Refs	
IMAGE_OPTIONAL_HEADER	01000260	4802A127	Time Date Stamp	2008/04/14 Mon 00:11:19 UTC
IMAGE_SECTION_HEADER .text	01000264	0071	Offset to Module Name	WINSPPOOL.DRV
IMAGE_SECTION_HEADER .data	01000266	0000	Number of Module Forwarder Refs	
IMAGE_SECTION_HEADER .rsrc	01000268	4802A094	Time Date Stamp	2008/04/14 Mon 00:08:52 UTC
BOUND_IMPORT Directory Table	0100026C	007E	Offset to Module Name	COMCTL32.dll
BOUND_IMPORT DLL Names	0100026E	0000	Number of Module Forwarder Refs	
SECTION .text	01000270	4802A094	Time Date Stamp	2008/04/14 Mon 00:08:52 UTC
SECTION .data	01000274	008B	Offset to Module Name	msvcrt.dll
SECTION .rsrc	01000276	0000	Number of Module Forwarder Refs	
	01000278	4802A0B2	Time Date Stamp	2008/04/14 Mon 00:09:22 UTC
	0100027C	0096	Offset to Module Name	ADVAPI32.dll
	0100027E	0000	Number of Module Forwarder Refs	
	01000280	4802A12C	Time Date Stamp	2008/04/14 Mon 00:11:24 UTC
	01000284	00A3	Offset to Module Name	KERNEL32.dll
	01000286	0001	Number of Module Forwarder Refs	
	01000288	4802A12C	Time Date Stamp	2008/04/14 Mon 00:11:24 UTC
	0100028C	00B0	Offset to Module Name	NTDLL.DLL
	0100028E	0000	Reserved	
	01000290	4802A0BE	Time Date Stamp	2008/04/14 Mon 00:09:34 UTC
	01000294	00BA	Offset to Module Name	GDI32.dll
	01000296	0000	Number of Module Forwarder Refs	
	01000298	4802A11B	Time Date Stamp	2008/04/14 Mon 00:11:07 UTC
	0100029C	00C4	Offset to Module Name	USER32.dll
	0100029E	0000	Number of Module Forwarder Refs	
	010002A0	00000000		
	010002A4	0000		
	010002A6	0000		

Non-zero number of forwarder refs

Therefore this ntdll entry is a
IMAGE_BOUND_FORWARDER_REF
Not a
IMAGE_BOUND_IMPORT_DESCRIPTOR
... I didn't notice it at first :)

Teaching to the test

- Although I told you to ignore `NumberOfModuleForwarderRefs...`
- For purposes of the game later, when I ask you how many `IMAGE_BOUND_IMPORT_DESCRIPTOR` structures are in the bound import directory table, you should NOT count `NumberOfModuleForwarderRefs` worth of entries after a thing that has. Also don't count the null terminating entry

9

NEW 2012

So how many IMAGE_BOUND_IMPORT_DESCRIPTORs here?

	VA	Data	Description	Value
notepad.exe	01000250	4802A0C9	Time Date Stamp	2008/04/14 Mon 00:09:45 UTC
IMAGE_DOS_HEADER	01000254	0058	Offset to Module Name	comdlg32.dll
MS-DOS Stub Program	01000256	0000	Number of Module Forwarder Refs	
IMAGE_NT_HEADERS	01000258	4802A111	Time Date Stamp	2008/04/14 Mon 00:10:57 UTC
Signature	0100025C	0065	Offset to Module Name	SHELL32.dll
IMAGE_FILE_HEADER	0100025E	0000	Number of Module Forwarder Refs	
IMAGE_OPTIONAL_HEADER	01000260	4802A127	Time Date Stamp	2008/04/14 Mon 00:11:19 UTC
IMAGE_SECTION_HEADER .text	01000264	0071	Offset to Module Name	WINSPOOL.DRV
IMAGE_SECTION_HEADER .data	01000266	0000	Number of Module Forwarder Refs	
IMAGE_SECTION_HEADER .rsrc	01000268	4802A094	Time Date Stamp	2008/04/14 Mon 00:08:52 UTC
BOUND_IMPORT Directory Table	0100026C	007E	Offset to Module Name	COMCTL32.dll
BOUND_IMPORT DLL Names	0100026E	0000	Number of Module Forwarder Refs	
SECTION .text	01000270	4802A094	Time Date Stamp	2008/04/14 Mon 00:08:52 UTC
SECTION .data	01000274	008B	Offset to Module Name	msvcrt.dll
SECTION .rsrc	01000276	0000	Number of Module Forwarder Refs	
	01000278	4802A0B2	Time Date Stamp	2008/04/14 Mon 00:09:22 UTC
	0100027C	0096	Offset to Module Name	ADVAPI32.dll
	0100027E	0000	Number of Module Forwarder Refs	
	01000280	4802A12C	Time Date Stamp	2008/04/14 Mon 00:11:24 UTC
	01000284	00A3	Offset to Module Name	KERNEL32.dll
	01000286	0001	Number of Module Forwarder Refs	
	01000288	4802A12C	Time Date Stamp	2008/04/14 Mon 00:11:24 UTC
	0100028C	00B0	Offset to Module Name	NTDLL.DLL
	0100028E	0000	Reserved	
	01000290	4802A0BE	Time Date Stamp	2008/04/14 Mon 00:09:34 UTC
	01000294	00BA	Offset to Module Name	GDI32.dll
	01000296	0000	Number of Module Forwarder Refs	
	01000298	4802A11B	Time Date Stamp	2008/04/14 Mon 00:11:07 UTC
	0100029C	00C4	Offset to Module Name	USER32.dll
	0100029E	0000	Number of Module Forwarder Refs	
	010002A0	00000000		
	010002A4	0000		
	010002A6	0000		

Non-zero number of forwarder refs →

Therefore this ntdll entry is a
IMAGE_BOUND_FORWARDER_REF
Not a
IMAGE_BOUND_IMPORT_DESCRIPTOR
... I didn't notice it at first :)

NEW 2012

Answer = 9

Notepad.exe's IMAGE_IMPORT_DESCRIPTORs and IAT with bound imports

CFF Explorer VII - [notepad.exe]

File Settings ?

notepad.exe

File: notepad.exe

- Dos Header
- Nt Headers
 - File Header
 - Optional Header
 - Data Directories [x]
- Section Headers [x]
- Import Directory
- Resource Directory
- Exception Directory
- Relocation Directory
- Debug Directory
- Address Converter
- Dependency Walker
- Hex Editor
- Identifier
- Import Adder
- Quick Disassembler
- Rebuilder
- Resource Editor

Module Name	Imports	OFTs	TimeDateStamp	ForwarderChain	Name RVA	FTs (IAT)
0000BFD4	N/A	0000BDF8	0000BDFC	0000BE00	0000BE04	0000BE08
szAnsi	(nFunctions)	Dword	Dword	Dword	Dword	Dword
ADVAPI32.dll	10	0000D1E8	FFFFFFFF	FFFFFFFF	0000D1D4	0000C000
KERNEL32.dll	67	0000D240	FFFFFFFF	FFFFFFFF	0000D1C4	0000C058
GDI32.dll	22	0000D460	FFFFFFFF	FFFFFFFF	0000D1B8	0000C278
USER32.dll	75	0000D518	FFFFFFFF	FFFFFFFF	0000D1AC	0000C330
msvcrt.dll	22	0000D778	FFFFFFFF	FFFFFFFF	0000D1A0	0000C590
COMDLG32.dll	9	0000D830	FFFFFFFF	FFFFFFFF	0000D190	0000C648

OFTs	FTs (IAT)	Hint	Name
Qword	Qword	Word	szAnsi
000000000000D9D8	000007FF7FF21ED0	027E	RegSetValueExW
000000000000D9EA	000007FF7FF2C2D0	026E	RegQueryValueExW
000000000000D9FE	000007FF7FF21F00	023C	RegCreateKeyW
000000000000DA0E	000007FF7FF30710	0230	RegCloseKey
000000000000DA1C	000007FF7FF306F0	0261	RegOpenKeyExW
000000000000DA2C	000007FF7FF30720	0180	IsTextUnicode

11

How does one go about binding imports?

- BindImageEx API, if you want to make your own program to bind your other programs (why?)
- Windows Installer "BindImage" action – ideal case, you bind at install time, so it will be correct until the next update of Windows.
- Bind.exe? Can't find it on my dev VM (VC++ 9.0, i.e. 2008 edition) but there's plenty of references to it in older documents (e.g. VC++ 6.0). Seems to be deprecated.
- However, we can use CFF Explorer, so let's do that to our hello world quick:
 - Open HelloWorld.exe in CFF Explorer.exe
 - Goto Data Directories [x] and note the zeros for Bound Import Directory RVA/Size.
 - Goto Import Directory and select kernel32.dll. Note the values in the FTs (IAT) column.
 - Go to "Rebuilder" helper plugin, select "Bind Import Table" only and then select "Rebuild"
 - Go back to the Data Directories to see the non-zero Bound Import Directory RVA and go to the Import Directory area to see the absolute VAs for the imported function addresses.

12

Example things mentioning bind.exe

<http://www.codeproject.com/KB/DLL/NeedBind.aspx>

<http://msdn.microsoft.com/en-us/magazine/cc301808.aspx>

Binding vs. ASLR: THERE CAN BE ONLY ONE!

- Address Space Layout Randomization makes binding pointless, because if the ASLR is doing its job, the bindings should be invalidated most of the time. So you end up being forced to resolve imports at load time anyway, and therefore any time you took to try and validate bound imports was pointless, so you may as well just not even use them.
- This is why I'm pretty sure binding is (going to be?) deprecated, and why bind.exe disappeared.



See notes for citation

13

<http://www.elfwood.com/~tommartin/Highlander.3294669.html>

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

From winnt.h

Which fields do we even care about, and why?



14



Delay Loaded DLLs

- Specifies that libraries will not even be loaded into the memory space until the first time they are used. This can potentially be a good thing to do for code
- Setting this option will generate extra information separate from normal DLL loading information to support the delayed loading.

- Linker
 - General
 - Input
 - Manifest File
 - Debugging
 - System
 - Optimization
 - Embedded IDL
 - Advanced
 - Command Line
- Manifest Tool
- XML Document Generator
- Browse Information
- Build Events
- Custom Build Step

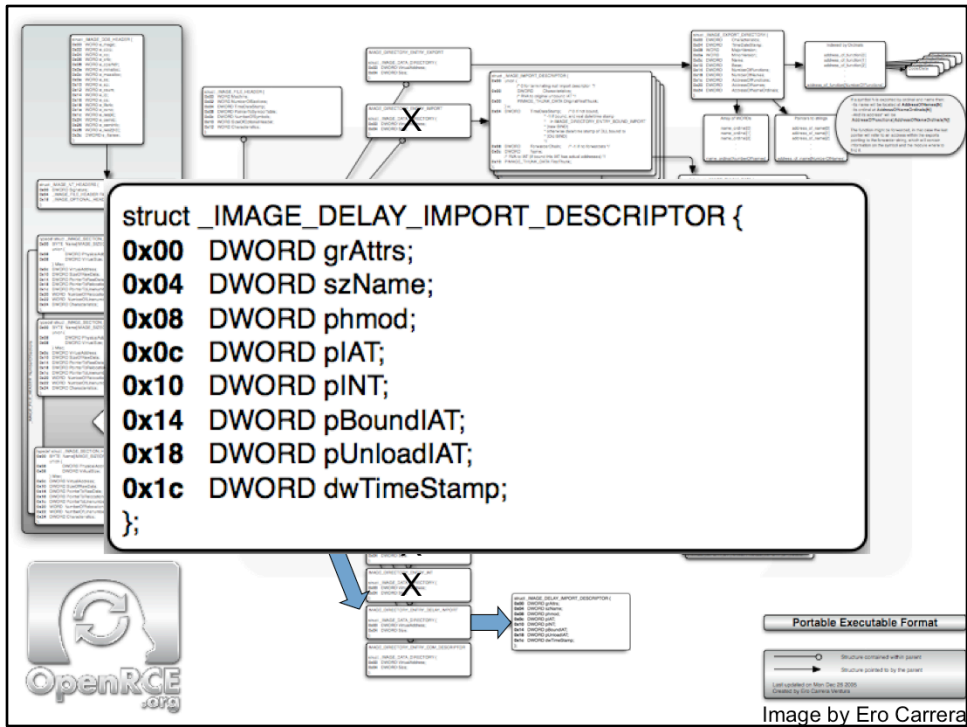
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Delay Loaded DLLs
Specifies one or more DLLs for delayed loading; use semi-colon delimited list if more than one. (/DELAYLOAD:[dll_name])

- Manifest File
 - Debugging
 - System
 - Optimization
 - Embedded IDL
 - Advanced
 - Command Line
- Manifest Tool
- XML Document Generator
- Browse Information
- Build Events
- Custom Build Step

Delay Loaded DLL	Don't Support Unload
Import Library	Don't Support Unload
Merge Sections	Support Unload (/DELAY:UNLOAD)
Target Machine	ARM64T802 (/MACHINE:X66)
Profile	No
CLR Thread Attribute	No threading attribute set
CLR Image Type	Default image type
Key File	
Key Container	
Delay Sign	No
Error Reporting	Prompt Immediately (/ERRORREPORT)
CLR Unmanaged Code Check	No

Delay Loaded DLL
Specifies to allow explicit unloading of the delayed load DLLs. (/DELAY:UNLOAD)



Delayed Imports

from DelayImp.H, dunno where he got _IMAGE_DELAY_IMPORT_DESCRIPTOR from

```
typedef struct ImgDelayDescr {
    DWORD      grAttrs;          // attributes
    RVA        rvaDLLName;      // RVA to dll name
    RVA        rvaHmod;         // RVA of module handle
    RVA        rvaIAT;          // RVA of the IAT
    RVA        rvaINT;          // RVA of the INT
    RVA        rvaBoundIAT;     // RVA of the optional bound IAT
    RVA        rvaUnloadIAT;    // RVA of optional copy of original IAT
    DWORD      dwTimeStamp;     // 0 if not bound,
                                // O.W. date/time stamp of DLL bound to (Old BIND)
} ImgDelayDescr, * PImgDelayDescr;
```

- We care about `rvaIAT` because it points at a separate IAT where stuff gets filled in as needed.
- Also `rvaDLLName` just because, you know, it tells us which DLL this is about.
- You can look up the rest on your own later (I recommend you check <http://msdn.microsoft.com/en-us/magazine/cc301808.aspx>), but really these fields are just there for the dynamic linker's benefit, so we don't care enough to go into any of them. The main takeaway will be about the procedure for resolving delayed imports.

The Delay-Loaded IAT

- We care about [rvalIAT](#) because this points to a **separate** IAT for delay-loaded functions only. But it's that IAT which is interesting.
- Initially the delay load IAT holds full virtual addresses of stub code. So the first time you call the delay-loaded function, it first calls the stub code.
- If necessary, the stub code loads the module which contains the function you want to call. Then it and resolves the address of the function within the module. It fills that address into the delay load IAT, and then calls the desired function. So the second time the code calls the function, it bypasses the dynamic resolution process, and just goes directly to the desired function.
- You can look up the rest on your own later (I recommend you check <http://msdn.microsoft.com/en-us/magazine/cc301808.aspx>), but these fields are mostly just there for the dynamic linker's benefit, so we don't care enough to go into them.

Delay Loading

hello
[...]
[...]

```
.text
...
call [0x103e6c4] <DrawThemeBackground>
...
call [0x103e6c4] <DrawThemeBackground>
...

stub code
0103540a <DLL Loading and Function Resolution Code>
...
01035425 mov    eax,offset mspaint+0x3e6c4 (0103e6c4)
0103542a jmp    mspaint+0x3540a (0103540a)

Delay Load IAT
...
0103e6c4 0x1035425 (DrawThemeBackground)
...
```



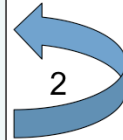
Delay Loading

hello
[...]

```
.text
...
call [0x103e6c4] <DrawThemeBackground>
...
call [0x103e6c4] <DrawThemeBackground>
...


stub code
0103540a <DLL Loading and Function Resolution Code>
...
01035425 mov    eax,offset mspaint+0x3e6c4 (0103e6c4)
0103542a jmp    mspaint+0x3540a (0103540a)

Delay Load IAT
...
0103e6c4 0x1035425 (DrawThemeBackground)
...
```



Delay Loading

UxTheme.dll ... 5ad72bef <DrawThemeBackground>	
.text ... call [0x103e6c4] <DrawThemeBackground> ... call [0x103e6c4] <DrawThemeBackground> ...	3
stub code 0103540a <DLL Loading and Function Resolution Code> ... 01035425 mov eax,offset mspaint+0x3e6c4 (0103e6c4) 0103542a jmp mspaint+0x3540a (0103540a)	
Delay Load IAT ... 0103e6c4 0x1035425 (DrawThemeBackground) ...	



hello

Delay Loading

```
UxTheme.dll  
...  
5ad72bef <DrawThemeBackground>
```

hello
[
...]

```
.text  
...  
call [0x103e6c4] <DrawThemeBackground>  
...  
call [0x103e6c4] <DrawThemeBackground>  
...
```

4

```
stub code  
0103540a <DLL Loading and Function Resolution Code>  
...  
01035425 mov    eax,offset mspaint+0x3e6c4 (0103e6c4)  
0103542a jmp    mspaint+0x3540a (0103540a)
```

```
Delay Load IAT  
...  
0103e6c4 0x5ad72bef (DrawThemeBackground)  
...
```

23

mspaint's delayed import descriptors

	RVA	Data	Description	Value
mspaint.exe	0003A5D8	00000001	Attributes	
IMAGE_DOS_HEADER	0003A5DC	000075E0	RVA to DLL Name	gdiplus.dll
MS-DOS Stub Program	0003A5E0	0003F460	RVA to HMODULE	
IMAGE_NT_HEADERS	0003A5E4	0003E6D4	RVA to Import Address Table	
IMAGE_SECTION_HEADER .text	0003A5E8	0003A648	RVA to Import Name Table	
IMAGE_SECTION_HEADER .data	0003A5EC	0003A880	RVA to Bound IAT ←	
IMAGE_SECTION_HEADER .rsrc	0003A5F0	00000000	RVA to Unload IAT	
SECTION .text	0003A5F4	00000000	Time Date Stamp	
IMPORT Address Table	0003A5F8	00000001	Attributes	
IMAGE_DEBUG_DIRECTORY	0003A5FC	000075F0	RVA to DLL Name	UxTheme.dll
DELAY_IMPORT_DLL Names	0003A600	0003F464	RVA to HMODULE	
IMAGE_LOAD_CONFIG_DIRECTORY	0003A604	0003E6C4	RVA to Import Address Table	
IMAGE_DEBUG_TYPE_CODEVIEW	0003A608	0003A638	RVA to Import Name Table	
DELAY_IMPORT Descriptors	0003A60C	0003A6D0	RVA to Bound IAT ←	
DELAY_IMPORT Name Table	0003A610	00000000	RVA to Unload IAT	
DELAY_IMPORT Hints/Names	0003A614	00000000	Time Date Stamp	
IMPORT Directory Table	0003A618	00000000		
IMPORT Name Table	0003A61C	00000000		
IMPORT Hints/Names & DLL Names	0003A620	00000000		
SECTION .data	0003A624	00000000		
DELAY_IMPORT Address Table	0003A628	00000000		
SECTION .rsrc	0003A62C	00000000		
	0003A630	00000000		
	0003A634	00000000		

Although the “RVA to Bound IAT” is filled in, this feature was reserved for a future version of bind, but I don't think it ever got implemented before deprecation so it just points at some nulls.

mspaint's delayed IAT

	RVA	Data	Description	Value
mspaint.exe	0003E6C4	01035425	Virtual Address	0000 DrawThemeBackground
IMAGE_DOS_HEADER	0003E6C8	01035400	Virtual Address	0000 OpenThemeData
MS-DOS Stub Program	0003E6CC	0103541B	Virtual Address	0000 CloseThemeData
IMAGE_NT_HEADERS	0003E6D0	00000000	End of Imports	UxTheme.dll
IMAGE_SECTION_HEADER .text	0003E6D0	010352B0	Virtual Address	0000 GdipSaveImageToStream
IMAGE_SECTION_HEADER .data	0003E6D8	010352C5	Virtual Address	0000 GdipGetImageRawFormat
IMAGE_SECTION_HEADER .rsrc	0003E6DC	010352DA	Virtual Address	0000 GdipGetPropertySize
SECTION .text	0003E6E0	010352EF	Virtual Address	0000 GdipGetAllPropertyItems
-IMPORT Address Table	0003E6E4	01035304	Virtual Address	0000 GdipCreateBitmapFromFile
-IMAGE_DEBUG_DIRECTORY	0003E6E8	01035319	Virtual Address	0000 GdipCreateBitmapFromFileICM
-DELAY_IMPORT_DLL Names	0003E6EC	0103532E	Virtual Address	0000 GdipGetImageDecodersSize
-IMAGE_LOAD_CONFIG_DIRECTORY	0003E6F0	0103529B	Virtual Address	0000 GdipDisposeImage
-IMAGE_DEBUG_TYPE_CODEVIEW	0003E6F4	01035358	Virtual Address	0000 GdipGetImageEncodersSize
-DELAY_IMPORT Descriptors	0003E6F8	0103536D	Virtual Address	0000 GdipGetImageEncoders
-DELAY_IMPORT Name Table	0003E6FC	01035382	Virtual Address	0000 GdipFree
-DELAY_IMPORT Hints/Names	0003E700	01035397	Virtual Address	0000 GdipAlloc
-IMPORT Directory Table	0003E704	010353AC	Virtual Address	0000 GdipCloneImage
-IMPORT Name Table	0003E708	010353C1	Virtual Address	0000 GdipSaveImageToFile
-IMPORT Hints/Names & DLL Names	0003E70C	010353D6	Virtual Address	0000 GdipSetPropertyItem
SECTION .data	0003E710	010353EB	Virtual Address	0000 GdipCreateBitmapFromHBITMAP
DELAY_IMPORT Address Table	0003E714	01035296	Virtual Address	0000 GdipStartUp
SECTION .rsrc	0003E718	01035343	Virtual Address	0000 GdipGetImageDecoders
	0003E71C	01035260	Virtual Address	0000 GdipShutdown
	0003E720	00000000	End of Imports	gdipplus.dll

These are (absolute) virtual addresses. Since the ImageBase for mspaint is 0x1000000 and the SizeOfImage is 0x57000, that means these virtual addresses start out **inside** mspaint itself. Each one just points at some stub code to call the dynamic linker.

mspaint's delayed imports in memory (some resolved, some not)

Resolved Not Resolved

The screenshot shows the WinDbg interface for process mspaint. The Memory view displays a table of memory addresses and their contents in Long Hex format. The Disassembly view shows the instruction at offset 0103541b, which is a jump to the start of stub code.

Address	Hex Value
0103e6c4	5ad72bef 5ad773b8 0103541b 00000000 010352b0 010352c5 010352da 010352ef 01035304
0103e6e8	01035319 0103532e 0103529b 01035358 0103536d 01035382 01035397 010353ac 010353e1
0103e70c	010353d6 010353eb 4ec67a79 01035343 01035260 00000000 00000000 00000000 00000000

```
Offset: 0103541b
01035419 ffe0 jmp eax
0103541b b8cce60301 mov eax,offset mspaint+0x3e6cc (0103e6cc)
01035420 e9e5ffff jmp mspaint+0x3540a (0103540a)
```

Start of stub code

26
Note to self, walk the stub code a bit in the debugger

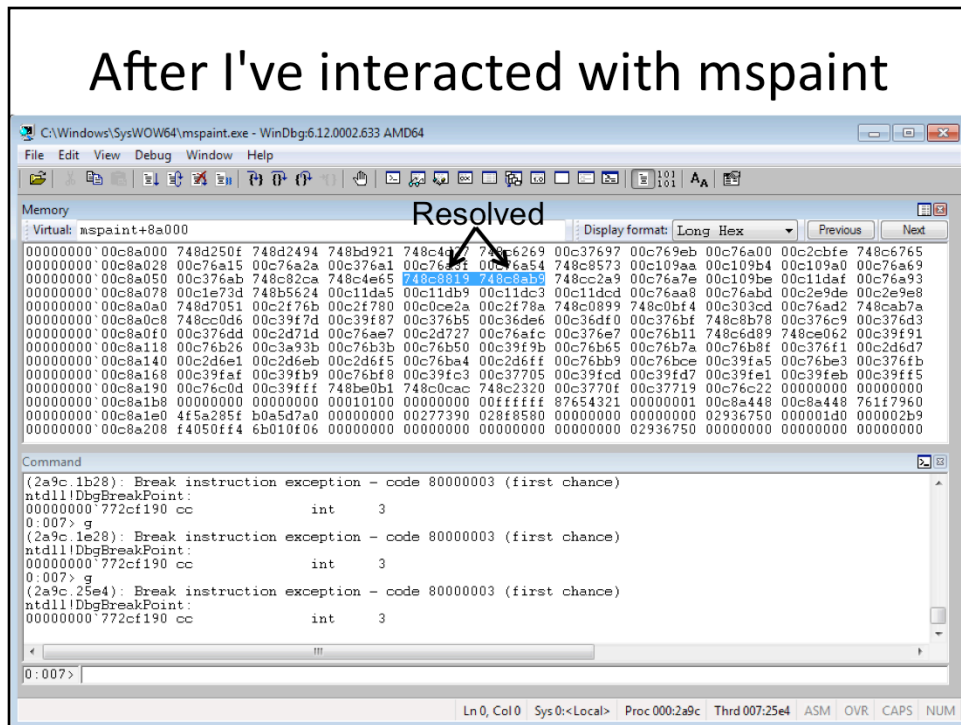
Before I've interacted with mspaint

The screenshot shows the WinDbg interface with the memory dump for `Virtual: mspaint+8a000`. The memory dump is divided into two sections: **Resolved** and **Unresolved**. The **Resolved** section shows memory addresses and their corresponding symbols, such as `00000000`00c8a000 748d250f 748d2498 748bd921 748c4d77 748c6269 00c37697 00c769eb 00c76a00 00c2cbfe 748c6765`. The **Unresolved** section shows memory addresses and their corresponding symbols, such as `00000000`00c8a028 00c76a15 00c76a2a 00c76a37 00c76a54 748c8573 00c109aa 00c109b4 00c109a0 00c76a69`. The **Command** window shows the following output:

```
* Use .symfix to have the debugger choose a symbol path.
* After setting your symbol path, use .reload to refresh symbol locations.
*****
Executable search path is:
ModLoad: 00000000`00c00000 00000000`01217000 mspaint.exe
ModLoad: 00000000`77420000 00000000`7742b000 ntdll.dll
ModLoad: 00000000`77420000 00000000`775e0000 ntdll32.dll
ModLoad: 00000000`74c10000 00000000`74cef000 C:\Windows\SYSTEM32\wow64.dll
ModLoad: 00000000`74c50000 00000000`74cac000 C:\Windows\SYSTEM32\wow64win.dll
ModLoad: 00000000`74c40000 00000000`74c48000 C:\Windows\SYSTEM32\wow64cpu.dll
(2a9c.147c): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ntdll.dll -
```

The **Base** window shows the address `0:007>`.

After I've interacted with mspaint



To interact with mspaint, just mouse over the blank canvas



Get your geek on



- Play through round 5 on your own, and then wait for the seed for the class deathmatch
- This will be the hardest round so far, on account of the gotchas:
 - CFF Explorer doesn't expose delay-load DLL information, and PEView doesn't parse it (or bound import info) for 64 bit executables. You're going to have to manually interpret per the struct definitions

See notes for citation

29

New 2012 – NOTE: I spent way more time on that token than I should have, so you must love and cherish it

From

<http://www.classicplastic.net/dvgi/pics-tokenstilt02.jpg>

<http://www.classicplastic.net/dvgi/pics-tokensgeneric02.jpg>

Dependency Walker, just 'cause

hehe depends.exe...that's right, potty humor, I went there

The screenshot shows the Dependency Walker interface for the application 'notepad.exe'. On the left, a tree view displays the dependency graph, including DLLs like COMDLG32.DLL, ADVAPI32.DLL, and KERNEL32.DLL. On the right, two tables provide details for the selected DLLs.

P	Ordinal	Hint	Function	Entry Point
✓	N/A	49 (0x0031)	CloseHandle	0x7C809B07
✓	N/A	76 (0x004C)	CreateEventW	0x7C80A739
✓	N/A	82 (0x0052)	CreateFileW	0x7C8107F0
✓	N/A	108 (0x006C)	CreateThread	0x7C8106C7
✓	N/A	125 (0x007D)	DelayLoadFailureHook	0x7C87EECD
✓	N/A	127 (0x007F)	DeleteCriticalSection	0x7C91135A
✓	N/A	130 (0x0082)	DeleteFileW	0x7C831F48
✓	N/A	137 (0x0089)	DisableThreadLibraryCalls	0x7C811326
✓	N/A	150 (0x0096)	EnterCriticalSection	0x7C901000
✓	N/A	186 (0x00BA)	ExpandEnvironmentStringsW	0x7C8305FA

E	Ordinal	Hint	Function	Entry Point
✓	1 (0x0001)	0 (0x0000)	ActivateActCtx	0x0000A6E4
✓	2 (0x0002)	1 (0x0001)	AddAtomA	0x0003551D
✓	3 (0x0003)	2 (0x0002)	AddAtomW	0x000326F1
✓	4 (0x0004)	3 (0x0003)	AddConsoleAliasA	0x00071DFF
✓	5 (0x0005)	4 (0x0004)	AddConsoleAliasW	0x00071DC1
✓	6 (0x0006)	5 (0x0005)	AddLocalAlternateComputerNameA	0x00059412
✓	7 (0x0007)	6 (0x0006)	AddLocalAlternateComputerNameW	0x000592F6
✓	8 (0x0008)	7 (0x0007)	AddRefActCtx	0x00028F11
✓	9 (0x0009)	8 (0x0008)	AddVectoredExceptionHandler	NTDLL.RtlAddVectoredExceptionHandler

Annotations in the image:

- Delay load:** Points to the 'Delay load' icon next to KERNEL32.DLL in the tree view.
- Forwarded-to DLL:** Points to the 'Forwarded-to DLL' icon next to NTDLL.DLL in the tree view.
- Forwarded-to Function:** Points to the 'Forwarded-to Function' icon next to the entry for NTDLL.RtlAddVectoredExceptionHandler in the function table.

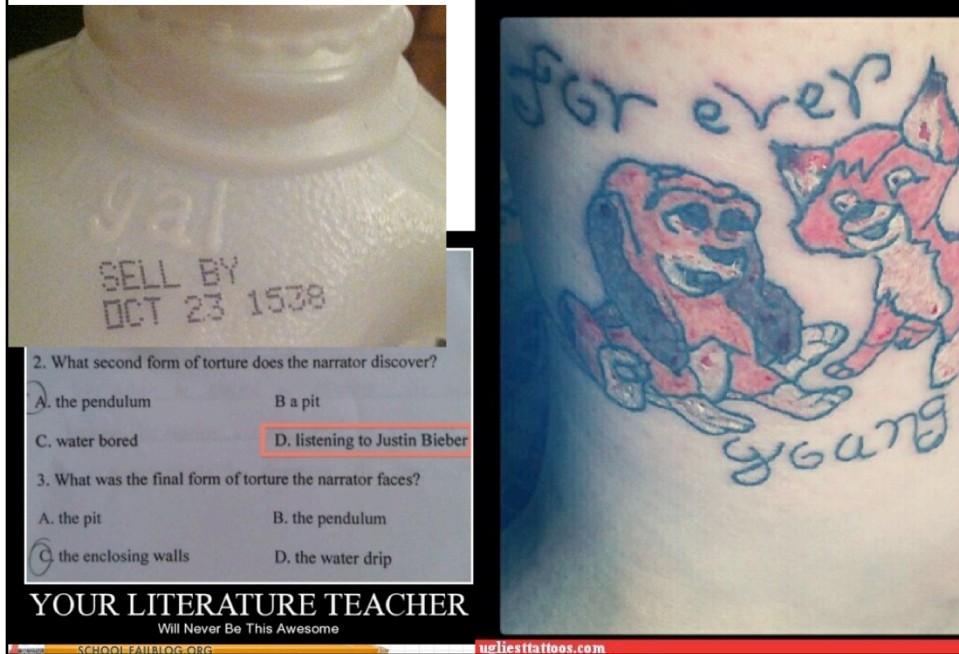
Runtime Importing

- Just for completeness, I should mention LoadLibrary() and GetProcAddress().
- LoadLibrary() can be called to dynamically load a DLL into the memory space of the process
- GetProcAddress() gives the address of a function specified by name, or by ordinal (which we will talk about soon). This address can then be used as a function pointer.
- Remember when we were seeing delay-loaded DLLs, and the dynamic linker "somehow" loaded the DLL and then resolved the function address? It's actually using LoadLibrary() and GetProcAddress().
- These functions are often abused to make it so that which functions the malware actually uses cannot be determined simply by looking at the INT. Rather, the malware will have the names of the imported libraries and functions obfuscated somewhere in the data, and then will deobfuscate them and dynamically resolve them before calling the imported functions.

TODO:

- Add picture & description of how rundll32.exe works

Uhg, *finally* done with imports. Treat yourself to some fail/win.



<http://cheezburger.com/6678531328>

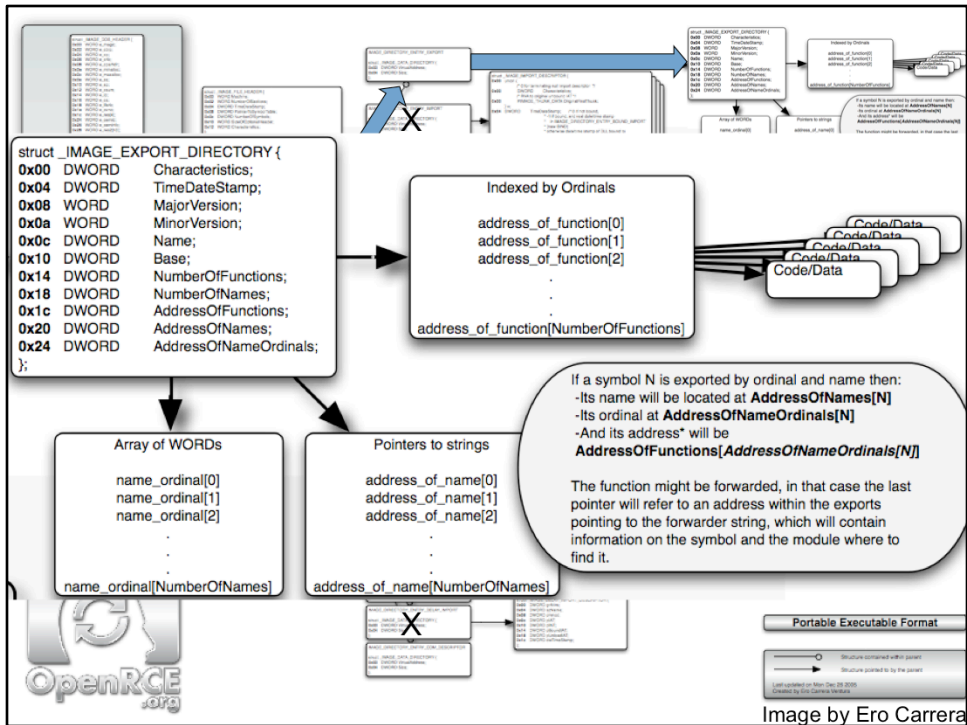
<https://i.chzbgr.com/completestore/12/10/16/TD8w0wqM5EetVkJaLdZROA2.jpg>

<https://i.chzbgr.com/completestore/12/10/12/SuxGUko-lUCvWyfpgOM6xQ2.jpg>

<https://i.chzbgr.com/completestore/12/10/7/C7UbjVtEHUmp-f9NS-A-Ag2.jpg>

Exporting Functions & Data

- For a library to be useful, other code which wants to use its functions must be able to import them, as already talked about.
- There are two options to export functions and data. They can be exported by name (where the programmer even has the option to call the exported name something different than he himself calls it), or they can be exported by ordinal.
- An ordinal is just an index, and if a function is exported by ordinal, it can only be imported by ordinal. While exporting by ordinal saves space, by not having extra strings for the names of symbols, and time by not having to search the strings, it also puts more work on the programmer which wants to import the export. But it can also be a way to make a private (undocumented) API more private.



Exports

from winnt.h

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Name;
    DWORD Base;
    DWORD NumberOfFunctions;
    DWORD NumberOfNames;
    DWORD AddressOfFunctions; // RVA from base of image
    DWORD AddressOfNames; // RVA from base of image
    DWORD AddressOfNameOrdinals; // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Exports 2

- The **TimeStamp** listed here is what's actually checked against when the loader is trying to determine if bound imports are out of date for instance. Can be different from the one in the File Header (see ntdll.dll). Presumably (wasn't able to confirm), the linker only updates this if there are meaningful changes to the RVAs or order for exported functions. That way, the TimeDateStamp "version" can stay backwards compatible as long as possible.
- **NumberOfFunctions** will be different from **NumberOfNames** when the file is exporting some functions by ordinal (talked about later). By knowing the number of names, when searching for an import by name, the loader can do a binary search.

Exports 3

- **Base** is the number to subtract from an ordinal to get the zero-indexed offset into the `AddressOfFunctions` array. Because ordinals start at 1 by default, this is usually 1. However ordinals could start at 10 if the programmer wants them to, and therefore `Base` would then be set to 10.
- **AddressOfFunctions** is an RVA which points to the beginning of an array which holds `DWORD` RVAs which point to the start of the exported functions. The pointed-to array should be `NumberOfFunctions` entries long. This would be the Export Address Table (EAT) like the flip side of the Import Address Table (IAT).
- Eat! I atè! :P

Exports 4

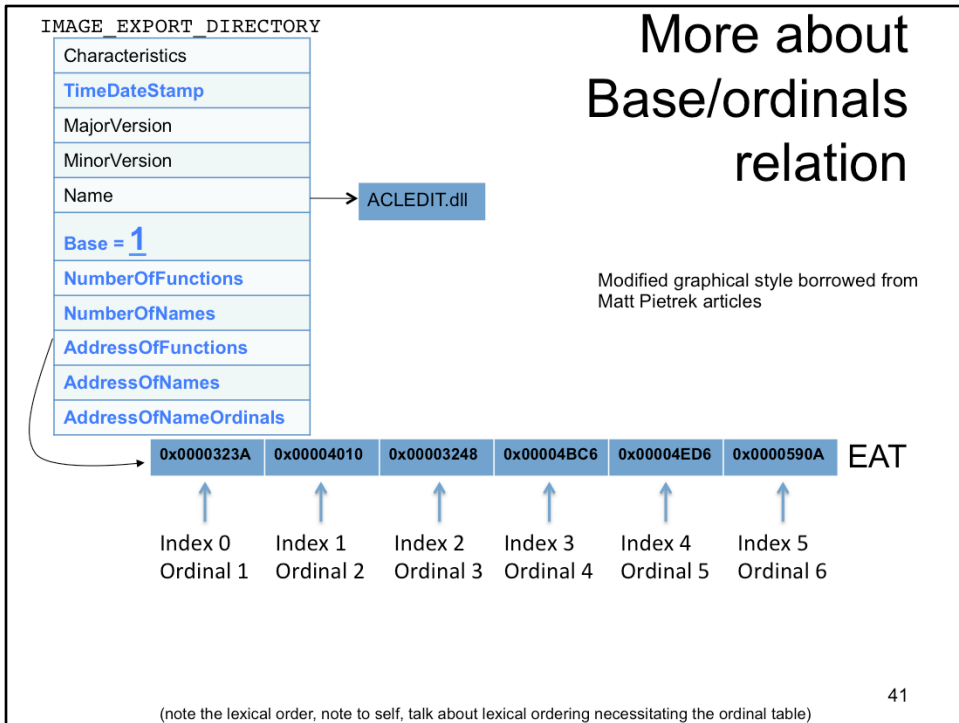
- **AddressOfNames** is an RVA which points to the beginning of an array which holds DWORD RVAs which point to the strings which specify function names. The pointed-to array should be NumberOfNames entries long. This would be the Export Names Table (ENT) like the flipside of the Import Names Table (INT).
- **AddressOfNameOrdinals** is an RVA which points to the beginning of an array which holds **WORD** (16 bit) sized ordinals. The entries in this array are already zero-indexed indices into the EAT, and therefore are unaffected by **Base**.

Ordinal says what?

- When importing by name, like I said, it can do a binary search over the strings in the ENT, because nowadays, they're lexically sorted. "Back in the day" they weren't sorted. Back then, it was strongly encouraged to "import by ordinal", that is, you could specify "I want ordinal 5 in kernel32.dll" instead of "I want AddConsoleAliasW in kernel32.dll", because if the names aren't sorted, you're doing a linear search. You can still import by ordinal if you choose, and that way your binary/library will load a bit faster.
- Even if you're importing by name, it is actually just finding the index in the ENT, and then selecting the same index in the AddressOfNameOrdinals, and then reading the value from the AddressOfNameOrdinals to use as an index into the EAT.
- Generally speaking, the downside of importing by ordinal is that if the ordinals change, your app breaks. That said, the developer who's exporting by ordinal has incentive to not change them, unless he wants those apps to break (e.g. to force a deprecated API to not be used any more).

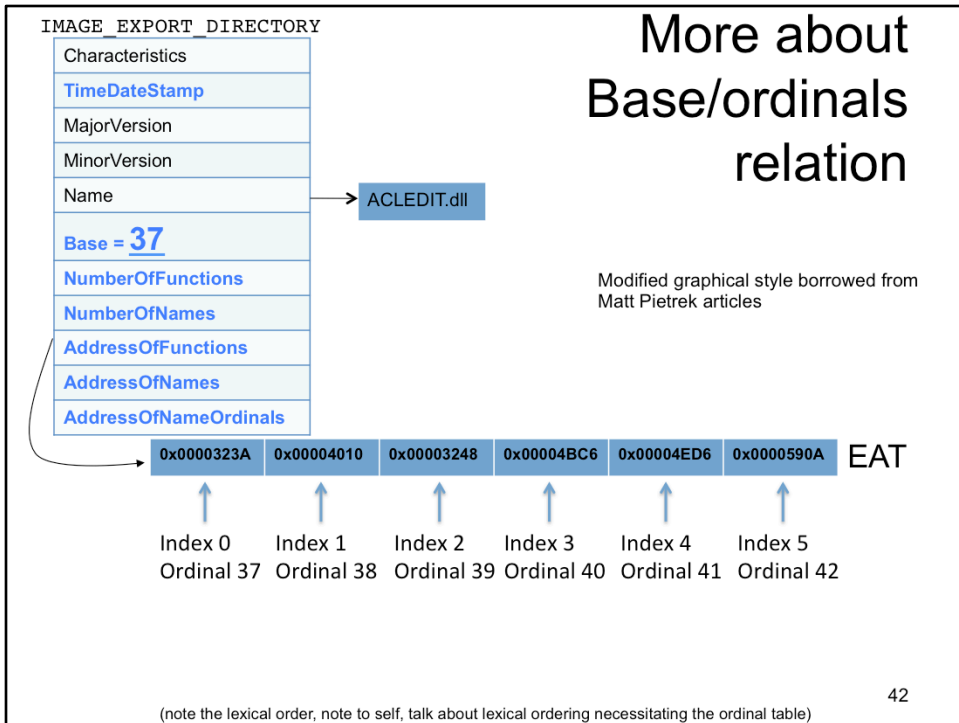
More about Base/ordinals relation

Modified graphical style borrowed from Matt Pietrek articles

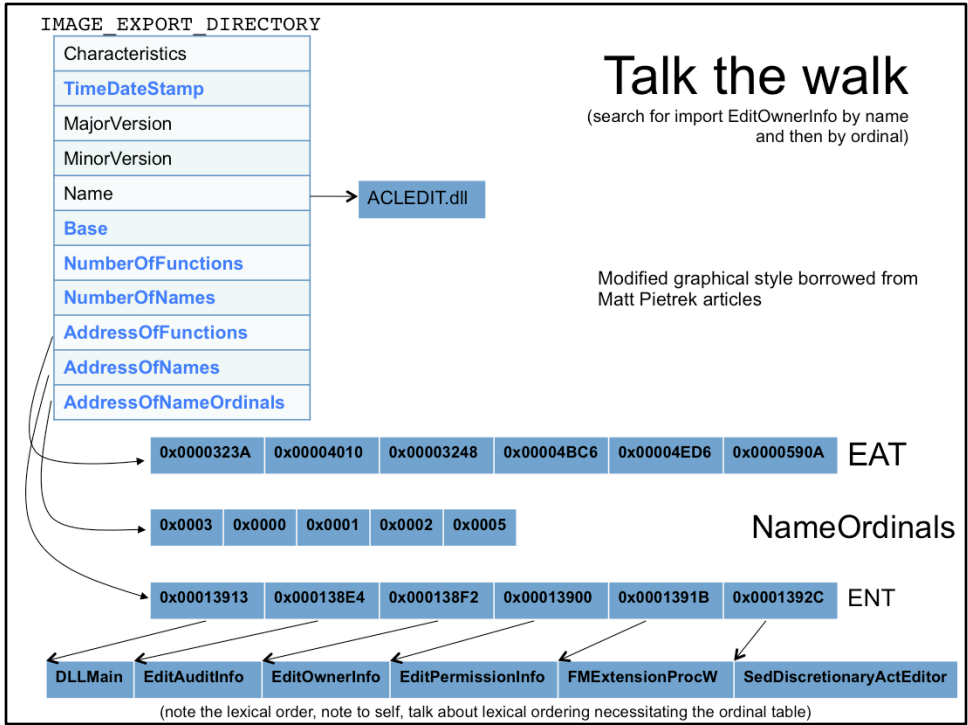


More about Base/ordinals relation

Modified graphical style borrowed from Matt Pietrek articles



(note the lexical order, note to self, talk about lexical ordering necessitating the ordinal table)



How does one go about specifying an export?

- [http://msdn.microsoft.com/en-us/library/hyx1zcd3\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/hyx1zcd3(VS.80).aspx)
- “There are three methods for exporting a definition, listed in recommended order of use:
 - The `__declspec(dllexport)` keyword in the source code
 - An `EXPORTS` statement in a `.def` file
 - An `/EXPORT` specification in a `LINK` command”

From HelloWorldDLL's DllMain.c

```
__declspec(dllexport) void __cdecl SayHello(  
    HWND hwnd,  
    HINSTANCE hinst,  
    LPSTR lpszCmdLine,  
    int nCmdShow)  
{  
    MessageBox(0, "Hello World!", 0, 0);  
}
```

Where to specify a .def file

The screenshot shows the 'Linker' configuration properties in Visual Studio. The left pane shows a tree view with 'Linker' expanded to 'Input'. The right pane shows a table of properties, with 'Module Definition File' highlighted. Below the table, a description for 'Module Definition File' is provided.

Additional Dependencies	
Ignore All Default Libraries	No
Ignore Specific Library	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded DLLs	
Assembly Link Resource	

Module Definition File
Use specified module definition file during executable creation. (/DEF:name)

Forwarded Exports

- There is an option to forward a function from one module to be handled by another one (e.g. it might be used if code was refactored to move a function to a different module, but you wanted to maintain backward compatibility.)
- As we just saw, normally **AddressOfFunctions** points to an array of RVAs which point at code. However, if a RVA in that array of RVAs points into the exports section (as defined by the base and size given in the data directory entry), then the RVA will actually be pointing at a string of the form `DllToForwardTo.FunctionName`

Kernel32.dll forwarded (to ntdll.dll) exports

kernel32.dll	RVA	Data	Description	Value
IMAGE_DOS_HEADER	00002654	0000A6E4	Function RVA	0001 ActivateActCtx
MS-DOS Stub Program	00002658	0003551D	Function RVA	0002 AddAtomA
IMAGE_NT_HEADERS	0000265C	000326F1	Function RVA	0003 AddAtomW
IMAGE_SECTION_HEADER .text	00002660	00071DFF	Function RVA	0004 AddConsoleAliasA
IMAGE_SECTION_HEADER .data	00002664	00071DC1	Function RVA	0005 AddConsoleAliasW
IMAGE_SECTION_HEADER .rsrc	00002668	00059412	Function RVA	0006 AddLocalAlternateComputerNameA
IMAGE_SECTION_HEADER .reloc	0000266C	000592F6	Function RVA	0007 AddLocalAlternateComputerNameW
SECTION .text	00002670	0002BF11	Function RVA	0008 AddRefActCtx
IMPORT Address Table	00002674	00009011	Forwarded Name RVA	0009 AddVectoredExceptionHandler -> NTDLL.RtlAddVectoredExceptionHandler
IMAGE_EXPORT_DIRECTORY	00002678	00072451	Function RVA	000A AllocConsole
EXPORT Address Table	0000267C	0005F6D4	Function RVA	000B AllocateUserPhysicalPages
EXPORT Name Pointer Table	00002680	0003597F	Function RVA	000C AreFileApisANSI
EXPORT Ordinal Table	00002684	0002E45A	Function RVA	000D AssignProcessToJobObject
-----	-----	-----	-----	-----

kernel32.dll	RVA	Raw Data	Value
IMAGE_DOS_HEADER	00008F88	00 6C 73 74 72 63 6D 70	.Istrcmp.Istrcmp
MS-DOS Stub Program	00008F98	41 00 6C 73 74 72 63 6D	70 57 00 6C 73 74 72 63 A.IstrcmpW.Istrc
IMAGE_NT_HEADERS	00008FA8	6D 70 69 00 6C 73 74 72	63 6D 70 69 41 00 6C 73 mp.IstrcmpiA.Is
IMAGE_SECTION_HEADER .text	00008FB8	74 72 63 6D 70 69 57 00	6C 73 74 72 63 70 79 00 trcmpiW.Istrcpy.
IMAGE_SECTION_HEADER .data	00008FC8	6C 73 74 72 63 70 79 41	00 6C 73 74 72 63 70 79 IstrcpyA.Istrcpy
IMAGE_SECTION_HEADER .rsrc	00008FD8	57 00 6C 73 74 72 63 70	79 6E 00 6C 73 74 72 63 W.Istrcpyn.Istrc
IMAGE_SECTION_HEADER .reloc	00008FE8	70 79 6E 41 00 6C 73 74	72 63 70 79 6E 57 00 6C pynA.IstrcpynW.I
SECTION .text	00008FF8	73 74 72 6C 65 6E 00 6C	73 74 72 6C 65 6E 41 00 strlen.IstrlenA.
IMPORT Address Table	00009008	6C 73 74 72 6C 65 6E 57	00 4E 54 44 4C 4C 2E 52 IstrlenW.NTDLL.R
IMAGE_EXPORT_DIRECTORY	00009018	74 6C 41 64 64 56 65 63	74 6F 72 65 64 45 78 63 tIAddVectoredExc
EXPORT Address Table	00009028	65 70 74 69 6F 6E 48 61	6E 64 6C 65 72 00 4E 54 eptionHandler.NT
EXPORT Name Pointer Table	00009038	44 4C 4C 2E 52 74 6C 44	65 63 6F 64 65 50 6F 69 DLL.RtlDecodePoi
EXPORT Ordinal Table	00009048	6E 74 65 72 00 4E 54 44	4C 4C 2E 52 74 6C 44 65 nter.NTDLL.RtlDe
EXPORT Names	00009058	63 6F 64 65 53 79 73 74	65 6D 50 6F 69 6E 74 65 codeSystemPointe

How does one go about forwarding exports?

- Statement in .def file of the form

EXPORTS

FunctionAlias=OtherDLLName.RealFunction

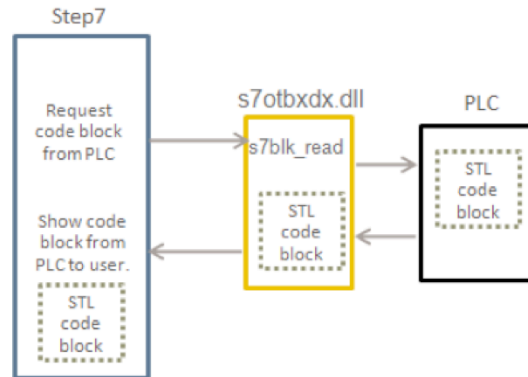
- or /export linker option
- /export:FunctionAlias=OtherDLLName.RealFunction
- Can even specify a linker comment in the code with
 - #pragma comment(linker, "/export:FunctionAlias=OtherDLLName.RealFunction")

Relevance to Stuxnet

- Stuxnet used forwarded exports for the 93 of 109 exports in s7otbxdx.dll which it didn't need to intercept.

Figure 18

Step7 and PCL communicating via s7otbxdx.dll



See notes for citation

50

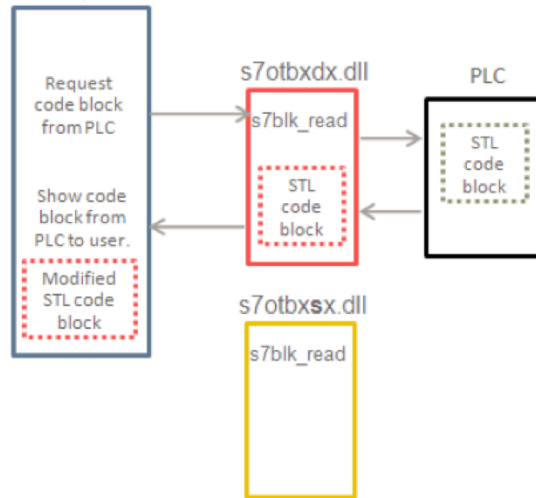
From http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

Stuxnet trojaned DLL

Figure 19

Communication with malicious version of s7otbxdx.dll

Step7



See notes for citation

51

From http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

Function Redirection Tutorial

- http://packetstormsecurity.org/papers/win/intercept_apis_dll_redirection.pdf
- Basically talks about making a trojan DLL which hooks or reimplements some functions for the intercepted DLL, and then forwards the rest on to the original. Basically exactly what Stuxnet did for the trojan PLC accessing DLL.

Returning to Bound Imports

- Just to fill this in, now that we know about forwarded functions, the point of NumberOfModuleForwarderRefs and IMAGE_BOUND_FORWARDER_REF is that when the linker is trying to validate that none of the bound imports are changed, it needs to make sure none of the versions (TimeDateStamps) of imported modules has changed. Therefore if a module is bound to any modules which forward to other modules, those forwarded-to modules must be checked as well

```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD TimeDateStamp;
    WORD  OffsetModuleName;
    WORD  NumberOfModuleForwarderRefs;
    // Array of zero or more IMAGE_BOUND_FORWARDER_REF follows
} IMAGE_BOUND_IMPORT_DESCRIPTOR, *PIMAGE_BOUND_IMPORT_DESCRIPTOR;

typedef struct _IMAGE_BOUND_FORWARDER_REF {
    DWORD TimeDateStamp;
    WORD  OffsetModuleName;
    WORD  Reserved;
} IMAGE_BOUND_FORWARDER_REF, *PIMAGE_BOUND_FORWARDER_REF;
```

WHILE we're thinking back...

- What are the three types of imports?
- What is the difference between importing by name vs. ordinal?
- Binding vs. ASLR: There can be only one?
- What did the life-size cut out of Anakin Skywalker look like?



See notes for citation

54

<http://i43.tower.com/images/mm107041173/hot-shots-part-deux-charlie-sheen-dvd-cover-art.jpg>

EAT Hooking

- IAT hooking can modify all *currently loaded* modules in a process' address space. If something new gets loaded (say, through LoadLibrary()), the attacker would need to be notified of this even to hook it's IAT too.
- Instead, if the attacker modifies the EAT in the module which contains the the functions which he is intercepting, when a new module is loaded, he can just let the loaded do its thing, and the new module will point at the attacker's code. Thus EAT hooking provides some "forward compatibility" assurance to the attacker that he will continue to hook the functions for all subsequently loaded modules.

EAT Hooking Lab

- beta: http://www.codeproject.com/KB/system/api_spying_hack.aspx



Get your geek on



- Play through round 6 on your own, and then wait for the seed for the class deathmatch

See notes for citation

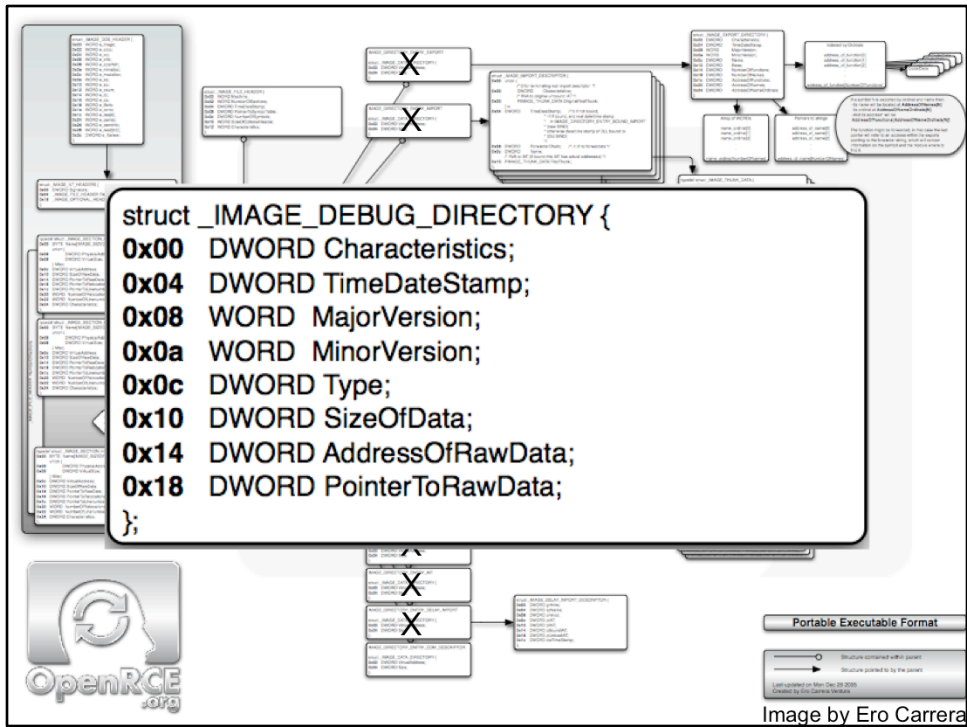
57

New 2012 – NOTE: I spent way more time on that token than I should have, so you must love and cherish it

From

<http://www.classicplastic.net/dvgi/pics-tokenstilt02.jpg>

<http://www.classicplastic.net/dvgi/pics-tokensgeneric02.jpg>



Debug Info

from winnt.h

```
typedef struct _IMAGE_DEBUG_DIRECTORY {
    DWORD Characteristics;
    DWORD TimeDateStamp;
    WORD MajorVersion;
    WORD MinorVersion;
    DWORD Type;
    DWORD SizeOfData;
    DWORD AddressOfRawData;
    DWORD PointerToRawData;
} IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;

#define IMAGE_DEBUG_TYPE_UNKNOWN 0
#define IMAGE_DEBUG_TYPE_COFF 1
#define IMAGE_DEBUG_TYPE_CODEVIEW 2
#define IMAGE_DEBUG_TYPE_FPO 3
#define IMAGE_DEBUG_TYPE_MISC 4
#define IMAGE_DEBUG_TYPE_EXCEPTION 5
#define IMAGE_DEBUG_TYPE_FIXUP 6
#define IMAGE_DEBUG_TYPE_OMAP_TO_SRC 7
#define IMAGE_DEBUG_TYPE_OMAP_FROM_SRC 8
#define IMAGE_DEBUG_TYPE_BORLAND 9
#define IMAGE_DEBUG_TYPE_RESERVED10 10
#define IMAGE_DEBUG_TYPE_CLSID 11
```

Debug Info 2

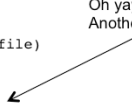
- **TimeStamp**, yet another to sanity check against. Should be the same as the one in the File Header I believe.
- **Type** and **SizeOfData** are what you would expect. The main Type we care about is `IMAGE_DEBUG_TYPE_CODEVIEW` as this is the common form now which points to a structure which holds a path to the pdb file which holds the debug symbols.
- **AddressOfRawData** is an RVA to the debug info.
- **PointerToRawData** is a file offset to the debug info.

Debug Info 3

From <http://www.debuginfo.com/examples/src/DebugDir.cpp>

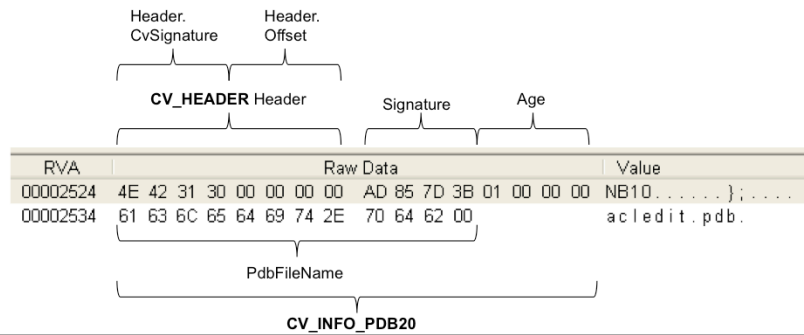
```
#define CV_SIGNATURE_NB10 '01BN'
#define CV_SIGNATURE_RSDS 'SDSR'
// CodeView header
struct CV_HEADER {
    DWORD CvSignature; // NBxx
    LONG Offset; // Always 0 for NB10
};
// CodeView NB10 debug information
// (used when debug information is stored in a PDB 2.00 file)
struct CV_INFO_PDB20 {
    CV_HEADER Header;
    DWORD Signature; // seconds since 01.01.1970
    DWORD Age; // an always-incrementing value
    BYTE PdbFileName[1]; // zero terminated string with the name of the PDB file
};
// CodeView RSDS debug information
// (used when debug information is stored in a PDB 7.00 file)
struct CV_INFO_PDB70 {
    DWORD CvSignature;
    GUID Signature; // unique identifier
    DWORD Age; // an always-incrementing value
    BYTE PdbFileName[1]; // zero terminated string with the name of the PDB file
};
```

Oh yay!
Another TimeDateStamp!



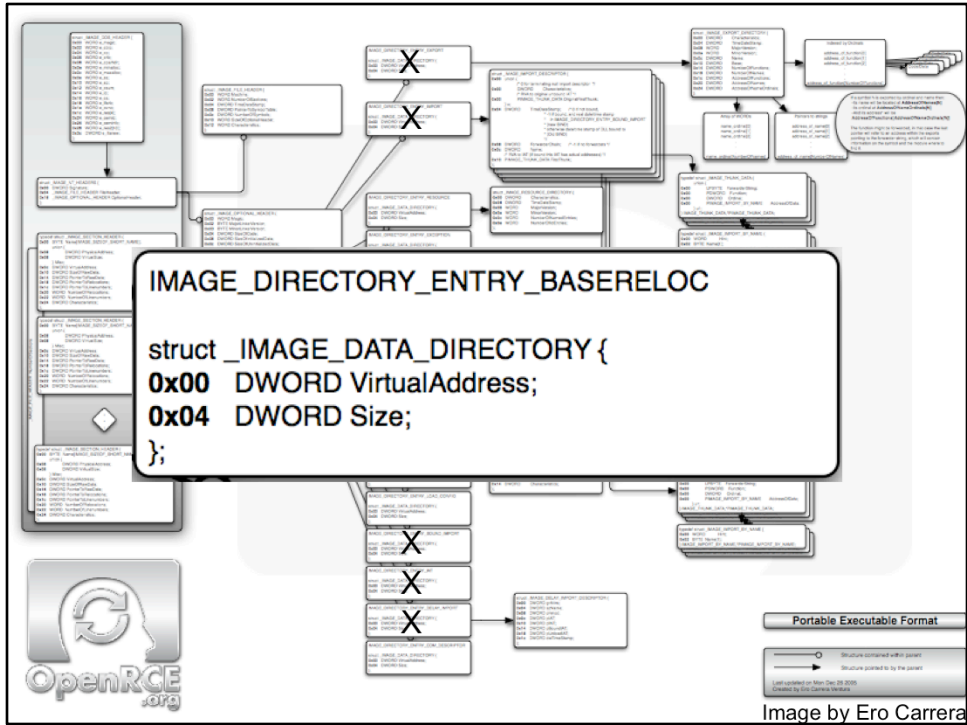
Therefore, how shall we interpret this?

RVA	Data	Description	Value
00001670	00000000	Characteristics	
00001674	3B7D85AD	Time Date Stamp	2001/08/17 Fri 20:59:25 UTC
00001678	0000	Major Version	
0000167A	0000	Minor Version	
0000167C	00000002	Type	IMAGE_DEBUG_TYPE_CODEVIEW
00001680	0000001C	Size of Data	
00001684	00002524	Address of Raw Data	
00001688	00001924	Pointer to Raw Data	



A thing of the past?

- Between pulling a pdb path from high profile malware like GhostNet, Aurora, and Stuxnet malware, and Greg Hoglund starting to talk (at BlackHat LV 2010) about using pdb paths and TimeDateStamps to provide better attribution for malware authors, are we going to see any meaningful values here anymore? Time will tell.
- e:\gh0st\server\sys\i386\RESSDT.pdb
- \Aurora_Src\Aurora\NC\Avc\Release\AVC.pdb
- b:\myrtus\src\objfre_w2k_x86\i386\guava.pdb



Relocations

from winnt.h

- Generally stored in the .reloc section
- Not shown on the picture the `IMAGE_DIRECTORY_ENTRY_BASERELOC` points at an array of `IMAGE_BASE_RELOCATION` structures.

```
typedef struct _IMAGE_BASE_RELOCATION {  
    DWORD    VirtualAddress;  
    DWORD    SizeOfBlock;  
    // WORD    TypeOffset[1];  
} IMAGE_BASE_RELOCATION;
```

65

Relocations 2

- **VirtualAddress** specifies the page-aligned virtual address that the specified relocation targets will be relative to.
- **SizeOfBlock** is the size of the IMAGE_BASE_RELOCATION itself + all of the subsequent relocation targets.
- Following SizeOfBlock are a variable number of WORD-sized relocation targets. The number of targets can be calculated as $(\text{SizeOfBlock} - \text{sizeof}(\text{IMAGE_BASE_RELOCATION})) / \text{sizeof}(\text{WORD})$.

Relocations example acledit.dll

	RVA	Data	Description	Value
acledit.dll	0002168C	3DEB	Type RVA	00002DEB IMAGE_REL_BASED_HIGHLOW
IMAGE_DOS_HEADER	0002168E	3F2B	Type RVA	00002F2B IMAGE_REL_BASED_HIGHLOW
MS-DOS Stub Program				
IMAGE_NT_HEADERS				
IMAGE_SECTION_HEADER .text	00021694	0000003C	Size of Block	
IMAGE_SECTION_HEADER .data	00021698	32FB	Type RVA	000032FB IMAGE_REL_BASED_HIGHLOW
IMAGE_SECTION_HEADER .rsrc	0002169A	3307	Type RVA	00003307 IMAGE_REL_BASED_HIGHLOW
IMAGE_SECTION_HEADER .reloc	0002169C	334A	Type RVA	0000334A IMAGE_REL_BASED_HIGHLOW
BOUND_IMPORT Directory Table	0002169E	33A2	Type RVA	000033A2 IMAGE_REL_BASED_HIGHLOW
BOUND_IMPORT DLL Names	000216A0	33DB	Type RVA	000033DB IMAGE_REL_BASED_HIGHLOW
SECTION .text	000216A2	3411	Type RVA	00003411 IMAGE_REL_BASED_HIGHLOW
SECTION .data	000216A4	341B	Type RVA	0000341B IMAGE_REL_BASED_HIGHLOW
SECTION .rsrc	000216A6	345A	Type RVA	0000345A IMAGE_REL_BASED_HIGHLOW
SECTION .reloc	000216A8	3473	Type RVA	00003473 IMAGE_REL_BASED_HIGHLOW
IMAGE_BASE_RELOCATION	000216AA	34B3	Type RVA	000034B3 IMAGE_REL_BASED_HIGHLOW
	000216AC	34D3	Type RVA	000034D3 IMAGE_REL_BASED_HIGHLOW
	000216AE	34E2	Type RVA	000034E2 IMAGE_REL_BASED_HIGHLOW
	000216B0	34FC	Type RVA	000034FC IMAGE_REL_BASED_HIGHLOW
	000216B2	3517	Type RVA	00003517 IMAGE_REL_BASED_HIGHLOW
	000216B4	351E	Type RVA	0000351E IMAGE_REL_BASED_HIGHLOW
	000216B6	3749	Type RVA	00003749 IMAGE_REL_BASED_HIGHLOW
	000216B8	3775	Type RVA	00003775 IMAGE_REL_BASED_HIGHLOW
	000216BA	3B13	Type RVA	00003B13 IMAGE_REL_BASED_HIGHLOW
	000216BC	3CF8	Type RVA	00003CF8 IMAGE_REL_BASED_HIGHLOW
	000216BE	3D12	Type RVA	00003D12 IMAGE_REL_BASED_HIGHLOW
	000216C0	3D82	Type RVA	00003D82 IMAGE_REL_BASED_HIGHLOW
	000216C2	3DF6	Type RVA	00003DF6 IMAGE_REL_BASED_HIGHLOW
	000216C4	3E15	Type RVA	00003E15 IMAGE_REL_BASED_HIGHLOW
	000216C6	3E35	Type RVA	00003E35 IMAGE_REL_BASED_HIGHLOW
	000216C8	3E3F	Type RVA	00003E3F IMAGE_REL_BASED_HIGHLOW
	000216CA	0000	Type RVA	
	000216CC	00004000	RVA of Block	
	000216D0	0000002C	Size of Block	
	000216D4	3256	Type RVA	00004256 IMAGE_REL_BASED_HIGHLOW

Relocations 3

- The upper 4 bits of the 16 bit relocation target specifies the type. The lower 12 bits specifies an offset, which will be used differently depending on the type.

Types are:

```
#define IMAGE_REL_BASED_ABSOLUTE      0
#define IMAGE_REL_BASED_HIGH         1
#define IMAGE_REL_BASED_LOW          2
#define IMAGE_REL_BASED_HIGHLOW     3
#define IMAGE_REL_BASED_HIGHADJ      4
#define IMAGE_REL_BASED_MIPS_JMPADDR 5
#define IMAGE_REL_BASED_MIPS_JMPADDR16 9
#define IMAGE_REL_BASED_IA64_IMM64   9
#define IMAGE_REL_BASED_DIR64       10
```

- We generally only care about `IMAGE_REL_BASED_HIGHLOW`, which when used says that the RVA for the data to be relocated is specified by **VirtualAddress** + the lower 12 bits.

Slice of life

00021690	00003000	RVA of Block	
00021694	0000003C	Size of Block	
00021698	32FB	Type RVA	000032FB IMAGE_REL_BASED_HIGHLOW
0002169A	3307	Type RVA	00003307 IMAGE_REL_BASED_HIGHLOW
0002169C	334A	Type RVA	0000334A IMAGE_REL_BASED_HIGHLOW

- So in the above if the file was being relocated, the loader would take the relocation target WORD 0x32FB, the upper 4 bits are 0x3 = IMAGE_REL_BASED_HIGHLOW. The lower 12 bits are 0x2FB. Given the type, we do (VirtualAddress (0x3000) + lower 12 bits (0x2FB)) == 0x32FB is the RVA of the location which needs to be fixed.
- Then the loader would just add whatever the delta is between the file's preferred load address and actual load address, and just add that delta to data at RVA 0x32FB.
- (Show example in WinDBG of what target for relocation can look like)

Memory Integrity Checking

- Let's say you want to make a memory integrity checker to look for inline hooks in running code. You know at this point that certain sections such as .text are marked as non-writable. Therefore you would think what is on disk should be the same as what's in memory. So to check for changes in memory, you should be able to hash the .text in memory, hash the .text read in from disk, and compare the hashes, right?
- Maybe. If the file isn't relocated when it's loaded into memory, yes that would work*. If the file is relocated when loaded, the application of the relocation fixups will change the bytes vs. what is on disk, and therefore change the hash. You can still compare hashes though if you now take the data read in from disk and apply relocations to it in the same way the loaded would have based on the delta between the preferred load address and the actual load address.
- *There are caveats such as the fact that things like the IAT can exist in "non-writable" memory, but it still gets written at load time, and thus differs from disk. That needs to be compensated for too.

Threads

- In modern OSes, processes generally have separate address spaces (as we talked about in the IAT/EAT hooking sections). Threads are distinct units of execution flow & context which are usually managed by the kernel, but which coexist within a single process address space. Therefore each thread can see the same global variables for instance, but care must be taken (mutual exclusion) to ensure they don't incur race conditions where two threads access and modify some variable in a way which alters the other's execution by screwing up its expectations.
- Therefore it is desirable sometimes to have variables (besides local (stack) variables) which are accessible only to a single thread. Thread Local Storage (TLS) is a mechanism which MS has provided in the PE spec to support this goal. They support both regular data as well as callback functions, which can initialize/destroy data on thread creation/destruction.



Get your geek on



- Play through round 7 on your own, and then wait for the seed for the class deathmatch

See notes for citation

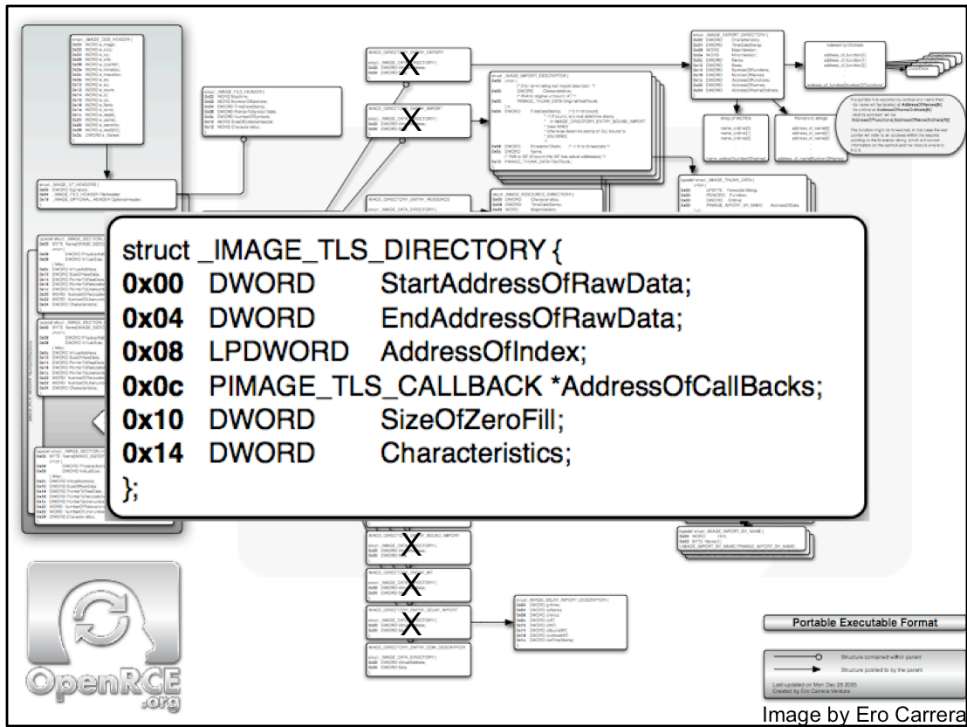
72

New 2012 – NOTE: I spent way more time on that token than I should have, so you must love and cherish it

From

<http://www.classicplastic.net/dvgi/pics-tokenstilt02.jpg>

<http://www.classicplastic.net/dvgi/pics-tokensgeneric02.jpg>



Thread Local Storage

from winnt.h

```
typedef struct _IMAGE_TLS_DIRECTORY32 {  
    DWORD    StartAddressOfRawData;  
    DWORD    EndAddressOfRawData;  
    DWORD     AddressOfIndex;  
    DWORD    AddressOfCallBacks;  
    DWORD     SizeOfZeroFill;  
    DWORD     Characteristics;  
} IMAGE_TLS_DIRECTORY32;
```

Thread Local Storage 2

- **StartAddressOfRawData** is the absolute virtual address (not RVA, and therefore subject to relocations) where the data starts.
- **EndAddressOfRawData** is the absolute virtual address (not RVA, and therefore subject to relocations) where the data ends.
- **AddressOfCallbacks** absolute virtual address points to an array of PIMAGE_TLS_CALLBACK function pointers.
- **SizeOfZeroFill** is interesting just because it's like a .bss zeroed blob tacked on after the TLS data.

C:\WINDOWS\system32\bootcfg.exe

(the only executable I could find that uses tls, thanks to a presumed bug in my property finder)

	RVA	Data	Description
bootcfg.exe			
IMAGE_DOS_HEADER	00001A20	01012000	Start Address of Raw Data
MS-DOS Stub Program	00001A24	01012014	End Address of Raw Data
IMAGE_NT_HEADERS	00001A28	01011068	Address of Index
IMAGE_SECTION_HEADER .text	00001A2C	01011018	Address of Callbacks
IMAGE_SECTION_HEADER .data	00001A30	00000000	Size of Zero Fill
IMAGE_SECTION_HEADER .tls	00001A34	00000000	Characteristics
IMAGE_SECTION_HEADER .rsrc			
BOUND_IMPORT Directory Table			
BOUND_IMPORT DLL Names			
SECTION .text			
IMPORT Address Table			
IMAGE_DEBUG_DIRECTORY			
IMAGE_TLS_DIRECTORY			
IMAGE_LOAD_CONFIG_DIRECTORY			
IMAGE_DEBUG_TYPE_CODEVIEW			
IMPORT Directory Table			
IMPORT Name Table			
IMPORT Hints/Names & DLL Names			
SECTION .data			
SECTION .tls			
SECTION .rsrc			

Note that End Address – Start Address = 0x14. Go to .tls and look at the likely file alignment padding resulting in a larger section.

76

How does one go about defining TLS?

- [http://msdn.microsoft.com/en-us/library/6yh4a9k1\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/6yh4a9k1(VS.80).aspx)
- `__declspec(thread) int tls_i = 1;`
- More info [http://msdn.microsoft.com/en-us/library/ms686749\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms686749(VS.85).aspx)
- Note: No way listed to create callbacks. For that we have to consult with unofficial sources:
- <http://www.nynaeve.net/?p=183>
- <http://hype-free.blogspot.com/2008/10/playing-tricks-with-windows-pe-loader.html>

More TLS Anti-Debug Tricks

```
/* TLS callback demonstration program.  
This program may be used to learn/illustrate the TLS callback concept.  
Copyright 2005 IIfak Guilfanov <ig@hexblog.com>
```

```
There is no standard way (from compiler vendors) of creating it.  
We use a special linker, Unilink, to create them.  
Please contact Yury Haron <yjh@styx.cabel.net> for more information  
about the linker.
```

```
*/  
  
#include <windows.h>  
#include <stdio.h>  
#include "ulnfeat.h"  
/* This is a TLS callback. It */  
void __stdcall callback(void * /*instance*/,  
                        DWORD reason,  
                        void * /*reserved*/)  
{  
    if (reason == DLL_PROCESS_ATTACH)  
    {  
        MessageBox(NULL, "Hello, world!", "Hidden message", MB_OK);  
        ExitProcess(0);  
    }  
}  
TLS_CALLBACK(c1, callback); // Unilink trick to declare callbacks  
/* This is the main function.  
It will never be executed since the callback will call ExitProcess().  
*/  
int main(void)  
{  
    return 0;  
}
```

From <http://www.hexblog.com/?p=9>

Lab: TSL Callbacks

- Use Ifak's example and Skywing's

TLS misc

- TLS callbacks can be executed when a process or thread is started or stopped. (DLL_PROCESS_ATTACH, DLL_PROCESS_DETACH, DLL_THREAD_ATTACH, DLL_THREAD_DETACH), the thing being that despite the name, an exe is called with DLL_PROCESS_ATTACH.
- TLS data generally stored in the .tls section
- Self-modifying TLS callbacks: https://www.openrce.org/blog/view/1114/Self-modifying_TLS_callbacks
- Tls callbacks could also not just bypass a breakpoint, but remove it too! :) More descriptions of possible actions here: <http://pferrie.tripod.com/papers/unpackers22.pdf>
 - In general you can find a ton of great documents at <http://pferrie.tripod.com> (I know, you thought tripod disappeared before geocities, right? ;))



Get your geek on



- Play through round 8 on your own, and then wait for the seed for the class deathmatch

See notes for citation

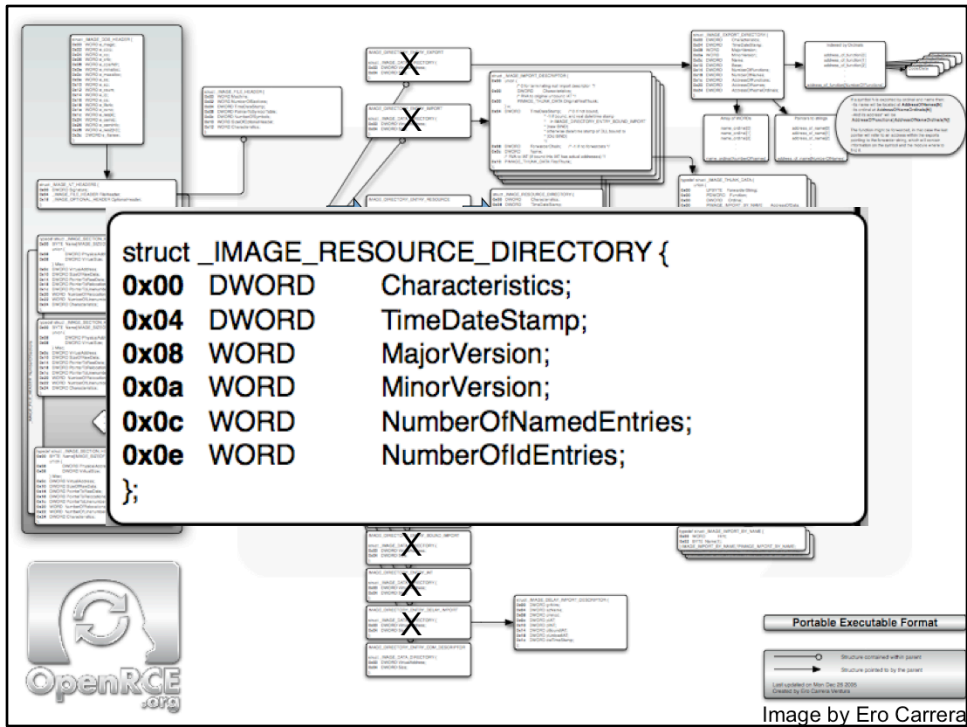
81

New 2012 – NOTE: I spent way more time on that token than I should have, so you must love and cherish it

From

<http://www.classicplastic.net/dvgi/pics-tokenstilt02.jpg>

<http://www.classicplastic.net/dvgi/pics-tokensgeneric02.jpg>



Resources

from winnt.h

- Generally stored in the .rsrc section

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD    Characteristics;
    DWORD    TimeDateStamp;
    WORD     MajorVersion;
    WORD     MinorVersion;
    WORD     NumberOfNamedEntries;
    WORD     NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY,
```

Resources 2

- Immediately following IMAGE_RESOURCE_DIRECTORY is an array of **NumberOfNamedEntries** + **NumberOfIdEntries** IMAGE_RESOURCE_DIRECTORY_ENTRY structs (with the Named entries first, followed by the ID entries.)
- A resource can be identified by a name or an ID, but not both.

Resources 3: What the...

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    union {
        struct {
            DWORD NameOffset:31;
            DWORD NameIsString:1;
        };
        DWORD Name;
        WORD Id;
    };
    union {
        DWORD OffsetToData;
        struct {
            DWORD OffsetToDirectory:31;
            DWORD DataIsDirectory:1;
        };
    };
} IMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Resources 4

- It's actually simpler than it looks. If the first DWORD's MSB is set (and therefore it starts with 8), that means the lower 31 bits are an offset to a string which is the name of the resource (and is specified like a wide character pascal string...that is, instead of being null terminated, it starts with a length which specifies the number of characters which follow...haven't been able to find what the actual type is).
- If the MSB is not set, it's treated as a WORD sized ID.
- If the MSB of the second DWORD is set, that means the lower 31 bits are an offset to another IMAGE_RESOURCE_DIRECTORY.
- If the MSB is not set, that means it's an offset to the actual data.
- All offsets are relative to the start of resource section.
- Let's walk an example

Resources 5

- Using resources in Visual Studio:
<http://msdn.microsoft.com/en-us/library/7zxb70x7.aspx> since I don't want to get into it.
- Both legitimate software and malware can embed additional binaries in the resources and then pull them out and execute them at runtime. E.g. ProcessExplorer and GMER .exes have kernel drivers embedded which they load on demand. Stuxnet also had numerous difference components such as kernel drivers, exploit code, dll injection templates, and config data embedded in resources.

ProcessExplorer.exe's resources

- Has embedded kernel drivers which it extracts and loads into memory on the fly. Different versions for x86 vs x86-64
- Look at the overloaded structs in PEView.



Get your geek on



- Play through round 9 on your own, and then wait for the seed for the class deathmatch

See notes for citation

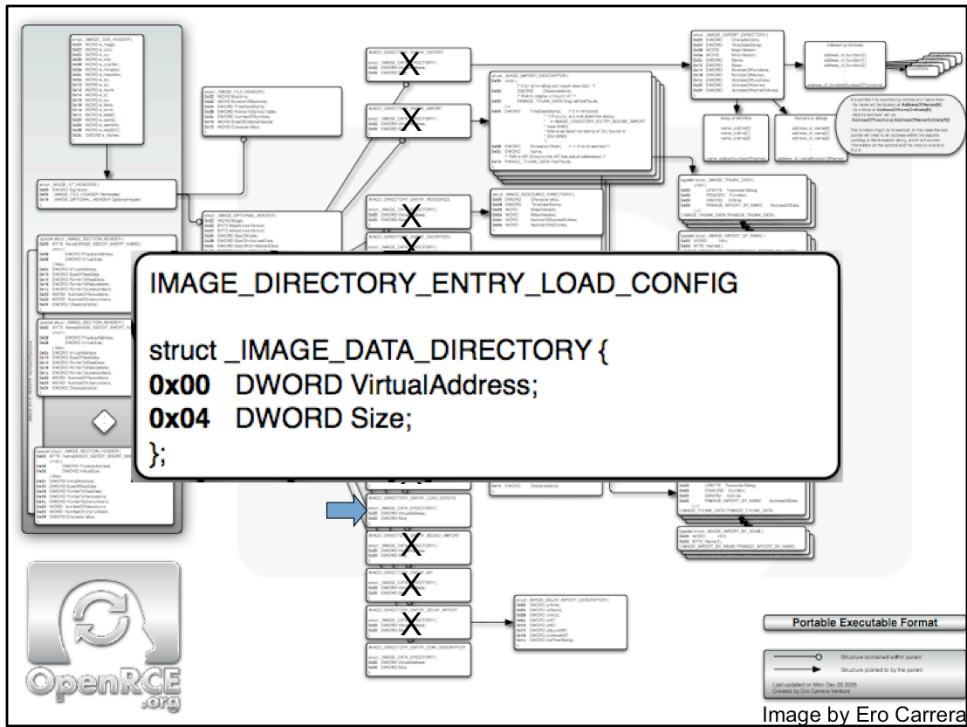
89

New 2012 – NOTE: I spent way more time on that token than I should have, so you must love and cherish it

From

<http://www.classicplastic.net/dvgi/pics-tokenstilt02.jpg>

<http://www.classicplastic.net/dvgi/pics-tokensgeneric02.jpg>



Load Configuration from winnt.h

- Another struct which doesn't rate inclusion in the picture

```
typedef struct {
    DWORD    Size;
    DWORD    TimeDateStamp;
    WORD     MajorVersion;
    WORD     MinorVersion;
    DWORD    GlobalFlagsClear;
    DWORD    GlobalFlagsSet;
    DWORD    CriticalSectionDefaultTimeout;
    DWORD    DeCommitFreeBlockThreshold;
    DWORD    DeCommitTotalFreeThreshold;
    DWORD    LockPrefixTable;           // VA
    DWORD    MaximumAllocationSize;
    DWORD    VirtualMemoryThreshold;
    DWORD    ProcessHeapFlags;
    DWORD    ProcessAffinityMask;
    WORD     CSDVersion;
    WORD     Reserved1;
    DWORD    EditList;                 // VA
    DWORD    SecurityCookie;          // VA
    DWORD    SEHandlerTable;          // VA
    DWORD    SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY32
```

91

Load Configuration from winnt.h

- Another struct which doesn't rate inclusion in the picture

```
typedef struct {
    DWORD      Size;
    DWORD      TimeDateStamp;
    WORD       MajorVersion;
    WORD       MinorVersion;
    DWORD      GlobalFlagsClear;
    DWORD      GlobalFlagsSet;
    DWORD      CriticalSectionDefaultTimeout;
    ULONGLONG DeCommitFreeBlockThreshold;
    ULONGLONG DeCommitTotalFreeThreshold;
    ULONGLONG LockPrefixTable;           // VA
    ULONGLONG MaximumAllocationSize;
    ULONGLONG VirtualMemoryThreshold;
    ULONGLONG ProcessAffinityMask;
    DWORD      ProcessHeapFlags;
    WORD       CSDVersion;
    WORD       Reserved1;
    ULONGLONG EditList;                 // VA
    ULONGLONG SecurityCookie;          // VA
    ULONGLONG SEHandlerTable;          // VA
    ULONGLONG SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY64, *PIMAGE_LOAD_CONFIG_DIRECTORY64;
```

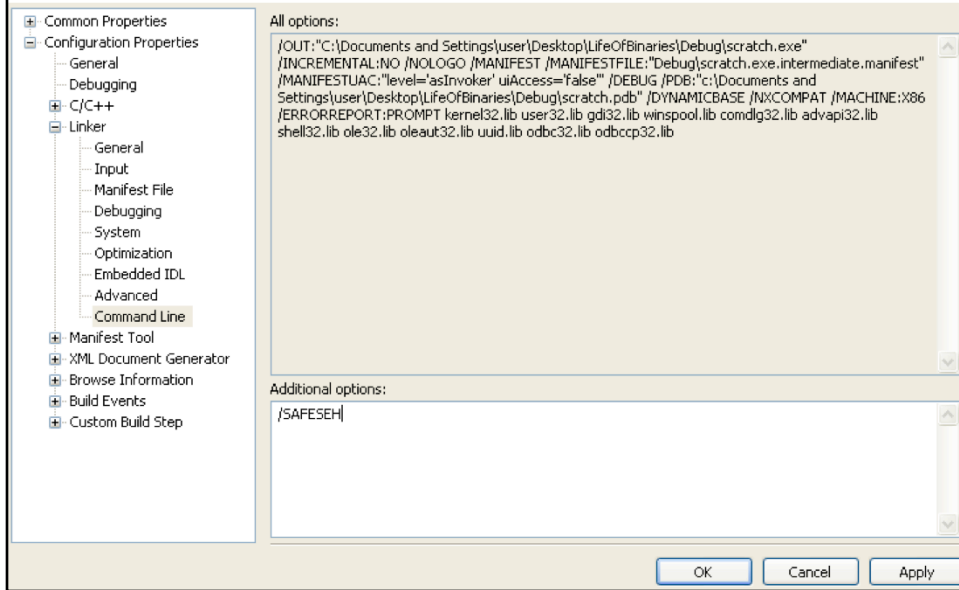
92

Load Config

- **SecurityCookie** is a VA (not RVA, therefore subject to fixups) which points at the location where the stack cookie used with the /GS flag will be.
- **SEHandlerTable** is a VA (not RVA) which points to a table of RVAs which specify the only exception handlers which are valid for use with Structured Exception Handler (SEH). The placement of the pointers to these handlers is caused by the /SAFESEH linker options.
- Take Corey Kallenberg's exploits class to see how SafeSEH mitigates (or fails to mitigate) exploits.
- **SEHandlerCount** is then just the number of entries in the array pointed to by SEHandlerTable.
- See [http://msdn.microsoft.com/en-us/library/ms680328\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms680328(VS.85).aspx) for a description of the rest of the fields

/SAFESEH

(There's no GUI option for this, and MS says to just set it manually)
[http://msdn.microsoft.com/en-us/library/9a89h429\(v=VS.100\).aspx](http://msdn.microsoft.com/en-us/library/9a89h429(v=VS.100).aspx)



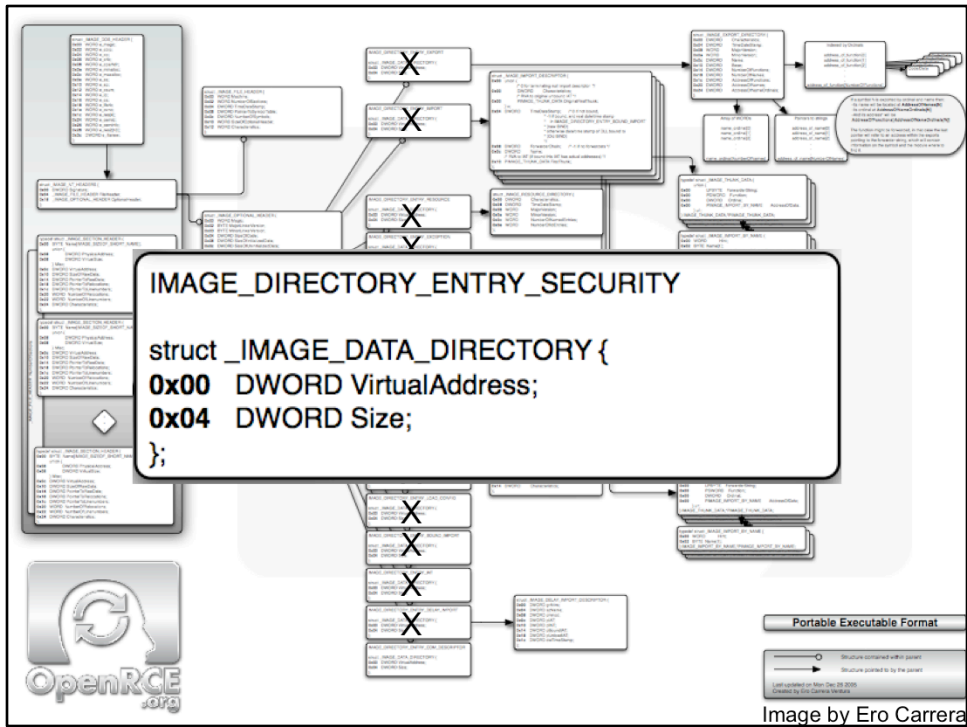
/GS "stack cookie/canary" option

Helps detect stack buffer overflows

The screenshot shows the Visual Studio Configuration Properties dialog for a project. The left-hand tree view is expanded to 'C/C++' > 'Code Generation'. The right-hand pane displays a list of compiler options. The 'Buffer Security Check' option is highlighted in blue and set to 'Yes'. Below the list, there is a section titled 'Buffer Security Check' with a descriptive text.

Enable String Pooling	No
Enable Minimal Rebuild	Yes (/Gm)
Enable C++ Exceptions	No
Smaller Type Check	No
Basic Runtime Checks	Default
Runtime Library	Multi-threaded Debug DLL (/MDd)
Struct Member Alignment	Default
Buffer Security Check	Yes
Enable Function-Level Linking	No (/GS-)
Enable Enhanced Instruction Set	Yes
Floating Point Model	<inherit from parent or project defaults>
Enable Floating Point Exceptions	No

Buffer Security Check
Check for buffer overruns; useful for closing hackable loopholes on internet servers. The default is enabled. (/GS-)



Digitally Signed Files ("Authenticode")

- Where certificates are stored
- [http://msdn.microsoft.com/en-us/library/ms537361\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms537361(VS.85).aspx)
- "The utility programs use the private key to generate a digital signature on a digest of the binary file and create a signature file containing the signed content of a public key certificate standard (PKCS) #7 signed-data object"
- ProcessExplorer as an example
- Remember way back on slide 30 we said there is that `IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY` characteristic which tells the loader to perform a digital signature check before it lets some code run? No? I can't believe you didn't remember that! It's like burned into my brain and I can't stop thinking about it no matter how hard I try!



Get your geek on



- Play through round 10 on your own, and then wait for the seed for the class deathmatch

See notes for citation

98

New 2012 – NOTE: I spent way more time on that token than I should have, so you must love and cherish it

From

<http://www.classicplastic.net/dvgi/pics-tokenstilt02.jpg>

<http://www.classicplastic.net/dvgi/pics-tokensgeneric02.jpg>

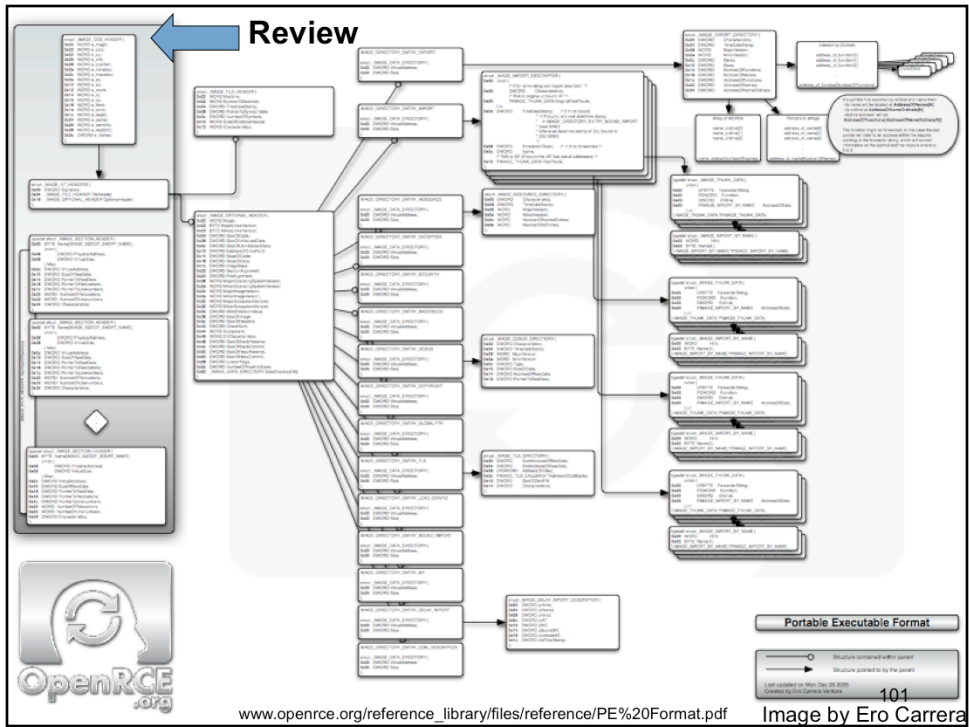
And the rest

- Most of the rest of the DataDirectory[] entries don't even apply to x86, therefore they have been moved to the backup slides

OS Loader: Load Time

(roughly based on the description of the Win2k loader here:
<http://msdn.microsoft.com/en-us/magazine/cc301727.aspx>)

1. Copy file from disk to memory per the section headers' specification of file offsets being mapped to virtual addresses. Select randomized base virtual address if ASLR compatible. Set the backend RWX permissions on the virtual memory pages (with NX if asked for.)
2. Fix relocations (if any)
3. Recursively check whether a DLL is already loaded, and if not, load imported DLLs (and any forwarded-to DLLs) and resolve imported function addresses placing them into the IAT. After every DLL is imported, call each DLL's entry point.
4. Resolve any bound imports in the main executable which are out of date.
5. Transfer execution to any TLS callbacks
6. Transfer execution to the executable's entry point specified in the OptionalHeader



www.openrce.org/reference_library/files/reference/PE%20Format.pdf

Image by Ero Carrera