MalwareTech SBK (Superpersistent Bootkit)

A Firmware Assisted Bootkit

Introduction

Coming up with the idea

- A few years ago, one of my friends was working on a piece of malware which resided in the BIOS firmware. I thought this was a cool idea, but was very impractical due to various constraints such as the small size of the BIOS ROM and the fact it's usually write protected.
- I decided to look into writing firmware based malware to target something more viable such as the GPU or Hard Disk, but after a few days of research, I came to the conclusion that I didn't have the skills required to do either.
- In late 2013 I saw a post on spritesmods.com about hard disk hacking using a JTAG, which inspired me to continue with my research.
- In the end I decided to focus on hard disk firmware, because I simply didn't have the funds to buy a bunch of modern GPUs and the potentially brick / short circuit them all.

Goals

- Find a way to modify the hard disk firmware without physical access to the disk.
- Create malware that is capable of surviving a full system reinstall, even if the entire disk is erased in the process.
- Come up with a way to conceal the infection and its components from the infected system and forensic tools.
- Make the malware completely impossible to remove on a software level, even from kernel mode or using a live CD.

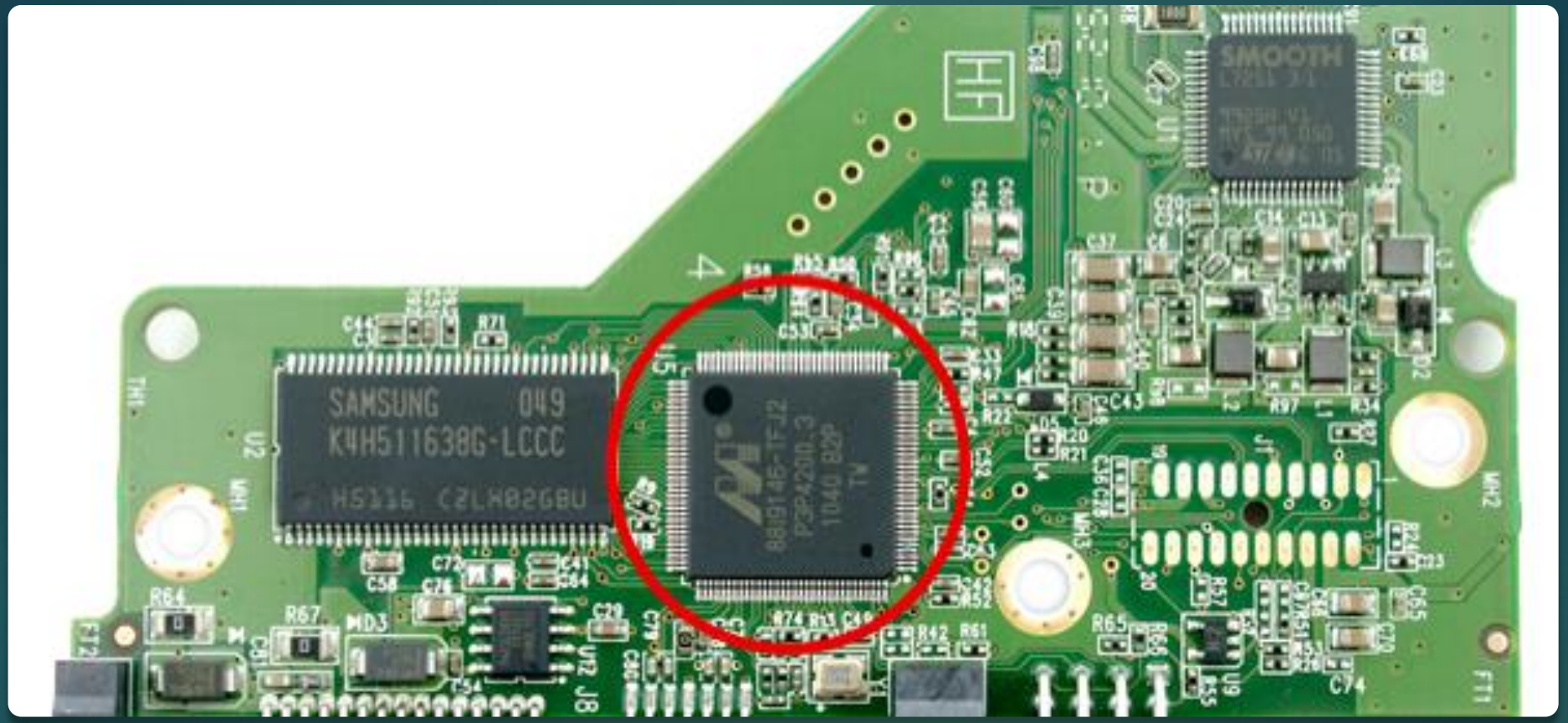
What is firmware?

- Almost all electronic devices contain a small computer that is specifically designed for said device (known as embedded systems).
- Your computer is made up of embedded systems, such as the hard disk and GPU.
- Firmware is the software used to control an embedded system, it's basically a tiny operating system.
- The firmware is generally stored in a small bit of flash memory on the circuit board.
- A computer simply sends ATA requests on the SATA port, the hard disk's firmware is responsible for the actual processing of the request.

Why firmware?

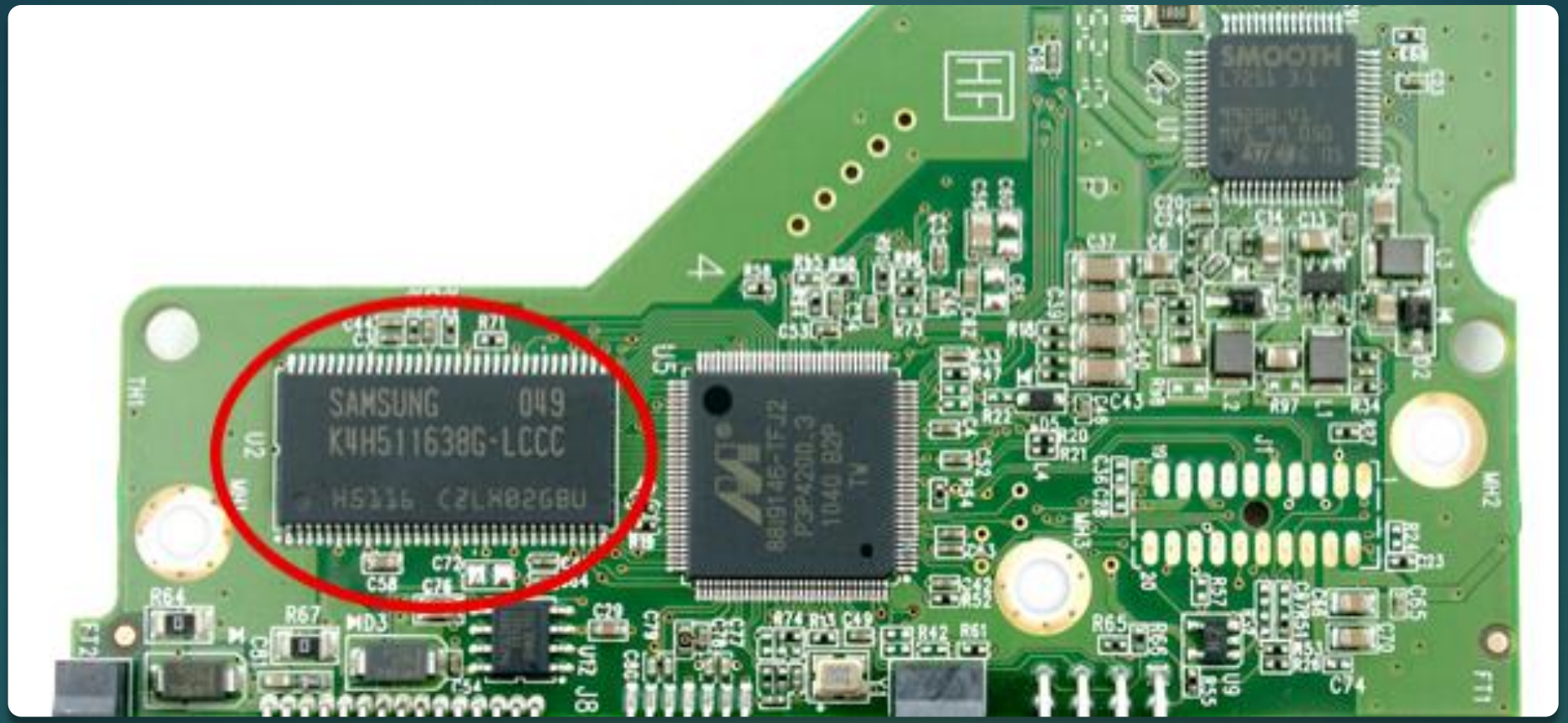
- The firmware allows for total control of the infected device, exceeding what's possible with a kernel rootkit.
- Device firmware is almost never modified and is likely to outlast many operating system upgrades and reinstalls.
- During forensic analysis most of the focus is towards what's on the disk, not what's hiding inside it.

Hard Disk PCB



Marvell 88i ARM Microcontroller

- Multi-core 32-bit ARM CPU
- Around 1 MB internal RAM (used for data and code)
- 64 KB internal ROM (stores MCU / PCB specific start up code)

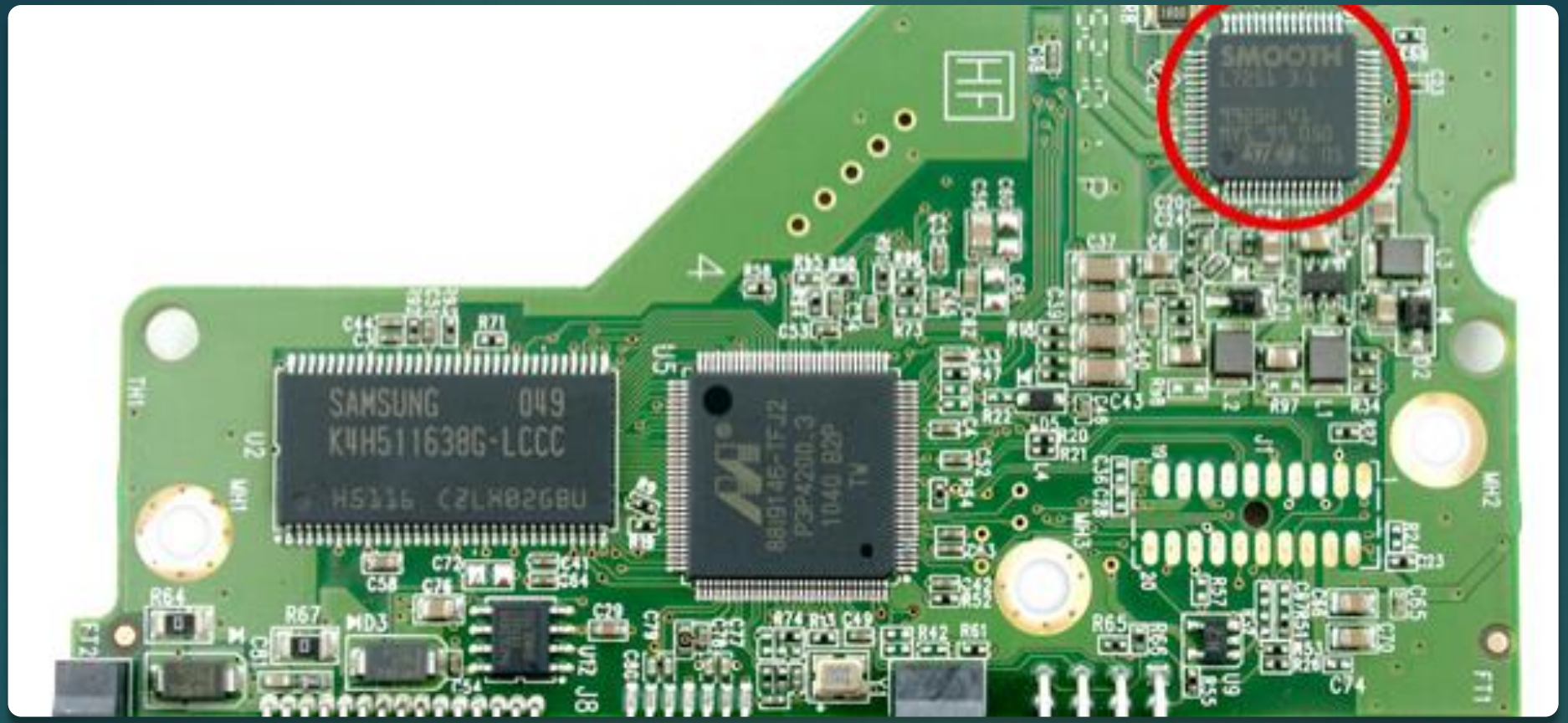


Samsung DDR400 SDRAM

- Used for DMA when reading/writing sectors and as the hard disk's cache
- Accessible from firmware
- Seems to be 64 MB even if the cache is 8 MB (Some space can be used for code)

Generic EEPROM Chip

- Usually around 192 KB in size
- Contains firmware kernel and loader
- Readable and writable



Smooth L7251

- Controls spindle motor
- Not really useful for what I'm doing

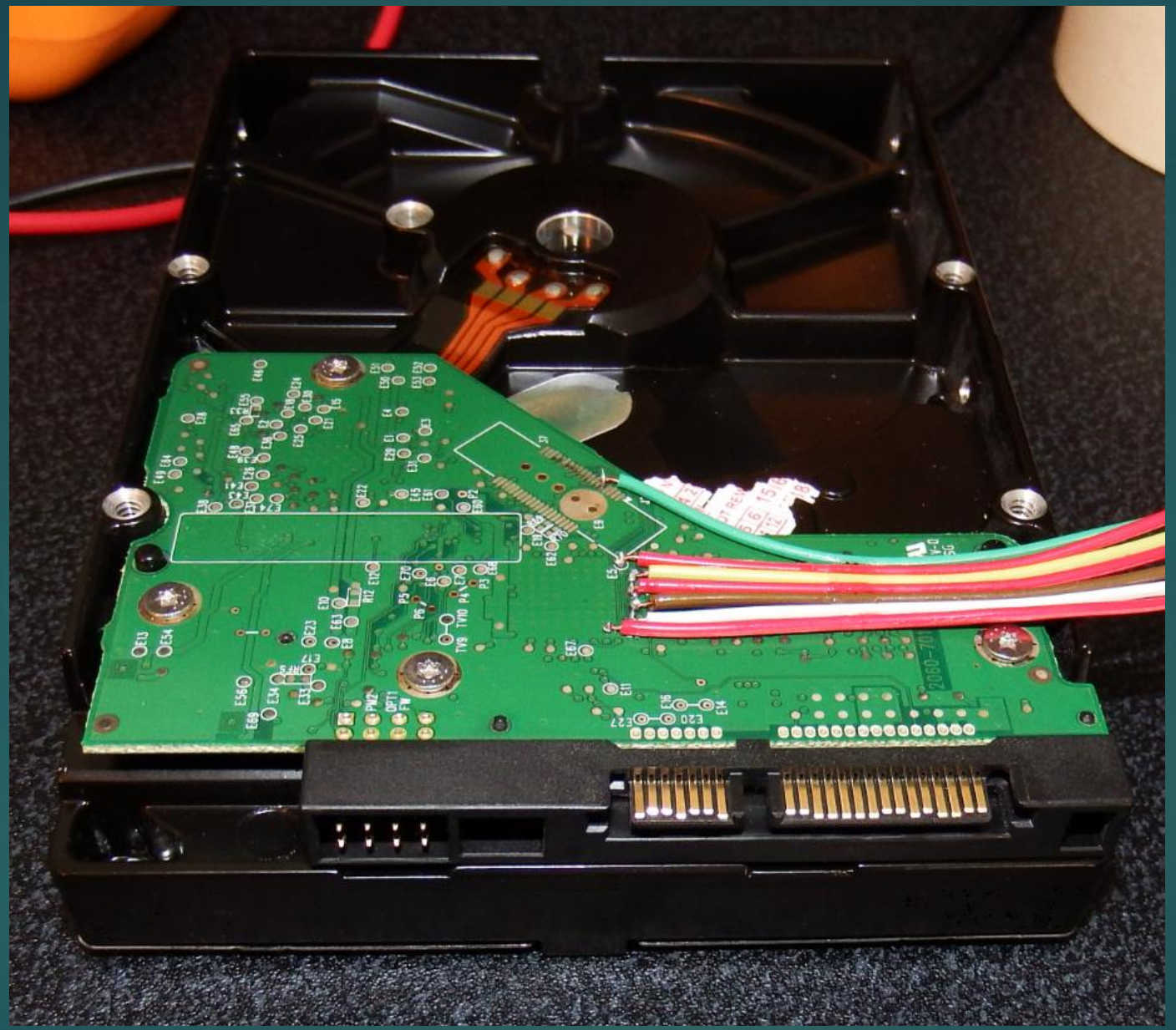
The Setup

Requirements

- A way to dump the disk firmware so I could reverse engineer it.
- The ability to use the disk outside of my computer and avoid any damage to its components if the disk short circuits.
- Something to let me debug the disk controller so I could test and modify code.
- A method of hard-flashing the firmware, should I brick the drive.
- A debugger which supports debugging ARM code via GDB.
- Way too much free time.

Solution

- JTAG for debugging & reading / writing the disk controller's memory.
- SPI programmer for dumping the firmware from the flash chip as well as hard-flashing.
- ATA to USB adapter for using the disk out side of the computer.
- AC to 12v DC Molex power supply for powering the disk.
- OpenOCD for driving the JTAG.
- IDA Pro for graphical debugging via OpenOCD.



JTAG and SPI wires are soldered directly to the PCB

I’m using a TIAO USB multi-protocol adapter for JTAG / SPI

Research

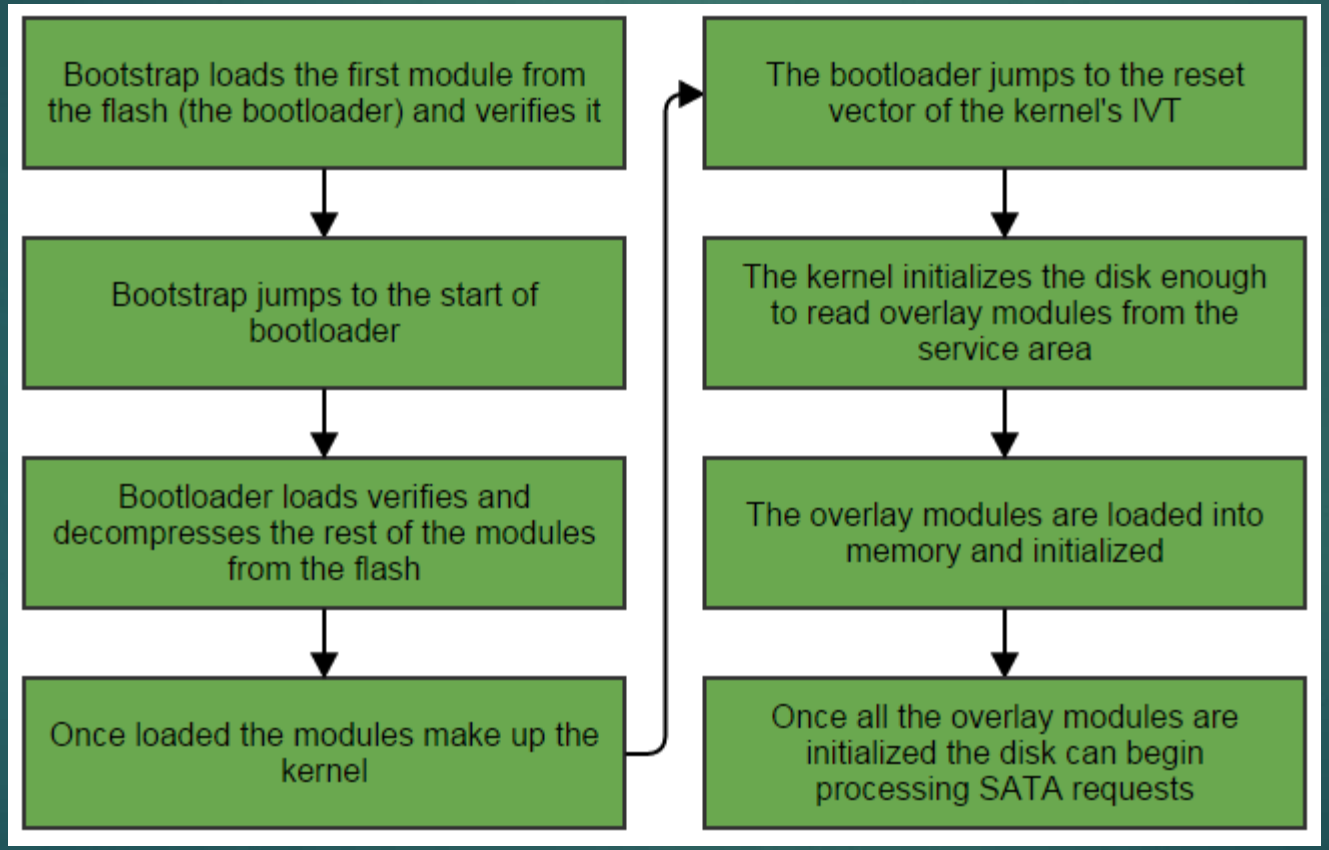
Research

- Week 1: Googling for anything that would help during the development phase, not much was found.
- Week 2 – 5: Finding my way around the hardware, Learning ARM basics, looking for ways to read/write flash, and mapping out memory.
- Week 5: Reversing the firmware load process.
- Week 6 - 8: Looking for the code responsible for handling disk read/write requests and getting an idea of how it works.
- Week 8 - 10: Finding out at which point the sector read is completed and where the sectors reside in the cache.

Setbacks

- Soldering wires to test points is much harder than it looks.
- Finding where in the cache sectors are read to took forever, this was due to the fact the address is calculated as an offset and uses several unknown variables.
- Before I was able to figure out how to hard-flash the firmware, several disks were bricked by failed attempts at writing the flash programmatically.
- Bits of the firmware are in read only memory which means software breakpoints cannot be used.
- Only 1 hardware breakpoint unit, no step-over functionality, and no watchpoints made debugging very time consuming.

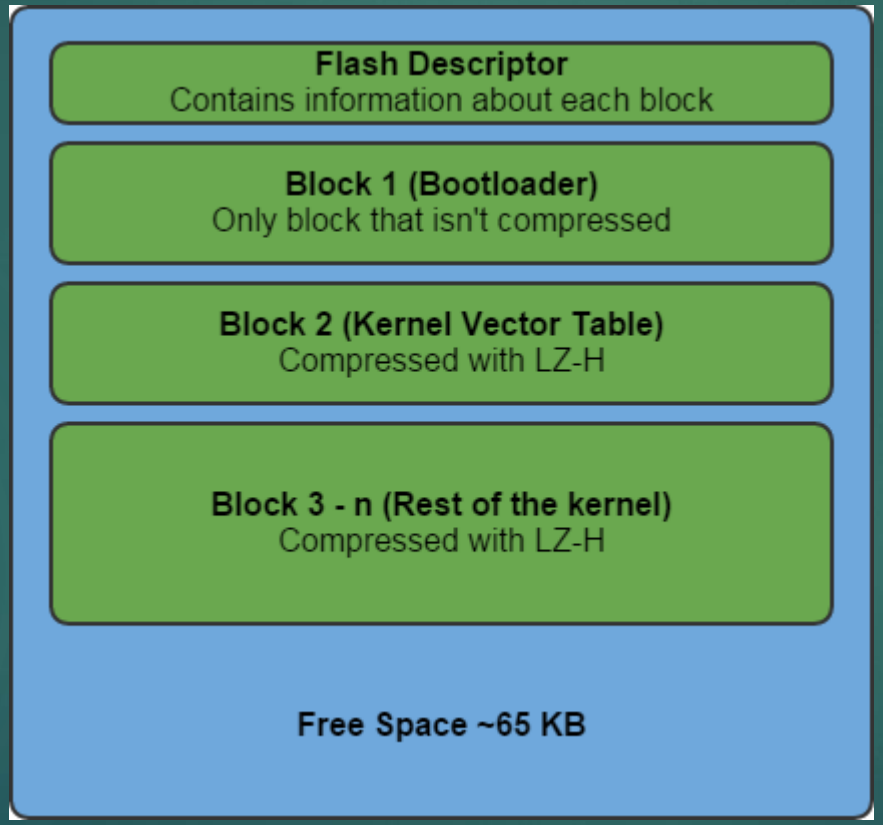
Hard disk boot process



Hard disk boot process

- Bootstrap code is embedded into the microcontroller and is the only part of the firmware which can't be changed.
- The bootloader and kernel modules are verified by performing a checksum-8
- All of the kernel modules are Lempel-Ziv compressed and encoded using Huffman, I'll call this LZ-H.
- Kernel modules are either data or code.
- Some kernel modules supply an entry point which is called by the bootloader, others do not.
- The first kernel module is a standard ARM vector table, which is loaded at address 0.

Flash format



Vendor specific commands

- The firmware implements a lot of custom commands which allows vendor software running on the computer to interface with the firmware.
- These commands are not documented and require a bit of reverse engineering to find out.
- Can be intercepted just like normal ATA commands.

Service area

- An area of the disk which is reserved for firmware use, this is why a disk will usually be a few GB smaller than its stated size.
- Stores various firmware data as well as overlay modules, which provide the main functionality for the disk. This means that the flash only has to store a small amount of code which can read the rest of the firmware from the service area.
- Usually spans the first million or so sectors of the disk. In my case sector 0 is actually sector 0x300000.
- Cannot be read using normal requests, only by the firmware directly or via vendor specific commands.

Overlay modules

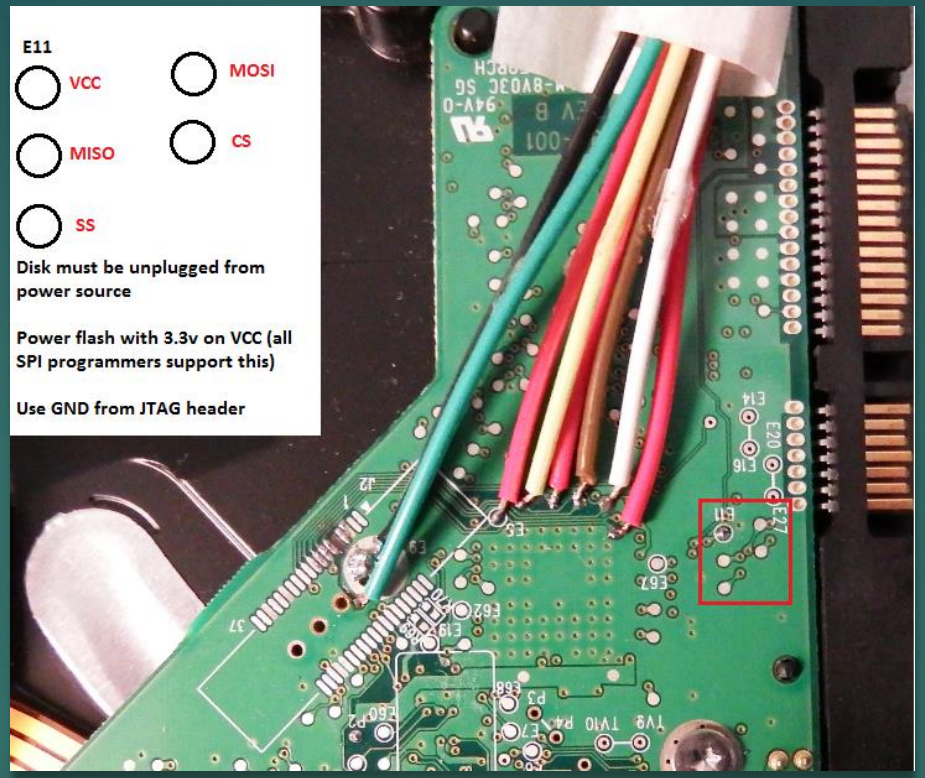
- Firmware modules that are stored in the service area of the hard disk.
- Referred to as overlay modules because due to limited memory capacity, the firmware will sometimes overwrite one overlay module with another in order to perform a different task.
- The code responsible for processing SATA requests is in one of the modules.
- Last parts of the firmware to be loaded.

Firmware flashing (Software)

- In order to make hard disk firmware easily updatable, vendors implemented a way for the firmware to flash itself.
- Vendors write software that uses vendor specific commands to flash the firmware from within the operating system.
- Reverse engineering vendor's software reveals the commands required to read/write the firmware.
- Vendor specific commands can be sent from user mode, as long as the sender had admin rights (this includes firmware flashing commands).

Firmware flashing (Hardware)

- Because software flashing requires the firmware to be functional, disks need a way to be flashed if the firmware is bricked (known as hard-flashing).
- Hard-flashing is performed by connecting an SPI programmer to the hard disk and directly interfacing the flash chip.
- It does not take more than about 10 seconds to perform a hard-flash and the disk does not need to be powered up.
- It would be possible to create a portable flasher to quickly infect hard drives, given physical access.



In my case the connection point for the SPI programmer is a cluster of 5 metal contact points on the bottom of the drive.

Firmware infection problems

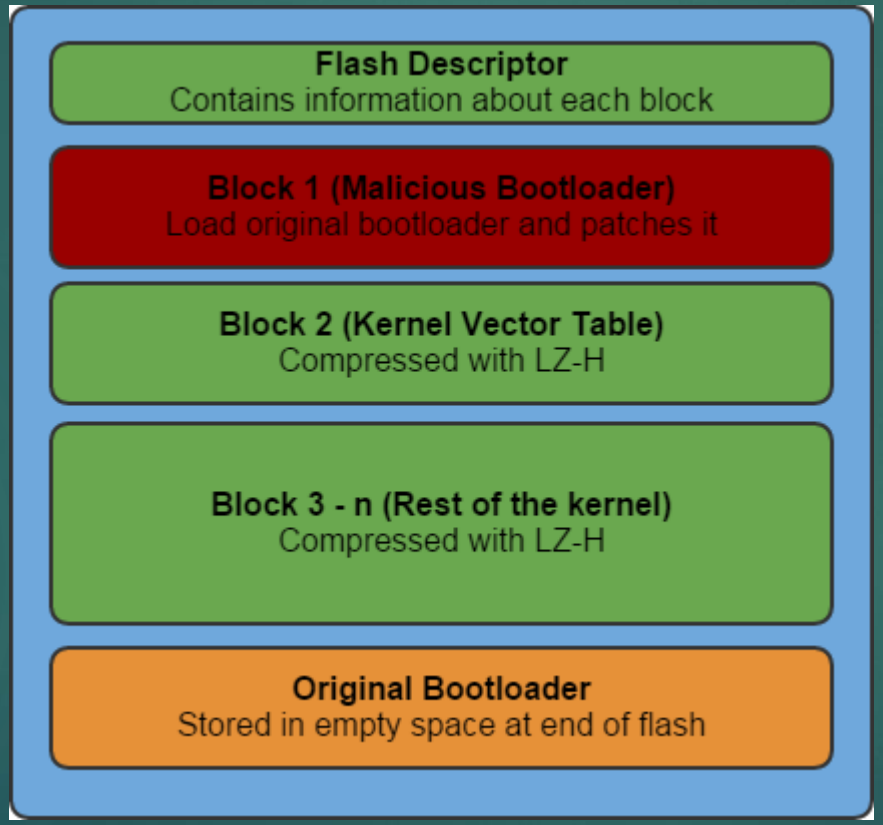
- Most firmware based rootkits can simply flash the firmware with a malicious version of the code and be done, but this isn't my case.
- The code I needed to target resides in modules loaded from the service area by the kernel. Although I could infect these modules, they would be detectable by disk repair utilities.
- The kernel is compressed with an algorithm that I've tried to replicate but failed miserably, so it can't be modified.

Firmware Bootkit

Bootkits

- Most of life's problems can be solved with bootkits, this is no exception.
- Firmware malware can adopt conventional bootkit techniques such as interrupt hooking and bootloader patching.
- There is enough free space in the flash to store the entire bootkit as well as the payload.
- On infection the firmware’s bootloader is relocated to the end of the flash and replaced with a malicious bootloader (stage 1).

Infected flash format

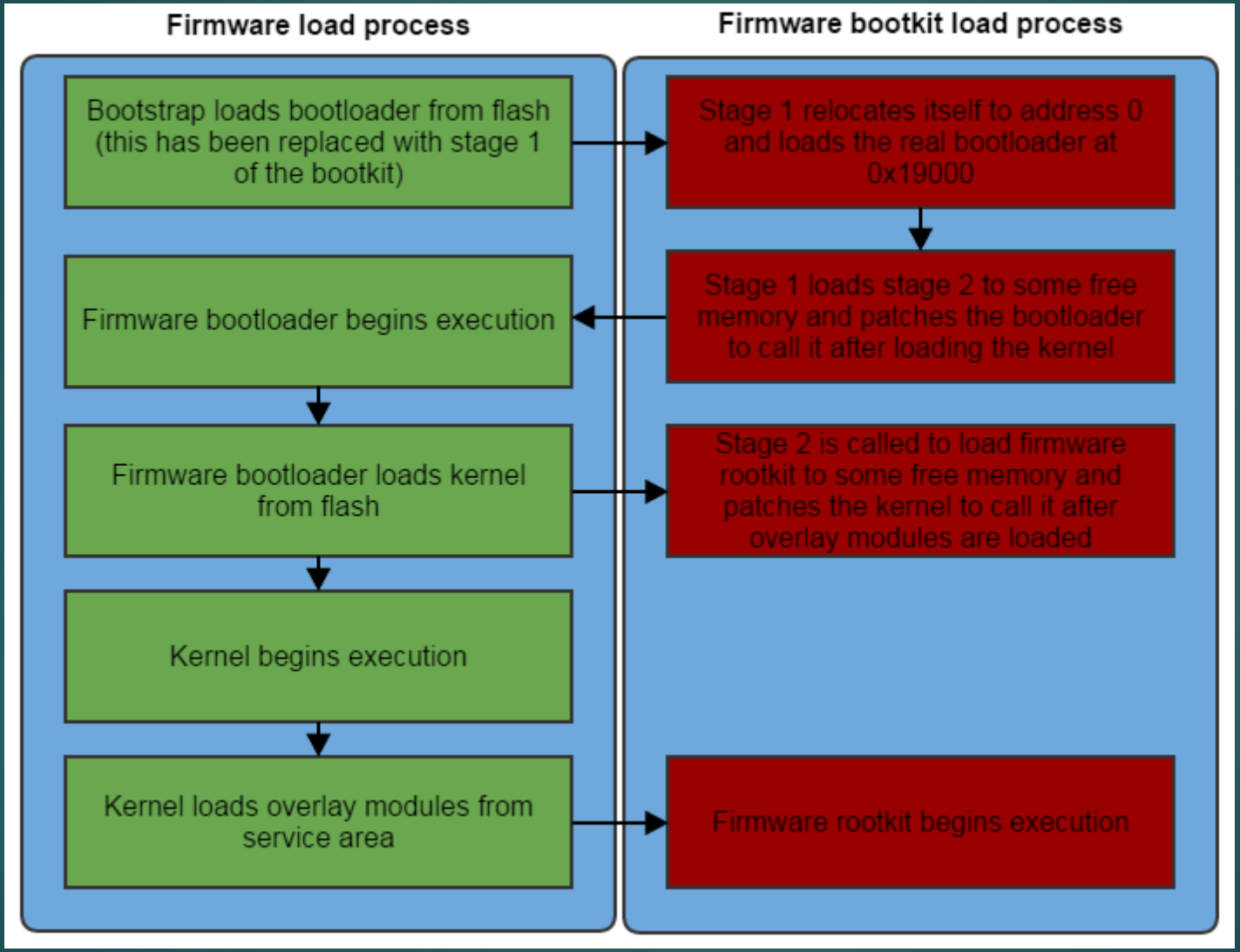


Bootkit – Stage 1

- Begins executing at address 0x19000 which is where the original bootloader should be.
- Relocates itself to address 0.
- Places stage 2 to an address which won't be overwritten by the kernel.
- Reads the original bootloader from flash and loads it at 0x19000.
- Patches the bootloader so stage 2 will be called right after the kernel is loaded.
- Jumps to the original bootloader.

Bootkit – Stage 2

- Loads the firmware rootkit to some free memory which won't be overwritten by the overlay modules.
- Patches some kernel code which is executed
- Transfers execution back to the kernel.


Firmware load process
Firmware bootkit load process
Bootstrap loads bootloader from flash (this has been replaced with stage 1 of the bootkit)
Stage 1 relocates itself to address 0 and loads the real bootloader at 0x19000
Stage 1 loads stage 2 to some free memory and patches the bootloader to call it after loading the kernel
Firmware bootloader begins execution
Firmware bootloader loads kernel from flash
Stage 2 is called to load firmware rootkit to some free memory and patches the kernel to call it after overlay modules are loaded
Kernel begins execution
Kernel loads overlay modules from service area
Firmware rootkit begins execution

Firmware Rootkit

Firmware rootkit

- The core of the malware.
- Capable of hooking kernel and overlay modules to intercept any I/O to and from the hard disk or its firmware.
- Only a few KB in size but still extremely powerful.
- Can read and write the hard disk service area.
- Can’t be detected by the operating system or antiviruses.
- Written entirely in ARM Assembly.

Sector spoofing

- The primary purpose of the firmware rootkit is to allow spoofing of disk sectors at firmware level.
- When the firmware receives a requests to read some sectors, it tells the read/write head to read them into the cache, then notifies the SATA port when the read is complete.
- The rootkit can overwrite sectors in the cache before they are sent to the host computer.
- By modifying sectors that are read from the disk, it's possible to run code on the host.
- But which sector(s) should be modified?

MBR spoofing

- The MBR is always logical block address 0, making it the perfect target.
- Firmware rootkit can simply intercept the MBR read request and replace it with a bootkit's MBR.
- Using a second bootkit is great because:
  - Saves the malware having to look for sectors that are part of an executable.
  - No need to worry about the modified sector(s) being part of a digitally signed executable.
  - Allows privileged access to the operating system.
  - Fairly easy to modify any bootkit to work with the firmware rootkit.
  - If your bootkit doesn't have a bootkit, you're probably doing it wrong anyway.

TinyXPB

- For those of you who are familiar with my previous work, you'll probably remember in 2014 I released TinyXPB, a simple Windows XP bootkit.
- On my blog I mentioned that TinyXPB was merely a payload for a much bigger project, this is the project I was referring to.
- I wanted to code something that would allow TinyXPB to survive an operating system reinstall or disk reformat.
- The firmware rootkit can load TinyXPB by spoofing the MBR as well as spoofing the first few sectors of the disk to contain TinyXPB's filesystem.
- The capabilities of the project are not limited to XP, but I decided to just go with that because it's quick and easy to install, and the entire bootkit is small enough to fit inside the firmware.

Persistence

- The code is stored in the firmware and not on the disk platters, meaning the malware can survive an operating system reinstall as well as the entire disk being erased.
- Although the firmware can be re-flashed using vendor specific commands, the firmware rootkit can intercept these and block modifications attempts.
- The only way to disinfect the disk would be to buy a new PCB or hard-flash the firmware using an SPI programmer (this is beyond most normal users).

Stealth

- Sectors are only ever modified in the cache, leaving no trace of modification.
- All of the malicious code resides in the firmware and not on the disk, so the firmware rootkit would not show up during forensic analysis.
- The first time the MBR gets read is during boot, after this the firmware can stop spoofing and let the real MBR be read, preventing detection while the system is running.
- Firmware read attempts using vendor specific commands can also be blocked, preventing any vendor utilities from dumping the infected firmware.

Possible Features

Hidden Filesystem

- Hard disk's service area usually has between 100 and 400 MB of free space, which is much better than the 65 KB in the flash.
- The firmware rootkit could use this area to store large components that exceed the size of the flash.
- Would also be possible to implement new vendor specific commands, allowing code running inside the operating system to access this area.
- Because the service area isn't readable by normal ATA commands, none of the data would show up during forensic analysis.
- Disk repair utilities can be blocked from accessing the hidden filesystem using the normal vendor specific commands.

Firmware Spoofing

- Instead of blocking reading/writing of the firmware using vendor specific commands, the rootkit could intercept them.
- Most hard disks store a backup copy of the firmware in the service area, on read attempts the rootkit could return this instead of the infected firmware.
- On write attempts the rootkit could infect new firmware on the fly, allowing the disk to be updated whilst maintaining persistence.

Weaknesses

- It's impossible to stop hard-flashing the firmware, because this is done by directly interfacing the flash chip with an SPI programmer.
- If the disk was plugged into an already booted system, it would be possible to observe the MBR being spoofed (though this may be preventable using some kind of heuristics).
- The hard disk doesn't lose power during a warm reboot; If the firmware rootkit was set to stop spoofing the MBR after the first read, the MBR bootkit would not be loaded upon restart.
- Secure boot defeats bootkits and thus defeats this; however, the malware could simply fall back to file infection or similar.
- The malware is only designed for a single (albeit popular) make of hard disk, as I don't have the resources to write separate code for all the major disk manufactures (this would be viable as most manufacturers firmware doesn't differ greatly between disk models).

Conclusion

This is usually the part where I'd post a proof of concept, but for obvious reasons it wouldn't be wise to do so. Instead you can just take this as an example of what one person can do, as well as an insight into what a state-backed agency with billions of dollar in funding could achieve, if one of those existed.

I'll probably upload a demo video of the malware surviving an operating system reinstall, but first I need to fix a few bugs which causes it to not work all the time. For now, here is a demonstration of how the MBR can be spoofed from the firmware by modifying sectors in the cache: https://www.youtube.com/watch?v=0gc-VF6bi3g

I hope you enjoyed reading, if you'd like to send feedback you can reach me on admin@malwaretech.com

Twitter: @MalwareTechBlog

Related Resources

- https://spritesmods.com/?art=hddhack
- http://www.s3.eurecom.fr/docs/acsac13_zaddach.pdf
- http://forum.hddguru.com/viewtopic.php?t=20324
- http://www.malwaretech.com/2015/04/hard-disk-firmware-hacking-part-1.html
- http://www.recover.co.il/SA-cover/SA-cover.pdf
- http://forum.hddguru.com/viewtopic.php?f=13&t=25552
- http://forum.hddguru.com/viewtopic.php?t=20346&f=1&start=320#p183693