PE continued

Xeno Kovah – 2012
xkovah at gmail

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

- [http://creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)

**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
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics;  // 0 for terminating null import descriptor
        DWORD OriginalFirstThunk;  // RVA to original unbound IAT (IMAGE_THUNK_DATA)
    };
    DWORD TimeDateStamp;  // 0 if not bound,
    DWORD ForwarderChain;  // -1 if no forwarders
    DWORD Name;  // RVA to IAT (if bound this IAT has actual addresses)
    DWORD FirstThunk;  // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;

- 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.
struct _IMAGE_DATA_DIRECTORY {
    0x00    DWORD VirtualAddress;
    0x04    DWORD Size;
};
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)

```c
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
    DWORD TimeDateStamp;
    WORD OffsetModuleName;
    WORD NumberOfModuleForwarderRefs;
} IMAGE_BOUND_IMPORT_DESCRIPTOR, *PIMAGE_BOUND_IMPORT_DESCRIPTOR;
```

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

- **TimeDateStamp** 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.
Notepad.exe's IMAGE_BOUND_IMPORT_DESCRIPTOR array

<table>
<thead>
<tr>
<th>VA</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000250</td>
<td>40002AB9</td>
<td>Time Date Stamp</td>
<td>2005/04/14 Mon 00:09:45 UTC</td>
</tr>
<tr>
<td>000000254</td>
<td>0058</td>
<td>Offset to Module Name</td>
<td>comdlg32.dll</td>
</tr>
<tr>
<td>000000256</td>
<td>0000</td>
<td>Number of Module ForwarderRefs</td>
<td></td>
</tr>
<tr>
<td>000000259</td>
<td>40002A11</td>
<td>Time Date Stamp</td>
<td>2005/04/14 Mon 00:10:57 UTC</td>
</tr>
<tr>
<td>000000260</td>
<td>0065</td>
<td>Offset to Module Name</td>
<td>SHELL32.dll</td>
</tr>
<tr>
<td>000000266</td>
<td>0000</td>
<td>Number of Module ForwarderRefs</td>
<td></td>
</tr>
<tr>
<td>000000269</td>
<td>40002A27</td>
<td>Time Date Stamp</td>
<td>2005/04/14 Mon 00:11:19 UTC</td>
</tr>
<tr>
<td>000000270</td>
<td>0071</td>
<td>Offset to Module Name</td>
<td>WINSPOOL.DRV</td>
</tr>
<tr>
<td>000000276</td>
<td>0000</td>
<td>Number of Module ForwarderRefs</td>
<td></td>
</tr>
<tr>
<td>000000279</td>
<td>40002A2B</td>
<td>Time Date Stamp</td>
<td>2005/04/14 Mon 00:11:52 UTC</td>
</tr>
<tr>
<td>000000280</td>
<td>007E</td>
<td>Offset to Module Name</td>
<td>COMCTL32.dll</td>
</tr>
<tr>
<td>000000286</td>
<td>0000</td>
<td>Number of Module ForwarderRefs</td>
<td></td>
</tr>
<tr>
<td>000000289</td>
<td>40002A48</td>
<td>Time Date Stamp</td>
<td>2005/04/14 Mon 00:12:52 UTC</td>
</tr>
<tr>
<td>000000290</td>
<td>0000</td>
<td>Offset to Module Name</td>
<td>msvcr73.dll</td>
</tr>
<tr>
<td>000000292</td>
<td>0000</td>
<td>Number of Module ForwarderRefs</td>
<td></td>
</tr>
<tr>
<td>000000295</td>
<td>40002A59</td>
<td>Time Date Stamp</td>
<td>2005/04/14 Mon 00:13:24 UTC</td>
</tr>
<tr>
<td>000000296</td>
<td>0000</td>
<td>Offset to Module Name</td>
<td>kernal32.dll</td>
</tr>
<tr>
<td>000000298</td>
<td>0000</td>
<td>Number of Module ForwarderRefs</td>
<td></td>
</tr>
<tr>
<td>0000002A0</td>
<td>00000000</td>
<td>Time Date Stamp</td>
<td>2005/04/14 Mon 00:13:47 UTC</td>
</tr>
<tr>
<td>0000002A4</td>
<td>0000</td>
<td>Offset to Module Name</td>
<td>ADVAPI32.dll</td>
</tr>
<tr>
<td>0000002A8</td>
<td>0000</td>
<td>Number of Module ForwarderRefs</td>
<td></td>
</tr>
</tbody>
</table>

Non-zero number of forwarder refs
Therefore this ntdll entry is an 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

NEW 2012
So how many IMAGE_BOUND_IMPORT_DESCRIPTORs here?

<table>
<thead>
<tr>
<th>VA</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>001002550</td>
<td>4002AD09</td>
<td>Time Date Stamp</td>
<td>2008/04/14 09:45 UTC</td>
</tr>
<tr>
<td>001002554</td>
<td>00000098</td>
<td>Offset to Module Name</td>
<td>cmdlg32.dll</td>
</tr>
<tr>
<td>001002556</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>001002558</td>
<td>4002A111</td>
<td>Time Date Stamp</td>
<td>2008/04/14 16:57 UTC</td>
</tr>
<tr>
<td>00100255C</td>
<td>00000000</td>
<td>Offset to Module Name</td>
<td></td>
</tr>
<tr>
<td>001002561</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>001002569</td>
<td>4002A127</td>
<td>Time Date Stamp</td>
<td>2008/04/14 11:19 UTC</td>
</tr>
<tr>
<td>00100256D</td>
<td>00000071</td>
<td>Offset to Module Name</td>
<td>winspool.drv</td>
</tr>
<tr>
<td>001002577</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>001002579</td>
<td>4002A484</td>
<td>Time Date Stamp</td>
<td>2008/04/14 08:52 UTC</td>
</tr>
<tr>
<td>00100257F</td>
<td>00000000</td>
<td>Offset to Module Name</td>
<td>comctl32.dll</td>
</tr>
<tr>
<td>001002583</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>001002587</td>
<td>4002A484</td>
<td>Time Date Stamp</td>
<td>2008/04/14 08:52 UTC</td>
</tr>
<tr>
<td>00100258D</td>
<td>00000000</td>
<td>Offset to Module Name</td>
<td>msonet.dll</td>
</tr>
<tr>
<td>001002591</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>001002595</td>
<td>4002A12C</td>
<td>Time Date Stamp</td>
<td>2008/04/14 11:24 UTC</td>
</tr>
<tr>
<td>001002599</td>
<td>00000001</td>
<td>Offset to Module Name</td>
<td>keshni.dll</td>
</tr>
<tr>
<td>00100259D</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>00100259F</td>
<td>4002A12C</td>
<td>Time Date Stamp</td>
<td>2008/04/14 11:24 UTC</td>
</tr>
<tr>
<td>0010025A3</td>
<td>00000000</td>
<td>Offset to Module Name</td>
<td>룁l.dll</td>
</tr>
<tr>
<td>0010025A5</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>0010025A9</td>
<td>4002A0BE</td>
<td>Time Date Stamp</td>
<td>2008/04/14 09:34 UTC</td>
</tr>
<tr>
<td>0010025AC</td>
<td>00000000</td>
<td>Offset to Module Name</td>
<td>iphlpapi.dll</td>
</tr>
<tr>
<td>0010025B6</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>0010025B9</td>
<td>4002A1B8</td>
<td>Time Date Stamp</td>
<td>2008/04/14 11:07 UTC</td>
</tr>
<tr>
<td>0010025C3</td>
<td>00000000</td>
<td>Offset to Module Name</td>
<td>ucrtbase.dll</td>
</tr>
<tr>
<td>0010025C7</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
<tr>
<td>0010025CA</td>
<td>00000000</td>
<td>Offset to Module Name</td>
<td></td>
</tr>
<tr>
<td>0010025CC</td>
<td>00000000</td>
<td>Number of Module Forwarder Refs</td>
<td></td>
</tr>
</tbody>
</table>

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 :)
Notepad.exe's IMAGE_IMPORT_DESCRIPTOR and IAT with bound imports
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.

Example things mentioning bind.exe
http://www.codeproject.com/KB/DLL/NeedBind.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.

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   SizeOfImage;
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaps;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY *DataDirectory[NUM_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

Which fields do we even care about, and why?
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 the support the delayed loading.
**Delay Loaded DLL**

Specifies one or more DLLs for delayed loading, use semicolon delimited list if more than one.

(DelayedLoad.dll[\_name])

<table>
<thead>
<tr>
<th>DLL Name</th>
<th>Load Delayed DLL</th>
<th>Support Unload</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Don't Support Unload</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Support Unload (DELAYLOAD)</td>
<td></td>
</tr>
</tbody>
</table>

**Delay Loaded DLL**

- Import Library: Default library
- Merge Sections: Default
- Target Machine: Any processor
- Preload: No
- CLR Thread Attribute: No threading attribute set
- CLR Image Type: Default image type
- Key File: No
- Key Container: No
- Delay Sign: No
- Error Reporting: Prompt Immediately (ERRORREPORT)
- CLR Unmanaged Code Check: No
struct _IMAGE_DELAY_IMPORT_DESCRIPTOR {
    0x00 DWORD grAttrs;
    0x04 DWORD szName;
    0x08 DWORD phmod;
    0x0c DWORD pIAT;
    0x10 DWORD pINT;
    0x14 DWORD pBoundIAT;
    0x18 DWORD pUnloadIAT;
    0x1c DWORD dwTimeStamp;
};
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    rvaMod;  // 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,
    // 0.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 rvaIAT 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.
```
.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

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

.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 0x5ad72bef (DrawThemeBackground)
...

23
mspaint's delayed import descriptors

24

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

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>003E5ECC</td>
<td>01035425</td>
<td>Virtual Address</td>
<td>0000 DrawThemeBackground</td>
</tr>
<tr>
<td>003E5EC2</td>
<td>01035400</td>
<td>Virtual Address</td>
<td>0000 OpenThemeData</td>
</tr>
<tr>
<td>003E5EC6</td>
<td>0103541E</td>
<td>Virtual Address</td>
<td>0000 CreateThemeData</td>
</tr>
<tr>
<td>003E5EC8</td>
<td>00000000</td>
<td>End of Imports</td>
<td>UsThemehl.dll</td>
</tr>
<tr>
<td>003E5ED0</td>
<td>01035280</td>
<td>Virtual Address</td>
<td>0000 GdipSaveImageToStream</td>
</tr>
<tr>
<td>003E5ED4</td>
<td>01035266</td>
<td>Virtual Address</td>
<td>0000 GdipGetImageReadFormat</td>
</tr>
<tr>
<td>003E5ED8</td>
<td>0103524A</td>
<td>Virtual Address</td>
<td>0000 GdipGetPropertySize</td>
</tr>
<tr>
<td>003E5EDC</td>
<td>01035230</td>
<td>Virtual Address</td>
<td>0000 GdipGetPropertyInfo</td>
</tr>
<tr>
<td>003E5EE0</td>
<td>0103521F</td>
<td>Virtual Address</td>
<td>0000 GdipGetImagePropertyItem</td>
</tr>
<tr>
<td>003E5EE4</td>
<td>01035204</td>
<td>Virtual Address</td>
<td>0000 GdipCreateBmpMapFromFile</td>
</tr>
<tr>
<td>003E5EE8</td>
<td>010351F9</td>
<td>Virtual Address</td>
<td>0000 GdipCreateBmpMapFromJobCM</td>
</tr>
<tr>
<td>003E5EEC</td>
<td>010351ED</td>
<td>Virtual Address</td>
<td>0000 GdipGetImageDecodersSize</td>
</tr>
<tr>
<td>003E5EF0</td>
<td>010351DE</td>
<td>Virtual Address</td>
<td>0000 GdipGetImageDecoders</td>
</tr>
<tr>
<td>003E5EF4</td>
<td>010351DC</td>
<td>Virtual Address</td>
<td>0000 GdipGetImageDecodersSize</td>
</tr>
<tr>
<td>003E5EF8</td>
<td>010351BB</td>
<td>Virtual Address</td>
<td>0000 GdipGetImageDecoders</td>
</tr>
<tr>
<td>003E5EFC</td>
<td>010351B9</td>
<td>Virtual Address</td>
<td>0000 GdipGetImageDecoders</td>
</tr>
<tr>
<td>003E5F00</td>
<td>010351AA</td>
<td>Virtual Address</td>
<td>0000 GdipGetImageDecoders</td>
</tr>
<tr>
<td>003E5F04</td>
<td>01035197</td>
<td>Virtual Address</td>
<td>0000 GdipAlloc</td>
</tr>
<tr>
<td>003E5F08</td>
<td>01035193</td>
<td>Virtual Address</td>
<td>0000 GdipConvertImage</td>
</tr>
<tr>
<td>003E5F0C</td>
<td>0103518F</td>
<td>Virtual Address</td>
<td>0000 GdipCreateBmpMapFromHBITMAP</td>
</tr>
<tr>
<td>003E5F10</td>
<td>01035185</td>
<td>Virtual Address</td>
<td>0000 GdipCreateBmpMapFromHBITMAP</td>
</tr>
<tr>
<td>003E5F14</td>
<td>0103517C</td>
<td>Virtual Address</td>
<td>0000 GdipCreateBmpMapFromHBITMAP</td>
</tr>
<tr>
<td>003E5F18</td>
<td>01035172</td>
<td>Virtual Address</td>
<td>0000 GdipCreateBmpMapFromHBITMAP</td>
</tr>
<tr>
<td>003E5F1C</td>
<td>01035168</td>
<td>Virtual Address</td>
<td>0000 GdipCreateBmpMapFromHBITMAP</td>
</tr>
<tr>
<td>003E5F20</td>
<td>00000000</td>
<td>End of Imports</td>
<td>gdylk.dll</td>
</tr>
</tbody>
</table>

These are (absolute) virtual addresses. Since the ImageBase for mspaint is 0x10000000 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

Start of stub code

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

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
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/completetestore/12/10/16/TD8w0wqM5EetVkJaLdZROA2.jpg
https://i.chzbgr.com/completetestore/12/10/12/SuxGUko-lUCvWypgOM6xQ2.jpg
https://i.chzbgr.com/completetestore/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.
struct IMAGE_EXPORT_DIRECTORY {
  0x00 DWORD Characteristics;
  0x04 DWORD TimeDateStamp;
  0x08 DWORD MajorVersion;
  0x0C WORD MinorVersion;
  0x0E DWORD Name;
  0x10 DWORD Base;
  0x14 DWORD NumberOfNames;
  0x18 DWORD AddressOfNames;
  0x1C DWORD AddressOfNameOrdinals;
};

Indexed by Ordinals

address_of_function[0]
address_of_function[1]
address_of_function[2]

... address_of_function[NumberOfFunctions]

Array of WORDs

name_ordinal[0]
name_ordinal[1]
name_ordinal[2]
...
name_ordinal[NumberOfNames]

Pointers to strings

address_of_name[0]
address_of_name[1]
address_of_name[2]
...
address_of_name[NumberOfNames]

If a symbol N is exported by ordinal and name then:
- its name will be located at AddressOfNames(N)
- its ordinal at AddressOfNameOrdinals(N)

And its address will be AddressOfFunctions(AddressOfNameOrdinals(N))

The function might be forwarded, in that case the last pointer will refer to an address within the exports pointing to the forwarder string, which will contain information on the symbol and the module where to find it.
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.

(image of diagram showing IMAGE_EXPORT_DIRECTORY and the ACEDIT.dll function)

(note the lexical order; note to self: talk about lexical ordering necessitating the ordinal table)
More about Base/ordinals relation

Modified graphical style borrowed from Matt Pietrek articles

Image Export Directory

Characteristics

TimeDateStamp

MajorVersion

MinorVersion

Name

Base = 37

NumberOfFunctions

NumberOfNames

AddressOfFunctions

AddressOfNames

AddressOfNameOrdinals

EAT

0x0000323A 0x00004010 0x00003248 0x00004BC6 0x00004ED6 0x0000596A

Index 0  Index 1  Index 2  Index 3  Index 4  Index 5
Ordinal 37  Ordinal 38  Ordinal 39  Ordinal 40  Ordinal 41  Ordinal 42

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

(search for import EditOwnerInfo by name and then by ordinal)

Modified graphical style borrowed from Matt Pietrek articles

(image showing a diagram of IMAGE_EXPORT_DIRECTORY with ACEDIT.dll, EAT, ENT, NameOrdinals, DLLMain, EditAuditInfo, EditOwnerInfo, EditPermissionInfo, FMExtensionProcW, and SedDiscretionaryActEditor)
How does one go about specifying an export?

- “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

<table>
<thead>
<tr>
<th>Additional Dependencies</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignore All Default Libraries</td>
<td>No</td>
</tr>
<tr>
<td>Ignore Specific Library</td>
<td>No</td>
</tr>
<tr>
<td>Module Definition File</td>
<td>Yes</td>
</tr>
<tr>
<td>Add Module to Assembly</td>
<td>No</td>
</tr>
<tr>
<td>Embed Managed Resource File</td>
<td>No</td>
</tr>
<tr>
<td>Force Symbol References</td>
<td>No</td>
</tr>
<tr>
<td>Delay-Loaded DLLs</td>
<td>No</td>
</tr>
<tr>
<td>Assembly Link Resource</td>
<td>No</td>
</tr>
</tbody>
</table>

**Module Definition File**

Use specified module definition file during executable creation. (DEF:maine)
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

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000006A4</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000690</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000676</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000672</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000668</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000664</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000660</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000656</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000652</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000648</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
<tr>
<td>00000644</td>
<td>Function RVA</td>
<td>ntoskrnl!ReadINIStream</td>
</tr>
</tbody>
</table>

RVA  | Raw Data | Value                               |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000006A4</td>
<td>00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>lstrcpyA, lstrncpyA</td>
</tr>
<tr>
<td>00000690</td>
<td>41 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>A, lstrcpyW, lstrncpyA</td>
</tr>
<tr>
<td>00000676</td>
<td>60 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>mp1, lstrncpyA, lstrncpyW</td>
</tr>
<tr>
<td>00000672</td>
<td>6C 73 74 6D 70 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>lstrncpyW, lstrncpyA</td>
</tr>
<tr>
<td>00000668</td>
<td>5C 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>W, lstrncpyA, lstrncpyW</td>
</tr>
<tr>
<td>00000664</td>
<td>79 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>W, lstrncpyA, lstrncpyW</td>
</tr>
<tr>
<td>00000660</td>
<td>7A 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>W, lstrncpyA, lstrncpyW</td>
</tr>
<tr>
<td>00000656</td>
<td>7C 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>W, lstrncpyA, lstrncpyW</td>
</tr>
<tr>
<td>00000652</td>
<td>7D 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>W, lstrncpyA, lstrncpyW</td>
</tr>
<tr>
<td>00000648</td>
<td>7E 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>W, lstrncpyA, lstrncpyW</td>
</tr>
<tr>
<td>00000644</td>
<td>7F 00 6C 73 74 72 63 6D 70 00 6C 73 74 72 63 6D 70</td>
<td>W, lstrncpyA, lstrncpyW</td>
</tr>
</tbody>
</table>

Note: The RVA values correspond to memory addresses within the Kernel32.dll module, indicating the locations of the exported functions.
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.

Function Redirection Tutorial

- 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.

```c
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?

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

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
struct _IMAGE_DEBUG_DIRECTORY {
  0x00  DWORD Characteristics;
  0x04  DWORD TimeDateStamp;
  0x08  WORD  MajorVersion;
  0x0a  WORD  MinorVersion;
  0x0c  DWORD  Type;
  0x10  DWORD SizeOfData;
  0x14  DWORD AddressOfRawData;
  0x18  DWORD PointerToRawData;
};
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_FIXUP 3
#define IMAGE_DEBUG_TYPE_MISC 4
#define IMAGE_DEBUG_TYPE_EXCEPTION 5
#define IMAGE_DEBUG_TYPE_SYMTAB 6
#define IMAGE_DEBUG_TYPEToLocal 7
#define IMAGE_DEBUG_TYPE_LOCAL 8
#define IMAGE_DEBUG_TYPE_BORLAND 9
#define IMAGE_DEBUG_TYPE_RESERVED10 10
#define IMAGE_DEBUG_TYPE_RESERVED11 11
Debug Info 2

- **TimeDateStamp**, 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_NT10 '0165'
#define CV_SIGNATURE_RSBS 'SSSR'

// CodeView header
struct CV_HEADER {
  DWORD CvSignature; // NTxx
  LONG Offset; // Always 0 for NT10
};

// CodeView NT10 debug information
// (used when debug information is stored in a PDB 2.00 file)
struct CV_INFO_NT10 {
  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 RSBS debug information
// (used when debug information is stored in a PDB 7.00 file)
struct CV_INFO_RSBS {
  DWORD CvSignature; // unique identifier
  DWORD Age; // an always-incrementing value
  BYTE PdbFileName[1]; // zero terminated string with the name of the PDB file
};
Therefore, how shall we interpret this?

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001670</td>
<td>00000000</td>
<td>Characteristics</td>
<td></td>
</tr>
<tr>
<td>00001674</td>
<td>3B7D65AD</td>
<td>Time Date Stamp</td>
<td>2001/08/17 Fri 20:59:25 UTC</td>
</tr>
<tr>
<td>00001678</td>
<td>0000</td>
<td>Major Version</td>
<td></td>
</tr>
<tr>
<td>0000167A</td>
<td>0000</td>
<td>Minor Version</td>
<td></td>
</tr>
<tr>
<td>0000167C</td>
<td>00000002</td>
<td>Type</td>
<td>IMAGE_DEBUG_TYPECODEVIEW</td>
</tr>
<tr>
<td>00001680</td>
<td>0000001C</td>
<td>Size of Data</td>
<td></td>
</tr>
<tr>
<td>00001684</td>
<td>00002524</td>
<td>Address of Raw Data</td>
<td></td>
</tr>
<tr>
<td>00001688</td>
<td>00001924</td>
<td>Pointer to Raw Data</td>
<td></td>
</tr>
</tbody>
</table>

Header, CvSignature, Offset

CV_HEADER, Header

Signature, Age

<table>
<thead>
<tr>
<th>RVA</th>
<th>Raw Data</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00002524</td>
<td>4E 42 31 30 00 00 00 00 AD 85 7D 3B 01 00 00 00 N810 ........</td>
<td>acledit.pdb.</td>
</tr>
<tr>
<td>00002534</td>
<td>61 63 6C 65 64 69 74 2E 70 64 62 00</td>
<td></td>
</tr>
</tbody>
</table>

PdbFileName

CV_INFO_PDB20
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\AuroraVNC\Avc\Release\AVC.pdb
- b:\myrtus\src\objfre_w2k_x86\i386\guava.pdb
struct IMAGE_DATA_DIRECTORY {
    0x00 DWORD VirtualAddress;
    0x04 DWORD Size;
};
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;
```
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 $(\text{SizeOfBlock} - \text{sizeof(IMAGE_BASE_RELOCATION)}) / \text{sizeof(WORD)}$. 
### Relocations example acedit.dll

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001618C</td>
<td>3DEB</td>
<td>Type RVA</td>
<td>0000020E8B IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>0001618E</td>
<td>3F2B</td>
<td>Type RVA</td>
<td>00000272B IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>00016193</td>
<td>00000000</td>
<td>RVA of Block</td>
<td></td>
</tr>
<tr>
<td>00016194</td>
<td>0000003C</td>
<td>Size of Block</td>
<td></td>
</tr>
<tr>
<td>00016196</td>
<td>32FB</td>
<td>Type RVA</td>
<td>0000032FB IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>0001619A</td>
<td>3507</td>
<td>Type RVA</td>
<td>00000307 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>0001619C</td>
<td>336A</td>
<td>Type RVA</td>
<td>0000036A IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>0001619E</td>
<td>3AA2</td>
<td>Type RVA</td>
<td>000003A2 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161A0</td>
<td>336B</td>
<td>Type RVA</td>
<td>0000036B IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161A2</td>
<td>3411</td>
<td>Type RVA</td>
<td>000003411 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161A4</td>
<td>341B</td>
<td>Type RVA</td>
<td>00000341B IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161A6</td>
<td>346A</td>
<td>Type RVA</td>
<td>00000346A IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161A8</td>
<td>3D21</td>
<td>Type RVA</td>
<td>000003D21 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161AC</td>
<td>34B3</td>
<td>Type RVA</td>
<td>0000034B3 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161AE</td>
<td>34E2</td>
<td>Type RVA</td>
<td>0000034E2 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161B0</td>
<td>34FC</td>
<td>Type RVA</td>
<td>0000034FC IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161B2</td>
<td>3517</td>
<td>Type RVA</td>
<td>000003517 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161B4</td>
<td>351E</td>
<td>Type RVA</td>
<td>00000351E IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161B6</td>
<td>3749</td>
<td>Type RVA</td>
<td>000003749 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161B8</td>
<td>3775</td>
<td>Type RVA</td>
<td>000003775 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161BA</td>
<td>3813</td>
<td>Type RVA</td>
<td>000003813 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161BC</td>
<td>3F0F</td>
<td>Type RVA</td>
<td>000003F0F IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161BE</td>
<td>3D12</td>
<td>Type RVA</td>
<td>000003D12 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161C0</td>
<td>3D02</td>
<td>Type RVA</td>
<td>000003D02 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161C2</td>
<td>3D06</td>
<td>Type RVA</td>
<td>000003D06 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161C4</td>
<td>3E15</td>
<td>Type RVA</td>
<td>000003E15 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161C6</td>
<td>3E36</td>
<td>Type RVA</td>
<td>000003E36 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161C8</td>
<td>3E3F</td>
<td>Type RVA</td>
<td>000003E3F IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
<tr>
<td>000161CA</td>
<td>3000</td>
<td>Type RVA</td>
<td></td>
</tr>
<tr>
<td>000161CC</td>
<td>00040000</td>
<td>RVA of Block</td>
<td></td>
</tr>
<tr>
<td>000161CD</td>
<td>00000020</td>
<td>Size of Block</td>
<td></td>
</tr>
<tr>
<td>000161DE</td>
<td>3256</td>
<td>Type RVA</td>
<td>000004256 IMAGE_BASED_BASE_HIGHLOW</td>
</tr>
</tbody>
</table>
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:

```c
#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_MIPS64     4
#define IMAGE_REL_BASED_MIPS64_MIPS 5
#define IMAGE_REL_BASED_MIPS64_MIPS16 6
#define IMAGE_REL_BASED_IA64     9
#define IMAGE_REL_BASED_D1644   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

<table>
<thead>
<tr>
<th>Offset</th>
<th>RVA of Block</th>
<th>Size of Block</th>
<th>Type RVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0021690</td>
<td>0x0003000</td>
<td>0x000003C</td>
<td>0x03FB</td>
</tr>
<tr>
<td>0x0021698</td>
<td>3307</td>
<td>0x0003007</td>
<td>0x034A</td>
</tr>
</tbody>
</table>

- So in the above if the file was being relocated, the loader would take the relocation target \( \text{WORD 0x32FB, the upper 4 bits are 0x3 = IMAGE\_REL\_BASED\_HIGHLOW} \). The lower 12 bits are 0x2FB. Given the type, we do \( \text{(VirtualAddress (0x3000) + lower 12 bits (0x2FB)) == 0x32FB is the RVA of the location which needs to be fixed.} \)
- Then the loaded 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

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
struct _IMAGE_TLS_DIRECTORY {
  0x00 DWORD StartAddressOfRawData;
  0x04 DWORD EndAddressOfRawData;
  0x08 LPDWORD AddressOfIndex;
  0x0c PIMAGE_TLS_CALLBACK *AddressOfCallBacks;
  0x10 DWORD SizeOfZeroFill;
  0x14 DWORD Characteristics;
};
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 was fine, thanks to a suspected bug in my snapstore-finder)

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001A20</td>
<td>01012000</td>
<td>Start Address of Raw Data</td>
</tr>
<tr>
<td>00001A24</td>
<td>01013014</td>
<td>End Address of Raw Data</td>
</tr>
<tr>
<td>00001A28</td>
<td>01011058</td>
<td>Address of Index</td>
</tr>
<tr>
<td>00001A2C</td>
<td>01011018</td>
<td>Address of Callbacks</td>
</tr>
<tr>
<td>00001A30</td>
<td>00000000</td>
<td>Size of Zero Fill</td>
</tr>
<tr>
<td>00001A34</td>
<td>00000000</td>
<td>Characteristics</td>
</tr>
</tbody>
</table>

Note that End Address – Start Address = 0x14. Go to .tls and look at the likely file alignment padding resulting in a larger section.
How does one go about defining TLS?

- __declspec( thread ) int tls_i = 1;
- Note: No way listed to create callbacks. For that we have to consult with unofficial sources:
  - http://www.nynaeve.net/?p=183
More TLS Anti-Debug Tricks

#include <windows.h>
#include <stdio.h>
#include "utf8t.h"

/* This is a TLS callback. */
void __stdcall callback(void **instance,
            WORD reason,
            void **reserved)
{
    if (reason == DUIL_PROCESS_ATTACH)
    {
        MessageBox(NULL, "Hello, world!", "hidden message", MB_OK);
        ExitProcess(0);
    }
    TLS_CALLBACK(ct, callback);  // Unlink 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 Ilfak'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

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
struct _IMAGE_RESOURCE_DIRECTORY {
    0x00   DWORD   Characteristics;
    0x04   DWORD   TimeDateStamp;
    0x08   WORD    MajorVersion;
    0x0a   WORD    MinorVersion;
    0x0c   WORD    NumberOfNamedEntries;
    0x0e   WORD    NumberOfIdEntries;
};
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 IMAGERESOURCEDIRECTORY is an array of NumberOfNamedEntries + NumberOfIdEntries IMAGERESOURCEDIRECTORYENTRY 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.
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

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
IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG

struct _IMAGE_DATA_DIRECTORY {
  0x00  DWORD VirtualAddress;
  0x04  DWORD Size;
};
Load Configuration from winnt.h

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

```c
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     CSVersion;
    WORD     Reserved1;
    DWORD    EditList;         // VA
    DWORD    SecurityCookie;   // VA
    DWORD    SEventHandlerTable;  // VA
    DWORD    SEventHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY32
```
Load Configuration from winnt.h

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

```c
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      CSVersion;
    WORD      Reserved1;
    ULONGLONG EditList;          // VA
    ULONGLONG SecurityCookie;    // VA
    ULONGLONG SEHandlerTable;    // VA
    ULONGLONG SEHandlerCount;
} IMAGE_LOAD_CONFIG_DIRECTORY64, *PIMAGE_LOAD_CONFIG_DIRECTORY64;
```
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.

/SAFESEH
(There’s no GUI option for this, and MS says to just set it manually)
/GS "stack cookie/canary" option
Helps detect stack buffer overflows
struct IMAGE_DATA_DIRECTORY {
    0x00   DWORD VirtualAddress;
    0x04   DWORD Size;
};
Digitally Signed Files
(“Authenticode”)

- Where certificates are stored
- “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 IMAGES_FILE_MUST_BE_SIGNED 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

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:

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