# TinyXPB  (Windows XP Bootkit)

Written by MalwareTech, Yes that’s my real name.

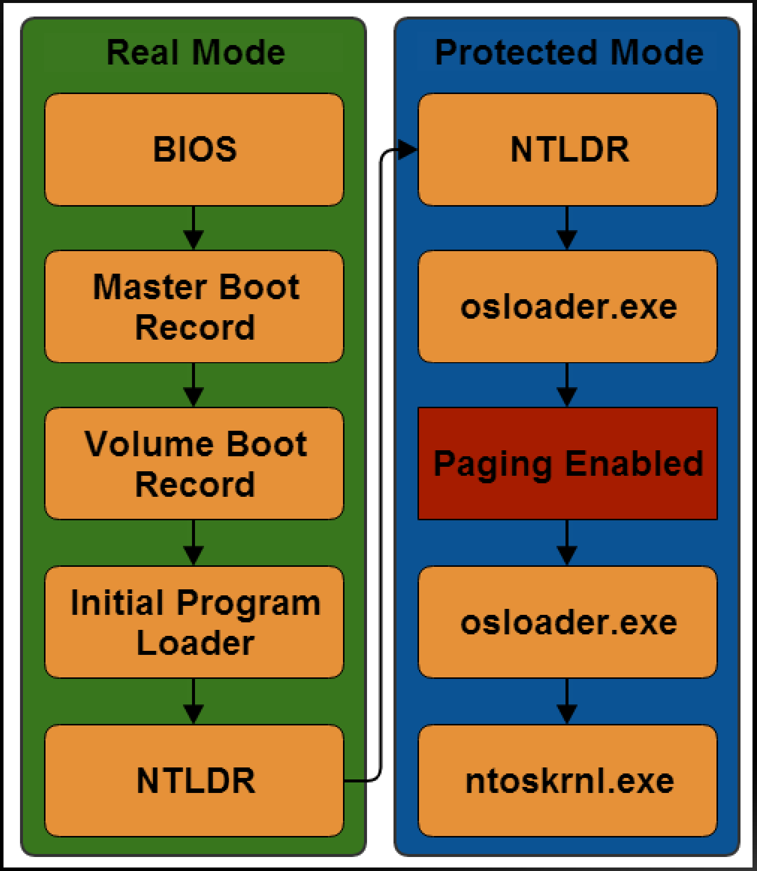
# Index

- Note: There are more slides than referenced in the index, use normal navigation.

## About

- TinyXPB is a 32-bit windows XP bootkit designed as a payload for another project. Although this bootkit could be programmed to work on Vista, 7, 8 (x86 & x64); I have limited it to 32-bit XP for simplicity and legal reasons.
- The bootkit can be booted from a floppy drive and will not modify any files on the disk, allowing it to be tested on real systems without risk of data loss.
- The purpose of a bootkit is to begin execution before windows is loaded, this is achieved by using a malicious MBR to hijack the boot process.
- When the MBR is executed, the CPU protection rings are not yet used; This means all code is run in ring 0 (full privileges), including ours.
- Any antiviruses are loaded very late in the boot process, which gives us lots of time to do what we want.
- The bootkit is written in a mix of 16-bit & 32-bit ASM and compiled with FASM, the driver is C and compiled with Visual Studio.

## Windows XP Boot Process



- Master Boot Record (MBR) is the first sector of the boot device and is 1 sector in size.
- Volume Boot Record (VBR) is the first sector of the NTFS partition and is 1 sector in size.
- MBR and VBR can exist on same disk, MBR is sector 0 of disk and VBR is sector 0 of partition.
- Initial Program Loader (IPL) is stored directly after the Volume Boot Record and is up to 15 sectors in size.
- The IPL + VBR take up the first 16 sectors of the NTFS partition and are referred to as $BOOT.
- NTLDR and ntoskrnl.exe are normal files on the file system.
- Osloader.exe is embedded in NTLDR.

## BIOS (Basic Input / Output System)

- BIOS code exists in an EEPROM (Electronically Erased Programmable Read Only Memory) chip on the motherboard.
- Most PCs do something known as “shadowing” where they copy and run the BIOS code from the RAM (at address 0x000F0000), RAM is faster than ROM so it speeds up boot.
- Reads the MBR from the first sector of the boot devices into address 0x7C00.
- Provides a collection of low level functions, known as BIOS Interrupts, which are accessible to real mode code via the “int” instruction.

## (MBR) Master Boot Record

- The MBR is the absolute first sector of the boot device and is 1 sector (512 bytes) in size, this code is loaded at 0x7C00 by the BIOS and then executed in real-mode.
- The most part the MBR is code, however; Inside the MBR is a table (the actual master boot record), this table consists of 4 x 16 byte entries and begins at offsets 0x1BE into the MBR.
- Once executed the code will look through the partition table, locate the active partition, and then read the first sector of the partition (VBR) into 0x7C00 then execute it.
- Because the MBR reads the VBR into 0x7C00, it will have relocated itself before-hand to avoid overwriting itself.
- The last 2 bytes of the MBR are the boot signature (0x55, 0xAA).

## (VBR) Volume Boot Record

- The VBR is the first sector of the bootable partition, like the MBR, it is 1 sector (512 bytes) in size. It is loaded and executed at address 0x7C00 (in real-mode) by the MBR.
- The first two bytes of the VBR is a jump instruction that jumps over the Bios Parameter Block (BPB) and into the main code.
- Directly after the first 2 bytes of the VBR is the BPB, this block contains some information about the driver and partition (Location of the master file table, Cylinder / Heads / Sectors setup of drive, Volume serial number, etc.).
- The VBR doesn't do much other than gather some information from the BPB, then reads the first 16 sectors of the partition into memory (usually starting at address 0xD000).
- The first 16 sectors of the partition are known as $BOOT, although all 16 are loaded into memory, only 7 are used (1 for VBR and 6 for the IPL).

## (IPL) Initial Program Loader

- The IPL can be up to 15 sectors (7680 bytes) in size and is located directly after the VBR on disk (sectors 1 -  15 of the partition), this code runs from address 0xD000 and onwards in real-mode.
- The Windows XP IPL only uses 6 of the 15 allocated sectors.
- It is the job of the IPL to locate NTDLR on the disk then read it into memory.
- The IPL will always read and execute NTLDR at address 0x20000.

## (NTLDR) New Technology Loader

- NTLDR is an “actual” file that resides at C:\NTLDR and has attributes FILE_ATTRIBUTE_HIDDEN and FILE_ATTRIBUTE_SYSTEM (The MBR, VBR and IPL aren’t accessible from the file system), it is executed at address 0x20000.
- NTLDR composes of 2 main parts:
  - NTLDR - a mixture of 16-bit real-mode and protected-mode code which switches between the two modes in order to use BIOS interrupts.
  - OSLOADER.exe - A PE file (compiled as a driver) which consists of 32-bit code and executes in protected-mode.
- NTLDR will setup the GDT and IDT then enter protected-mode (paging still not setup).
- OSLOADER.exe is extracted from within NTLDR and loaded with its entry point at 0x401000, this is because OSLOADER is a PE file and if located at 0x401000 it does not require a relocation table.
- NTLDR calls 0x401000 (OSLOADER Entry point).

## OSLOADER.EXE

- OSLOADER is a PE file which was compiled as a kernel driver, however it is used somewhat like a flat-file as opposed to a driver as the kernel isn’t even loaded yet.
- Although OSLOADER is PE file, the PE header isn’t loaded into memory, only the code.
- The code in OSLOADER does quite a lot so I’ll only highlight the main things:
    - Enables paging and sets up the page-table.
    - Uses NTDETECT to detect if the hardware required to run windows is present.
    - Parses boot.ini for boot settings and in the case of dual boot, ask the user which OS they wish to load.
    - Load boot drivers.
    - Locate ntoskrnl.exe and load it into memory before calling KiSystemStartup (ntoskrnl entry point).

## NTOSKRNL.EXE

- Ntoskrnl is a PE file located at C:\Windows\System32\ntoskrnl.exe, it will be loaded by OSLOADER.EXE (the load address changes between service packs).
- The entry point of ntoskrnl is KiSystemStartup which is responsible for Initializing the kernel and starting the OS.
- KiSystemStartup will also reinitialize and execute the boot drivers loaded by OSLOADER.EXE
- After the kernel is done initializing, the OS will be ready to log-in.

## The Bootkit

- The bootkit is packed into a fat-13 floppy image and consists of 4 files.
    - MBR – The floppy MBR is responsible for loading the other 3 components from the image, into memory.
    - Loader16.bin – The 16-bit real-mode component of the bootkit (flat-file).
    - Loader32.bin – The 32-bit protected-mode component of the bootkit (flat-file).
    - Driver32.sys – The 32-bit kernel driver which is loaded by the bootkit (PE file).

- Loader16 and Loader32 are written in ASM using flat-assembler.
- Driver32 is written in C using Visual Studio 2012.

# Bootkit MBR

- The bootkit MBR will originally be loaded and executed at 0x7C00, however; as soon as it is executed it will copy itself to address 0x80000 and execute from there.
- Once at 0x8000 the MBR will parse the floppy's 12-bit File Allocation Table (FAT) looking for LOADER16BIN, LOADER32BIN, and DRIVER32SYS.
- The MBR will use the INT 0x13, AH=2 (Standard Disk Read) BIOS interrupt to read files from the disk into memory. We can't use the Extended Read on floppy disks, so the MBR will also convert Logical Block Addresses to Cylinder-Head-Sector values.
  - Loader16 will be loaded at address 0x80200
  - Loader32 will be loaded at address 0x80400
  - Driver32 will be loaded at address 0x81000
- The driver is loaded at a 64kb aligned address which allows it to be up to 64kb in size, to support a driver bigger: the boot-loader would have to be modified to navigate segment boundaries.
- Once the MBR has finished loading the bootkit components, it will transfer execution to 0x80200 (Loader16).

## Loader16

- Loader16 will execute from address 0x80200 and will execute entirely in real-mode.
- The code will be responsible for hooking INT 0x13 (Disk read interrupt) and processing calls, as well as hooking code in NTLDR and the entry point of OSLOADER.exe.
- We use INT 0x13, AH = 42, (Extended Read) to perform disk reads, because all modern BIOS firmware supports extended read operations, this shouldn't be an issue.
- Until now, all code has been run from our floppy disk. Loader16 will read the real windows MBR from the hard drive into 0x7C00 (like the BIOS would), hook INT 0x13, then execute the windows MBR.

## Loader16 – INT 0x13 Hook

- INT 0x13 is the disk service interrupt provided by the BIOS. The MBR, VBR, IPL, and NTLDR will use it to read sectors from the disk (among other things).
- A BIOS interrupt (INT 0x??) is a special call instruction, when executed: the CPU will push the EFLAGS register onto the stack then perform a far call ptr.
    - A 16-bit far call ptr means the CPU will take the segment:offset value pointed to by IVT entry 0x?? and call it.
    - The IVT (Interrupt Vector Table) is simply a table of 4 byte entries, 2-bytes for segment, 2-bytes for offset.
    - INT 0x13 means IVT entry number 0x13 (19), because each IVT entry is 4 bytes, the 19th entry will be at address 0x4C (76) AKA (19 * 4).
    - If 0x4C held the segment:offset value 0000:7C00 INT 0x13 would be equivalent to pushfd; call 0x7C000; (You may need to read up on real-mode segmentation).
- By storing the original segment:offset pair for INT 0x13, then replacing it with a segment:offset pair that points to our code, we can intercept all calls to the disk service and use the original segment:offset to process the calls.

## Loader16 – INT 0x13 Hook

- We will hook INT 0x13 (Disk Service) and check the AH register (this contains the function ID), if we find the ID 0x02 (Disk Read) or 0x42 (Extended Disk Read) we will intercept it, if not we will call the original INT 0x13 and let it process the call.
- On intercepted disk read calls we will store some of the info used in the call (namely the address which the disk sectors will be read to and the number of sectors to read). We will then call the original INT 0x13 to read the sectors, then we will scan all the sectors that were read, before giving control back to the code which called INT 0x13.
- By hooking the disk interrupt, we can scan every sector read by the MBR, VBR, IPL or NTLDR.
- The bytes we are scanning for are FC F3 67 66 A5 66 8B 4E 0C 66 83 E1 03 which are the op-codes for the following code:
  **cld**
  **rep movsw**
  **mov cx, [esi+0x0C]**
  **and cx, 0x03**
- This code is used by NTLDR to load OSLOADER into memory, when the IPL uses INT 0x13 to read NTLDR into memory, the signature will be found and a hook places after "**rep movsw**" (right after OSLOADER is loaded to 0x401000).

## Loader16 – INT 0x13 Hook

- When the NTLDR code used to move OSLOADER.exe to 0x401000 is called, the CPU is in protected-mode but the code segment is 16-bit. This means that we can't use far jumps for our hook, because in protected-mode a far jump changes the segment selector whereas in real-mode it combines the segment with the offset to perform a 24-bit jump.
- As we can't use segments to perform far jumps and a 16-bit jump isn't big enough to get from 0x2???? (Address in NTLDR we want to hook) to 0x80600 (Our Loader32), we have to use a hack.
- As it happens there is a 32-bit code segment in the GDT (selector 0x08, Base address 0x00000000), which means if we do a far call with segment selector 0x08, we will switch the CPU into 32-bit mode, but because this is a 16-bit jump we can only jump to a 16-bit offset from 0x00000000 (32-bit code segment).
- Luckily address 0x600 is free, so we can place a 32-bit relative jump at that address, then once we perform the far jump, we will be at address 0x600 and the CPU will be in 32-bit mode (It can execute the 32-bit jump at 0x600 and we can make it to our code).
- The jump at 0x600 will point to 0x80400 (Loader32), this means Loader16 is no longer needed and INT 0x13 will be unhook (protected-mode code doesn't use BIOS interrupts).

## Loader32

- Loader32 will be executed by special hook we placed in NTLDR right after it movs OSLOADER is moved to 0x40100.
- Now that OSLOADER is at it's base address, we can remove our hook in NTLDR, then scan OSLOADER for a place to put a new hook).
- We will scan OSLOADER for the following bytes: 08 51 6A 01 52 50 6A 05 E8 which are op-codes for these instructions:
  **push ecx**
  **push 0x01**
  **push edx**
  **push eax**
  **push 0x05**
  **call**
- The "call" op-code is followed by the address of BlAllocateDescriptor, a function in OSLOADER that is used to allocate memory, we will hook this call so that when the loader tries to call BlAllocateDescriptor, it will call our code instead.
- We can now manipulate the return address placed on the stack by the far call that called Loader32, in order to return to NTLDR and allow it to resume execution.

## Loader32 – BlAllocateDescriptor Hook

- Once NTLDR has finished and OSLOADER begins executing, at some point it will call BlAllocateDescriptor, which is hooked by us.
- Our hook on BlAllocateDescriptor lets us know when OSLOADER calls said function (which means the function is initialized and ready for use, which means we can use it).
- Now that we know BlAllocateDescriptor is ready for use, we will remove the hook and make 3 calls to BlAllocateDescriptor.
    - The first call will be to allocate some memory to move Loader32 (ourselves), this is because when paging is enabled the memory we currently execute from will be paged out and our code gone.
    - The second call is to allocate some memory for our driver to run from, because the address the drivers PE file is stored at will also be paged out, we might as well map the driver's PE file into memory now, ready for execution. (I wrote the PE loader that maps and relocate the driver while slightly drunk, it works fine but may be a little confusing). We do not resolve imports yet because ntoskrnl isn't loaded so it's impossible (Yes, I spent 20 minutes debugging before I realized that).
    - The final call will be to process the original call that OSLOADER was trying to make when our hook stole the show.
- It is important to note that the entire boot process is single-threaded so we need not worry about any race-conditions with our hooks. ☺

## Loader32 – BlAllocateDescriptor Hook

- Just before our BlAllocateDescriptor transfers execution back to OSLOADER, we will relocate loader32 to the allocated memory, then perform another byte scan in OSLOADER.
- this time we scan for the following bytes: 8B F0 85 F6 74 11 68 4C 23 these correspond to the instructions:
  **mov esi, eax**
  **test esi, esi**
  **jz 0x11**
  **push 234Ch**
- The code directly after these instructions is used by OSLOADER to call KiSystemStartup in ntoskrnl, right before "**mov esi, eax**" is a push followed by a call, the push will push "**LOADER_PARAMETER_BLOCK**" to the stack, we'll change the address of the call to point to our code.
- The KiSystemStartup hook will point to code in the loader32 we copied to the allocated memory (Somewhere in the 0x80000000 range).
- Now we return control to OSLOADER so it can continue loading the OS.

## Loader32 – KiSystemStartup Hook

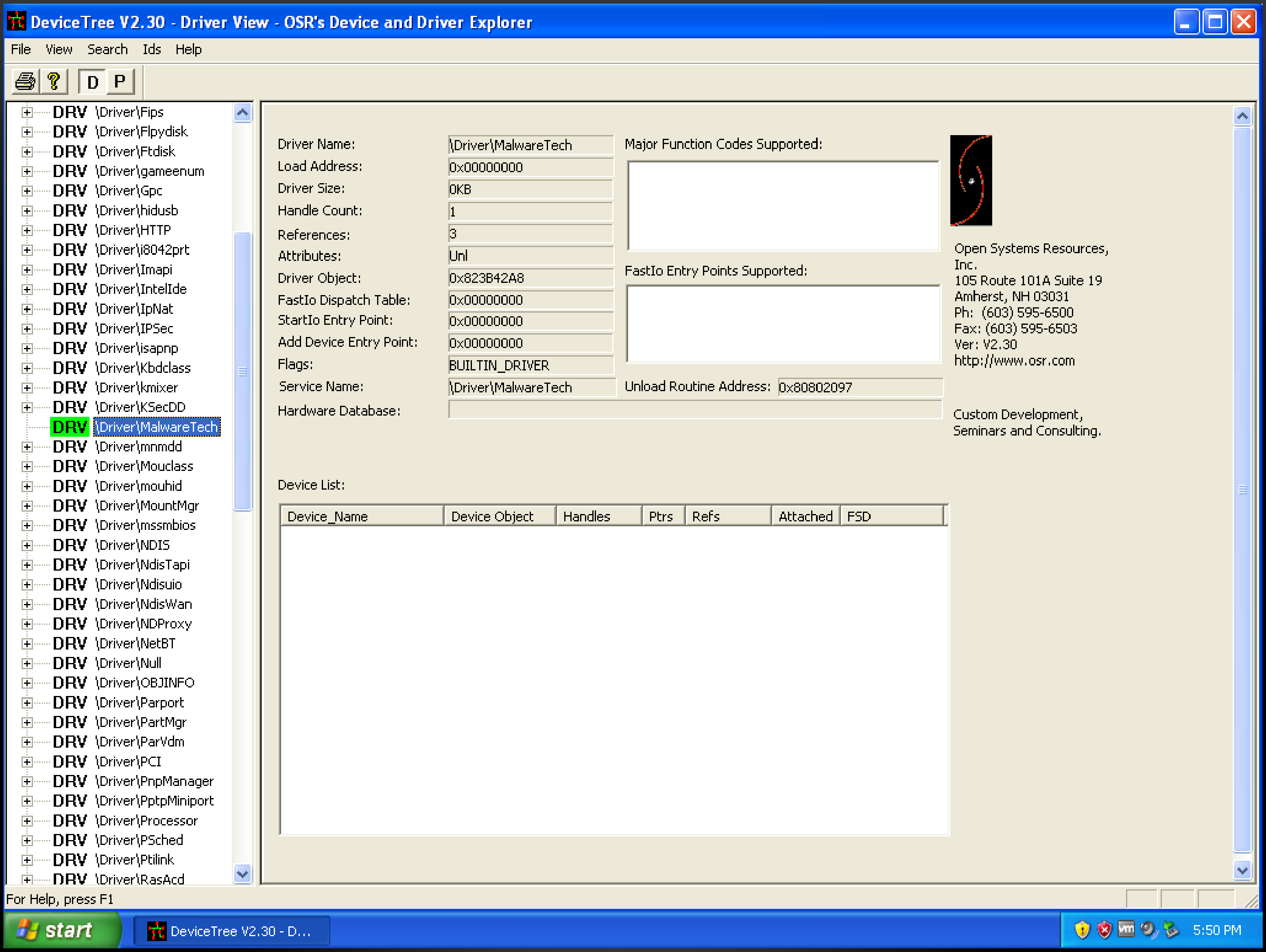
- Right before OSLOADER calls KiSystemStartup, we will receive control.
- Now that ntoskrnl is loaded, we can hook ntoskrnl functions. No need to scan for anymore byte signatures, because ntoskrnl has a Export Address Table.
- We don't know the base address of ntoskrnl and we need it to resolve the exports, luckily right before this code was called, OSLOADER pushed the address of the "**LOADER_PARAMETER_BLOCK**" structure to the stack. We can use the parameter block's "**InLoadOrderListHead**" (**LDR_DATA_TABLE_ENTRY) structure** to get the ntoskrnl base address.
- Loader32 uses a custom GetProcAddress implementation to resolve IoCreateDriver in the kernel, we will then store and overwrite the first 5 bytes of IoCreateDriver with a call to our code.
- Finally we remove the hook in OSLOADER that was placed before the call to KiSystemStartup, then we transfer execution back to OSLOADER so that it can call the real KiSystemStartup.

## Loader32 – IoCreateDriver Hook

- Our code will be called when the kernel first calls IoCreateDriver, which will be early on during IoInitSystem (called by a function that's called by a function that's called during KiSystemStartup, or something like that).
- I chose to hook IoCreateDriver because it can be used to execute a driver, it's only called when the kernel is ready to execute drivers, and it's called before the kernel reinitializes boot drivers (meaning we can act as a boot driver and we're executed early on in system initialization).
- We will resolve the driver's imports (works better now that the kernel is actually loaded).
- Now that the driver has all the imports resolved, we will unhook IoCreateDriver and call it twice:
    - The first call is to execute our driver.
    - The second call is to process the original call that the kernel was trying to make before our hook intercepted it.
- This is the last stage of the bootkit, our driver is created and executed as \Driver\MalwareTech.

## Driver32

- This is just a template driver used to demonstrate the bootkit functionality.
- We will output “System successfully infected.” to the kernel debugger, then register a callback using “**IoRegisterBootDriverReinitialization**”, the callback will be called after all the boot drivers have been loaded (this is to make sure the filesystem is loaded).
- Once the callback is called, the driver will create a file named “MalwareTech.txt” in C:\.



DeviceTree V2.30 - Driver View - OSR's Device and Driver Explorer

File View Search Ids Help

```
DRV \Driver\Fips
DRV \Driver\Flpydisk
DRV \Driver\Ftdisk
DRV \Driver\gameenum
DRV \Driver\Gpc
DRV \Driver\hidusb
DRV \Driver\HTTP
DRV \Driver\i8042prt
DRV \Driver\Imapi
DRV \Driver\IntelIde
DRV \Driver\IpNat
DRV \Driver\IPSec
DRV \Driver\isapnp
DRV \Driver\Kbdclass
DRV \Driver\kmixer
DRV \Driver\KSecDD
DRV \Driver\MalwareTech
DRV \Driver\mnmdd
DRV \Driver\Mouclass
DRV \Driver\mouhid
DRV \Driver\MountMgr
DRV \Driver\mssmbios
DRV \Driver\NDIS
DRV \Driver\NdisTapi
DRV \Driver\Ndisuio
DRV \Driver\NdisWan
DRV \Driver\NDProxy
DRV \Driver\NetBT
DRV \Driver\Null
DRV \Driver\OBJINFO
DRV \Driver\Parport
DRV \Driver\PartMgr
DRV \Driver\ParVdm
DRV \Driver\PCI
DRV \Driver\PnpManager
DRV \Driver\PptpMiniport
DRV \Driver\Processor
DRV \Driver\PSched
DRV \Driver\Ptilink
DRV \Driver\RasAcd
```

| | |
|---|---|
| Driver Name: | \Driver\MalwareTech |
| Load Address: | 0x00000000 |
| Driver Size: | 0KB |
| Handle Count: | 1 |
| References: | 3 |
| Attributes: | Unl |
| Driver Object: | 0x823B42A8 |
| FastIo Dispatch Table: | 0x00000000 |
| StartIo Entry Point: | 0x00000000 |
| Add Device Entry Point: | 0x00000000 |
| Flags: | BUILTIN_DRIVER |
| Service Name: | \Driver\MalwareTech |
| Hardware Database: | |

Major Function Codes Supported:

FastIo Entry Points Supported:

Unload Routine Address: 0x80802097

Open Systems Resources, Inc.
105 Route 101A Suite 19
Amherst, NH 03031
Ph: (603) 595-6500
Fax: (603) 595-6503
Ver: V2.30
http://www.osr.com

Custom Development, Seminars and Consulting.

Device List:

| Device_Name | Device Object | Handles | Ptrs | Refs | Attached | FSD |
|---|---|---|---|---|---|---|

For Help, press F1

## Playtime (VMware)

- download an XP SP3 ISO and install it in Vmware (disk format must be NTFS).
- Enable floppy disk in the VM settings and select “use floppy image”.
- Use the floppy.img in the bootkit directory.
- During boot when the VMWare BIOS screen shows, hit “esc” to enter the boot menu, make sure it is set to boot from removable drive first.

## Playtime (Bochs)

- For this I recommend using IDA with bochs, if you have IDA Pro 6.1, you will need to download version 2.4.6 of bochs, any higher version will not work.
- Download an XP SP3 ISO.
- From the command-line run bximage.exe in the bochs folder and use the following settings: [hd], [flat], 2048, <whatever name you want>.
- Now create a bochsrc.bxrc file and then edit & use the config displayed on the next slide.
- Open bochs.exe and complete the XP installation (disk format must be NTFS).
- Point the floppy to floppy image in the bootkit directory.
- Open the bochsrc.bxrc file with IDA and press play.

- #Megabytes of RAM
- megs: 256
- #Bios and VGABios locations
- romimage: file=$BXSHARE\BIOS-bochs-latest
- vgaromimage: file=$BXSHARE\VGABIOS-lgpl-latest
- #Points to bootkit floppy image
- floppya: 1_44="PATH TO BOOTKIT FLOPPY IMAGE HERE", status=inserted
- #hard disk (XP Image)
- ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
- ata0-master: type=disk, path="PATH TO DISK IMAGE YOU CREATED WITH BXIMAGE HERE", mode=flat, cylinders=4161, heads=16, spt=63
- ata1-master: type=cdrom, path="PATH TO WINDOWS XP ISO", status=inserted
- #Boot from floppy
- boot: <SET TO "floppy" FOR BOOTKIT AND "cdrom"  FOR OS INSTALL>
- #Config interface
- config_interface: textconfig
- display_library: win32
- #Logfile
- log: bochsout.txt
- #Disable mouse
- mouse: enabled=0
- keyboard_mapping: enabled=1, map=$BXSHARE/keymaps/x11-pc-us.map

## Resources

- http://thestarman.narod.ru/asm/mbr/NTFSBR.htm - A great deal of information about windows boot records.
- http://www.stoned-vienna.com/ - An aswesome bootkit working for Windows XP, Vista, and 7 (32-bit system). Probably not for the faint hearted as there is a LOT of asm spread out across many different files.
- http://sourceforge.net/projects/bochs/ - An x86 emulator with powerful debugger, great for debugging bootkits.
- http://www.hex-rays.com/products/ida/ - Probably the best disassembler ever, works great as a GUI interface for bochs (which is command line).
- http://www.malwaretech.com/ - My blog, reading it will increase your chance of getting laid 10 folds.