Malware Dynamic Analysis
Part 3

Veronica Kovah
vkovah.ost at gmail

http://opensecuritytraining.info/MalwareDynamicAnalysis.html
All materials is licensed under a Creative Commons “Share Alike” license
http://creativecommons.org/licenses/by-sa/3.0/

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

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

See notes for citation
Where are we at?

• **Part 3: Maneuvering techniques**
  – (How malware strategically positions itself to access critical resources)
  – DLL/code injection
  – DLL search order hijacking...

• **Part 4: Malware functionality**
  – Keylogging, Phone home, Security degrading, Self-destruction, etc.
 Maneuvering

• DLL injection
• Direct code injection
• DLL search order hijacking
• Asynchronous Procedure Call (APC) injection
• IAT/EAT hooking
• Inline hooking
DLL/code Injection

- Load a malicious DLL/code into one or more processes
- Run malicious code on behalf of a legitimate process
- Bypass host-based security software
  - HIDS, Personal Firewall

IE process’s memory
DLL Injection Methods (1)

• AppInit_DLLs
  – HKLM\Software\Microsoft\Windows NT \CurrentVersion\Windows\AppInit_DLLs is set to a space or comma separated list of DLLs to load into processes that load user32.dll
  – On Windows Vista and newer you also have to set a few other values in that path like LoadAppInit_DLLs = 1 and RequireSignedAppInit_DLLs = 0

[References]
• Michael Ligh et al., Chapter 9. Dynamic Analysis, Malware Analyst's Cookbook and DVD
Observing Parite's Maneuvering

• Using Regshot on the victim VM
  1) Start Regshot (MalwareClass/tools/v5_regshot_1.8.3...)
  2) Click 1st shot button → Shot
  3) Run parite/malware.exe
  4) Click 2nd shot button → Shot
  5) Click Compare button

Q1. Which DLL is used for maneuvering?
Q2. Where is it maneuvering?
Q3. Open question:
   Any theories why it’s maneuvering to there?

[References]
Answers for Parite Lab

A1. “fmsiopcps.dll” is added to HKLM\Software\Microsoft\Windows NT \CurrentVersion\Windows\AppInit_DLLs

A2. All Windows applications, which use user32.dll
Application Programming Interface (API)

• “Specifies a software component in terms of its operations, their inputs and outputs and underling types”
  [en.wikipedia.org/wiki/Application_programming_interface](http://en.wikipedia.org/wiki/Application_programming_interface)

• `char *strncpy(char *dest, const char *src, size_t n)`;
  
  — 3 inputs:
  • `dest`: destination string
  • `src`: source string
  • `n`: number of characters to copy from source string
  
  — 1 output: returns a pointer to the destination string

[References]


• `strcpy(3)` - Linux man page, [http://linux.die.net/man/3/strcpy](http://linux.die.net/man/3/strcpy)

See notes for citation
DLL Injection Methods (2)

• **CreateRemoteThread** Windows API
  – Manipulate a victim process to call LoadLibrary with the malicious DLL name
  – Malicious code is located in DllMain, which is called once a DLL is loaded into memory
  – A common API call pattern:
    • OpenProcess→VirtualAllocEx→
      WriteProcessMemory→GetModuleHandle→
      GetProcAddress→CreateRemoteThread

• Also, a **direct code injection** method

[References]
• Michael Sikorski et al., Chapter 12. Covert Malware Launching, Practical Malware Analysis
HANDLE WINAPI OpenProcess(  
  _In_  DWORD dwDesiredAccess,  
  _In_  BOOL bInheritHandle,  
  _In_  DWORD dwProcessId  
);  

• dwProcessId [in]  
  – The identifier of the local process to be opened...  

• Return value  
  – If the function succeeds, the return value is an open  
    handle to the specified process...

[References]  
OpenProcess®VirtualAllocEx®WriteProcessMemory®
GetModuleHandle®GetProcAddress®CreateRemoteThread

\[
\begin{align*}
\text{LPVOID WINAPI VirtualAllocEx}( & \\
_\text{In}_ & \text{HANDLE hProcess,} \\
_\text{In opt}_ & \text{LPVOID lpAddress,} \\
_\text{In}_ & \text{SIZE_T dwSize,} \\
_\text{In}_ & \text{DWORD flAllocationType,} \\
_\text{In}_ & \text{DWORD flProtect} \\
\end{align*}
\]

- **hProcess** [in]
  - The handle to a process. The function allocates memory within the virtual address space of this process...
- **dwSize** [in]
  - The size of the region of memory to allocate, in bytes...
- **Return value**
  - If the function succeeds, the return value is the base address of the allocated region of pages...

**References**
OpenProcess ➔ VirtualAllocEx ➔ WriteProcessMemory ➔
GetModuleHandle ➔ GetProcAddress ➔ CreateRemoteThread

`HMODULE WINAPI GetModuleHandle(  
   _In_opt_ LPCTSTR lpModuleName  
);`

- pModuleName [in, optional]
  - The name of the loaded module (either a .dll or .exe file)...
- Return value
  - If the function succeeds, the return value is a handle to the specified module...

[References]
GetProcAddress

FARPROC WINAPI GetProcAddress(
  _In_  HMODULE hModule,
  _In_  LPCSTR lpProcName
);

- hModule [in]
  - A handle to the DLL module that contains the function or variable...
- lpProcName [in]
  - The function or variable name, or the function's ordinal value...
- Return value
  - If the function succeeds, the return value is the address of the exported function or variable...

[References]
OpenProcess→VirtualAllocEx→WriteProcessMemory→GetModuleHandle→GetProcAddress→CreateRemoteThread

HANDLE WINAPI CreateRemoteThread(
  _In_ HANDLE hProcess,
  _In_ LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_ SIZE_T dwStackSize,
  _In_ LPTHREAD_START_ROUTINE lpStartAddress,
  _In_ LPVOID lpParameter,
  _In_ DWORD dwCreationFlags,
  _Out_ LPDWORD lpThreadId

• hProcess [in] — A handle to the process in which the thread is to be created...
• lpStartAddress [in]
  — A pointer to the application-defined function of type
    LPTHREAD_START_ROUTINE to be executed by the thread and
    represents the starting address of the thread in the remote process...
• lpParameter [in]
  — A pointer to a variable to be passed to the thread function.

[References]
CreateRemoteThread() cont.

- IpStartAddress’s type is LPTHREAD_START_ROUTINE, which is defined as:
  \[
  \text{typedef DWORD (__stdcall *LPTHREAD_START_ROUTINE) (}
  \text{[in] LPVOID IpThreadParameter)}
  \]
  
- You can’t put any function as IpStartAddress. It has to be one which matches the above prototype.

- One (popular) example is:
  \[
  \text{HMODULE WINAPI LoadLibrary(}
  \text{ _In_ LPCTSTR IpFileName)}
  \]
DLL Injection API Call Example

malicious process

LoadLibrary(filename)

myInjectDll()
{
    buf = "evil.dll"
}

Internet Explorer process
PID: 109

kernel32.dll

LoadLibrary(filename)
DLL Injection API Call Example

```
LoadLibrary(filename)

myInjectDll()
    { buf = "evil.dll"
      h=OpenProcess(_,proc_id)
    }

malicious process
See notes for citation

kernel32.dll

LoadLibrary(filename)

Internet Explorer process
PID: 109
```
DLL Injection API Call Example

LoadLibrary(filename)

kernel32.dll

LoadLibrary(filename)

myInjectDll()
{
  buf = "evil.dll"
  h=OpenProcess(_,proc_id)
  addr = VirtualAllocEx(h, size,)
}

malicious process

Internet Explorer process
PID: 109
DLL Injection API Call Example

```c
LoadLibrary(filename)

myInjectDll()
{
    buf = "evil.dll"
    h = OpenProcess(_, proc_id)
    addr = VirtualAllocEx(h, _, size, _)
}
```

```
kernel32.dll
LoadLibrary(filename)

0x4000
```

malicious process

Internet Explorer process

PID: 109
DLL Injection API Call Example

```c
LoadLibrary(filename)

myInjectDll()
{  buf = "evil.dll"
   h=OpenProcess(.,proc_id)
   addr = VirtualAllocEx(h.,size,.)
   WriteProcessMem(h,addr,buf,size,...)
}
```

malicious process

Table:

<table>
<thead>
<tr>
<th>DLL (kernel32.dll)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadLibrary(filename)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DLL (kernel32.dll)</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4000</td>
<td></td>
</tr>
</tbody>
</table>

Internet Explorer process

PID: 109
DLL Injection API Call Example

```c
LoadLibrary(filename)

myInjectDll()
{
    buf = "evil.dll"
    h = OpenProcess(proc_id, 0, 0)
    addr = VirtualAllocEx(h, 0x4000, size, MEM_COMMIT, PAGE_EXECUTE_READWRITE)
    WriteProcessMem(h, addr, buf, size, 0)
}
```

See notes for citation

Internet Explorer process
PID: 109
**DLL Injection API Call Example**

malicious process

```
myInjectDII()
{
    buf = "evil.dll"
    h = OpenProcess(_, proc_id)
    addr = VirtualAllocEx(h, size, _)
    WriteProcessMem(h, addr, buf, size,...)
    CreateRemoteThread(h, start, param,...)
}
```

Internet Explorer process

PID: 109

See notes for citation
**DLL Injection API Call Example**

<table>
<thead>
<tr>
<th>LoadLibrary(filename)</th>
<th>kernel32.dll</th>
</tr>
</thead>
<tbody>
<tr>
<td>myInjectDll()</td>
<td></td>
</tr>
<tr>
<td>{ buf = &quot;evil.dll&quot;</td>
<td>0x4000</td>
</tr>
<tr>
<td>h=OpenProcess(proc_id)</td>
<td></td>
</tr>
<tr>
<td>addr = VirtualAllocEx(h, size,)</td>
<td></td>
</tr>
<tr>
<td>WriteProcessMem(h,addr,buf,size,...)</td>
<td></td>
</tr>
<tr>
<td>CreateRemoteThread(h,,start,param,...)</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

malicious process

See notes for citation

Internet Explorer process
PID: 109
Observing “Onlinegame2” Maneuvering

• For this lab, we will use WinApiOverride (an API monitor) to analyze onlinegames/2/malware.exe

• Hint: new process will be invoked

Q1. What is the address of LoadLibrary()?  
Q2. Where is it maneuvering to?  
Q3. What’s the path of the DLL being injected?
Answers for “Onlinegame2” Lab

A1. 0x7C801D7B
A2. Explorer.exe, OpenProcess takes PID as its parameter
A3. C:\WINDOWS\system32\ailin.dll
Observing “Onlinegame1” Maneuvering

- Spot the direct code injection
- Use WinApiOverride (an API monitor) to analyze onlinegames/1/malware.exe

**Q1.** What is the size of the code being injected?

**Q2.** Where is it maneuvering?

**Q3.** What’s the path of DLL being injected?

- Take a dump of the process using Process Explorer.
Answers for “Onlinegame1” Lab

A1. 0x457
A2. Explorer.exe, OpenProcess takes PID as its parameter
A3. C:\Windows\System32\nmdfgds0.dll
   • Process Explorer provides process memory dump. In order to open the dump file, use windbg’s File→Open Dump menu option
Thread

- AKA light weight process who has own program counter (EIP), a register set, and a stack
- Multiple threads can exist in a process and share a process's resources, such as opened file and network connection, concurrently
- Thread context switching is much cheaper than process context switching

[References]
- Silberscharz Galvin, Chapter 5 Threads, Operating System Concepts 5th Edition

[Image Sources]
- http://www.cs.cf.ac.uk/Dave/C/mthread.gif
**DLL Injection Methods (3a)**

- **SetWindowsHookEX Windows API**
  - Monitor certain types of events (e.g. key strokes)
  - HHOOK WINAPI SetWindowsHookEx(
    _In_  int idHook,
    _In_  HOOKPROC lpfn,
    _In_  HINSTANCE hMod,
    _In_  DWORD dwThreadId
  );

[References]
- Michael Sikorski et al., Chapter 12. Covert Malware Launching, Practical Malware Analysis
DLL Injection Methods (3b)

- If `dwThreadId` is zero, it injects DLL into memory space of every process in the same Windows “desktop” (which is a memory organization term, not the desktop you see when looking at your computer)
- If `dwThreadId` belongs to another process, it injects DLL into the process
- For the sake of simple DLL injection, use uncommon message type (e.g. WH_CBT)
DLL Injection Methods (4)

• Codecave *(a redirection of program execution to another location and then returning back to the area where program execution had previously left.)*
  – Inject a snippet of code, which calls LoadLibrary, to a victim process
  – Suspend a thread in the victim process and restart the thread with the injected code
  – API call pattern
    • OpenProcess → VirtualAllocEx → WriteProcessMemory → SuspendThread → GetThreadContext → SetThreadContext → ResumeThread

[References]
Maneuvering

- DLL injection
- Direct code injection
- **DLL search order hijacking**
- Asynchronous Procedure Call (APC) injection
- IAT/EAT hooking
- Inline hooking
**DLL Search Order Hijacking (1)**

- (default) DLL search order in Windows XP SP3
  1. KnownDLLs and its dependent DLLs
     HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs
  2. Directory from which the application loaded
  3. System directory (e.g. c:\WINDOWS\system32)
  4. 16-bit System Directory (e.g. c:\WINDOWS\system)
  5. Windows Directory
  6. Current working directory
  7. Directories in %Path%

[References]
DLL Search Order Hijacking (2)

• Also an obfuscated method to be persistent
• A malware can make a legitimate looking DLL
  – Loaded by an application
  – In the directory where the application is located or the current working directory
  – Which is not listed in KnownDLLs and its dependent DLLs
  – Identically named dll as the one in system32 directory

[References]
• Nick Harbour, Malware Persistence without the Windows Registry, https://www.mandiant.com/blog/malware-persistence-windows-registry/
Checking KnownDLLs

- Use Regedit
  1) Start → Run... → regedit
  2) Search for the following registry key
     HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet
     \Control\Session Manager\KnownDLLs

- Use Winobj.exe to see all dependent DLLs of KnownDLL
  - On desktop, SysinternalSuite\Winobj.exe
  - Check \KnownDLLs

See notes for citation
Observing Nitol's Maneuvering

- For this lab, we will use Process Monitor to analyze nitol/malware.exe
- **Q1.** Which DLL is used for maneuvering?
- **Q2.** Where is it maneuvering to?
- **Q3.** Open question: Any theories why it’s maneuvering to there?
- **Q4.** Bonus question: How does it persist?

[References]

- Rex Plantado, MSRT October '12 - Nitol: Counterfeit code isn't such a great deal after all, http://blogs.technet.com/b/mmpc/archive/2012/10/15/msrt-october-12-nitol-counterfeit-code-isn-t-such-a-great-deal-after-all.aspx
Answers for Nitol Lab

A1. lpk.dll was written to multiple directories where executable files exist
   – C:\Program Files\Internet Explorer\lpk.dll
   – C:\Program Files\Messenger\lpk.dll etc.
   – Check where lpk.dll is loaded from with iexplorer.exe

A2. All executable which has lpk.dll in the same directory and uses lpk.dll

Just for fun, 基础类应用程序 means "Foundation Classes application" according to Google Translate
Maneuvering

- DLL injection
- Direct code injection
- DLL search order hijacking
- Asynchronous Procedure Call (APC) injection
- IAT/EAT hooking
- Inline hooking
Asynchronous Procedure Call (APC) Injection

• A function executed asynchronously when a thread is in an alertable state
• A thread enters to alertable states when it calls some functions such as SleepEx, WaitForSingleObjectEx, WaitForMultipleObjectEx
• Each thread has a queue of APCs
• Kernel-mode APC is generated by the system
• User-mode APC is generated by an application
• API call pattern
  – OpenThread→QueueUserAPC
  – From kernel-space to run user-mode code: KeInitializeAPC→KeInsertQueueApc

[References]
• Michael Sikorski et al., Chapter 12. Covert Malware Launching, Practical Malware Analysis
IAT/EAT Hooking

• Import Address Table (IAT) holds addresses of dynamically linked library functions
• Export Address Table (EAT) holds addresses of functions a DLL allows other code to call
• Overwrite one or more IAT/EAT entries to redirect a function call to the attacker controlled code
• IAT hooking only affects a module
• EAT hooking affects all modules loaded after EAT hooking takes place
• IAT & EAT hooking only affect one process memory space

[References]
• Xeno Kovah, Rookits: What they are, and how to find them, http://opensecuritytraining.info/Rootkits.html
Normal Inter-Module Function Call

WickedSweetApp.exe

... push 1234
call [0x40112C]
add esp, 4
...
Import Address Table
0x40112C: SomeFunc
0x401130: SomeJunk
0x401134: ScumDunk
...

WickedSweetLib.dll

... SomeFunc:
mov edi, edi
push ebp
mov ebp, esp
sub esp, 0x20
...
ret

From the Rootkits class

See notes for citation
Normal Inter-Module Function Call

WickedSweetApp.exe

... push 1234
call [0x40112C]
add esp, 4
...
Import Address Table
0x40112C:MySomeFunc
0x401130:SomeJunk
0x401134:ScumDunk
...

WickedWickedDll.dll

MySomeFunc:
... call SomeFunc()
... ret

WickedSweetLib.dll

... SomeFunc:
  mov edi, edi
  push ebp
  mov ebp, esp
  sub esp, 0x20
  ...
  ret

From the Rootkits class

See notes for citation
Inline Hooking

- There are a few first meaningless bytes at the beginning of a function for hooking if it is compiled with /hotpatch option
- Overwrite the first 5 or so bytes of a function with jump to the attacker's code
- This redirect the program control from the called function to the malicious code
- Execute any instructions overwritten in the first 5 bytes as the last part of the malicious code before jumping back to wherever it came from

[References]
- Greg Hoglund et al., Chapter 4. The Age-Old Art of Hooking, Rootkits
Normal Intra-Module Function Call

WickedSweetApp.exe

1

... push 1234
   call SomeFunc()
   add esp, 4
   ...
   SomeFunc:
   mov edi, edi
   push ebp
   mov ebp, esp
   sub esp, 0x20
   ...
   ret

2

From the Rootkits class
Inline Hooked Intra-Module Function Call

WickedSweetApp.exe

... push 1234
call SomeFunc()
add esp, 4
...
...
SomeFunc:
jmp MySomeFunc
sub esp, 0x20
...
ret

WickedWickedDll.dll

MySomeFunc:
<stuff>
...
mov edi, edi
push ebp
mov ebp, esp
jmp SomeFunc+5

From the Rootkits class
Many processes, each with their own view of memory, and the kernel schedules different ones to run at different times.