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/

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

- Common Object File Format (COFF) was introduced with UNIX System V.
- Windows has Portable Executable (PE) format. Derived from COFF.
- Modern unix derivatives tend to use the Executable and Linkable Format (ELF).
- Mac OS X uses the Mach Object (Mach-o) format.
Different target binary formats

• Executable (.exe on Windows, no suffix on Linux)
  – A program which will either stand completely on its own, containing all code necessary for its execution, or which will request external libraries that it will depend on (and which the loader must provide for the executable to run correctly)

• Dynamic Linked Library (.dll) on Windows == Shared Library aka Shared Object (.so) on Linux
  – Needs to be loaded by some other program in order for any of the code to be executed. The library *may* have some code which is automatically executed at load time (the DllMain() on windows or init() on Linux). This is as opposed to a library which executes none of its own code and only provides code to other programs.

• Static Library (.lib on Windows, .a on Linux)
  – Static libraries are just basically a collection of object files, with some specific header info to describe the organization of the files.
Common Windows PE File Extensions

- .exe - Executable file
- .dll - Dynamic Link Library
- .sys/.drv - System file (Kernel Driver)
- .ocx - ActiveX control
- .cpl - Control panel
- .scr - Screensaver

Note: .lib files (Static Libraries) don't have the same "DOS Header then PE Header" format that the rest of these do.
Building Windows Executable, Dynamic Linked Library, Static Library

<table>
<thead>
<tr>
<th>Common Properties</th>
<th>General Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Properties</td>
<td>Output Directory =</td>
</tr>
<tr>
<td>General</td>
<td>Intermediate Directory =</td>
</tr>
<tr>
<td>C/C++</td>
<td>Extensions to Delete on Clean =</td>
</tr>
<tr>
<td>Linker</td>
<td>Build Log File =</td>
</tr>
<tr>
<td>Manifest Tool</td>
<td>Inherited Project Property Sheets =</td>
</tr>
<tr>
<td>XML Document Generator</td>
<td>Enable Managed Incremental Build =</td>
</tr>
<tr>
<td>Browse Information</td>
<td>Project Defaults</td>
</tr>
<tr>
<td>Build Events</td>
<td>Configuration Type = Application (.exe)</td>
</tr>
<tr>
<td>Custom Build Step</td>
<td>Use of MFC = Makfile</td>
</tr>
<tr>
<td></td>
<td>Use of ATL = Application (.exe)</td>
</tr>
<tr>
<td></td>
<td>Character Set = DynamicLibrary (.dll)</td>
</tr>
<tr>
<td></td>
<td>Common Language Runtime support = StaticLibrary (.lib)</td>
</tr>
<tr>
<td></td>
<td>Whole Program Optimization = Utility</td>
</tr>
</tbody>
</table>

**Configuration Type**

Specifies the type of output this configuration generates.
Further Reading

- The definitions of all of the structures for a PE file are in WINNT.h
- An In-Depth Look into the Win32 Portable Executable File Format Part 1 & 2 – An excellent set of reference articles by Matt Pietrek (this is how I first learned)
  http://msdn.microsoft.com/en-us/magazine/cc301805.aspx,
- The official spec:
  http://www.microsoft.com/whdc/system/platform/firmware/pecoff.mspx
- All the VisualStudio compiler options (note, some aren't in the GUI, you have to add them manually): http://msdn.microsoft.com/en-us/library/fwkeyyhe(v=VS.90).aspx
Your new best friends: PEView and CFF Explorer

- I like PEView (http://www.magma.ca/~wjr/PEview.zip) by Wayne Radburn for looking at PE files. It's no frills and gives you a view very close to what you would see if you were looking at the structs in a program which was parsing the file.
- Once you've seen and understood stuff in PEView, you can graduate to the much more feature-full CFF Explorer by Daniel Pistelli (it lets you hex edit the file or disassemble code! :D) (http://www.ntcore.com/exsuite.php)
Tools: WinDbg

- We're going to be using WinDbg for basic userspace debugging (as opposed to kernel debugging like in the Intermediate x86 class)
Terminology

- RVA - Relative Virtual Address. This indicates some displacement relative to the start (base) of a binary in memory.
- AVA – Absolute Virtual Address, more often just "Virtual Address", but I want to be exact. This is a specific address memory where something can be found.
- So if the base is 0x80000000, and the AVA was 0x80001000, then the RVA would be 0x1000.
- If the base is 0x80000000, and the AVA was 0xC123000f, then the RVA would be 0x4123000f.
- RVA = VA – Base
- Windows uses RVAs extensively in the PE format, unlike ELF which uses just AVAs.
Terminology 2

- Windows uses the following variable size names:
  - CHAR = character = 1 byte
  - WORD = word = 2 bytes
    - SHORT = short integer = 2 bytes
  - DWORD = double-word = 4 bytes
    - LONG = long integer = 4 bytes
  - QWORD = quad-word = 8 bytes
    - LONGLONG = long long integer = 8 bytes
New 2012 -> Q: Ask students what the next offset after 0x3C would be. A: 0x40 (ensures they get what I just said about sizes, and they have their hex math down)
The MS-DOS File Header
(from winnt.h)
BLUE means the stuff we actually care about

typedef struct _IMAGE_DOS_HEADER {
    WORD  e_magic;       // Magic number
    WORD  e_cblp;        // Bytes on last page of file
    WORD  e_cp;          // Pages in file
    WORD  e_crlc;        // Relocations
    WORD  e_cparhdr;     // Size of header in paragraphs
    WORD  e_minalloc;    // Minimum extra paragraphs needed
    WORD  e_maxalloc;    // Maximum extra paragraphs needed
    WORD  e_ss;          // Initial (relative) SS value
    WORD  e_sp;          // Initial SP value
    WORD  e_dp;          // Initial IP value
    WORD  e_csum;        // Checksum
    WORD  e_ip;          // Initial IP value
    WORD  e_cs;          // Initial (relative) CS value
    WORD  e_lfarlc;      // File address of relocation table
    WORD  e_ovno;        // Overlay number
    WORD  e_res[4];      // Reserved words
    WORD  e_oemid;       // OEM identifier (for e_oeminfo)
    WORD  e_oeminfo;     // OEM information; e_oemid specific
    WORD  e_res2[10];    // Reserved words
    LONG  e_lfanew;      // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
The DOS Header

- **e_magic** is set to ASCII 'MZ' which is from Mark Zbikowski who developed MS-DOS
- For most Windows programs the DOS header contains a stub DOS program which does nothing but print out “This program cannot be run in DOS mode”
- The main thing we care about is the **e_lfanew** field, which specifies a file offset where the PE header can be found (a file pointer if you will)
struct _IMAGE_NT_HEADERS {
  0x00  DWORD Signature;
  0x04  _IMAGE_FILE_HEADER FileHeader;
  0x18  _IMAGE_OPTIONAL_HEADER OptionalHeader;
};
NT Header or “PE Header”  
(from winnt.h)

typedef struct IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGEFILEHEADER FileHeader;
    IMAGEOPTIONALHEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;

• Signature == 0x00004550 aka ASCII string “PE” in little endian order in a DWORD  
• Otherwise, just a holder for two other *embedded* (not pointed to) structs
struct _IMAGE_FILE_HEADER {
    0x00    WORD Machine;
    0x02    WORD NumberOfSections;
    0x04    DWORD TimeDateStamp;
    0x08    DWORD PointerToSymbolTable;
    0x0c    DWORD NumberOfSymbols;
    0x10    WORD SizeOfOptionalHeader;
    0x12    WORD Characteristics;
};
File Header
(from winnt.h)

typedef struct _IMAGE_FILE_HEADER {
    WORD   Machine;
    WORD   NumberOfSections;
    DWORD  TimeDateStamp;
    DWORD  PointerToSymbolTable;
    DWORD  NumberOfSymbols;
    WORD   SizeOfOptionalHeader;
    WORD   Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
File Header 2

- **Machine** specifies what architecture this is supposed to run on. This is our first indication about 32 or 64 bit binary
- Value of **014C** = x86 binary, aka 32 bit binary, aka **PE32** binary
- Value of **8664** = x86-64 binary, aka AMD64 binary, aka 64 bit binary, aka **PE32+** binary
File Header 3

- The **TimeDateStamp** field is pretty interesting. It's a Unix timestamp (seconds since epoc, where epoc is 00:00:00 UTC on January 1st 1970) and is set at link time.
  - Can be used as a “unique version” for the given file (the version compiled on Jan 1 2010 may or may not be meaningfully different than that compiled on Jan 2 2010)
  - Can be used to know when a file was linked (useful for determining whether an attacker tool is “fresh”, or correlating with other forensic evidence, keeping in mind that attackers can manipulate it)
File Header 4

- Oh hay, Hoglund started using the TimeDateStamp as a characteristic for malware attribution (BlackHat Las Vegas 2010, slides not posted yet)
- **NumberOfSections** tells you how many section headers there will be later
• The **Characteristics** field is used to specify things like:

  ```
  #define IMAGE_FILE_EXECUTABLE_IMAGE 0x0002
  // File is executable (i.e. no unresolved external references).
  #define IMAGE_FILE_LINE_NUMS_STRIPPED 0x0004
  // Line numbers stripped from file.
  #define IMAGE_FILE_LARGE_ADDRESS_AWARE 0x0020
  // App can handle >2gb addresses
  #define IMAGE_FILE_32BIT_MACHINE 0x0100
  // 32 bit word machine.
  #define IMAGE_FILE_SYSTEM 0x1000
  // System File. (Xeno: I don't see that set on .sys files)
  #define IMAGE_FILE_DLL 0x2000
  // File is a DLL.
  ```
File Header 4

- SizeOfOptionalHeader can *theoretically* be shrunk to exclude “data directory” fields (talked about later) which the linker doesn't need to include. But I don't think it ever is in practice.
- PointerToSymbolTable, NumberOfSymbols not used anymore now that debug info is stored in separate file
Welcome to the Binary Scavenger Hunt

We got fun and games!

New 2012 – changed this to a screen shot to save size

From

http://www.defensereview.com/stories/predatorcamo/Predator%20Camo_Large.jpg
http://media.moddb.com/images/mods/1/12/11314/00004.jpg
http://remingtons.files.wordpress.com/2010/07/arnold-predator.jpg
http://images.alphacoders.com/178/178993.jpg
http://www.iamexpat.nl/app/webroot/upload/files/Topics/Lifestyle/Whats-on/Guns-n-Roses-guns-n-roses-589484_655_475(1).jpg
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

How to play

• Open 2 instances of cmd.exe
  – One will be for independent work, one will be for class-competition
• Start the game in both instances by doing "python BinHunt.py"
• In the independent work one, enter 0 for the mode
• In the class one, enter 2 for the mode
About This Game

- Part of a larger effort to create games to reinforce material from security classes
- Allows for interesting data collection. Inspired by this picture from khanacademy.org/about:
Example of me playing the same round multiple times and getting better each time

(I tried not to memorize any of the answers, and go through the motions of looking them up with the tools, I basically just got faster with the tools)

Xeno playing BinaryScavengerHunt Round 1 and 2 three times in a row
(seed = 1349311990)
Example of Entire First Class to Beta Test BinaryScavengerHunt

• TODO
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    SizeOfSubSystem;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
typedef struct _IMAGE_OPTIONAL_HEADER64 {  
    WORD Magic;  
    BYTE MajorLinkerVersion;  
    BYTE MinorLinkerVersion;  
    DWORD SizeOfCode;  
    DWORD SizeOfInitializedData;  
    DWORD SizeOfUninitializedData;  
    DWORD AddressOfEntryPoint;  
    DWORD BaseOfCode;  
    ULONGLONG 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;  
    ULONGLONG SizeOfStackReserve;  
    ULONGLONG SizeOfStackCommit;  
    ULONGLONG SizeOfHeapReserve;  
    ULONGLONG SizeOfHeapCommit;  
    DWORD LoaderFlags;  
    DWORD NumberOfRvaAndSizes;  
    IMAGE_DATA_DIRECTORY DataDirectory[NUM_DIRECTORY_ENTRIES];  
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
Optional Header 0

- It's not at all optional ;)
- **Magic** is the true determinant of whether this is a PE32 or PE32+ binary
- Depending on the value, the optional header will be interpreted as having a couple 32 or 64 bit fields.
- 0x10C = 32 bit, PE32
- 0x20B = 64 bit, PE32+
Optional Header 1

- It's not at all optional ;)
- **AddressOfEntryPoint** specifies the RVA of where the loader starts executing code once it's completed loading the binary. Don't assume it just points to the beginning of the .text section, or even the start of `main()`.
- **SizeOfImage** is the amount of contiguous memory that must be reserved to load the binary into memory
Optional Header 2

- **SectionAlignment** specifies that sections (talked about later) must be aligned on boundaries which are multiples of this value. E.g. if it was 0x1000, then you might expect to see sections starting at 0x1000, 0x2000, 0x5000, etc.

- **FileAlignment** says that data was written to the binary in chunks no smaller than this value. Some common values are 0x200 (512, the size of a HD sector), and 0x80 (not sure what the significance is)
Optional Header 3

- **ImageBase** specifies the preferred virtual memory location where the beginning of the binary should be placed.
- Microsoft recommends developers “rebase” DLL files. That is, picking a non-default memory address which will not conflict with any of the other libraries which will be loaded into the same memory space.
- If the binary cannot be loaded at ImageBase (e.g. because something else is already using that memory), then the loader picks an unused memory range. Then, every location in the binary which was compiled assuming that the binary was loaded at ImageBase must be fixed by adding the difference between the actual ImageBase minus desired ImageBase.
- The list of places which must be fixed is kept in a special “relocations” (.reloc) section.
- This is because MS doesn't support position-independent code.
Optional Header 4

- **DLLCharacteristics** specifies some important security options like ASLR and non-executable memory regions for the loader, and the effects are not limited to DLLs.

  - `#define IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE 0x0100` // DLL run more.
  - `#define IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY 0x0200` // Code Integrity Image
  - `#define IMAGE_DLLCHARACTERISTICS_NO_SEH 0x0400` // image does not use SEH. No SE handler may reside in this image

- **IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE** is set when linked with the `/DYNAMICBASE` option. This is the flag which tells the OS loader that this binary supports ASLR. Must be used with the `/FIXED:NO` option for .exe files otherwise they won’t get relocation information.

- **IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY** says to check at load time whether the digitally signed hash of the binary matches.

- **IMAGE_DLLCHARACTERISTICS_NX_COMPAT** is set with the `/NXCOMPAT` linker option, and tells the loader that this image is compatible with Data Execution Prevention (DEP) and that non-executable sections should have the NX flag set in memory (we learn about NX in the Intermediate x86 class)

- **IMAGE_DLLCHARACTERISTICS_NO_SEH** says that this binary never uses structured exception handling, and therefore no default handler should be created (because in the absence of other options that SEH handler is potentially vulnerable to attack.)
Security-Relevant Linker Options

- `/DYNAMICBASE` – Mark the properties to indicate that this executable will work fine with Address Space Layout Randomization (ASLR)
- `/FIXED:NO` – This will force the linker to generate relocations information for an executable, so that it is capable of having its base address modified by ASLR (otherwise usually .exe files don't have relocations information, and therefore can't be moved around in memory)
- `/NXCOMPAT` – Mark the properties to indicate that this executable will work fine with Data Execution Protection (which marks data memory regions such as the stack and heap as non-executable). DEP is just MS's name for utilizing the NX/XD bit to mark memory pages as non-executable (Which we'll talk about more in the Intermediate x86 class)
- `/SAFESEH` – Safe Structured Exception Handling. Enforces that the only SEH things you can use are ones which are specified in the binary (it will automatically add any ones defined in your code to a list that will be talked about later)
# ASLR & DEP/NX

**ASLR**

- Entry Point: No Entry Point
- Set Checksum: No
- Base Address
- Randomized Base Address: Enable Image Randomization (DYNAMICBASE)
- **Fixed Base Address**
  - Generate a relocation section (FIXED:NO)
- Data Execution Prevention (DEP): Image is compatible with DEP (NX/COMPAT)
- Turn Off Assembly Generation: No
- Delay Loaded DLL
- Import Library
- Merge Sections
- Target Machine: MachineX86 (/MACHINE:X86)
- 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.PROMPT)
- CLR Unmanaged Code Check: No

**DEP/NX**

- Fixed Base Address
  - Specifies if image must be loaded at a fixed address. (FIXED:No)
### ASLR & DEP/NX in the Binary

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000100</td>
<td>Section Header</td>
<td>IMAGE_DOS_HEADER</td>
<td>Signature</td>
</tr>
<tr>
<td>0x000200</td>
<td>Section</td>
<td>IMAGE_NT_HEADERS</td>
<td>Signature</td>
</tr>
<tr>
<td>0x000300</td>
<td>Section</td>
<td>IMAGE_FILE_HEADER</td>
<td>Size of Optional Header</td>
</tr>
<tr>
<td>0x000400</td>
<td>Section</td>
<td>IMAGE_FILE_HEADER</td>
<td>Size of Data</td>
</tr>
<tr>
<td>0x000500</td>
<td>Section</td>
<td>IMAGE_FILE_HEADER</td>
<td>Number of Sections</td>
</tr>
<tr>
<td>0x000600</td>
<td>Section</td>
<td>IMAGE_FILE_HEADER</td>
<td>Imports</td>
</tr>
<tr>
<td>0x000700</td>
<td>Section</td>
<td>IMAGE_FILE_HEADER</td>
<td>Resources</td>
</tr>
<tr>
<td>0x000800</td>
<td>Section</td>
<td>IMAGE_FILE_HEADER</td>
<td>Export</td>
</tr>
<tr>
<td>0x000900</td>
<td>Section</td>
<td>IMAGE_FILE_HEADER</td>
<td>Symbolic</td>
</tr>
<tr>
<td>0x000A00</td>
<td>Section</td>
<td>IMAGE_FILE_HEADER</td>
<td>String</td>
</tr>
</tbody>
</table>

**Relocations**

**ASLR**

- IMAGE_DLLCHARACTERISTIC_DYNAMIC_BASE
- IMAGE_DLLCHARACTERISTIC_FORCE_INTEGRITY

**DEP/NX**

- IMAGE_DLLCHARACTERISTIC_NOActivityIndicator
- IMAGE_DLLCHARACTERISTIC_TERMINAL_SERVER_AWARE

Loader Flags

- 0x00000000
- 0x00000000
- 0x00000000
- 0x00000000
- 0x00000000
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES 16
(from winnt.h)

Therefore, while FileHeader.SizeOfOptionalHeader could technically change, in practice it's fixed.
Optional Header 3

- The type of `DataDirectory[16]` is `IMAGE_DATA_DIRECTORY`

```c
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

- `VirtualAddress` is a RVA pointer to some other structure of the given Size
Optional Header 4
(from winnt.h)

- There is a predefined possible structure for each index in DataDirectory[]

```c
#define IMAGE_DIRECTORY_ENTRY_EXPORT 0 // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT 1 // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE 2 // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION 3 // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY 4 // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASE_RELOC 5 // Base Reolocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG 6 // Debug Directory
  // IMAGE_DIRECTORY_ENTRY_COPYRIGHT 7 // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE 7 // Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR 8 // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS 9 // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG 10 // Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT 11 // Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT 12 // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT 13 // Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14 // COM Runtime descriptor
```

- We will return to each entry in the DataDirectory[] later.
- Note that while the array is 16 elements, only 15 (0-14) are defined.
typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD    e_magic;  // Magic number
    WORD    e_cblp;  // Bytes on last page of file
    WORD    e_cp;  // Pages in file
    WORD    e_crlc;  // Relocations
    WORD    e_cparhdr; // Size of header in paragraphs
    WORD    e_minalloc; // Minimum extra paragraphs needed
    WORD    e_maxalloc; // Maximum extra paragraphs needed
    WORD    e_ss;  // Initial (relative) SS value
    WORD    e_sp;  // Initial SP value
    WORD    e_csum;  // Checksum
    WORD    e_ip;  // Initial IP value
    WORD    e_cs;  // Initial (relative) CS value
    WORD    e_lfarlc; // File address of relocation table
    WORD    e_ovno; // Overlay number
    WORD    e_res[4]; // Reserved words
    WORD    e_oemid; // OEM identifier (for e_oeminfo)
    WORD    e_oeminfo; // OEM information; oemid specific
    WORD    e_res2[10]; // Reserved words
    LONG    e_lfanew;  // File address of new exe header
} IMAGE_DOS_HEADER, *IMAGE_DOS_HEADER;
Get your geek on

• Play through round 2 on your own, and then wait for the seed for the class deathmatch
• You can skip to level 2 by starting the game with "python BinHunt.py 2"

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
Sections

- Sections group portions of code or data (Von Neumann sez: “What's the difference?! :P”) which have similar purpose, or should have similar memory permissions (remember the linking merge option? That would be for merging sections with "similar memory permissions")
Sections 2

- Common section names:
- `.text` = Code which should never be paged out of memory to disk
- `.data` = read/write data (globals)
- `.rdata` = read-only data (strings)
- `.bss` = (Block Started by Symbol or Block Storage Segment or Block Storage Start depending on who you ask (the CMU architecture book says the last one))
- MS spec says of `.bss` "Uninitialized data (free format)"
  which is the same as for ELF.
- In practice, the `.bss` seems to be merged into the `.data`
  section by the linker for the binaries I've looked at
- `.idata` = import address table (talked about later). In
  practice, seems to get merged with `.text` or `.rdata`
- `.edata` = export information
Sections 3

- PAGE* = Code/data which it's fine to page out to disk if you're running low on memory (not in the spec, seems to be used primarily for kernel drivers)
- .reloc = Relocation information for where to modify hardcoded addresses which assume that the code was loaded at its preferred base address in memory
- .rsrc = Resources. Lots of possible stuff from icons to other embedded binaries. The section has structures organizing it sort of like a filesystem.
typedef struct _IMAGE_SECTION_HEADER {
  0x00  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    0x08  DWORD  PhysicalAddress;
    0x08  DWORD  VirtualSize;
    } Misc;
  0x0c  DWORD  VirtualAddress;
  0x10  DWORD  SizeOfRawData;
  0x14  DWORD  PointerToRawData;
  0x18  DWORD  PointerToRelocations;
  0x1c  DWORD  PointerToLineNumbers;
  0x20  WORD   NumberOfRelocations;
  0x22  WORD   NumberOfLineNumbers;
  0x24  DWORD  Characteristics;
};
Section Header
(from winnt.h)

#define IMAGE_SIZEOF_SHORT_NAME 8

typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLinenumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLinenumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
Refresher: C Unions

union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
} Misc;

- Used to store multiple different interpretations of the same data in the same location.
- Accessed as if the union were a struct. So if you have
  IMAGE_SECTION_HEADER sectHdr;
  You don't access sectHdr.VirtualSize, you access sectHdr.Misc.VirtualSize
- We will only ever consider it as the VirtualSize field.
Section Header 2

- **Name[8]** is a byte array of ASCII characters. It is **NOT** guaranteed to be null-terminated. So if you're trying to parse a PE file yourself you need to be aware of that.
- **VirtualAddress** is the RVA of the section relative to OptionalHeader.ImageBase
- **PointerToRawData** is a relative offset from the beginning of the file which says where the actual section data is stored.
Section Header 3

- There is an interesting interplay between Misc.VirtualSize and SizeOfRawData. Sometimes one is larger, and other times the opposite.
- Why would VirtualSize be greater than SizeOfRawData? This indicates that the section is allocating more memory space than it has data written to disk.
- Think about the .bss portion of the .rdata section. It just needs a bunch of space for variables. The variables are uninitialized, which is why they don't have to be in the file. Therefore the loader can just give a chunk of memory to store variables in, by just allocating VirtualSize worth of data. Thus you get a smaller binary.
VirtualSize > SizeOfRawData

(on your own slide, draw the correspondence between the 0x200 in the first picture and the 0x300 in the second)
Section Header 4

- Why would SizeOfRawData be greater than VirtualSize?
- Remember that PE has the notion of file alignment. (OptionalHeader.FileAlignment) Therefore, if you had a FileAlignment of 0x200, but you only had 0x100 bytes of data, the linker would have had to write 0x100 bytes of data followed by 0x100 bytes of padding.
- By having the VirtualSize < SizeOfRawData, the loader can say “ok, well I see I really only need to allocate 0x100 bytes of memory and read 0x100 bytes of data from disk.”
VirtualSize < SizeOfRawData

(on your own slide, draw the correspondence between the 0x200 in the first picture and the 0x100 in the second)
Section Header 5
(from winnt.h)

- **Characteristics** tell you something about the section. Examples:

```c
#define IMAGE_SCN_CNT_CODE 0x00000020
// Section contains code.
#define IMAGE_SCN_CNT_INITIALIZED_DATA 0x00000040
// Section contains initialized data.
#define IMAGE_SCN_CNT_UNINITIALIZED_DATA 0x00000080
// Section contains uninitialized data.
#define IMAGE_SCN_MEM_DISCARDABLE 0x02000000
// Do not cache this section
#define IMAGE_SCN_MEM_NOT_CACHED 0x04000000
// Section can be discarded.
#define IMAGE_SCN_MEM_NOT_PAGED 0x08000000
// Section is not pageable.
#define IMAGE_SCN_MEM_SHARED 0x10000000
// Section is shareable.
#define IMAGE_SCN_MEM_EXECUTE 0x20000000
// Section is executable.
#define IMAGE_SCN_MEM_READ 0x40000000
// Section is readable.
#define IMAGE_SCN_MEM_WRITE 0x80000000
// Section is writeable.
```
Section Header

- PointerToRelocations,
  PointerToLinenumbers,
  NumberOfRelocations,
  NumberOfLinenumbers aren't used anymore
Renaming Sections

Merge Sections

Copy the link to merge section from to section To if section To does not exist; section From is renamed to To. (IMAGE from .txt)
Merge Sections

Entry Point
No Entry Point
No
Set Checksum
No
Base Address
Randomized Base Address
Enable Image Randomization (OPTIMAGE)
Predefined Base Address
Generate a relocation section (PEREL)
Data Execution Prevention (DEP)
Image is compatible with DEP (NONCOMPAT)
Turn Off Assembly Generation
No
Delay Loaded DLL
Do not support unload
Import Library
Target Machine
MachineType (MACHINEBIN64)
Profile
No
CLR Thread Attribute
No thread attribute set
CLR Image Type
Default image type
Key File
No
Key Container
No
Delay Sign
No
Error Reporting
Prompt immediately (ERRORREPORT/PROMPT)
CLR Unmanaged Code Check
No

Merge Sections
Causes the linker to merge section <from> into section <to> if section <to> does not exist; section <from> is renamed as <to> (MERGE <source>).

scratch.c
Linking...

LINE: warning LDX1654: section `.rdata' (00000040) merged into `.data' (C0000040) with different attributes
typedef struct _IMAGE_FILE_HEADER {
  WORD     Machine;
  WORD     NumberOfSections;
  DWORD    TimeDateStamp;
  DWORD    PointerToSymbolTable;
  DWORD    NumberOfSymbols;
  WORD     SizeOfOptionalHeader;
  WORD     Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
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
Static Linking vs Dynamic Linking

- With static linking, you literally just include a copy of every helper function you use inside the executable you're generating.
- Dynamic linking is when you resolve pointers to functions inside libraries at runtime.
- Needless to say, a statically linked executable is bloated compared to a dynamically linked one. But on the other hand, it's standalone, without outside dependencies. But on the other hand, patches or fixes to libraries are not applied to the statically linked binary until it's re-linked, so it can potentially have vulnerable code long after a library vulnerability is patched.
- Going to learn a bunch about how dynamic linking works, in service to learning a bit about how it is abused.
Calling Imported Functions

- As a programmer, this is transparent to you, but what sort of assembly does the compiler actually generate when you call an imported function like `printf()`?
- We can use the handy-dandy `HelloWorld.c` to find out quickly.

```
printf("Hello World!\n");
004113BE 8B F4  mov    esi,esp
004113C0 68 3C 57 41 00  push  41573Ch
004113C5 5F 15 BC 82 41 00  call  dword ptr ds:[004102BCh]
```

(Nota to self, show imports in PEView too)
IMAGE_DIRECTORY_ENTRY_IMPORT

struct _IMAGE_DATA_DIRECTORY {
  0x00  DWORD VirtualAddress;
  0x04  DWORD Size;
};
struct IMAGE_IMPORT_DESCRIPTOR {
    union {
        /* 0 for terminating null import descriptor */
        DWORD Characteristics;
        /* RVA to original unbound IAT */
        PIMAGE_THUNK_DATA OriginalFirstThunk;
    } u;
    DWORD TimeDateStamp; /* 0 if not bound,
        * -1 if bound, and real date/time stamp
        * in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT
        * (new BIND)
        * otherwise date/time stamp of DLL bound to
        * (Old BIND)
        */
    DWORD ForwarderChain; /* -1 if no forwarders */
    DWORD Name;
    /* RVA to IAT (if bound this IAT has actual addresses) */
    PIMAGE_THUNK_DATA FirstThunk;
};
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics;  // 0 for terminating null import descriptor
        DWORD OriginalFirstThunk;  // RVA to original unbound IAT [IMAGE_IMPORT_DATA]
    };
    DWORD TimeDateStamp;  // 0 if not bound,
    DWORD ForwarderChain;  // -1 if no forwarders
    DWORD Name;  // -1 if no forwarders
    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.
Import Descriptor 2

- **OriginalFirstThunk** ("is badly named" according to Matt Pietrek) is the RVA of the Import Name Table (INT). It's so named because the INT is an array of IMAGE_THUNK_DATA structs. So this field of the import descriptor is trying to say that it's pointing at the first entry in that array.
Import Descriptor 3

- **FirstThunk** like OriginalFirstThunk except that instead of being an RVA which points into the INT, it's pointing into the Import Address Table (IAT). The IAT is also an array of IMAGE_THUNK_DATA structures (they're heavily overloaded as we'll see).

- **Name** is just the RVA which will point at the specific name of the module which imports are taken from (e.g. hal.dll, ntdll.dll, etc)
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString;  // PBYTE
        DWORD Function;         // PWORD
        DWORD Ordinal;
        DWORD AddressOfData;    // IMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;

• We just learned that both the INT (pointed to by OriginalFirstThunk) and the IAT (pointed to by FirstThunk) point at arrays of IMAGE_THUNK_DATA32s.
• The INT and IAT IMAGE_THUNK_DATA32 structures are all interpreted as pointing at IMAGE_IMPORT_BY_NAME structures to begin with. That is they are u1.AddressOfData. This is actually the RVA of an IMAGE_IMPORT_BY_NAME structure.
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD   Hint;
    BYTE   Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;

- **Hint** specifies a possible “ordinal” of an imported function. Talked about later, when we talk about exports, but basically it’s just a way to look up the function by an index rather than a name.

- **Name** on the other hand is to look up the function by name. It’s not one byte long, it’s a null terminated ASCII string which follows the hint. But usually it’s just null in our examples.
On the impersistence of being: INT vs IAT

- The **INT** IMAGE_THUNK_DATA structures are always interpreted as pointing at IMAGE_IMPORT_BY_NAME structures, that is they are `u1.AddressOfData`, the RVA of an IMAGE_IMPORT_BY_NAME.
- The **IAT** IMAGE_THUNK_DATA structures *start out* are all interpreted as the `u1.AddressOfData`, but once the OS loader resolves each import, it overwrites the IMAGE_THUNK_DATA structure with the actual virtual address of the start of the function. Therefore it is *subsequently* interpreted as `u1.Function`. 
Review: Import data structures
ON DISK

IMAGE_IMPORT_DESCRIPTOR

- OriginalFirstThunk
- TimeDateStamp
- ForwarderChain
- Name
- FirstThunk

Zero-filled IMAGE_IMPORT_DESCRIPTOR entry terminates the array

Graphical style borrowed from the Matt Pietrek articles

Import Names Table
(IMAGE_THUNK_DATA array)

- 0x014B, NIQuerySysInfo
- 0x040B, RtlInitUnicodeString
- 0x01DA, IoIoCompleteRequest

Array of IMAGE_IMPORT_BY_NAME Structures stored wherever in the file

Import Address Table
(IMAGE_THUNK_DATA array)

ntdll.dll
Review:
Import data structures
IN MEMORY
AFTER IMPORTS RESOLVED

IMAGE_IMPORT_DESCRIPTOR

OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name
FirstThunk

0
0
0
0
0

...
Import data structures
ON DISK

IMAGE_IMPORT_DESCRIPTOR
- OriginalFirstThunk
- TimeDateStamp
- ForwarderChain
- Name
- FirstThunk

0
0
0
0
0
...

Import Names Table
(IMAGE_THUNK_DATA array)
- 0x014B, IoDeleteSymbolicLink
- 0x040B, RtlInitUnicodeString
- 0x01DA, IoCompleteRequest

Array of IMAGE_IMPORT_BY_NAME
Structures stored wherever in the file

Import Address Table
(IMAGE_THUNK_DATA array)

ntoskrnl.exe

Zero-filled
IMAGE_IMPORT_DESCRIPTOR
entry terminates the array

Graphical style borrowed from the Matt Pietrek articles
Import data structures

IN MEMORY
AFTER IMPORTS
RESOLVED

IMAGE_IMPORT_DESCRIPTOR

OriginalFirstThunk
TimeStamp
ForwarderChain
Name
FirstThunk

0
0
0
0
0
...

Import Names Table (IMAGE_THUNK_DATA array)

Array of IMAGE_IMPORT_BY_NAME Structures stored wherever in the file

0x014B, IoDeleteSymbolicLink
0x040B, RtlInitUnicodeString
0x01DA, IoCompleteRequest

Import Address Table (IMAGE_THUNK_DATA array)

IAT entries now point to the full virtual addresses where the functions are found in the other modules (just ntskrnl.exe in this case)

Graphical style borrowed from the Matt Pietrek articles
Look through null.sys
(note to self: start from the data directory)

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000010</td>
<td>0x00000630</td>
<td>Import Name Table RVA</td>
</tr>
<tr>
<td>0x00000014</td>
<td>0x00000000</td>
<td>Time Date Stamp</td>
</tr>
<tr>
<td>0x00000018</td>
<td>0x00000000</td>
<td>Forwarder Chain</td>
</tr>
<tr>
<td>0x0000001C</td>
<td>0x000006D4</td>
<td>Name RVA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ntoskrnl.exe</td>
</tr>
<tr>
<td>0x00000020</td>
<td>0x00000300</td>
<td>Import Address Table RVA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00000024</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000028</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x0000002C</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000030</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0x00000034</td>
<td>0x00000000</td>
<td></td>
</tr>
</tbody>
</table>
Import data structures
ON DISK

**IMAGE_IMPORT_DESCRIPTOR**
- OriginalFirstThunk
- TimeDateStamp
- ForwarderChain
- Name
- FirstThunk

**Import Names Table** (IMAGE_THUNK_DATA array)
- 0x0001, ExReleaseFastMutex
- 0x004E, KlRaiseIrql
- 0x004D, KlLowerIrql
- 0x029D, MmLockPageableDataSection
- 0x01EE, KeCancelTimer
- 0x02BC, MmUnlockPageableImageSection
- ... Array of IMAGE_IMPORT_BY_NAME Structures stored wherever in the file...
Import data structures
IN MEMORY
AFTER IMPORTS RESOLVED

IMAGE_IMPORT_DESCRIPTOR
  OriginalFirstThunk
  TimeDateStamp
  ForwarderChain
  Name
  FirstThunk
  OriginalFirstThunk
  TimeDateStamp
  ForwarderChain
  Name
  FirstThunk

Import Names Table
(IMAGE_THUNK_DATA array)
- 0x0001, ExReleaseFastMutex
- 0x004E, KlRaiseIrql
- 0x004D, KlLowerIrql
- 0x029D, MmLockPageableDataSection
- 0x01EE, KeCancelTimer
- 0x02BC, MmUnlockPageableImageSection

Array of IMAGE_IMPORT_BY_NAME Structures stored somewhere in the file

Import Address Table
(IMAGE_THUNK_DATA array)

... IAT entries now point to the full virtual addresses where the functions are found in the other modules

HAL.dll
ntoskrnl.exe

Graphical style borrowed from the Matt Pietrek articles
Look through beep.sys

nt then hal, no special significance, just sayin'
Look through beep.sys 2

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000789</td>
<td>00000A00</td>
<td>Hint_Name RVA</td>
<td>000D EReleaseFastMutex</td>
</tr>
<tr>
<td>0000078C</td>
<td>00000AC2</td>
<td>Hint_Name RVA</td>
<td>0046 HrRaiseException</td>
</tr>
<tr>
<td>00000788</td>
<td>00000A84</td>
<td>Hint_Name RVA</td>
<td>0044 StartWinlog</td>
</tr>
<tr>
<td>0000078C</td>
<td>00000A46</td>
<td>Hint_Name RVA</td>
<td>0018 StartMakeBeep</td>
</tr>
<tr>
<td>00000780</td>
<td>00000A45</td>
<td>Hint_Name RVA</td>
<td>0000 ExAcquireFastMutex</td>
</tr>
<tr>
<td>00000784</td>
<td>00000000</td>
<td>End of imports</td>
<td>HAL.dll</td>
</tr>
<tr>
<td>00000789</td>
<td>00000A6C</td>
<td>Hint_Name RVA</td>
<td>026D MemLockPageableDataSection</td>
</tr>
<tr>
<td>0000078C</td>
<td>000009C8</td>
<td>Hint_Name RVA</td>
<td>01EE KeCancelsTimer</td>
</tr>
<tr>
<td>0000078E</td>
<td>000009D6</td>
<td>Hint_Name RVA</td>
<td>026C MemUnlockPageableImageSection</td>
</tr>
<tr>
<td>000007A4</td>
<td>00000976</td>
<td>Hint_Name RVA</td>
<td>0184 IoStartNextPacket</td>
</tr>
<tr>
<td>000007A8</td>
<td>00000A9A</td>
<td>Hint_Name RVA</td>
<td>0254 KeSetTimer</td>
</tr>
<tr>
<td>000007AC</td>
<td>00000A18</td>
<td>Hint_Name RVA</td>
<td>0568 _local</td>
</tr>
<tr>
<td>000007A0</td>
<td>0000099C</td>
<td>Hint_Name RVA</td>
<td>0185 IoStartPacket</td>
</tr>
<tr>
<td>00000754</td>
<td>00000A34</td>
<td>Hint_Name RVA</td>
<td>020C KeInitializeEvent</td>
</tr>
<tr>
<td>0000079E</td>
<td>00000A48</td>
<td>Hint_Name RVA</td>
<td>0213 KeInitializeTimer</td>
</tr>
<tr>
<td>000007BC</td>
<td>00000A3C</td>
<td>Hint_Name RVA</td>
<td>026B KeInitializeDpc</td>
</tr>
<tr>
<td>000007E2</td>
<td>00000A8E</td>
<td>Hint_Name RVA</td>
<td>0181 IoCreateDevice</td>
</tr>
<tr>
<td>000007C4</td>
<td>00000A65</td>
<td>Hint_Name RVA</td>
<td>040B RtlGetUnicodeString</td>
</tr>
<tr>
<td>000007CB</td>
<td>000009B2</td>
<td>Hint_Name RVA</td>
<td>0118 IoAcquireCancelSpinLock</td>
</tr>
<tr>
<td>000007CC</td>
<td>0000096C</td>
<td>Hint_Name RVA</td>
<td>023A IoRemoveDriverQueue</td>
</tr>
<tr>
<td>000007C0</td>
<td>0000090C</td>
<td>Hint_Name RVA</td>
<td>0228 IoRemoveEntryDeviceQueue</td>
</tr>
<tr>
<td>000007C4</td>
<td>00000986</td>
<td>Hint_Name RVA</td>
<td>0199 IoReleaseCancelSpinLock</td>
</tr>
<tr>
<td>000007C8</td>
<td>00000A22</td>
<td>Hint_Name RVA</td>
<td>0149 IoDeleteDevice</td>
</tr>
<tr>
<td>000007DC</td>
<td>00000050</td>
<td>Hint_Name RVA</td>
<td>010A IoCompleteRequest</td>
</tr>
<tr>
<td>000007E0</td>
<td>00000000</td>
<td>End of imports</td>
<td>ntoskrnl.exe</td>
</tr>
</tbody>
</table>

hal then nt, no special significance, just sayin' it's backwards from the previous
Lab: appverif.exe

- appverif.exe was chosen because it has only "normal" imports; no "bound" or "delayed" imports as will be talked about later
- View Imports of C:\Windows \SysWOW64\appverif.exe with PEView
- View imports in memory by attaching with WinDbg
The WOW Effect

- On Win 7 x64…
  - C:\Windows\System32 = where the 64 bit binaries are stored
  - C:\Windows\SysWOW64 = where the 32 bit binaries are stored.
    - Try opening C:\Windows\SysWOW
    - 32 bit executables, like PEView currently is, will open SysWOW64 instead of System32
  - C:\Windows\Sysnative = how you can force 32 bit executables to find the 64 bit executables to find the 64 bit executables
- For more: http://www.cert.at/static/downloads/papers/cert.at-the_wow_effect.pdf
Did I mention?

It's a MADHOUSE!!

Did someone call me?

NO!

http://s1.picofile.com/file/6417096576/mad_house.jpg
http://mimg.ugo.com/201111/9/0/1/214109/cuts/brighteyes_528x297.jpg
Open WinDbg as Administrator
Open executable
1) Create a "memory" window
2) Drag the window over the grey to split the window top to bottom
Set to appverif.exe + RVA of IAT

Hmm... this still seems to match what's on disk? Turns out WinDog hit a breakpoint before the loader had a chance to resolve the IAT

Set to "Long Hex"

Base where this got loaded this time
Hit Go and you will see this

IAT looks updated!

Just to be sure, let's use the "list nearby symbols" (ln) command on an IAT entry
IMAGE_DIRECTORY_ENTRY_IAT

struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
};
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

Get your geek on

• Play through round 4 on your own, and then wait for the seed for the class deathmatch
• If you see something like the following: "user32.dll!Foofus" that means the function Foofus() in user32.dll
• You can skip to level 4 by starting the game with "python BinHunt.py 4"
IAT Hooking

- When the IAT is fully resolved, it is basically an array of function pointers. Somewhere, in some code path, there's something which is going to take an IAT address, and use whatever's in that memory location as the destination of the code it should call.
- What if the "whatever's in that memory location" gets changed after the OS loader is done? What if it points at attacker code?
IAT Hooking 2

• Well, that would mean the attacker's code would functionally be "man-in-the-middle"ing the call to the function. He can then change parameters before forwarding the call on to the original function, and filter results that come back from the function, or simply never call the original function, and send back whatever status he pleases.
  – Think rootkits. Say you're calling OpenFile. It looks at the file name and if you're asking for a file it wants to hide, it simply returns “no file found.”

• But how does the attacker change the IAT entries? This is a question of assumptions about where the attacker is.
IAT Hooking 3

- In a traditional memory-corrupting exploit, the attacker is, by definition, in the memory space of the attacked process, upon successfully gaining arbitrary code execution. The attacker can now change memory such as the IAT for this process only, because remember (from OS class or Intermediate x86) each process has a separate memory space.
- If the attacker wants to change the IAT on other processes, he must be in their memory spaces as well. Typically the attacker will format some of his code as a DLL and then perform “DLL Injection” in order to get his code in other process' memory space.
- The ability to do something like DLL injection is generally a prerequisite in order to leverage IAT hooking across many userspace processes. In the kernel, kernel modules are generally all sharing the same memory space with the kernel, and therefore one subverted kernel module can hook the IAT of any other modules that it wants.
DLL Injection

• See http://en.wikipedia.org/wiki/DLL_injection for more ways that this can be achieved on Windows/*nix
• We're going to use the AppInit_DLLs way of doing this, out of laziness
• (Note: AppInit_DLLs' behavior has changed in releases > XP, it now has to be enabled with Administrator level permissions.)
TODO:

- First thing tomorrow, show the fastjump r0x0r
Can also read more here: http://www.codeproject.com/KB/system/api_spying_hack.aspx
Before IAT hooking
After IAT hooking

http://knowyourmeme.com/memes/oh-crap-omg-rage-face