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

- Todays 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

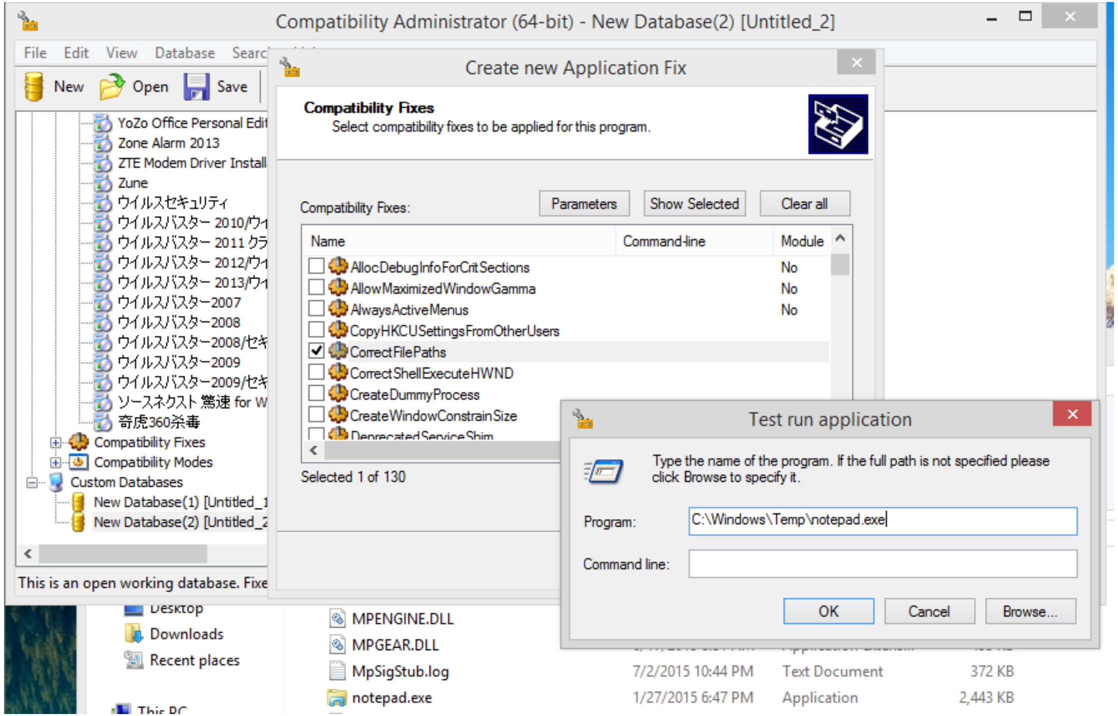
SDB contents

```
<EXE>
  <NAME type='stringref'>OREGON32.EXE</NAME>
  <APP_NAME type='stringref'>The Oregon Trail v1.2</APP_NAME>
  <VENDOR type='stringref'>Minnesota Educational Computing Corp.</VENDOR>
  <EXE_ID type='hex'>568058f1-da4f-4105-8f72-edd5d2a4aaf3</EXE_ID>
     <APP_ID type='hex'>82f31111-af62-4849-b866-14c4e748e33c</APP_ID>
  <MATCH_MODE type='integer'>0x2</MATCH_MODE>
  <MATCHING_FILE>
     <NAME type='stringref'>OREGON32.DLL</NAME>
  </MATCHING_FILE>
  <SHIM_REF>
     <NAME type='stringref'>EmulateGetDiskFreeSpace</NAME>
     <SHIM_TAGID type='integer'>0x23298</SHIM_TAGID>
  </SHIM_REF>
 </EXE>
```

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

Shim Name	Purpose
`DisableWindowsDefender`	*"The fix disables Windows Defender for security applications that do not work with Windows Defender."*
`CorrectFilePaths`	Redirect file system paths
`LoadLibraryRedirectFlag`	Change load directory of DLLs
`NoSignatureCheck`	??? ☺
`RelaunchElevated`	Ensure an EXE runs as admin
`TerminateExe`	??? ☺
`VirtualRegistry`	Registry redirection and expansion

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\`
    - `C:\Windows\AppPatch\Custom\Custom64\`

- 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

```
00000000 (04)   opcode: PATCH_MATCH
0000000c (04)   rva: 0x00053c2e
00000014 (64)   module_name:
u'kernel32.dll'
00000054 (05)   pattern: 9090909090

  disassembly:
    0x53c2e: nop
    0x53c2f: nop
    0x53c30: nop
    0x53c31: nop
    0x53c32: nop
```

```
00000000 (04)   opcode: PATCH_REPLACE
0000000c (04)   rva: 0x00053c2e
00000014 (64)   module_name:
u'kernel32.dll'
00000054 (07)   pattern: e8321a0700ebf9

  disassembly:
    0x53c2e: call 0x000c5665
    0x53c33: jmp 0x00053c29
```

Patch details, II

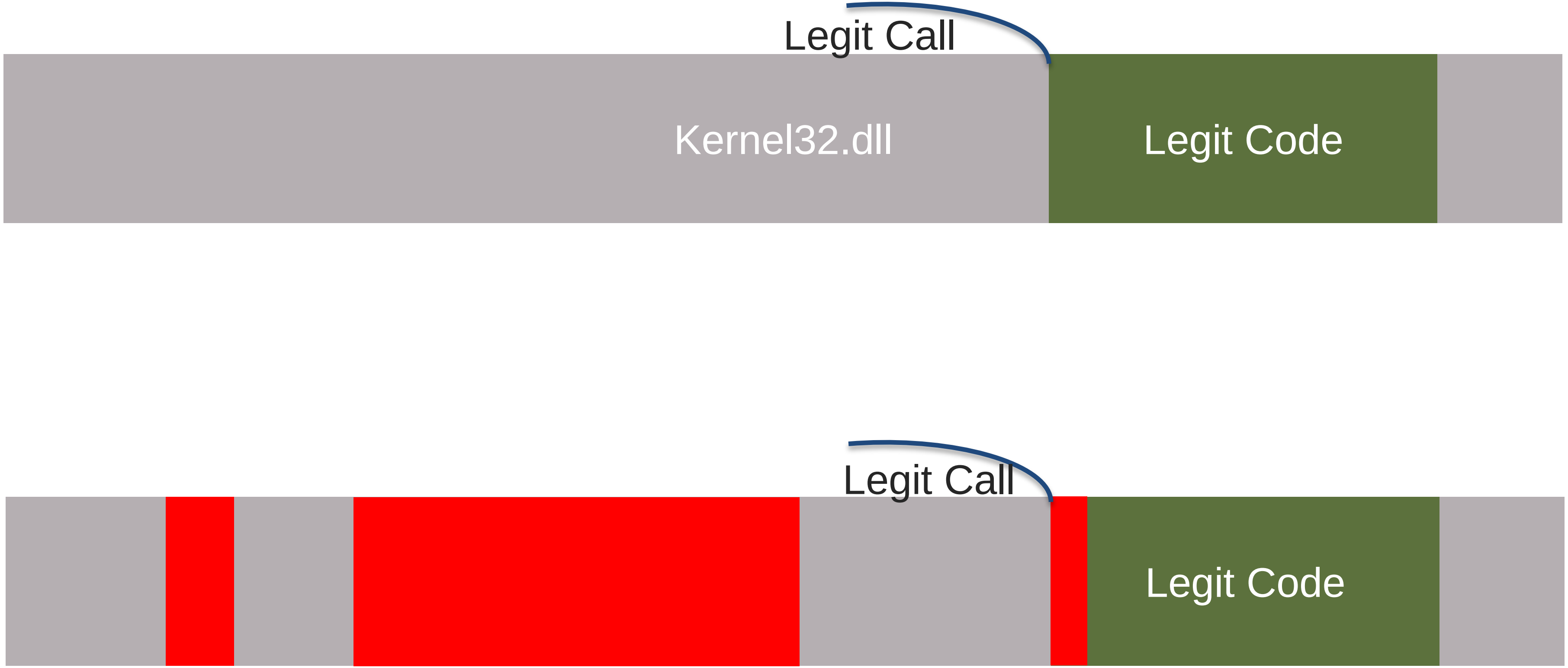
```
00000000 (04)   opcode: PATCH_MATCH
0000000c (04)   rva: 0x000c5665
00000014 (64)   module_name:
u'kernel32.dll'
00000054 (08)   pattern:
0000000000000000
```

```
00000000 (04)   opcode: PATCH_REPLACE
0000000c (04)   rva: 0x000c5665
00000014 (64)   module_name: u'kernel32.dll'
00000054 (14)   pattern:
8304240260 9ce8030000009d61c3
  disassembly:
    0xc5665: add dword [esp],2
    0xc5669: pushad
    0xc566a: pushfd
    0xc566b: call 0x000c566d
    0xc5670: popfd
    0xc5671: popad
    0xc5672: ret
```

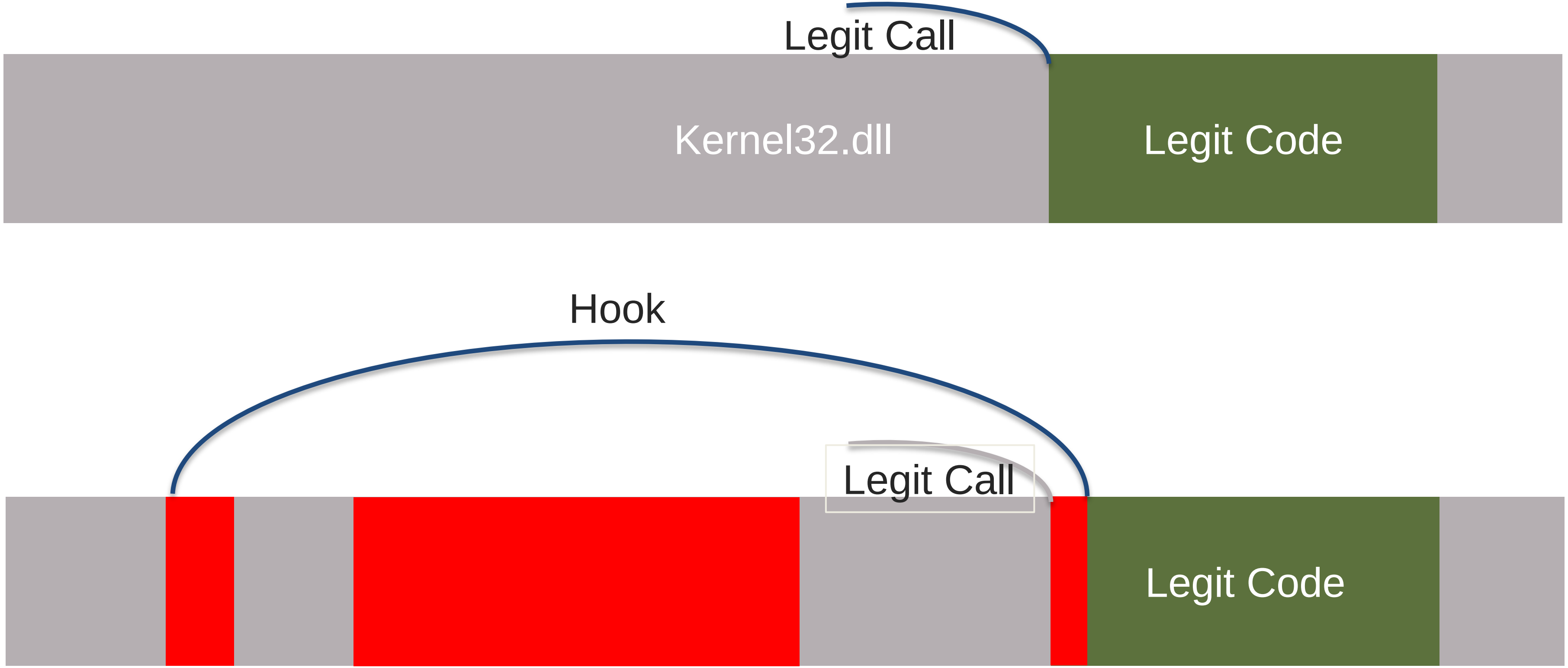
Patch details, III

< Multi-kilobyte shellcode downloader >

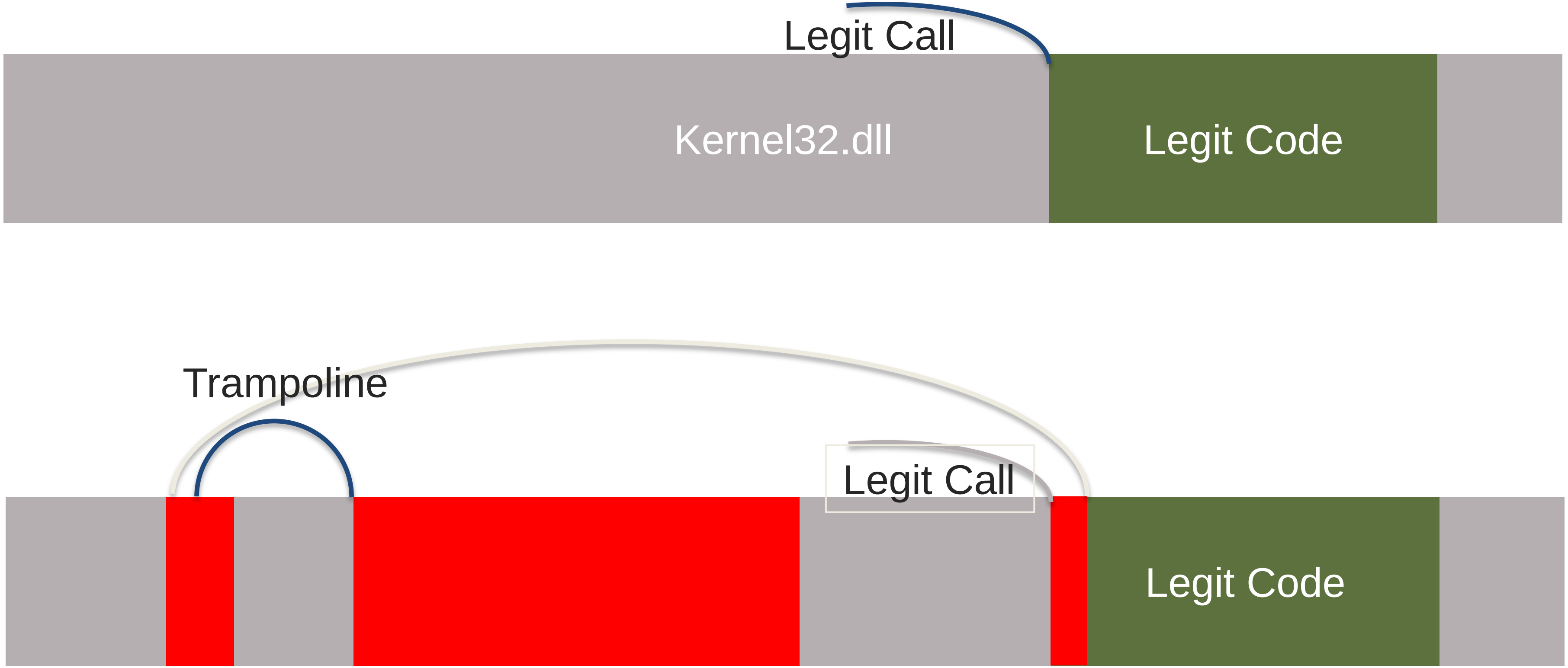
Patch details, summary



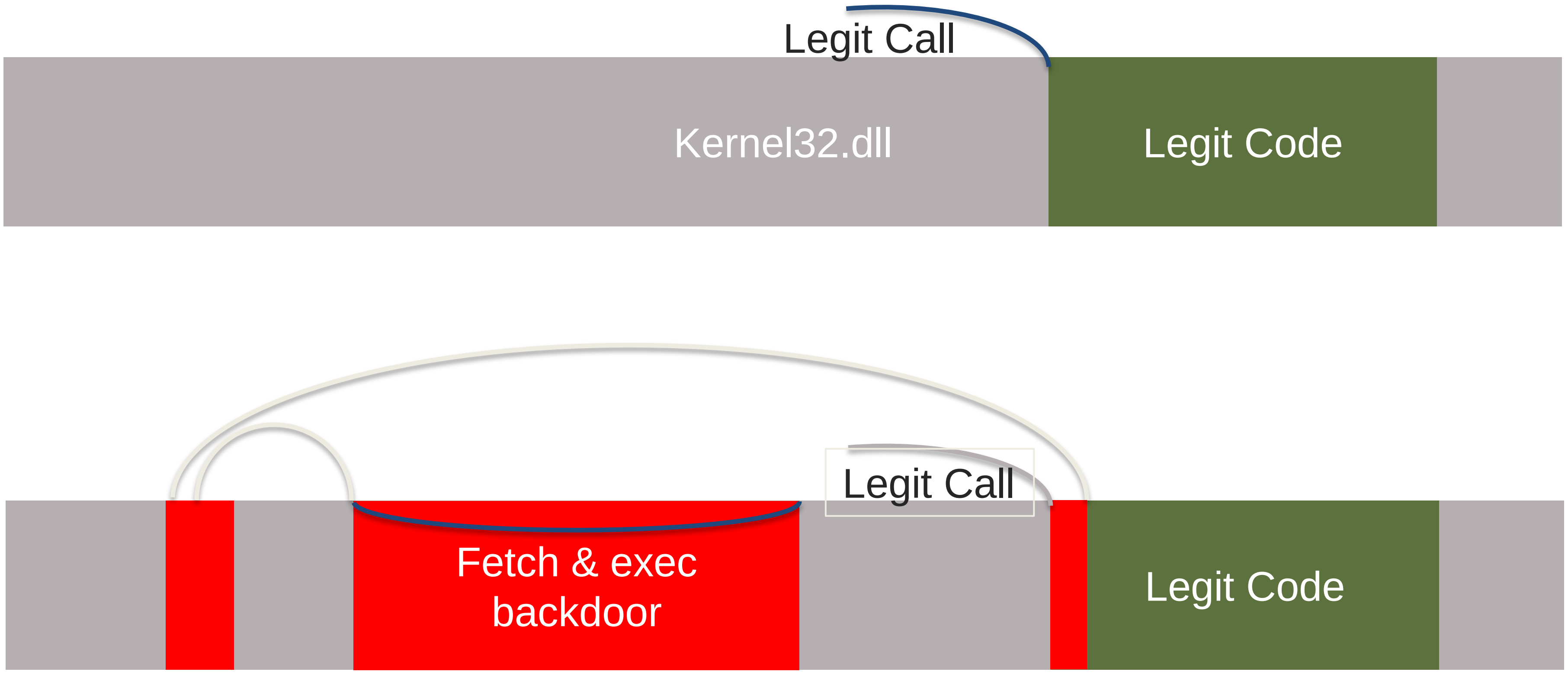
Patch details, summary



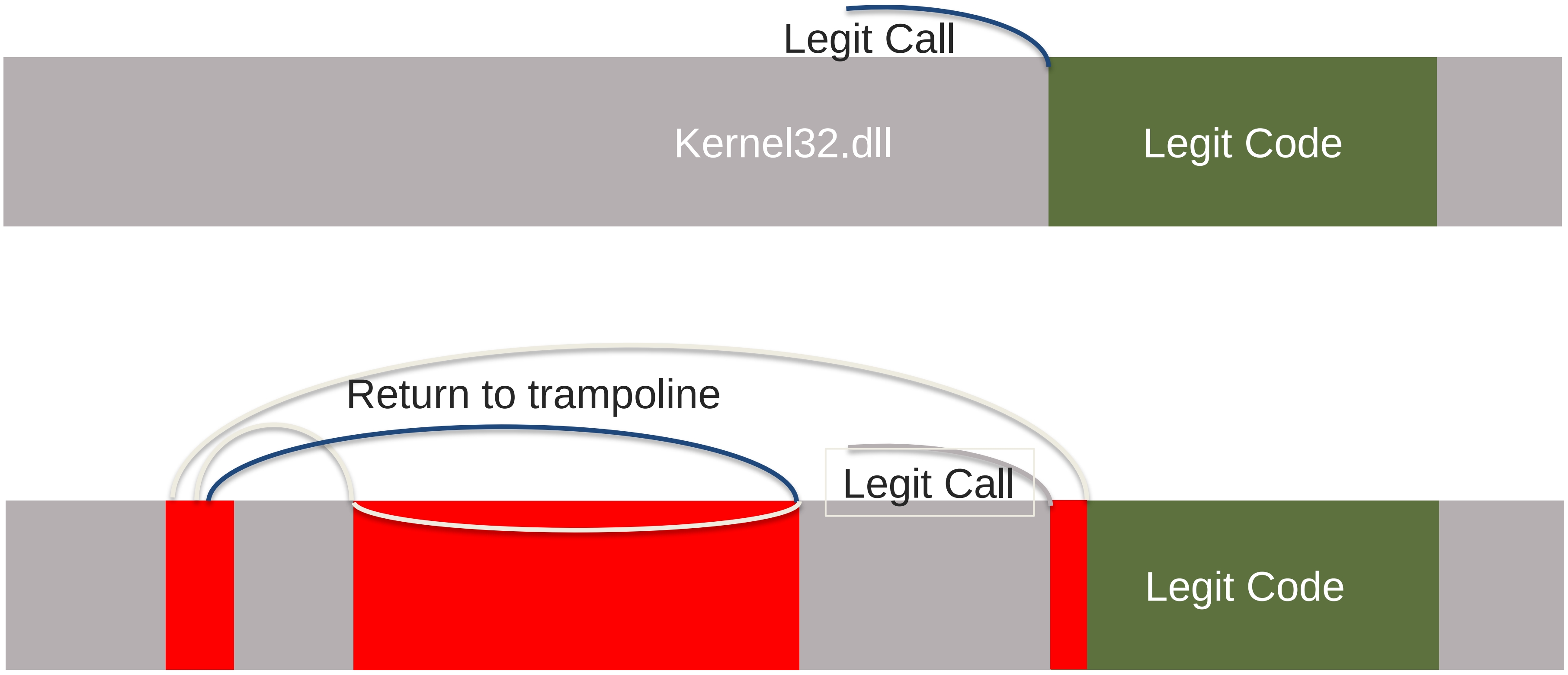
Patch details, summary



Patch details, summary

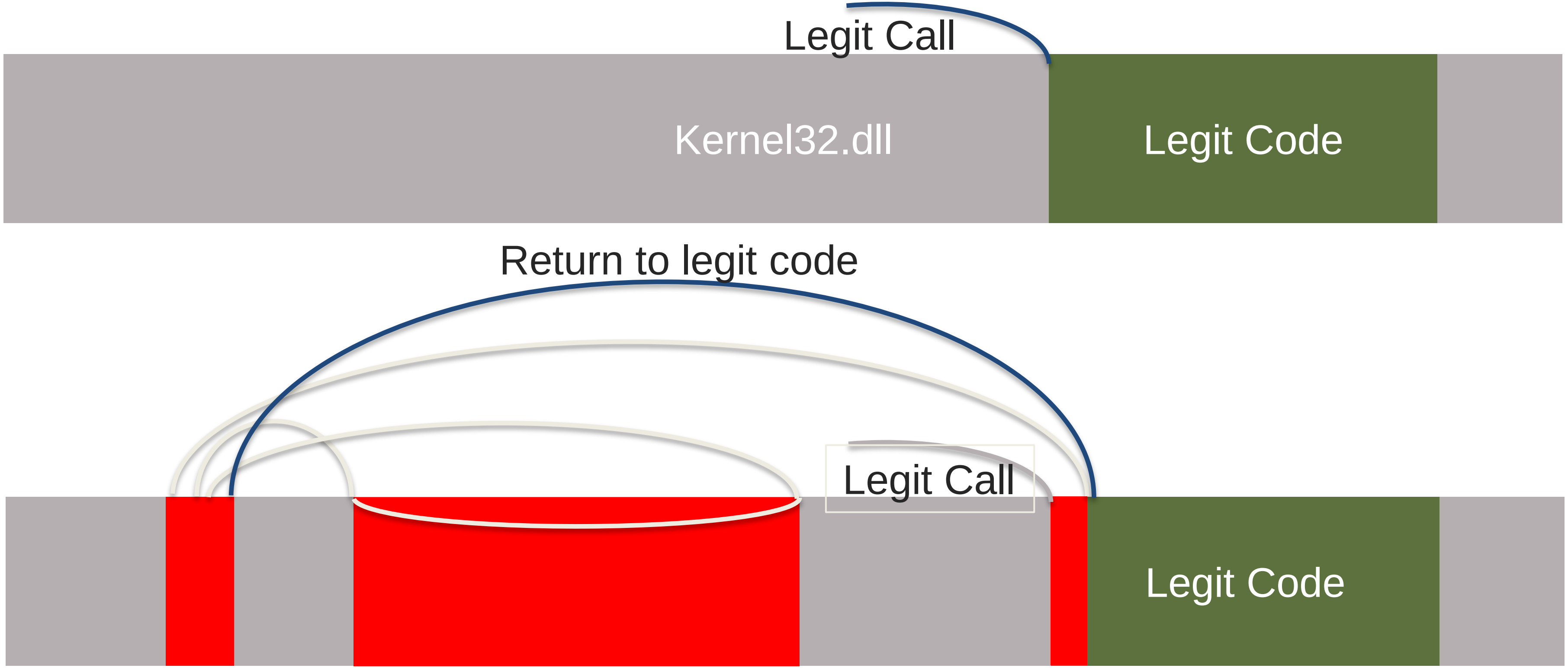


Patch details, summary

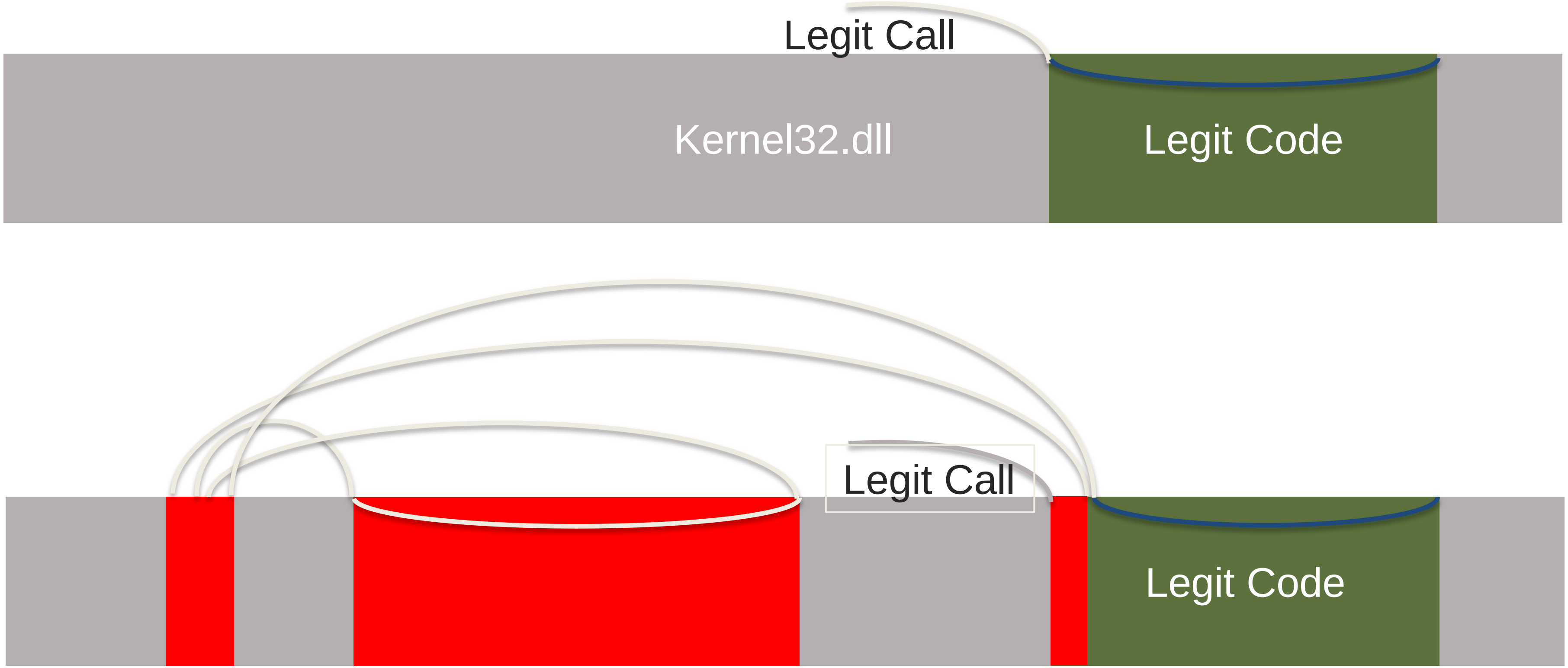


FireEye

Patch details, summary



Patch details, summary



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

SDB file format

- The SDB file format is an undocumented Microsoft format
  - `apphelp.dll` exposes ~254 exports for manipulating shims
  - That doesn't help for forensic analysis!

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 02 00 00 00 01 00 00 00 73 64 62 66 02 78 5A 11   ........sdbf.xZ.
0010h: 02 00 03 78 34 36 01 00 02 38 07 70 03 38 01 60   ...x46...8.p.8.`
0020h: 16 40 01 00 00 00 01 98 20 36 01 00 45 58 45 2E   .@.....˜ 6..EXE.
0030h: 45 54 47 21 E2 9C 04 00 45 2E 31 45 52 53 49 21   ETG!âœ..E.1ERSI!
0040h: 5E 9D 04 00 45 2E 32 45 52 53 49 21 DA 9D 04 00   ^...E.2ERSI!Ú...
0050h: 45 2E 41 45 52 53 49 21 56 9E 04 00 45 2E 42 45   E.AERSI!Vž..E.BE
0060h: 52 53 49 21 D2 9E 04 00 45 58 45 2E 45 54 4B 21   RSI!Òž..EXE.ETK!
0070h: 4E 9F 04 00 4C 4F 41 5F 5F 24 24 24 CA 9F 04 00   NŸ..LOA__$$$ÊŸ..
0080h: 5F 53 43 5F 5F 24 24 24 C6 A0 04 00 2E 30 30 30   _SC__$$$Æ ...000
0090h: 35 39 32 30 C2 A1 04 00 2E 30 30 31 35 39 32 30   5920Â¡...0015920
```

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

```
<LIBRARY> <SHIM>
    <NAME type='stringref'>Brucon_Shim</NAME>
    <DLLFILE type='stringref'>Custom\elogger.dll</DLLFILE>
</SHIM></LIBRARY>
```

- Application definition: target & shim pointer

```
<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 sdbs: `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`) □
  - 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

  https://www.blackhat.com/docs/asia-14/materials/Erickson/Asia-14-Erickson-Persist-It-Using-And-Abusing-Microsofts-Fix-It-Patches.pdf

- "Shim: A new method of injection" (in Russian)

  ftp://os2.fannet.ru/fileechoes/programming/XA_159.PDF

- "Roaming Tiger" - Anton Cherepanov/ESET @ ZeroNights '14

  http://2014.zeronights.org/assets/files/slides/roaming_tiger_zeronights_2014.pdf

- "Windows - Owned By Default!" – Mark Baggett @ DerbyCon 2013
- "Compatibility Fix Descriptions" - MSDN
  https://technet.microsoft.com/en-us/library/cc722305%28v=ws.10%29.aspx

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:

elogger.dat	"Roaming Tiger"
`Brucon_Shim`	`AcProtect_Shim`
`Brucon_Apps`	`AcProtect_Apps`
`Brucon_Database`	`AcProtect_Database`

Shim DLL exports

Shim DLL export name	Shim DLL export purpose
SE_DllLoaded	Callback during DLL load
SE_DLLUnloaded	Callback during DLL unload
SE_DynamicShim	Unknown
SE_GetProcAddress	Callback during GetProcAddress
SE_InstallAfterInit	Callback after shim complete
SE_InstallBeforeInit	Callback before shim application
SE_IsShimDLL	Callback when shimming shim DLL
SE_Process	Callback when EXE exiting