Persistent BIOS Infection

“The early bird catches the worm”

Anibal L. Sacco (Ssr Exploit writer)
Alfredo A. Ortega (Ssr Exploit writer)
Agenda

- Introduction
- A bit of history
- A better choice
- What is the BIOS
- BIOS Structure
- How it works
- Update/flashing process
- A Simple way to patch BIOS
- Where to patch
- What can be done
- Shellcodes
- Virtual machine demo
- Real hardware demo
Introduction

- Practical approach to generic & reliable BIOS code injection
- True Persistency
- Rootkit(ish) behavior
- OS independant
A little bit of history:

Commonly used persistency methods:

- User mode backdoor
- Kernel mode backdoor

How can this be done more effectively?
BIOS Level backdoor:

- Takes control before any other software
- Stealth behavior
- Generally forgotten by almost all Antiviruses
- OS Independent (Runs outside the OS context)
What is the BIOS?

- BIOS stands for Basic Input Output System
- Boot firmware
- Hardware initialization (RAM, North Bridge, etc.)
- Size: 256 Kb and bigger
- Commonly stored on EEPROM or flash memory
BIOS Structure

- It is composed of various LZH compressed modules
- Each module has an 8 bit checksum
- There are some uncompressed modules:
  - Bootblock: In charge of the POST, and emergency boot
  - Decompression routine: decompresses the rest of the modules
- Various checksum checks.
<table>
<thead>
<tr>
<th>Class.Instance (Name)</th>
<th>Packed</th>
<th>Expanded</th>
<th>Compression</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.03 ( BIOSCODE)</td>
<td>06DAF (28079) =&gt; 093F0 (37872)</td>
<td>LZINT (74%)</td>
<td>446DFh</td>
<td></td>
</tr>
<tr>
<td>B.02 ( BIOSCODE)</td>
<td>05B87 (23431) =&gt; 087A4 (34724)</td>
<td>LZINT (67%)</td>
<td>4B4A9h</td>
<td></td>
</tr>
<tr>
<td>B.01 ( BIOSCODE)</td>
<td>05A36 (23094) =&gt; 080E0 (32992)</td>
<td>LZINT (69%)</td>
<td>5104Bh</td>
<td></td>
</tr>
<tr>
<td>C.00 ( UPDATE)</td>
<td>03010 (12304) =&gt; 03010 (12304)</td>
<td>NONE (100%)</td>
<td>5CFDFh</td>
<td></td>
</tr>
<tr>
<td>X.01 ( ROMEXEC)</td>
<td>01110 (4368) =&gt; 01110 (4368)</td>
<td>NONE (100%)</td>
<td>6000Ah</td>
<td></td>
</tr>
<tr>
<td>T.00 ( TEMPLATE)</td>
<td>02476 (9334) =&gt; 055E0 (21984)</td>
<td>LZINT (42%)</td>
<td>63D78h</td>
<td></td>
</tr>
<tr>
<td>S.00 ( STRINGS)</td>
<td>020AC (8364) =&gt; 047EA (18410)</td>
<td>LZINT (45%)</td>
<td>66209h</td>
<td></td>
</tr>
<tr>
<td>E.00 ( SETUP)</td>
<td>03AE6 (15078) =&gt; 09058 (36952)</td>
<td>LZINT (40%)</td>
<td>682D0h</td>
<td></td>
</tr>
<tr>
<td>M.00 ( MISER)</td>
<td>03095 (12437) =&gt; 046D0 (18128)</td>
<td>LZINT (68%)</td>
<td>6BDD1h</td>
<td></td>
</tr>
<tr>
<td>L.01 ( LOGO)</td>
<td>01A23 (6691) =&gt; 246B2 (149170)</td>
<td>LZINT (4%)</td>
<td>6EE81h</td>
<td></td>
</tr>
<tr>
<td>L.00 ( LOGO)</td>
<td>00500 (01280) =&gt; 03752 (14162)</td>
<td>LZINT (9%)</td>
<td>7088Fh</td>
<td></td>
</tr>
<tr>
<td>X.00 ( ROMEXEC)</td>
<td>06A6C (27244) =&gt; 06A6C (27244)</td>
<td>NONE (100%)</td>
<td>70DDAh</td>
<td></td>
</tr>
<tr>
<td>B.00 ( BIOSCODE)</td>
<td>001DD (00477) =&gt; 0D740 (55104)</td>
<td>LZINT (0%)</td>
<td>77862h</td>
<td></td>
</tr>
<tr>
<td><em>.00 ( TCPA_</em>)</td>
<td>00004 (00004) =&gt; 00004 (004)</td>
<td>NONE (100%)</td>
<td>77A5Ah</td>
<td></td>
</tr>
<tr>
<td>D.00 ( DISPLAY)</td>
<td>00AF1 (02801) =&gt; 00FE0 (4064)</td>
<td>LZINT (68%)</td>
<td>77A79h</td>
<td></td>
</tr>
<tr>
<td>G.00 ( DECOMPCODE)</td>
<td>006D6 (01750) =&gt; 006D6 (1750)</td>
<td>NONE (100%)</td>
<td>78585h</td>
<td></td>
</tr>
<tr>
<td>A.01 ( ACPI)</td>
<td>0005B (00091) =&gt; 00074 (116)</td>
<td>LZINT (78%)</td>
<td>78C76h</td>
<td></td>
</tr>
<tr>
<td>A.00 ( ACPI)</td>
<td>012FE (04862) =&gt; 0437C (17276)</td>
<td>LZINT (28%)</td>
<td>78CECh</td>
<td></td>
</tr>
<tr>
<td>B.00 ( BIOSCODE)</td>
<td>00BD0 (03024) =&gt; 00BD0 (3024)</td>
<td>NONE (100%)</td>
<td>7D6AAh</td>
<td></td>
</tr>
</tbody>
</table>
How it works

- The first instruction executed by the CPU is a 16 byte opcode located at F000:FFF0
- The Bootblock POST (Power On Self Test) initialization routine is executed.
- Decompression routine is called and every module is executed.
- Initializes PCI ROMs.
- Loads bootloader from hard-disk and executes it.
Update/flashing process

- BIOS is upgradeable.

- Vendors provide periodic updates to add new features and fix bugs. They also provide their own tools to flash from DOS, Windows, and even from ActiveX!

- BIOS update procedure depends on South-Bridge and chip used.

- CoreBOOT project provides a generic BIOS flashing tool: flashrom, that supports most motherboard/chip combination.
A Simple way to patch BIOS

- BIOS contains several checksums
- Any modification leads to an unbootable system.

We used two techniques:
  1) Use a BIOS building tool (Pinczakko's method)
  2) Patch and compensate the 8-bit checksum

Three easy steps:
  1) Dump BIOS using flashrom
  2) Patch and compensate
  3) Re-flash
Where to patch

- Anywhere is valid:
  - f000:fff0: First instruction executed.
  - INT 0x19: Executed before booting
  - Insert a ROM module: Executing during POST

- The most practical place: Decompressor
  - It's uncompressed!
  - Located easily by pattern matching
  - Almost never change
  - Called multiple times during boot
What can be done

- Depends. What resources are available from BIOS?
  - Standardized Hard Disk access (Int 13h)
  - Memory Manager (PMM)
  - Network access (PXE, Julien Vanegue technique)
  - Modem and other hardware (Needs a driver)

- Our choice was to modify hard-disk content:
  1) Modify shadow file on unix
  2) Code injection on windows binaries
Shellcodes

- Shellcodes are all in 16 bit
- We use BIOS services for everything
- Easy to debug: BIOS execution environment can be emulated running the code as a COM file over DOS
- Pseudocode:
  1) Checks ready-signal
  2) Checks for services initialization
  3) Runs
Decompression Routine:

- pushad
- push cx
- push es
- push esi
- ...
- ...
- ...

Jmp +0xnnn

Hook

PADDING:
...

[Start of our injected code]
Check for int 0x10 availability

Execute the code
Smashed by the hook

Execute main Shellcode
...
...

No

Yes
Virtual machine demo

- Virtual machines also have a BIOS!
  - In VMWARE, it's embedded as a section of the main VM process, shared on all Vms.
  - Also can be specified on the VMX file for each VM.
  - Is a phoenix BIOS.
  - Very easy to develop because of the **embedded** GDB server.
  - Using Interrupt Vector Table as ready-signal

- Two attacks:
  - OpenBSD shadow file
  - Windows code injection

- This method will infect multiple virtual machines.
Real hardware demo

- We infected an Phoenix-Award BIOS
- Extensively used BIOS
- Using the VGA ROM signature as ready-signal.
- No debug allowed here, all was done by Reverse-Engineering and later, Int 10h (Not even printf!)
- Injector tool is a 100-line python script!
Future research

- Virtualized Rootkit
- PCI device placement (Modems, VGA, Ethernet and RAID controllers)
- The ultimate BIOS rootkit...

Thank you for your attention!