The Real Shim Shady

William Ballenthin, FireEye
Jonathan Tomczak, Mandiant
Bio, plan

- William Ballenthin, Reverse Engineer
  - FLARE (FireEye Labs Advanced Reverse Engineering) team
  - Malware analysis, forward and backward engineering

- Jonathan Tomczak, Consultant
  - Mandiant Professional Services
  - Incident response, forensics, tool development

- Today's Topic: Case Study and Investigative Techniques for Hijacked Application Compatibility Infrastructure.
Put out the Fire!

- Working the malware triage queue, encountered interesting situation:
  - Client targeted by phishing emails
  - Large deployment FireEye boxes didn’t fire
  - Malware maintained persistence, somehow

- What’s going on? How to fix detection & investigative methodology?
DLL Injection via Shims

- Malware: self-extracting RAR
  - drops KORPLUG launcher (`elogger.dll`)
  - loading shellcode backdoor (`elogger.dat`)
- `elogger.dat` does a little of everything: manually loads PE payload, injects, privesc, installs service, HTTP protocol
- Also, installs an ACI shim:
  - Writes two (32/64-bit) hardcoded, embedded SDB files to disk
  - Invokes `sdbinst.exe`
WHAT’S THE ACI?

What are shims and why are they on my system?
Application Compatibility Infrastructure

- Manages and resolves application compatibility issues with updates to Microsoft Windows
- Configured via freely available Application Compatibility Toolkit (ACT)
- API hooking (& more) built into the executable Loader
  - “Shims” typically implemented as code (DLLs) or configuration (disable feature)
  - Shims described by databases (SDB files) indicating source and target
  - SDBs registered with the OS, queried by loader
Application Compatibility Infrastructure, II

- Targets specified by executable file metadata, including:
  - Filename
  - PE checksum
  - File size
  - Version info fields, etc.

- Lots of shims to play with
  - Dozens of preconfigured quickfixes (redirect file reads, change heap behavior)
  - Thousands of SDB entries distributed by MS
  - Some undocumented features
    - EMET uses ACI to inject its DLL into processes on execution
SHIM TECHNIQUES
Shim development, creation, and deployment
The Application Compatibility Toolkit
SDB deployment

- `sdbinst.exe` registers SDB files with operating system
  - Creates uninstallation entries in the control panel
  - Add values to Registry keys:
    - HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom
    - HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\InstalledSDB

- Microsoft recommends packaging in an MSI and deploying via GPO
- Directly adding the Registry values circumvent `sdbinst.exe` and extra control panel entries
# Fun shims

<table>
<thead>
<tr>
<th>Shim Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisableWindowsDefender</td>
<td>“The fix disables Windows Defender for security applications that do not work with Windows Defender.”</td>
</tr>
<tr>
<td>CorrectFilePaths</td>
<td>Redirect file system paths</td>
</tr>
<tr>
<td>LoadLibraryRedirectFlag</td>
<td>Change load directory of DLLs</td>
</tr>
<tr>
<td>NoSignatureCheck</td>
<td>??? 😊</td>
</tr>
<tr>
<td>RelaunchElevated</td>
<td>Ensure an EXE runs as admin</td>
</tr>
<tr>
<td>TerminateExe</td>
<td>??? 😊</td>
</tr>
<tr>
<td>VirtualRegistry</td>
<td>Registry redirection and expansion</td>
</tr>
</tbody>
</table>
Trick 1: DLL Injection via shims (seen in wild)

- Self-extracting RAR
  - drops KORPLUG launcher (elogger.dll)
  - loading shellcode backdoor (elogger.dat)
- elogger.dat does some of everything: manually loads PE payload, injects, privesc, installs service, HTTP protocol
- Also, installs an ACI shim:
  - Writes two (32/64-bit) hardcoded, embedded SDB files to disk
  - Invokes sdbinst.exe
SDB contents

<DATABASE><NAME type='stringref'>Brucon_Database</NAME>
<DATABASE_ID type='guid'>503ec3d4-165b-4771-b798-099d43b833ed</DATABASE_ID>
<LIBRARY><SHIM><NAME type='stringref'>Brucon_Shim</NAME><DLLFILE type='stringref'>Custom\elogger.dll</DLLFILE></SHIM></LIBRARY><EXE><NAME type='stringref'>svchost.exe</NAME><APP_NAME type='stringref'>Brucon_Apps</APP_NAME><EXE_ID type='hex'>e8cc2eb6-469d-43bc-9d6a-de089e497303</EXE_ID><MATCHING_FILE><NAME type='stringref'>*</NAME></MATCHING_FILE><SHIM_REF><NAME type='stringref'>Brucon_Shim</NAME></SHIM_REF></EXE></DATABASE>
Analysis

- Persistence configured via opaque file format
- Hardcoded SDB file easily sig-able via filenames, IDs
  - Payload file exists in the clear, in very limited set of directories
    - C:\Windows\AppPatch\Custom\n    - C:\Windows\AppPatch\Custom\Custom64\n
- FireEye identified filename elogger.dll often reused in KORPLUG & SOGU campaigns.
Trick 2: Argument replacement via shims (seen in lab)

- CorrectFilePath fix redirects arguments from the application’s path to an attacker’s specified path
  - Trivial to hook into CreateProcess, WinExec, ShellExecute

- Custom program mine.exe, launches C:\windows\temp\1.exe
  - Add shim: redirects C:\windows\temp\1.exe to C:\dump\1.exe
  - CorrectFilePath: “C:\windows\temp\1.exe; C:\dump\1.exe”
SDB contents

<DATABASE><TIME type='integer'>0x1d100fac0a4a7fc</TIME>
 <NAME type='stringref'>minesdb</NAME>
 <DATABASE_ID type='guid'>2840a82e-91ff-4f29-bff2-fd1e9780b6eb</DATABASE_ID>
 <EXE>
  <APP_NAME type='stringref'>mine.exe</APP_NAME>
  <MATCHING_FILE><NAME type='stringref'>*</NAME></MATCHING_FILE>
  <SHIM_REF>
   <NAME type='stringref'>CorrectFilePaths</NAME>
   <COMMAND_LINE type='stringref'>"C:\Windows\Temp\1.exe; C:\dump\1.exe"
   </COMMAND_LINE>
  </SHIM_REF></EXE></DATABASE>
Trick 2: Argument replacement via shims, II

- Analysis:
  - Consider the targeted process is `cmd.exe`
    - Hidden persistence, MITM of process creation
    - #DFIR confusion
  - Configured via opaque file format
  - Payload not limited to specific directories
Trick 3: Shellcode injection via shims (seen in wild)

- Phishing email leads to dropper
  - dropper installs template SDB and modifies them dynamically
    - *SDB declares shellcode that it injects on executable load*
    - payload is a downloader for other stages

- First identified by TrendMicro…
SDB contents

<DATABASE><NAME type='StringRef'>opera.exe</NAME>
<DATABASE_ID>
   538f5e1c-932e-4426-b1c9-60a6e15bcd7f
</DATABASE_ID>
<LIBRARY><SHIM_REF><PATCH>
   <NAME type='StringRef'>patchdata0</NAME>
   <PATCH_BITS type='hex'>040000c...0000000000000000
</PATCH_BITS>
</PATCH></SHIM_REF></LIBRARY>
<EXE><APP_NAME type='StringRef'>opera.exe</APP_NAME>
<MATCHING_FILE><NAME>opera.exe</NAME></MATCHING_FILE>
<PATCH_REF><NAME type='StringRef'>patchdata0</NAME>
   <PATCH_TAGID type='integer'>0x6c</PATCH_TAGID>
</PATCH_REF></EXE></DATABASE>
PATCH_BITS

- Windows loader writes arbitrary bytes into module memory
  - PATCH_MATCH to verify target of memory write
  - PATCH_REPLACE stamps in raw bytes
  - Can target both EXE and DLL modules
## Patch details

<table>
<thead>
<tr>
<th>Offset</th>
<th>Opcode</th>
<th>Address</th>
<th>Module Name</th>
<th>Pattern</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 (04)</td>
<td>PATCH_MATCH</td>
<td>0x53c2e</td>
<td>u'kernel32.dll'</td>
<td>9090909090</td>
<td>0x53c2e: nop</td>
</tr>
<tr>
<td>0x0000000c (04)</td>
<td></td>
<td>0x53c2f</td>
<td></td>
<td></td>
<td>0x53c2f: nop</td>
</tr>
<tr>
<td>0x0000014 (64)</td>
<td></td>
<td>0x53c30</td>
<td></td>
<td></td>
<td>0x53c30: nop</td>
</tr>
<tr>
<td>0x0000054 (05)</td>
<td></td>
<td>0x53c31</td>
<td></td>
<td></td>
<td>0x53c31: nop</td>
</tr>
<tr>
<td>0x0000054 (07)</td>
<td>PATCH_REPLACE</td>
<td>0x53c32</td>
<td></td>
<td>e8321a0700ebf9</td>
<td>0x53c32: call 0x000c5665</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x53c33: jmp 0x00053c29</td>
</tr>
</tbody>
</table>
## Patch details, II

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Module Name</th>
<th>Pattern</th>
<th>Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>PATCH_MATCH</td>
<td>u'kernel32.dll'</td>
<td>83042402609ce80300000009d61c3</td>
<td>0xc5665: add dword [esp],2</td>
</tr>
<tr>
<td>0000000c</td>
<td>PATCH_REPLACE</td>
<td>u'kernel32.dll'</td>
<td></td>
<td>0xc5669: pushad</td>
</tr>
<tr>
<td>00000014</td>
<td></td>
<td>u'kernel32.dll'</td>
<td></td>
<td>0xc566a: pushfd</td>
</tr>
<tr>
<td>00000054</td>
<td></td>
<td>u'kernel32.dll'</td>
<td></td>
<td>0xc566b: call 0x000c566d</td>
</tr>
<tr>
<td>0000000000</td>
<td></td>
<td></td>
<td></td>
<td>0xc5670: popfd</td>
</tr>
<tr>
<td>000000000c</td>
<td></td>
<td></td>
<td></td>
<td>0xc5671: popad</td>
</tr>
<tr>
<td>00000014</td>
<td></td>
<td></td>
<td></td>
<td>0xc5672: ret</td>
</tr>
</tbody>
</table>
Patch details, III

< Multi-kilobyte shellcode downloader >
Patch details, summary

Kernel32.dll

Legit Call

Legit Code

Legit Call

Legit Code
Patch details, summary

Kernel32.dll

Legit Code

Legit Call

Hook

Legit Call

Legit Code
Patch details, summary

Kernel32.dll

Legit Call

Legit Code

Trampoline

Legit Call

Legit Code
Patch details, summary

Kernel32.dll

Legit Call

Fetch & exec backdoor

Legit Call

Legit Code

Legit Call

Legit Code
Patch details, summary

Kernel32.dll

Legit Code

Return to trampoline

Legit Call

Legit Code
Patch details, summary

Kernel32.dll

Legit Call

Legit Code

Legit Call

Legit Code
Analysis

- Persistence & injection by MS infrastructure!
- External storage of shellcode in opaque format

- Dynamic modification of SDB files from template
  - Generates unique GUIDs for database ID
  - Extensible payloads
  - `PATCH_BYTES` not documented
FLYING THROUGH THE MATRIX

Understanding SDB files
The SDB file format is an undocumented Microsoft format

- `apphelp.dll` exposes ~254 exports for manipulating shims
- That doesn’t help for forensic analysis!
SDB file format, II

- So, we reverse engineered it
- Conceptually, like an indexed XML document
  - Three main nodes: the index, the database structure, and a string table
  - No compression, encryption, signatures, nor checksums
Consider the scenario

- Shim definition: name & shim action
  ```xml
  <LIBRARY>  <SHIM>
    <NAME type='stringref'>Brucon_Shim</NAME>
    <DLLFILE type='stringref'>Custom\elogger.dll</DLLFILE>
  </SHIM></LIBRARY>
  ```

- Application definition: target & shim pointer
  ```xml
  <NAME type='stringref'>svchost.exe</NAME>
  <APP_NAME type='stringref'>Brucon_Apps</APP_NAME>
  <SHIM_REF>
    <NAME type='stringref'>Brucon_Shim</NAME>
    <SHIM_TAGID type='integer'>0x47c</SHIM_TAGID>
  </SHIM_REF>
  ```
python-sdb

- Some tools exist for unpacking SDB files
  - But they rely on the Windows API

- python-sdb is a cross platform, pure Python library for parsing SDBs
  - Python API makes it easy to build scripts that inspect SDB features
  - Provided sample scripts dump database as various XML flavors

- https://github.com/williballenthin/python-sdb
DETECTION METHODOLOGY

Investigating malicious shims at scale in a large environment
Consider the scenario

- **Trojan.mambashim**
  - Python (what, just read the source!?!)
  - Obfuscated bytecode
  - Installs service, or uses `ctypes` to dynamically create sdb and install
  - sdb causes Windows loader to inject DLL payload launcher into `putty44.exe`

Would you have any idea this was happening to your environment?
Existing administrative tools?

- Fact: *Trojan.mambashim* generates random sdb path using a dictionary of English words, installs using *sdbinst.exe*

- ACI Fails:
  - No central management for SDBs on a system
  - No Active Directory tools for SDB management
  - No accounting of ACI changes or rollback features

- Win?
  - Maybe catch *sdbinst.exe* via process auditing?
ACI Integrity checking?

- SDB files are not signed 😞
- Whitelisting SDBs by hash does **not** work
  - eg. collection across 6,000 hosts yields 18,000 unique SDB files
- Embedded timestamps and installation order affect SDB integrity checks
  - If Office is installed before Visual Studio, and then vice versa on another system, it may result in a different SDB.
Mass inspection & anomaly detection

- Acquire, inspect `%systemdrive%\*.sdb`
  - Legitimate SDBs typically reside in Windows and Program Files
  - Attacker SDBs found in `%USERSPROFILE%\, working directories`
- Acquire, inspect
  - HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom
  - HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\InstalledSDB
- Default sdb's: drvmain, frxmain, msimain, pcamain, sysmain
Mass inspection & anomaly detection

- **Trojan.mambashim**
  - Random header timestamp (range 0-max int64 (!!!))
  - Random compiler version (rand.rand.rand.rand.rand)
  - EXE vendor name vendor
  - Random database ID (well, it’s a GUID…)
  - Random EXE ID (also GUID…)

- But, blacklist won’t scale
- Good for hunting, not fire and forget
Mass inspection & anomaly detection, II

Microsoft-Windows-Application-Experience-Program-Telemetry.evtx

Compatibility fix applied to C:\PROGRAM FILES\Putty\putty44.exe.
Fix information: vendor, {7e4053fe-ade9-426f-9dc2-0bbfa76b5366}, 0x80010156.

- Do you have technology that can detect “unusual entries”?
  - Count tuple (hostname, vendor, application) & sort ASC
  - Alert on new tuples?
Domain specific hashing

- Realistically, *Trojan.mambashim* could be much nastier.

- We don’t expect blacklisting to scale, that’s just playing catch up

- We really want to whitelist:
  - But, can’t whitelist entire files by hash (see earlier)
  - **Can** hash shim & application definitions
    - Don’t expect these to change
    - Use this to build a whitelist!
      - `shims_hash_shims.py`
Prepare for this scenario

- https://github.com/ganboing/sdb_packer
  - Extract existing legit sysmain.sdb
  - Add new shim for explorer.exe, etc.
    - Payload: keylog data & shellcode that does exfil
  - Re-pack sysmain.sdb
  - Deploy
  - ???
  - Profit
Shims are real. Don’t get shimmed.

- Both targeted and commodity threats are actively using ACI shims
- There is no existing infrastructure for detection
- Consider the risk

- You are now the front line.
Prior work

- “Persist It - Using and Abusing Microsoft Fix It Patches” - Jon Erickson/iSIGHT @ BH ’14

- “Shim: A new method of injection” (in Russian)
  ftp://os2.fannet.ru/fileechoes/programming/XA 159.PDF

- “Roaming Tiger” - Anton Cherepanov/ESET @ ZeroNights ’14

- “Windows - Owned By Default!” – Mark Baggett @ DerbyCon 2013

- “Compatibility Fix Descriptions” - MSDN
THE END
Questions?
File Timestamp Indicators

- Filesystem created timestamp indicates installation of SDB to the system
  - Windows Patch
  - Application Install
  - Malicious SDB that was pre-compiled before installation.
- Registry timestamps show installation timestamp
- Filesystem modified timestamp indicates that the SDB was recompiled.
  - Windows Patch
  - Application Install
  - Malicious injection into an existing SDB such as sysmain.sdb
Notes on artifacts

- FireEye identified filename `elogger.dll` often reused in KORPLUG & SOGU campaigns.

- `elogger.dll` exports `ShimMain` and `NotifyShims`, which are undocumented shim entry points. Some KORPLUG loaders also export these without SOGU payloads referencing the ACI.

- “Roaming Tiger” (ESET) campaign distributed SDB files with similar naming schemes:

<table>
<thead>
<tr>
<th><code>elogger.dat</code></th>
<th>“Roaming Tiger”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brucon_Shim</td>
<td>AcProtect_Shim</td>
</tr>
<tr>
<td>Brucon_Apps</td>
<td>AcProtect_Apps</td>
</tr>
<tr>
<td>Brucon_Database</td>
<td>AcProtect_Database</td>
</tr>
</tbody>
</table>
## Shim DLL exports

<table>
<thead>
<tr>
<th>Shim DLL export name</th>
<th>Shim DLL export purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SE_DllLoaded</td>
<td>Callback during DLL load</td>
</tr>
<tr>
<td>SE_DLLUnloaded</td>
<td>Callback during DLL unload</td>
</tr>
<tr>
<td>SE_DynamicShim</td>
<td>Unknown</td>
</tr>
<tr>
<td>SE_GetProcAddress</td>
<td>Callback during GetProcAddress</td>
</tr>
<tr>
<td>SE_InstallAfterInit</td>
<td>Callback after shim complete</td>
</tr>
<tr>
<td>SE_InstallBeforeInit</td>
<td>Callback before shim application</td>
</tr>
<tr>
<td>SE_IsShimDLL</td>
<td>Callback when shimming shim DLL</td>
</tr>
<tr>
<td>SE_Process</td>
<td>Callback when EXE exiting</td>
</tr>
</tbody>
</table>