These Boots are Made for Hacking:
An Introduction to Secure Boot

Ira Ray Jenkins
How did we get here?
Bootstrapping

1. RAM controller initialized
2. BIOS decompressed into shadow RAM
3. Shadow RAM write-protected
4. Peripherals enumerated + POST
5. Boot sector loaded
6. Boot candidates scanned for signature
7. Potential boot devices evaluated
8. Expansion ROM hooks installed
9. Boot sector loads stage 2
10. Stage 2 loads kernel and RAMdisk
11. Kernel re-enumerates hardware
12. RAM disk mounted
13. Device driver modules loaded
14. Filesystem mounted (flash, hard disk, ...)
15. Device driver modules loaded
16. Run startup scripts and user apps
What is “secure boot”? 

- The validation of cryptographically signed kernels prior to execution by the bootloader.
- Conceptualized in the 1980s & early '90s
- Implemented in 1997, known as AEGIS.

Figure 3. AEGIS boot control flow

Source: Arbaugh, Farber, and Smith
Why care about secure boot?
Firmwares are under siege.

Nearly 50 exploits published since 2004!
Thunderstrike

• POC bootkit created by Trammel Hudson in 2014.
• Given CVE-2014-4498:
  – “The CPU Software in Apple OS X before 10.10.2 allows physically proximate attackers to modify firmware during the EFI update process by inserting a Thunderbolt device with crafted code in an Option ROM…”

• How bad is it?
  – Complete system control from boot
    • Backdoors, logging, export of encryption keys, etc...
  – Persistent
    • Hides in non-volatile flash as part of the firmware
  – Viral
    • Infected Thunderbolt → Macbook → Thunderbolt
    • Rinse & Repeat
• POC SMM Malware developed by Xeno Kovah & Corey Kallenberg in 2015.
• Asks what is possible using BIOS-level malware?
  – Answer: Any & every thing.
• Scenario 1: Running “live”, non-persistent Oss, like Tails:
  – Advantage: Tails runs independently from host-machine OS.
  – Problem: Host-machine OS exploit leads to Tails exploit.
  – POC: Export PGP keys used for encrypted communications.
• Scenario 2: Serial-over-lan management
  – Advantage: Remotely control servers, etc…
  – Problem: BIOS has access to virtual serial ports.
  – POC: Remote command and control.
• Automated malware infection analyses:
  – Obtained BIOS firmwares from several vendors:
    • Lenovo, HP, Dell, Acer, & Asus
  – 3161 firmware images, 12 exploit misses
Extensible Firmware Interface

“All problems in computer science can be solved by another level of indirection… but that will usually create another problem.” – David Wheeler

• Created by Intel in 1998.
• Features:
  – Modular design
  – CPU-independent architecture & drivers
  – Flexible pre-OS environment with network capability
Unified EFI

- UEFI Forum started in 2005.
- Introduced “secure boot” in 2010.
- Version 2.5 released in 2015
  - 2637 pages long
Microsoft Controversy

• In 2011, Microsoft announced Windows 8 certified systems would ship with secure boot enabled.
  – Required OEMs to include Microsoft keys on new systems.
• Free Software Foundation and the open-source community reacted in opposition.
  – Vendor lock-in, DRM, etc…
• Firmware was required to provide a mechanism for disablement or reconfiguration.
  – Required third-party bootloaders & OSs to be signed by Microsoft.
• As of Windows 10, OEMS no longer required to provide these mechanisms.
UEFI Secure Boot

• Standard specifies:
  – API for cryptographically protected UEFI “variables”
  – How to validate signed binaries
  – Procedures to revoke bad certificates & hashes
  – Requires non-volatile flash that can be made read-only
UEFI Variables

- **Platform Key**
  - Used to authorize changes to KEKs
  - Owned by OEM, e.g., Lenovo
- **Key Exchange Key (KEK)**
  - Used to authorize changes to DB & DBX
  - Owned by OS vendor, e.g., Microsoft
- **DB**
  - “whitelist” of certificates, signatures, and hashes
- **DBX**
  - “blacklist” of certificates, signatures, and hashes
Thinking like a hacker…

- What if the keys become corrupted or erased?
  - “A Tale of One Software Bypass of Windows 8 Secure Boot”
  - “Setup for Failure”

- How does firmware or hardware stop me from writing my own data?
  - “The Empire Strikes Back Apple – how your Mac firmware security is completely broken”
• Allow signed UEFI firmware updates only
• Protect UEFI firmware in SPI flash from direct modification
• Protect firmware update components (inside SMM or DXE on reboot)
• Program SPI controller and flash descriptor securely
• Protect SecureBootEnable/CustomMode/PK/KEK/db(x) in NVRAM
• Implement VariableAuthenticated in SMM and physical presence checks
• Protect SetVariable runtime API
• Securely disable Compatibility Support Module (CSM), unsigned legacy Option ROMs and MBR boot loaders
• Configure secure image verification policies (no ALLOW_EXECUTE)
• Build platform firmware using latest UEFI/EDK sources
• Correctly implement signature verification and crypto functionality
• **And don’t introduce a single bug in all of this...**
Conclusions

• Secure boot is a simple idea:
  – Don’t run code you don’t trust

• Complicated in practice:
  – Who do you trust? Microsoft, sys admins, yourself?
  – Implementation is hard

“Layers of abstraction become boundaries of competence.” – Sergey Bratus
References

- https://tomicarter.files.wordpress.com/2012/08/conradbootstraps-1.jpg
- http://frameshaped.net/scratch/wp-content/uploads/bootstrap01-1.jpg
- http://www.clker.com/cliparts/3/3/B/g/Z/9/mono-power-button-md.png
- http://f.tqn.com/y/cplus/1/W/N/L/-/-/Amiga_1200_Kickstart_3.0_ROMs.jpg
- Thunderstrike. https://trmn.net/Thunderstrike
- Google trends. https://www.google.com/trends/explore#q=biosmalware,bootkit,firmwaremalware