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/)

**You are free:**
- to Share — to copy, distribute and transmit the work
- to Remix — to adapt the work

**Under the following conditions:**
- Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

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

- It's not Virii (http://en.wikipedia.org/wiki/Plural_form_of_words_ending_in_-_us)
- Viruses are malware which self-replicate, but which need humans to initiate their execution. This is in contrast to worms which can self-replicate in the absence of humans.
- We're primarily interested in viruses which infect PE files, but they can infect other file types as well.
Conceptual:
Executable file infection

Headers specify entry point

File gets infected

I'm a Virus! :D
Or Maybe…

Headers specify entry point

<table>
<thead>
<tr>
<th>Headers</th>
<th>Headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>.text</td>
</tr>
<tr>
<td>.data</td>
<td>inline hook</td>
</tr>
<tr>
<td>.bss</td>
<td>.bss</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

File gets infected

I'm a Virus! :D
Or Maybe…

Headers specify entry point

Headers
- .text
- .data
- .bss
- ...

File gets infected

Headers
- .text
- .data
- .bss
- TLS Callback
- I'm a Virus! :D
Or Maybe…

Headers specify entry point

Headers
- .text
- padding
- .data
- .bss
- ...

File gets infected

Headers
- inline_book
- I'm a Virus! :D
- .data
- .bss
- ...

8
Padding

• Show how between 0x4a4 and 0xf14 is all just padding in the hello ELF binary, and therefore it can be infected.
• But we're going to go for a virus which tacks itself to the end of the file, and alters the headers to make the extra data get loaded into memory.
Interlude

- For the PE infector I was sitting around thinking "How am I going to get the address of CreateFile/WriteFile in a way where I don't just do opportunistic infection of files which already have those or GetProcAddress in their IAT" and then Corey Kallenberg reminded me that Skape talked about EAT searching in his excellent shellcode paper: http://www.hick.org/code/skape/papers/win32-shellcode.pdf, so thanks go out to Corey and Skape.
Lab: BabysFirstPhageForPE.c

- Is like the conceptual picture 1
- Can only infect binaries in C:\VirusTarget
- Has built in kill switch logic so that the parents can infect binaries to create children, and children can infect binaries to make grandchildren, but the grandchildren are sterile and can't have children of their own.
- Also, it doesn't fix the OptionalHeader.Checksum. So that must be fixed before an infected file will run to infect another file.
- Doesn't do anything malicious other than replicate, and whether that's malicious or not is in the eye of the beholder
Steps

• Copy HelloWorld and any other executables to infect to C:\VirusTarget
• Run BabysFirstPhageForPE.exe in VS debug mode talking through the stages
• Open infected file up with CFF Explorer, go to Rebuilder section, and select only the "Update Checksum" option, then close it and save the changes.
• Open infected binary in WinDbg, set a breakpoint on the entry point in the PE header (which is the virus entry point), and step through the code
• NOTE: If you're doing this on your own, make sure you don't have any breakpoints set in the virus code at the time that it copies itself into the buffer, otherwise the breakpoints will get copied too! (We learn about how breakpoints modify the code in Intermediate x86)
Throwback…TO THE MAX!

• A virus in the days of botnets!: Virut
  • (see what they did there? VirusX=>1? They're so clever. Also called Virus... because X is awesome!)
  • http://www.f-secure.com/v-descs/virus_w32_virut.shtml

• “Variants in the Virut family are polymorphic, memory-resident, appending file infectors that have Entry Point Obscuring (EPO) capabilities.”
  • http://www.symantec.com/connect/blogs/w32virutcf-collateral-damage

• “All of this sounds quite grim, but this threat can be removed from infected networks by following best practices.”

• Of course! Best practices! In retrospect it's all so obvious!
Fortune favors the bold

- Xpaj: Another misc virus
  - “It is not very common for a file infector to do more than simply introduce trivial modifications to the files it infects. Virus authors usually avoid complex modifications to the files because of the possibility of corruption. W32.Xpaj.B is one of exceptions.”


- Core principals the same, just another way of going about it, but the point is, now you can read and understand
Further Reading
(with the exception of the ~nemo link these are just misc googling that I haven’t read, so no guarantee of quality)

• PE infection
  – http://vx.netlux.org/29a/29a-7/Articles/29A-7.023

• ELF infection
  – http://vxheavens.com/lib/static/vdat/tuunix02.htm

• Mach-O infection
  – http://felinemenace.org/~nemo/slides/mach-o_infection.ppt
  – http://vxheavens.com/lib/vrg01.html

• Old sk00l
  – http://www.textfiles.com/virus/
  – http://vx.netlux.org/29a/main.html
  – http://vx.netlux.org/lib/
Packers

- Originally used to compress executables back when disk space was at a premium
- The executable would then decompress itself in memory and run as normal
- Nowadays they are mostly used for obfuscating binaries. Specifically since all the data for the original binary is compressed and/or encrypted, it prevents analysts from being able to infer things about the binary based on strings or function imports.
Conceptual Packing: File On Disk

Headers specify entry point

Original file on disk

.text
.data
.bss
...

Pack

Headers specify entry point

Packed file on disk

Headers

Compressed / Encrypted Blob

Unpacking code
Conceptual Unpacking: Load Time

File on disk

| Headers |
| Compressed / Encrypted Blob |
| Unpacking code |

Read headers and map file to memory accordingly.

Headers for packed file must still reserve >= virtual memory space used by original executable.

File in Virtual Memory

| Headers |
| Compressed / Encrypted Blob |
| Empty space |
| Unpacking code |
Conceptual
Unpacking: Run Time

Unpacking code runs, replacing its own memory with original layout. Final step is to jump to original entry point, at which point the original executable runs as normal.
The Ultimate Packer for eXecutables (UPX)

- http://upx.sourceforge.net/
- Easy to understand, very cross-platform compatible, legitimate packer which also has an automatic unpacking ability as well.
- Run it as "upx File -o PackedFile" to pack, and "upx -d PackedFile -o File" to decompress.
- Demo the header changes made by UPX to both a PE and ELF file
### UPX applied to PE hello.c

#### Before UPX

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000100</td>
<td>0100</td>
<td>Magic</td>
</tr>
<tr>
<td>0000010A</td>
<td>09</td>
<td>Major Linker Version</td>
</tr>
<tr>
<td>00000110</td>
<td>80</td>
<td>Minor Linker Version</td>
</tr>
<tr>
<td>00000110C</td>
<td>00000400</td>
<td>Size of Code</td>
</tr>
<tr>
<td>00000110E</td>
<td>00000050</td>
<td>Size of Initialized Data</td>
</tr>
<tr>
<td>000001114</td>
<td>00000000</td>
<td>Size of Uninitialized Data</td>
</tr>
<tr>
<td>00000119</td>
<td>000007C2</td>
<td>Address of Entry Point</td>
</tr>
<tr>
<td>0000011C</td>
<td>00001000</td>
<td>Base of Code</td>
</tr>
<tr>
<td>00000120</td>
<td>00002000</td>
<td>Base of Data</td>
</tr>
<tr>
<td>00000124</td>
<td>00400000</td>
<td>Image Base</td>
</tr>
<tr>
<td>00000128</td>
<td>00010000</td>
<td>Section Alignment</td>
</tr>
<tr>
<td>0000012C</td>
<td>00000200</td>
<td>File Alignment</td>
</tr>
<tr>
<td>00000130</td>
<td>0005</td>
<td>Major OS Version</td>
</tr>
<tr>
<td>00000132</td>
<td>0000</td>
<td>Minor OS Version</td>
</tr>
<tr>
<td>00000134</td>
<td>0000</td>
<td>Major Image Version</td>
</tr>
<tr>
<td>00000136</td>
<td>0000</td>
<td>Minor Image Version</td>
</tr>
<tr>
<td>00000138</td>
<td>0005</td>
<td>Major Subsystem Version</td>
</tr>
<tr>
<td>0000013A</td>
<td>0000</td>
<td>Minor Subsystem Version</td>
</tr>
<tr>
<td>0000013C</td>
<td>00000000</td>
<td>Win32 Version Value</td>
</tr>
<tr>
<td>00000140</td>
<td>00000000</td>
<td>Size of Image</td>
</tr>
<tr>
<td>00000144</td>
<td>00000000</td>
<td>Size of Headers</td>
</tr>
<tr>
<td>00000148</td>
<td>0030</td>
<td>checksum</td>
</tr>
<tr>
<td>0000014C</td>
<td>0003</td>
<td>Subsystem</td>
</tr>
<tr>
<td>0000014E</td>
<td>0140</td>
<td>Dll Characteristics</td>
</tr>
<tr>
<td></td>
<td>000B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>

#### After UPX

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000100</td>
<td>0100</td>
<td>Magic</td>
</tr>
<tr>
<td>0000010A</td>
<td>09</td>
<td>Major Linker Version</td>
</tr>
<tr>
<td>00000110</td>
<td>00</td>
<td>Minor Linker Version</td>
</tr>
<tr>
<td>00000110C</td>
<td>00001000</td>
<td>Size of Code</td>
</tr>
<tr>
<td>00000110E</td>
<td>00001000</td>
<td>Size of Initialized Data</td>
</tr>
<tr>
<td>000001114</td>
<td>00001000</td>
<td>Size of Uninitialized Data</td>
</tr>
<tr>
<td>00000119</td>
<td>000017F2</td>
<td>Address of Entry Point</td>
</tr>
<tr>
<td>0000011C</td>
<td>00001000</td>
<td>Base of Code</td>
</tr>
<tr>
<td>00000120</td>
<td>00001000</td>
<td>Base of Data</td>
</tr>
<tr>
<td>00000124</td>
<td>00400000</td>
<td>Image Base</td>
</tr>
<tr>
<td>00000128</td>
<td>00010000</td>
<td>Section Alignment</td>
</tr>
<tr>
<td>0000012C</td>
<td>00000200</td>
<td>File Alignment</td>
</tr>
<tr>
<td>00000130</td>
<td>0005</td>
<td>Major OS Version</td>
</tr>
<tr>
<td>00000132</td>
<td>0000</td>
<td>Minor OS Version</td>
</tr>
<tr>
<td>00000134</td>
<td>0000</td>
<td>Major Image Version</td>
</tr>
<tr>
<td>00000136</td>
<td>0000</td>
<td>Minor Image Version</td>
</tr>
<tr>
<td>00000138</td>
<td>0005</td>
<td>Major Subsystem Version</td>
</tr>
<tr>
<td>0000013A</td>
<td>0000</td>
<td>Minor Subsystem Version</td>
</tr>
<tr>
<td>0000013C</td>
<td>00000000</td>
<td>Win32 Version Value</td>
</tr>
<tr>
<td>00000140</td>
<td>00000000</td>
<td>Size of Image</td>
</tr>
<tr>
<td>00000144</td>
<td>00000000</td>
<td>Size of Headers</td>
</tr>
<tr>
<td>00000148</td>
<td>0030</td>
<td>checksum</td>
</tr>
<tr>
<td>0000014C</td>
<td>0003</td>
<td>Subsystem</td>
</tr>
<tr>
<td>0000014E</td>
<td>0140</td>
<td>Dll Characteristics</td>
</tr>
<tr>
<td></td>
<td>0040</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>
UPX applied to PE hello.c 2

<table>
<thead>
<tr>
<th>RVA</th>
<th>Data</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00</td>
<td>Name</td>
<td>Text</td>
</tr>
<tr>
<td>00000100</td>
<td>02</td>
<td>Virtual Size</td>
<td>7E 74 65 78</td>
</tr>
<tr>
<td>00000100</td>
<td>04</td>
<td>RVA</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000100</td>
<td>03</td>
<td>Size of Raw Data</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000100</td>
<td>02</td>
<td>Pointer to Relocations</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000100</td>
<td>03</td>
<td>Pointer to Line Numbers</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000100</td>
<td>04</td>
<td>Number of Relocations</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000100</td>
<td>01</td>
<td>Number of Line Numbers</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>00</td>
<td>Characteristics</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>02</td>
<td>IMAGE_SCN_CNT_CODE</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>04</td>
<td>IMAGE_SCN_MEM_DISCARDABLE</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>03</td>
<td>IMAGE_SCN_MEM_READ</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>00</td>
<td>IMAGE_SCN_CNT_INITIALIZED_DATA</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>02</td>
<td>IMAGE_SCN_MEM_DISCARDABLE</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>04</td>
<td>IMAGE_SCN_MEM_READ</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>01</td>
<td>IMAGE_SCN_CNT_UNINITIALIZED_DATA</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>03</td>
<td>IMAGE_SCN_MEM_DISCARDABLE</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00000000</td>
<td>04</td>
<td>IMAGE_SCN_MEM_READ</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

Total virtual size = 0x518A
Total file size = 0x1C00
UPX applied to PE hello.c 3
UPX applied to ELF hello.c

readelf -l hello-static
ELF file type is EXEC (Executable file)
Entry point: 0x80481e0
There are 6 program headers, starting at offset 52
Program Headers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
<th>FileOff</th>
<th>Section Name</th>
<th>Flags</th>
<th>At End</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>0x08080000</td>
<td>0x08080000</td>
<td>0x08080000</td>
<td>0x851df</td>
<td>Hello</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x08090000</td>
<td>0x08090000</td>
<td>0x08090000</td>
<td>0x85200</td>
<td>Hello</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>NOTE</td>
<td>0x08090000</td>
<td>0x08090000</td>
<td>0x08090000</td>
<td>0x85200</td>
<td>Hello</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>TLS</td>
<td>0x08090000</td>
<td>0x08090000</td>
<td>0x08090000</td>
<td>0x85200</td>
<td>Hello</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>GNU_STACK</td>
<td>0x08090000</td>
<td>0x08090000</td>
<td>0x08090000</td>
<td>0x85200</td>
<td>Hello</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Section to Segment mapping:
Segment Sections:
00 .note.ABI-tag .note.gnu.build-id .rel.plt .init .plt .text __libc_freeres_fn .fini .rodata
   __libc_atexit __libc_subroutines .eh_frame .got.plt .got.plt .data .bss __libc_freeres_ptr
01 .data .ctors .dtors .jcr .data.rel.ro .got .got.plt .data .bss __libc_freeres_ptr
02 .note.ABI-tag .note.gnu.build-id
03 .data .bss
04 .data .ctors .dctors .jcr .data.rel.ro .got

readelf -l hello-static-packed
ELF file type is EXEC (Executable file)
Entry point: 0x8049208
There are 2 program headers, starting at offset 52
Program Headers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
<th>FileOff</th>
<th>Section Name</th>
<th>Flags</th>
<th>At End</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>0x08000000</td>
<td>0x08000000</td>
<td>0x08000000</td>
<td>0x83f08</td>
<td>Hello</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x08000000</td>
<td>0x08000000</td>
<td>0x08000000</td>
<td>0x83f08</td>
<td>Hello</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

| Original executable 2nd LOAD segment VirtAddr + MemSize |
Pseudo Execution Flow Description
my wife pointed out if I don't obfuscate the addresses you could skip straight to step 4 :)

- 0. Starts out running at its entry point
- 1. mmaps a page <SOMEWHERE>
- 2. Copies some of its code to <SOMEWHERE>
- 3. You will be stuck in loops the range between <SOMEWHERE> and <SOMEWHERE> for a while. Find the escape hatch!
- 4. Transfers control flow to <SOMEWHERE>
- 5. mmap allocates memory space sufficient to cover original memory space
- 6. Decompresses data into original location
- 7. munmaps the original compressed memory area (but leaves the 1 page at <SOMEWHERE>) immediately before going to the original entry point (OEP)
Other packers

- Other packers introduce extensive anti-debug tricks into the unpacking code.
- Themida/VMProtect will take x86 code and convert it into a series of equivalent instructions in a custom bytecode language which is subsequently interpreted into machine code (which may not necessarily be the exact same code that existed in the application pre-modification.)
  - Therefore the "VM" in VMProtect is a "software virtual machine" in the "Java virtual machine" sense. Java interprets bytecode in order to eventually execute equivalent machine code, these VMs interpret bytecode to execute equivalent machine code.
Packers

More than 100 files detected with packer listed

Panda speculates long tail for packers, seems true
Further Reading

- PE
  - Lots and lots of good stuff - http://pferrie.tripod.com/
  - http://securitylabs.websense.com/content/Assets/HistoryofPackingTechnology.pdf
  - http://www.lmgfly.com/?q=PE+packer+filetype%3Apdf

- ELF
  - Burneye packer - http://packetstormsecurity.org/groups/teso/burneye-1.0.1-src.tar.bz2
  - Userland Exec - Just another name for what unpacking does as far as I'm concerned, but still worth a read - http://www.securityfocus.com/archive/1/348638
  - http://www.lmgfly.com/?q=ELF+packer+filetype%3Apdf
Other self-decompressors

- The concept of self-decompression at runtime is also used in other areas. For instance, Cisco's IOS is stored on the router in a compressed form which decompresses in memory. (http://www.coresecurity.com/files/attachments/Killing_the_myth_of_Cisco_IOS_rootkits.pdf.rar)
- BIOS can play the same game. (http://www.coresecurity.com/files/attachments/Persistent_BIOS_Infection_CanSecWest09.pdf)
- Both of these are obviously doing this because storage space is expected to be smaller than memory.
Reflective DLL Injection

- http://www.harmonysecurity.com/ReflectiveDllInjection.html
- In contrast with normal DLL injection, where you might rely on the OS to load the DLL for you, a reflective DLL is self-sufficient. The code within it will handle the necessary initialization (that the OS loader would normally do) in order to ensure it can execute normally. The benefit is that this DLL will not be registered anywhere by the OS as being a DLL, it will just be some blob of code in memory somewhere.
- Note, that this property can also potentially be used to find it. That is, if you are doing memory analysis and you see something that indicates control flow eventually transfers to "some blob of code in memory somewhere", that is sort of suspicious, and therefore bears investigation. (Of course then the question becomes, how did you find this control flow divergence ;))
- That said, there are "legitimate" reasons there may be code blobs running around in memory; see the Adobe Flash ActionScript Just-In-Time (JIT) code generation engine...and the subsequent utilization for exploits ("JIT Spray" - http://www.semantiscope.com/research/BHDC2010/BHDC-2010-Paper.pdf)
Hot Patching Running Binaries

*topic added under threat of physical violence by my wife!*

*"No! You have to add it! Patching is part of a binary's life! Don't make me give you to the back of my hand!"

- Good talk here covering all the various ways you can hotpatch, including microsoft's way: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Sotirov.pdf
- `/hotpatch` option can be added manually to the additional compiler options. Then it will generate a "mov edi, edi" instruction before the normal "push ebp; mov ebp, esp" function prolog which will make your functions easier to hotpatch in the future if necessary.
- The "mov edi, edi" obviously does nothing, but it's 2 bytes, which when combined with the 3 bytes for the "push ebp; mov ebp, esp" == 5 bytes. If some code needs to hotpatch a function, the first 5 bytes can then be overwritten with a jmp instruction to jump to the new implementation of the function, and that function knows it only needs to execute the "push ebp; mov ebp, esp" instructions before it jumps back to the original code.
5-Byte JMP Overwrite

Before overwriting the function prologue, we need to save the overwritten instructions. The hook routine should execute the saved instructions before returning to the patched function.

Patched function:

- jmp hook
- mov esi, [ebp+arg_0]
- ...
- ret

Hook routine:

- // do some work
- // saved instructions
- push ebp
- mov ebp, esp

// return
- jmp patched_function+5


TODO: replace this slide
Games Nerds Play: TinyPE/TeenyELF/TinyMach-O

- http://www.phreedom.org/solar/code/tinype/ - 97 bytes
Explore your world!
Read the Mach-O Spec! :D

• How is it similar to the binary formats we've covered in this class?
• How is it different?
• Where's the 0xbeef?
Bad Moon Rising

- Unified Extensible Firmware Interface (UEFI) is the replacement for BIOS. It's going to bring BIOS into the modern day by making people not have to program as much 16 bit x86.
- It's also standardizing on the PE format for the binaries. That means, now your firmware can have modules which look a lot like Windows executables and are a lot easier to programs.
- Does anyone see why this might start to create more problems for security at the firmware level?
Binject

- http://www.rnicrosoft.net/tools/binject_v0.1.zip
- Can break apart a binary and put it back together in a convenient trojaned form.
  - The only thing which is going to catch that sort of thing are filesystem integrity checkers.
- DLL Entry Point Redirection, Import Table DLL Additions, TLS Callback and more, made easy
Teardown - What did we learn about?

- Not this year! Go watch last year's video at OpenSecurityTraining.info!
- **Compilation**
  - Lexical Analysis — turning characters into lexemes, which can be grouped into tokens
  - Syntax Analysis — Context Free Grammars in Backus Naur form (BNF), and Parse Trees (aka Concrete Syntax Trees)
  - Abstract Syntax Trees (ASTs), Abstract Assembly Trees (AATs), tiling on AATs to generate assembly
- **Linking**
  - Splicing output object files together into a final binary which may be standalone or may depend on the loader to find external libraries for it
Teardown - What did we learn about?

- Portable Executable (PE) binary format used on Windows systems
  - 3 flavors of imports + hooking, exports + hooking & export forwarding, relocations w/ relevance to memory integrity checks, thread local storage (TLS) & TLS callbacks, resources & file embedding, digital signature files

- Executable and Linking Format (ELF) binary format used on *nix systems
  - imports & dynamic linking, exports, relocations
Teardown - What did we learn about?

- F*cking viruses, how do they work?!
- Packing/Unpacking and the effects on binary format and memory contents
- Reflective DLL injection
- Smallest possible binaries
- And the rest!

- Fly little birdie fly! It's time for you to explore on your own. Go back through and re-read explanations, read cited materials, etc.
Extra Slides
IMAGE_DIRECTORY_ENTRY_EXCEPTION

- “array of IMAGE_RUNTIME_FUNCTION_ENTRY structures, which are CPU-specific. Pointed to by the IMAGE_DIRECTORY_ENTRY_EXCEPTION slot in the DataDirectory. Used for architectures with table-based exception handling, such as the IA-64. The only architecture that doesn't use table-based exception handling is the x86.”
IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR

- C:\WINDOWS\System32\TsWpfWrp.exe has a non-zero instance of this
- "This value has been renamed to IMAGE_DIRECTORY_ENTRY_COMHEADER in more recent updates to the system header files. It points to the top-level information for .NET information in the executable, including metadata. This information is in the form of an IMAGE_COR20_HEADER structure."
IMAGE_DIRECTORY_ENTRY_ARCHIVE
aka IMAGE_DIRECTORY_ENTRY_COPYRIGHT

- From Matt Pietrek's PE part 2 figures:
- "Points to architecture-specific data, which is an array of
  IMAGE_ARCHITECTURE_HEADER structures. Not used for x86 or IA-64,
  but appears to have been used for DEC/Compaq Alpha."
IMAGE_DIRECTORY_ENTRY_GLOBALPTR

- "The VirtualAddress field is the RVA to be used as the global pointer (gp) on certain architectures. Not used on x86, but is used on IA-64. The Size field isn't used. See the November 2000 Under The Hood column for more information on the IA-64 gp."