LA-UR-18-25474
Approved for public release; distribution is unlimited.

Title: Covert Malware Launching and Data Encoding: Malware Analysis Day 5

Author(s): Pearce, Lauren

Intended for: Presentation for 2 week course on malware analysis

Issued: 2018-06-21

Covert Malware Launching and Data Encoding

Malware Analysis Day 5

laurenp@lanl.gov

Launchers

- We’ve mentioned launchers previously – what is their role?
- Where do launchers often to hide the malicious code?
- Why are we bringing them back up today?

DLL Injection

- A method by which malware forces a remote process to load a malicious DLL.
- All the actions taken by the malicious DLL appear to come from the injected process.
- The malicious DLL will have the permissions of the process it was injected into.

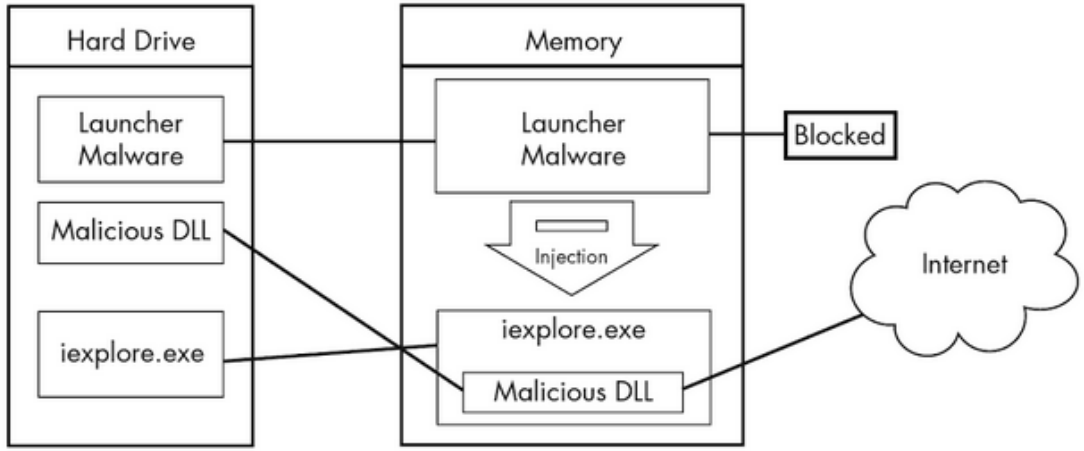


Figure 12-1: DLL injection—the launcher malware cannot access the Internet until it injects into iexplore.exe.

Practical Malware Analysis Chapter 12 Figure 12-1

DLL Injection: The Launcher Malware Must…

1. Obtain a handle to the chosen victim process
   a. `CreateToolhelp32Snapshot, Process32First, Process32Next`
2. Open a handle to the victim process
   a. `OpenProcess`
3. Use the victim process's handle to allocate space in the victim process's memory for shenanigans
   a. `VirtualAllocEx`
4. Write the evil library's name into the allocated memory space
   a. `WriteProcessMemory`
5. Create a remote thread in the victim's process space
   a. `CreateRemoteThread` – requires 3 arguments: the process handle, the address of the code where the thread should start running, and an argument for the code at the specified address.
      i. What are those arguments going to be?

DLL Injection: C Code

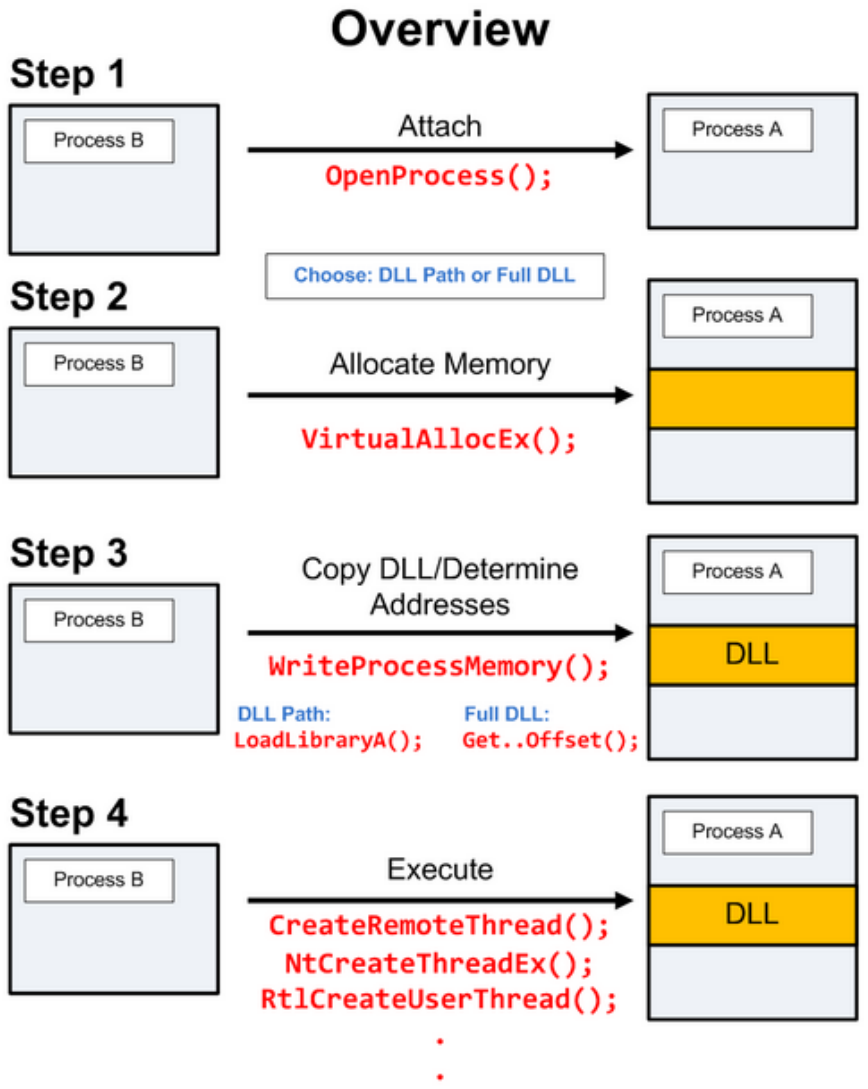
```
hVictimProcess = OpenProcess(PROCESS_ALL_ACCESS, 0, victimProcessID ❶);

pNameInVictimProcess = VirtualAllocEx(hVictimProcess,...,sizeof(maliciousLibraryName),...,...);
WriteProcessMemory(hVictimProcess,...,maliciousLibraryName, sizeof(maliciousLibraryName),...);
GetModuleHandle("Kernel32.dll");
GetProcAddress(...,"LoadLibraryA");
❷ CreateRemoteThread(hVictimProcess,...,...,LoadLibraryAddress,pNameInVictimProcess,...,...);
```

Listing 12-1: C Pseudocode for DLL injection

Practical Malware Analysis Listing 12-1

DLL Injection - Review

- Clear as mud?
- What does malware achieve by doing this?



http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html ← An excellent article and how-to

DLL Direct Injection

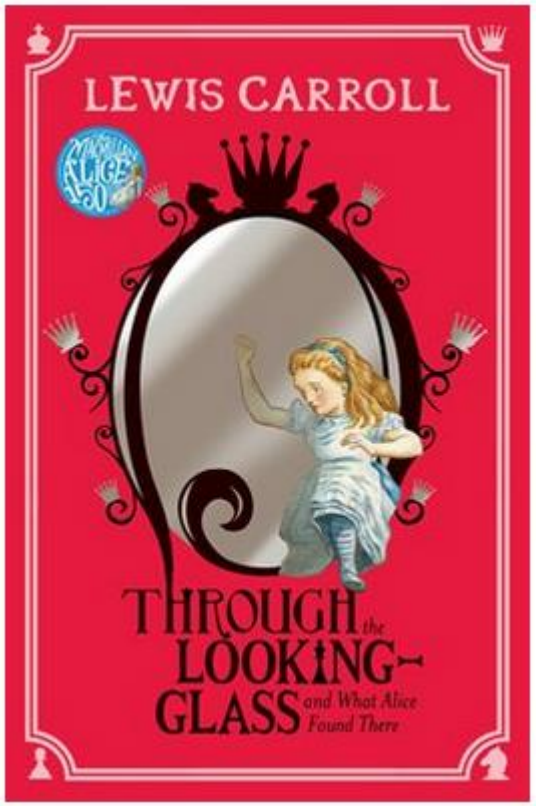
- Looks very similar to DLL injection in the assembly, but writes the actual DLL directly into the memory space of the victim process.
  - How is this different from what we just talked about?
- It is possible to inject compiled code, but most often this method is used to inject shell code
  - It’s very difficult to inject code into a running process without causing the process to crash.
- If you see this and it’s injecting more than the most simple shellcode, you have an extremely skilled author.
- From my perspective, analyzing this involves memory forensics.

DLL Direct Injection

- Call Sequence:
  - `VirtualAllocEx`
  - `WriteProcessMemory`
  - `VirtualAllocEx`
  - `WriteProcessMemory`
  - `CreateRemoteThread`
- Why two calls to VirtualAllocEx and WriteProcessMemory?
  - One set to write the data that the remote thread will take as an argument.
  - One set to write the actual code for the remote thread
- What do you think this would pass as arguments to WriteProcessMemory
  - hProcess – Handle to the process to be injected
  - lpBaseAddress – Pointer to the start of the code written in the remote process's memory
  - lpParameters – Pointer to the data already written in the remote process's memory

Process Replacement

- Process replacement is used to overwrite the memory space of a running process with a malicious executable.
- Less risk of crashing the process than direct injection.



https://www.panmacmillan.com/aliceinwonderlandbooks

Process Replacement

1. Create a process, but launch it in a suspended state
   a. `CreateProcess` with flag `CREATE_SUSPENDED`
2. Free the memory that the target process controls
   a. `ZwUnmapViewOfSection`
3. Allocate the now freed memory for use by your malware
   a. `VirtualAllocEx`
4. Write evil code to your freshly allocated memory space
   a. `WriteProcessMemory`
5. Set the entry point of the process to point at the malicious code
   a. `SetThreadcontext`
6. Fire off the suspended process
   a. `ResumeThread`

Process Replacement

- Why would the malware author use this technique?
- How can you detect this in dynamic analysis?
- What API calls alert you to the possibility of process replacement?

Demo

Lab 12-1

Hook Injection

- The Windows OS uses “messages” for communication between the OS and applications. Hooks are used to intercept messages that are bound for applications from the OS.
- Windows uses hooks for things like macro recording and hot keys. Malware uses hooks for:
  - Run malicious code whenever a XXXX message is sent
  - Ensure a malicious DLL is loaded or loads into a victim process’s memory space

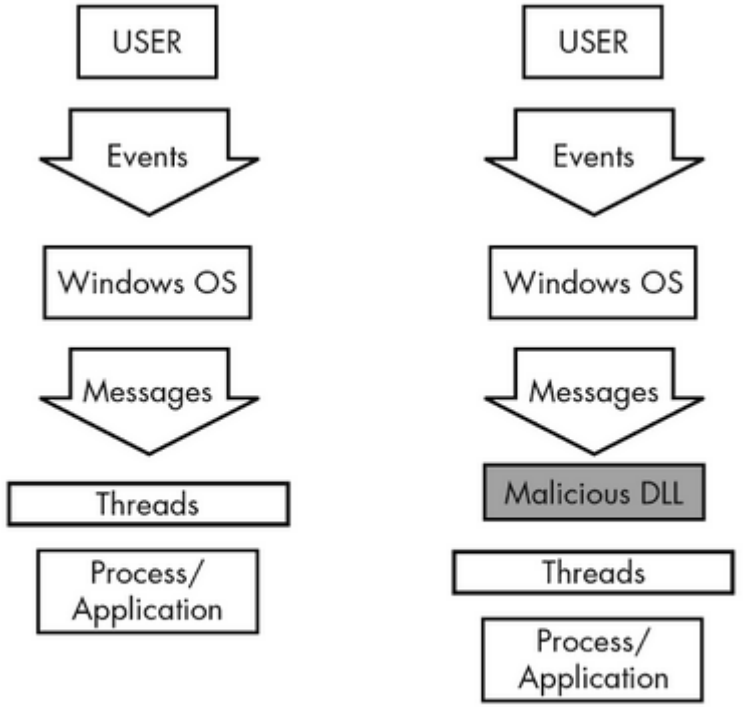


Figure 12-3: Event and message flow in Windows with and without hook injection

Hook Injection: Vocabulary

- Local Hooks – Destination is an internal process
- Remote Hook – Destination is a remote process
  - High-Level – Hook procedure must be a DLL export. A high level hook procedure will be mapped into the process space of a hooked thread.
  - Low Level – Hook procedure must be contained in the process that installed the hook
- Hook Chain – A list of pointers to hook procedures.
  - When a message occurs that is associated with a particular type of hook, the system passes the message to each procedure in the chain, one after the other.
  - A hook in the chain may prevent the message from making it to the next link.

Hook Injection: Keyloggers

- Windows hook types WH_KEYBOARD and WH_KEYBOARD_LL monitor keystrokes.

WH_KEYBOARD_LL

The **WH_KEYBOARD_LL** hook enables you to monitor keyboard input events about to be posted in a thread input queue.

For more information, see the ***LowLevelKeyboardProc*** callback function.

WH_KEYBOARD

The **WH_KEYBOARD** hook enables an application to monitor message traffic for WM_KEYDOWN and WM_KEYUP messages about to be returned by the GetMessage or PeekMessage function. You can use the **WH_KEYBOARD** hook to monitor keyboard input posted to a message queue.

For more information, see the ***KeyboardProc*** callback function.

Hook Injection: SetWindowsHookEx

- idHook – the type of hook procedure to install
- lpfn – pointer to the hook procedure
- hMod –
  - High-level Hooks – handle to the DLL containing the procedure specified in lpfn
  - Low Level Hooks – handle to the local module where the procedure specified in lpfn is defined
- dwThreadId – the thread identifier for the thread that the hook procedure will be associated with
  - If 0 – the hook procedure is associated with all existing threads running on the same desktop as the calling thread

Syntax

C++

```
HHOOK WINAPI SetWindowsHookEx(
  _In_ int       idHook,
  _In_ HOOKPROC  lpfn,
  _In_ HINSTANCE hMod,
  _In_ DWORD     dwThreadId
);
```

Hook Injection: Thread Targeting

- Target a thread or load into all?
  - Target a specific thread, the malware will include instructions to find the thread identifier it's looking for. Sufficient if your goal is to load a DLL into a remote process.
    - Search for the target process, if found get the thread you want, if not launch it yourself
  - Load into all threads – degrades the performance of the system and more likely to be detected. Necessary if you need to see every occurrence of a message, such as in keylogging.

Hook Injection: An Example

```
00401100        push    esi
00401101        push    edi
00401102        push    offset LibFileName ; "hook.dll"
00401107        call    LoadLibraryA
0040110D        mov     esi, eax
0040110F        push    offset ProcName ; "MalwareProc"
00401114        push    esi             ; hModule
00401115        call    GetProcAddress
0040111B        mov     edi, eax
0040111D        call    GetNotepadThreadId
00401122        push    eax             ; dwThreadId
00401123        push    esi             ; hmod
00401124        push    edi             ; lpfn
00401125        push    WH_CBT   ; idHook
00401127        call    SetWindowsHookExA
```

Listing 12-4: Hook injection, assembly code

Practical Malware Analysis Listing 12-4

Detours

- Microsoft library that (theoretically) makes it easy to extend existing application and OS functionality. Malware authors like this.
  - Modify import tables
  - Attach DLLs to existing programs
  - Add function hooks to running processes

roadtrafficsigns.com

Detours: How

- Malware targets an existing on-disk binary
- Malware modifies the PE structure of the targeted binary to add a section named .detour. This section contains the original PE header, but a new IAT.
- Malware uses the `setdll` tool provided by the Detours library to modify the original PE header to point to the modified IAT

Detours: Example

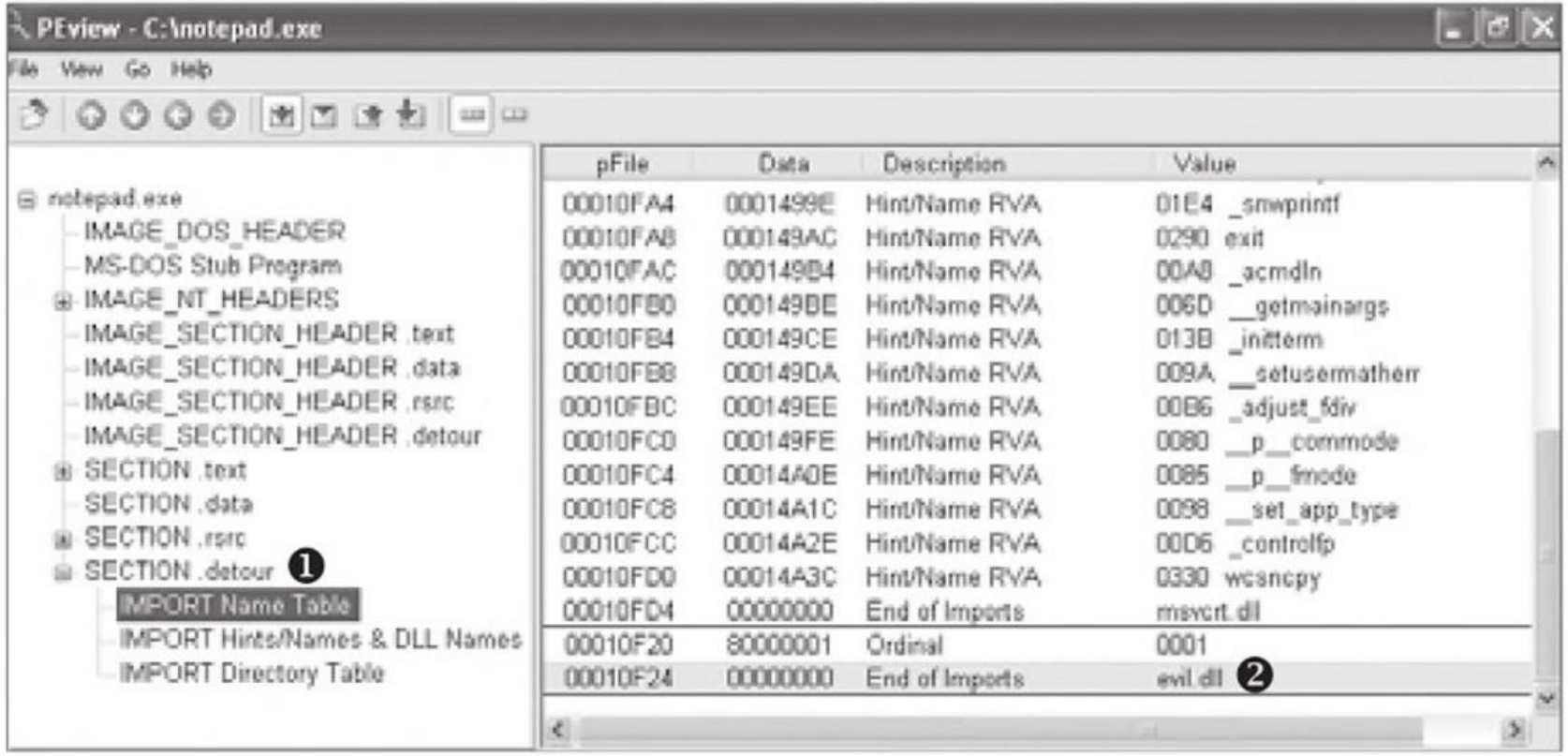


Figure 12-4: A PEview of Detours and the evil.dll

Practical Malware Analysis Figure 12-4

APC Injection

- What?
  - Asynchronous Procedure Call – A Windows feature that allows a thread to execute some other code before executing its normal execution path.
  - Every thread has a queue of APCs attached to it that are processed when the thread is in an “alertable State”
    - WaitForSingleObjectEx, WaitForMultipleObjectsEx, SleepEx
- Why?
  - Thread creation has overhead, it’s more efficient to invoke a function on a thread that already exists
- For Malware
  - Get an existing thread to execute their code

APC Injection: Vocab

- Kernel-Mode APC – An APC generated for the system or a driver
- User-Mode APC – An APC generated for an application

Identifying APC Injection from User Space

- Identifying thread targeting code:
  - `CreateToolhelp32Snapshot, Process32First, Process32Next`
  - `ZwQuerySystemInformation` with the `SYSTEM_PROCESS_INFORMATION` argument
    - These are used to identify a target process
  - `Thread32First, Thread32Next`
- Once a target thread has been identified, malware can queue a function to be invoked in a remote thread using the call `QueueUserAPC`

QueueUserAPC

- pfnAPC – a pointer to the malware-supplied APC function to be called when the specified thread enters an alertable state.
- hThread – A handle to the target thread.
- dwData – A single value that is passed to the function pointed to by pfnAPC

C++

```
DWORD WINAPI QueueUserAPC(
  _In_ PAPCFUNC  pfnAPC,
  _In_ HANDLE    hThread,
  _In_ ULONG_PTR dwData
);
```

APC Injection: Example

```
00401DA9        push    [esp+4+dwThreadId]      ; dwThreadId
00401DAD        push    0                       ; bInheritHandle
00401DAF        push    10h                     ; dwDesiredAccess
00401DB1        call    ds:OpenThread ❶
00401DB9        test    esi, esi
00401DBB        jz      short loc_401DCE
00401DBD        push    [esp+4+dwData]          ; dwData = dbnet.dll
00401DC1        push    esi                     ; hThread
00401DC2        push    ds:LoadLibraryA ❷       ; pfnAPC
00401DC8        call    ds:QueueUserAPC
```

Listing 12-5: APC injection from a user-mode application

APC Injection from Kernel Space

- Why, if your malware already has access to kernel space, would you be concerned with APC injection?
  - Malicious drivers and rootkits still need to execute code in user space, but living in kernel space there isn't an easy way for them to do it.
  - APC Injection gives kernel space malware a way to run code in user space.
- How?
  - Create and dispatch a new thread with the APC
  - That new thread executes the APC in a user-mode process
  - Often involves shellcode

APC Injection from Kernel Space: Example

```
000119BD        push    ebx
000119BE        push    1 ❶
000119C0        push    [ebp+arg_4] ❷
000119C3        push    ebx
000119C4        push    offset sub_11964
000119C9        push    2
000119CB        push    [ebp+arg_0] ❸
000119CE        push    esi
000119CF        call    ds:KeInitializeApc
000119D5        cmp     edi, ebx
000119D7        jz      short loc_119EA
000119D9        push    ebx
000119DA        push    [ebp+arg_C]
000119DD        push    [ebp+arg_8]
000119E0        push    esi
000119E1        call    edi         ;KeInsertQueueApc
```

Listing 12-6: User-mode APC injection from kernel space

Demo

Lab 12-3

Data Encoding

“content modification for the purpose of hiding intent”

Hiding Intent

- How does data encoding allow malware to hide its intent?
  - Hide config information – log file paths, C&C domains/Ips
  - Hide the nature of content leaving the network
  - Hide API calls or sequences of calls which would raise attention
  - Hide strings that would reveal the malicious nature of the program

Custom Encoding

- Why would the malware author use custom encoding?
  - All of the benefits of simple encoding mechanisms – lightweight, and nonobvious.
  - Actually MORE difficult for the analyst to decode than standard crypto
    - With standard crypto, once you have the key and the know the algorithm they're using, it's trivial to put together a decoder

Decoding: Turn the Malware Against Itself

- Use a debugger to manipulate the malware into decoding all of its encoded strings.
- Write a script to feed your encoded blobs to the decoder and spit out the output.
  - This requires a paid Ida license, but is magical
- This is the ONLY method I have ever used to decode strings in malware.

Decoding: Write a Decoder

- Use your favorite programming language and standard libraries to write a decoder.
- This is what you turn to when method 1 doesn’t work. Used to be a more standard approach, modern tools have changed that.
- Often the only feasible method to decode encoded network communications – why?

Simple Ciphers

- Why would a malware author use an XOR when he could use DES?
  - Small size and simplicity makes them viable for use in exploit shellcode
  - Much less obvious in the code
  - Lightweight – less overhead

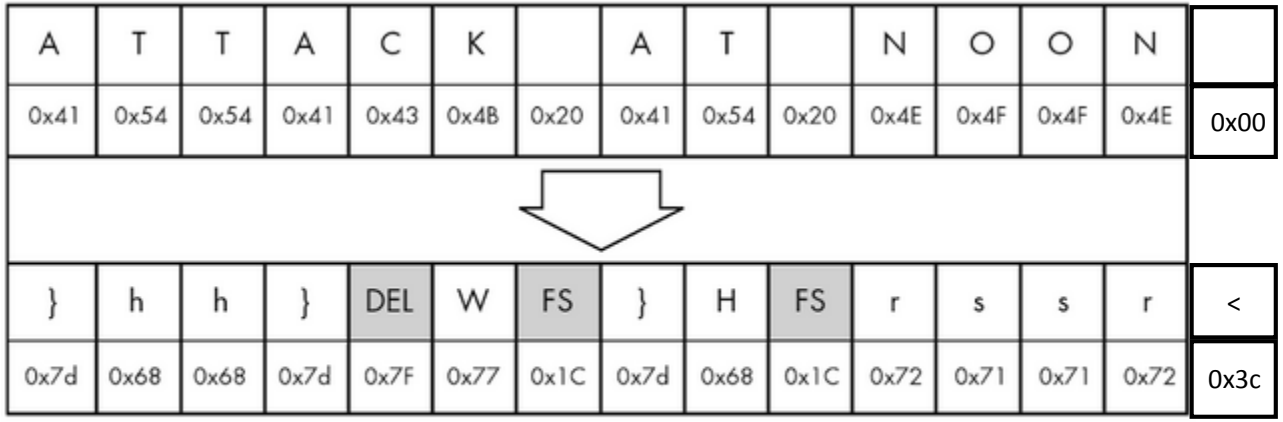
- Simple Ciphers are to obscure – often this is sufficient.

Some Simple Ciphers

- Caesar Cipher
  - Shift characters of the alphabet X characters to the right
- Double Transposition Cipher
  - The plaintext into a matrix, shift rows and columns in a way determined by the key, read the ciphertext from the array
- One Time Pad
  - A standard non-secret mapping exists between letters and bits
  - Pad = string of randomly selected bits same length as bits representing the plaintext
  - Plaintext is encoded by xoring the plaintext with the pad. Ciphertext is decoded by xoring the ciphertext with the pad.
    - Illustrates an important principle of XOR – one than many more complex ciphers rely on

Single Byte XOR

- Simple and reversible
  - Same function to encode and decode



A	T	T	A	C	K		A	T		N	O	O	N	
0x41	0x54	0x54	0x41	0x43	0x4B	0x20	0x41	0x54	0x20	0x4E	0x4F	0x4F	0x4E	0x00
}	h	h	}	DEL	W	FS	}	H	FS	r	s	s	r	<
0x7d	0x68	0x68	0x7d	0x7F	0x77	0x1C	0x7d	0x68	0x1C	0x72	0x71	0x71	0x72	0x3c

Figure 13-1: The string ATTACK AT NOON encoded with an XOR of 0x3C (original string at the top; encoded strings at the bottom)

Single Byte XOR Weakness

```
5F 48 42 12 10 12 12 12 16 12 1D 12 ED ED 12 12    _HB.............
AA 12 12 12 12 12 12 12 52 12 08 12 12 12 12 12    ........R.......
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12    ................
12 12 12 12 12 12 12 12 12 12 12 12 13 12 12 12    ................
A8 02 12 1C 0D A6 1B DF 33 AA 13 5E DF 33 82 82    ........3..^.3..
46 7A 7B 61 32 62 60 7D 75 60 73 7F 32 7F 67 61    Fz{a2b`}u`s.2.ga
```

Listing 13-1: First bytes of XOR-encoded file a.gif

NULL-Preserving Single Byte XOR

- How would you write a single byte xor encoding/decoding function that didn’t obviously reveal the key?

```
If plaintextChar != key AND plaintextChar != NULL
    cyphertextByte = plaintextChar XOR key
```

Finding XOR Encoding Functions

- You can search code in Ida – it may be useful to search for xor instructions:
  1. Switch so that your context is in Ida View
  2. Search → Text
  3. Enter xor, check the “find all occurrences box”, click OK
- Remember – xor is used for all sorts of compiler shortcuts. What you’re looking for is xor in a loop, maybe with a cmp before it.

Single Byte XOR Example

- What argument holds the limit for our counter?
- Where is our counter incremented?
- What is our xor key?
- What instruction writes the ciphertext into a new string?

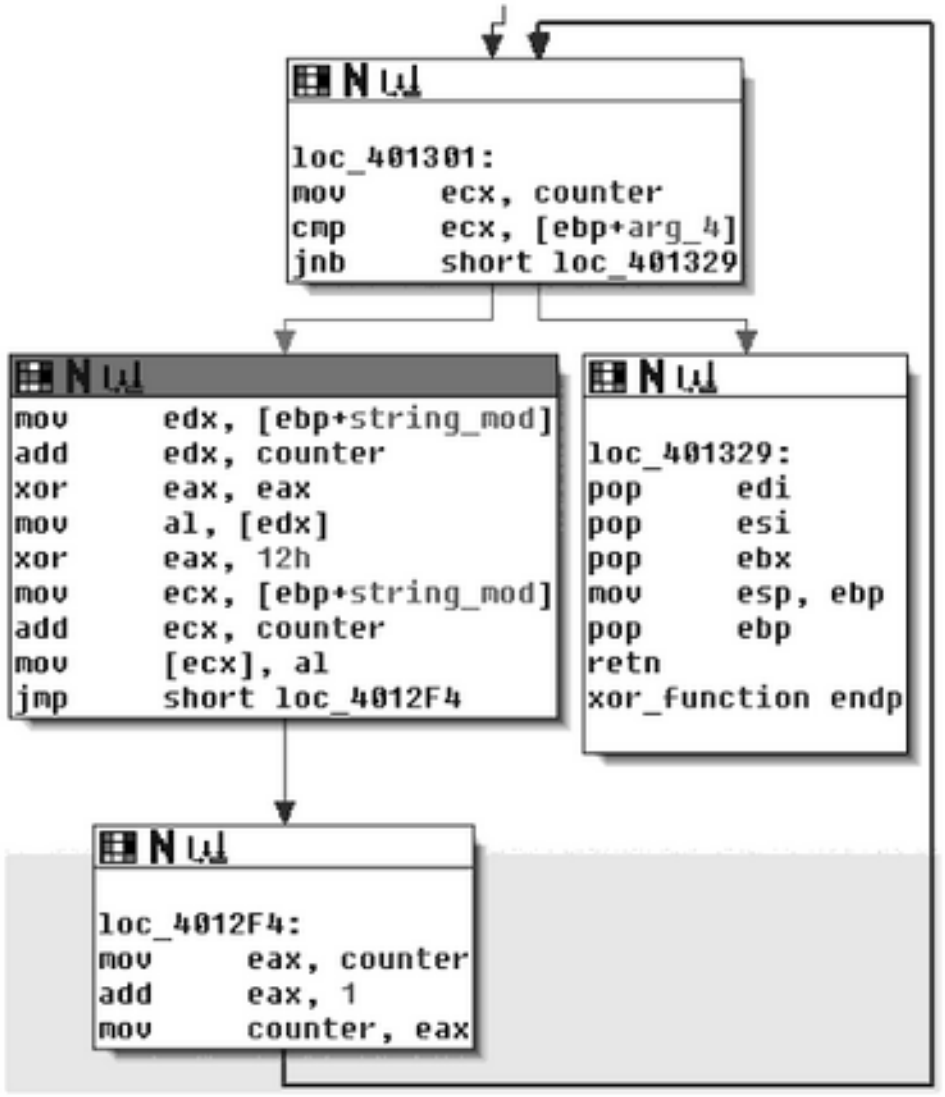


Figure 13-3: Graphical view of single-byte XOR loop

Practical Malware Analysis Figure 13-3

More instructions of Interest

- ADD and SUB
- SHL SHR
- ROL ROR
- ROT

Base64 Encoding

- Converts binary data into a character set of only 64 characters.
- MIME Base64 uses A-Z, a-z, + and -, and = for padding.
- Squeezing binary into a confined space of 64 characters.
- Takes a 3 byte (24 bit) chunk and divides it into 4 6 byte chunks.
- Each 6 byte chunk is converted to a decimal number.
- That decimal number is an index to a character

Base64 Bit Encoding

A				T				T			
0x4				0x1				0x5				0x4				0x5				0x4			
0	1	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	1	0	1	0	1	0	0
16				21				17				20			
Q				V				R				U			

Figure 13-4: Base64 encoding of ATT

Recognizing Base64 Encoding

- Look for strings of 64 different characters, then look where they are used.
  - Malware authors can use custom indexing strings – doesn’t have to be A-Z, a-z, + and -.
  - You need the malware’s indexing string to decode the base64 encoded blob
  - Malware authors may encode their indexing string and only decode it when needed

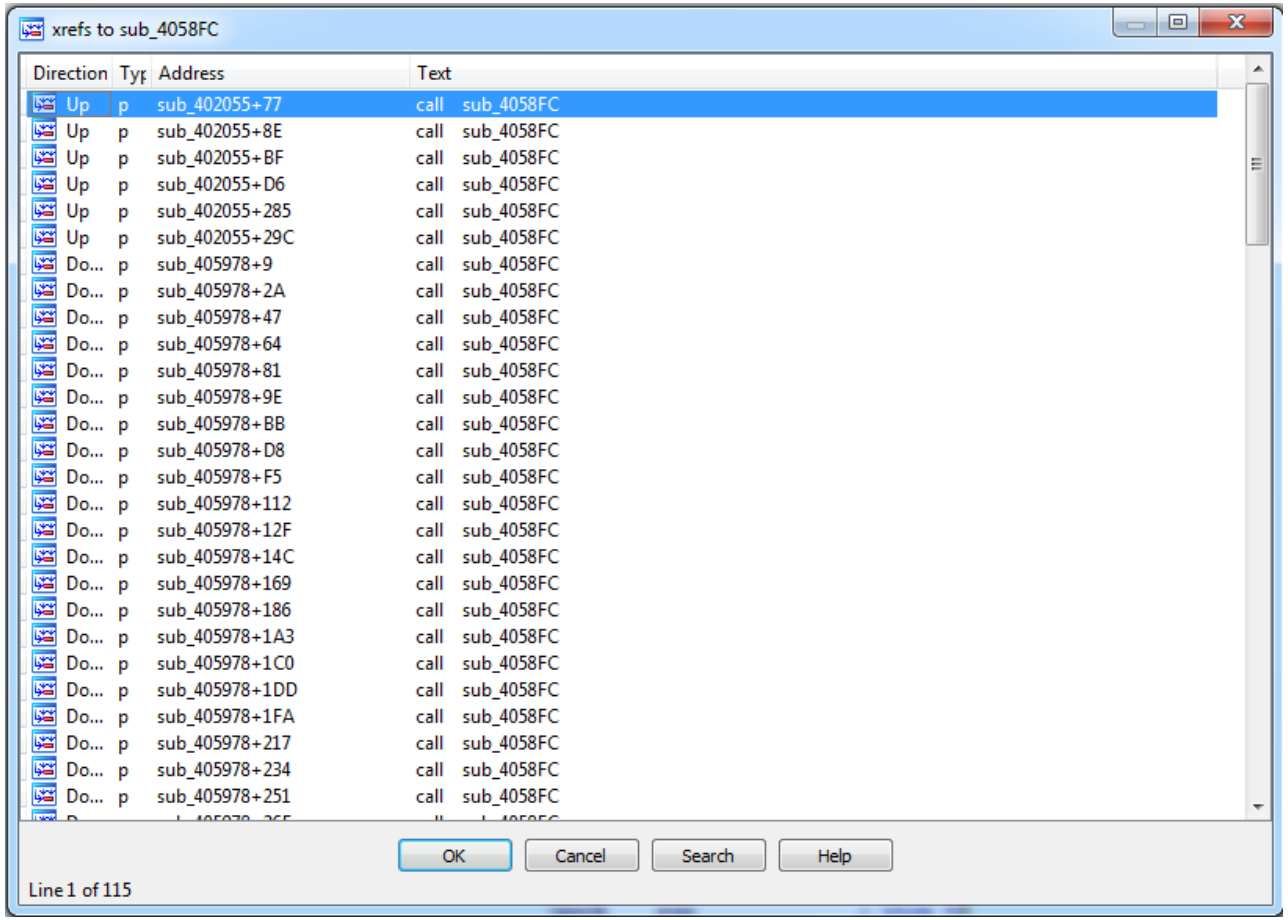
Demo

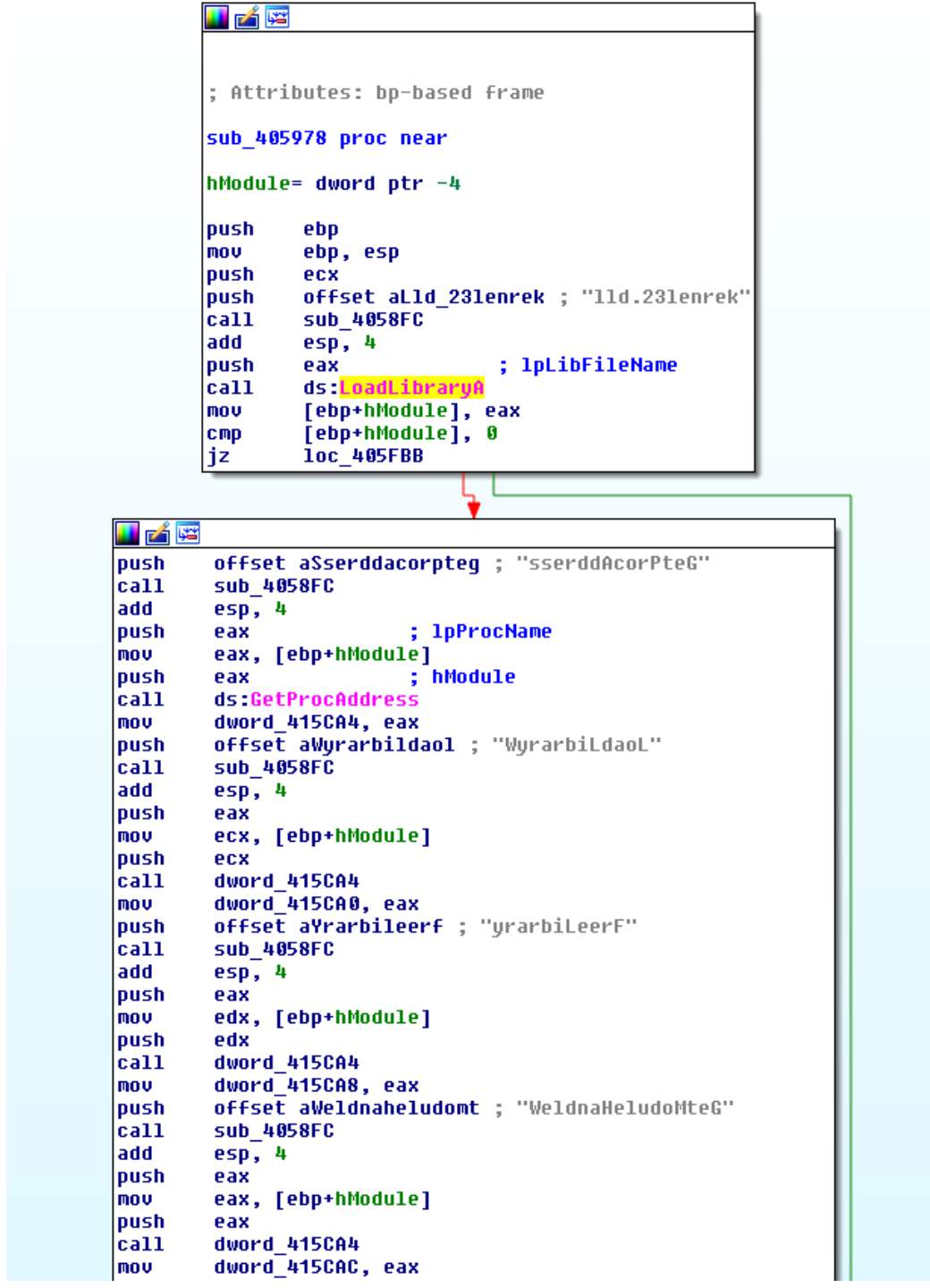
13-1

Demo

Ida Python Magic

When you Find a Decoding Function…





```
; Attributes: bp-based frame

sub_405978 proc near

hModule= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
push    offset aLld_23lenrek ; "lld.23lenrek"
call    sub_4058FC
add     esp, 4
push    eax             ; lpLibFileName
call    ds:LoadLibraryA
mov     [ebp+hModule], eax
cmp     [ebp+hModule], 0
jz      loc_405FBB
```

```
push    offset aSserddacorpteg ; "sserddAcorPteG"
call    sub_4058FC
add     esp, 4
push    eax             ; lpProcName
mov     eax, [ebp+hModule]
push    eax             ; hModule
call    ds:GetProcAddress
mov     dword_415CA4, eax
push    offset aWyrarbildaol ; "WyrarbiLdaoL"
call    sub_4058FC
add     esp, 4
push    eax
mov     ecx, [ebp+hModule]
push    ecx
call    dword_415CA4
mov     dword_415CA0, eax
push    offset aYrarbileerf ; "yrarbiLeerF"
call    sub_4058FC
add     esp, 4
push    eax
mov     edx, [ebp+hModule]
push    edx
call    dword_415CA4
mov     dword_415CA8, eax
push    offset aWeldnaheludomt ; "WeldnaHeludoMteG"
call    sub_4058FC
add     esp, 4
push    eax
mov     eax, [ebp+hModule]
push    eax
call    dword_415CA4
mov     dword_415CAC, eax
```

```
; Attributes: bp-based frame

sub_405978 proc near

hModule= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
push    offset aLld_23lenrek ; 'kernel32.dll'
call    sub_4058FC
add     esp, 4
push    eax             ; lpLibFileName
call    ds:LoadLibraryA
mov     [ebp+hModule], eax
cmp     [ebp+hModule], 0
jz      loc_405FBB
```

```
push    offset aSserddacorpteg ; 'GetProcAddress'
call    sub_4058FC
add     esp, 4
push    eax             ; lpProcName
mov     eax, [ebp+hModule]
push    eax             ; hModule
call    ds:GetProcAddress
mov     GetProcAddress_0, eax
push    offset aWyrarbildaol ; 'LoadLibraryW'
call    sub_4058FC
add     esp, 4
push    eax
mov     ecx, [ebp+hModule]
push    ecx
call    GetProcAddress_0
mov     LoadLibraryW, eax
push    offset aYrarbileerf ; 'FreeLibrary'
call    sub_4058FC
add     esp, 4
push    eax
mov     edx, [ebp+hModule]
push    edx
call    GetProcAddress_0
mov     FreeLibrary, eax
push    offset aWeldnaheludomt ; 'GetModuleHandleW'
call    sub_4058FC
add     esp, 4
push    eax
mov     eax, [ebp+hModule]
push    eax
call    GetProcAddress_0
```

Modern Standard Cryptographic Algorithms

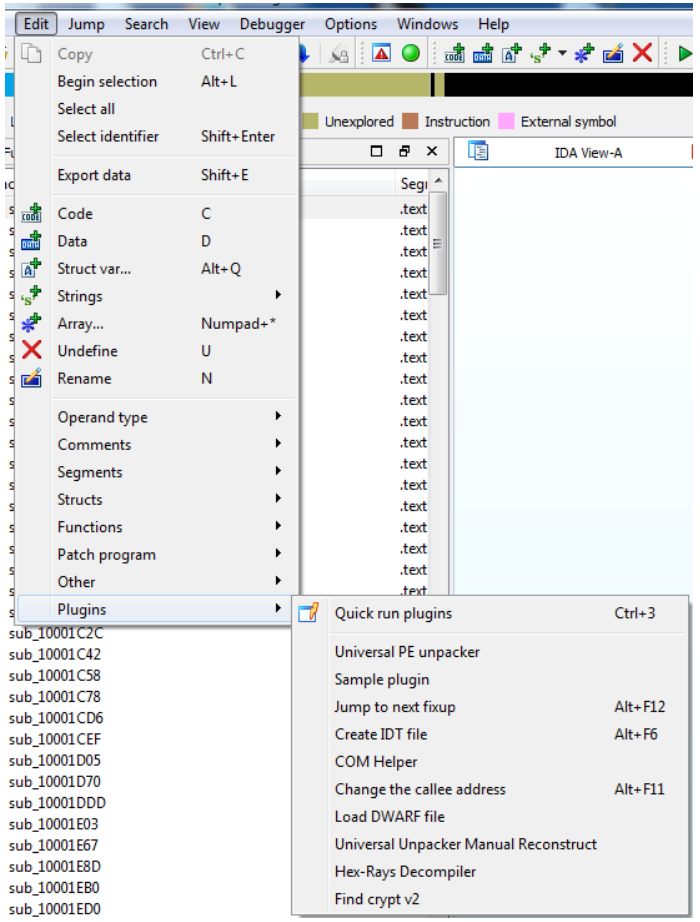
- Why use these?
  - Nearly impossible to decrypt ciphertext without possessing the key.
  - You really want to make sure your target to never knows what you exfilled

Identifying Standard Crypto: Strings and Imports

- Sometimes malware authors will compile static crypto libraries into their malware – this leaves strings behind as evidence.
- If they rely on Microsoft crypto libraries, there will be imports. They're usually pretty easily identifiable as they tend to start with "crypt"

Identifying Standard Crypto: Cryptographic Constants

- Most crypto algorithms use “magic constants” – a fixed number that is necessary to the functioning of the algorithm.
  - Exceptions are RC4 and IDEA. Guess which standard algorithms we see the most of in malware?
- Ida has a free plugin called Find Crypt that searches for magic constants.




```
The initial autoanalysis has been finished.
40EA78: found const array zinflate_lengthExtraBits (used in zlib)
40EAEC: found const array zinflate_distanceExtraBits (used in zlib)
40F4A8: found const array CRC32_m_tab (used in CRC32)
Found 3 known constant arrays in total.
```

Demo

Lab 13-3

Lab Work

- Labs 12-2, 12-4, and 13-2. Chapter 13 lab may be challenging - KANAL more or less no longer exists and Ida plugins don’t work in the free version of Ida.
  - We’ve discussed methods that don’t require plugins – look for loops with math operations inside them, imports of crypt functions, xors, etc.

- Continue working on the Obfuscated Malware Lab – much of what we went over today is applicable.

Sources/Questions/Comments/Corrections

- As usual, much credit to Andrew Honig and Michael Sikorski's Practical Malware Analysis.

- Note that animations (mostly highlighting on click) are extremely useful when teaching from this slide deck. Email me for slide originals.

- Questions/Comments/Corrections to Lauren Pearce – Laurenp@lanl.gov