Booty

Subverting the Windows 7 Bootloader

Alex Hixon

Contents

1 Abstract

With the new Windows NT6 series, Microsoft have removed the traditional NTLDR method of booting the system. It's replacement, bootmgr and winload, aim to simplify boot management, minimise code duplication and provide decent support for up and coming UEFI systems running on Intel and AMD machines.

With this, however, comes an increased number of attack vectors. This report will look at these, and discuss the implementation details on an attack which circumvents the boot process and attempts to provide a method to steal user data without alerting the user of its existence.

2 Background

The standard boot procedure is shown in figure 1. Follows is a brief overview of the boot process for Windows NT6.

Upon startup, `bootmgr` will be executed by the MBR or VBR (volume boot record). It then verfies the checksum of `bootmgr.exe`, decompresses it, loads it into memory, and finally jumps to protected mode and begins to execute it.

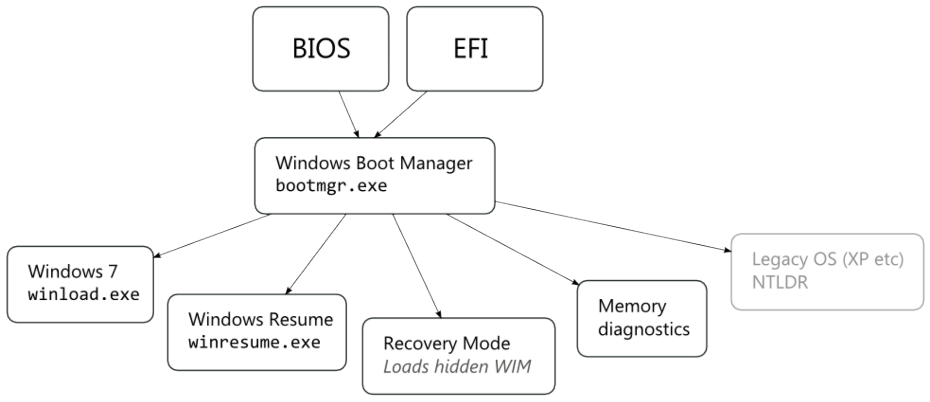


Fig. 1: Windows NT6 Startup Procedure

On Windows 7, `bootmgr` is comprised of 3 concatenated exectuables:

- a 16-bit stub which does intialisation
- uncompressed PE executable
- a 64-bit (or 32-bit, depending on the platform) PE executable that contains the real 'bootmgr.exe' application. This handles selecting the chosen boot device, and any boot options.

  Note: Vista, too, has a bootmgr application, however its 16-bit stub does not include decompression as the native `bootmgr.exe` appended to the end itself is not compressed. Although it should be possible to use Vista's 16-bit stub with uncompressed native code following it on Windows 7, this was not tested.

On an UEFI system, bootmgr itself is an EFI boot application. It does not have a stub that does initialisation; instead, it is already executing in protected mode and begins to run as normal. In either case, the same `winload.exe` is executed whether the system is booted from BIOS or from UEFI. This, of course, has the implication that any modifications upon `winload.exe` would be pre-boot environment agnostic.

It may be useful to think of `bootmgr` as a similar application as GRUB - it enables you to select boot targets and runs them.

Once `bootmgr.exe` is loaded into memory, it verifies itself and key Windows system files using certificates built into itself.

Once the user selects a valid boot option (or one is automatically selected), bootmgr.exe loads the boot entry's loader program and passes any boot flags. By default, this is `%windir%\system32\winload.exe`. Winload loads the registry hive, any boot drivers, and loads `NTOSKRNL.exe` and its dependencies into memory, sets up paging and starts executing the kernel. Drivers must be signed against one of the 5 built-in root CAs.

3 Attack vectors

The following issues stood out:

- `winload.exe` assumes that the `bootmgr.exe` that called it was valid and unmodified. (this however may be intentional; to enable another bootloader - such as GRUB - to boot Windows)
- `bootmgr.exe` assumes that it is not modified
- if a certificate is compromised, it cannot be revoked (without an update to the binary being pushed out, which may have its own implications).

I have chosen to take advantage of the first two issues.

I will now proceed to explain how I was able to circumvent the bootloader, and in doing so, was able to load abitrary kernel drivers.

This following information applies to PC/AT BIOS systems, however to the best of the authors' knowledge, *this should also apply to UEFI systems*.

I was targeting a Windows 7 x64 system for this project, though these ideas can be carried over to different architectures and different versions of Windows (provided they're in the NT6 series).

4 Implementation

4.1 Permissions

By default, `bootmgr` and `winload.exe` cannot be modified, as they are owned by the TrustedInstaller group. However, since it assumed the attacker has gained admin privilleges to perform this attack, it is possible just to change ownership of the file to the Administrators group, and then grant ourselves write permissions.

This can be easily done programatically by calling `SetNamedSecurityInfo` and `AdjustTokenPrivilleges` from `advapi32.dll`. Using the `takeown` and `cacls` utilities included with Windows can be used to do the job from a batch script if required.

4.2 Removing self-verification

I first needed to disable any verification done on bootmgr and winload. This was surprisingly trivial.

Other papers discussing Windows Vista bootloader security[1][2] (only found *after* this project had already half-started, unfortunately) have attempted to circumvent the checks by "monkey-patching" throughout, by changing *several* functions, which if their definitions change or more/less code is added, means that they require modification and different offset targets.

However, I opted to modify the single `BlImgQueryCodeIntegrityBootOptions` function. This function, given the current bootmgr application context and two pointers, asks the BCD if certain debugging options are set, and returns their values. One or both are checked to see particular checks should be performed (such as code signing). The modified function always returns that these flags are set, and so allows booting the system with an unsigned `winload.exe` and `bootmgr.exe` (see 2).

The reason this works is that if the first of these flags is set when code integrity is initialised, `bootmgr` sets an internal state so that any calls to `ImgpValidateImageHash` and similar calls (`BlImgLoadBootApplication`, ResInitializeMuiResource - checking if the MUI is valid, and so on) will always return true.

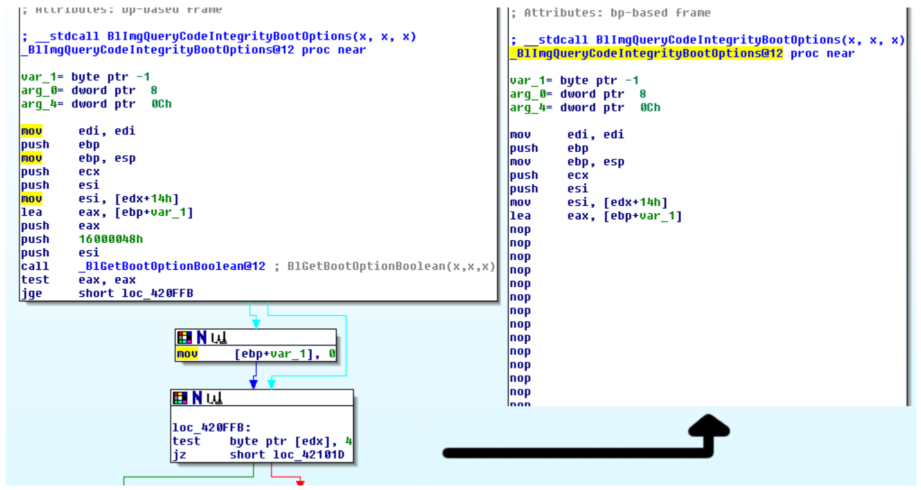
To create the modified bootmgr executable, I utilised a tool called `bmzip`[3]. This decompresses and extracts the PE+ (MZ) `bootmgr.exe` from the bootmgr file, instead of writing one myself. I then recompressed `bootmgr.exe`, and appended the 16-bit initialisation code to complete the binary.

The easiest way would merely to have turned these options on in the BCD. However, these can easily be read out by a system tool to determine whether or not the system is in a non-standard state. Also, this would only have disabled file verification for bootmgr, *not* for the drivers. Previously in Windows Vista,

[1] http://www.symantec.com/avcenter/reference/Windows_Vista_Kernel_Mode_Security.pdf

[2] http://www.blackhat.com/presentations/bh-europe-07/Kumar/Whitepaper/bh-eu-07-Kumar-WP-apr19.pdf

[3] Available from http://www.coderforlife.com/projects/win7boot/extras/bmzip64.exe - closed source freeware tool released under the Windows 7 EULA



```
; Attributes: bp-based frame

; __stdcall BlImgQueryCodeIntegrityBootOptions(x, x, x)
_BlImgQueryCodeIntegrityBootOptions@12 proc near

var_1= byte ptr -1
arg_0= dword ptr  8
arg_4= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    esi
mov     esi, [edx+14h]
lea     eax, [ebp+var_1]
push    eax
push    16000048h
push    esi
call    _BlGetBootOptionBoolean@12 ; BlGetBootOptionBoolean(x,x,x)
test    eax, eax
jge     short loc_420FFB

mov     [ebp+var_1], 0

loc_420FFB:
test    byte ptr [edx], 4
jz      short loc_42101D
```

```
; Attributes: bp-based frame

; __stdcall BlImgQueryCodeIntegrityBootOptions(x, x, x)
_BlImgQueryCodeIntegrityBootOptions@12 proc near

var_1= byte ptr -1
arg_0= dword ptr  8
arg_4= dword ptr  0Ch

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    esi
mov     esi, [edx+14h]
lea     eax, [ebp+var_1]
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
```

Fig. 2: Modified `BlImgQueryCodeIntegrity`

`DISABLE_INTEGRITY_CHECKS` could be passed as a boot option string to the loader application (`winload`), but in Windows 7 this has been disabled.

4.3 Unsigned drivers

By this stage, it is now possible to modify winload.exe without any repercussions. To test this hypothesis, I changed a string inside winload, updated the file checksum and booted to see the on-screen text had changed in response to the string change (see figure 3).

I then tried to integrate the payload into the bootkit, and it was decided that modifying winload would be the best option. This however, turned out to take a few tries - although each stage provided valuable experience and information, so it wasn't completely a waste to have prior attempts.

4.3.1 Attempt 1

Initally, a code cave was implemented. This was supposed to load the driver image into memory and in doing so load the driver. This was done as follows:

1. Create an additional segment at the end of the file (I made this 0x1000 bytes).
2. Update the PE header and image size.
3. After the driver linked list had been enumerated and each driver called with `LoadImageEx`, add code to jump out to the new segment (codecave).
4. In the code cave, load the driver image into memory (also via `LoadImageEx`), and return to the original program flow.

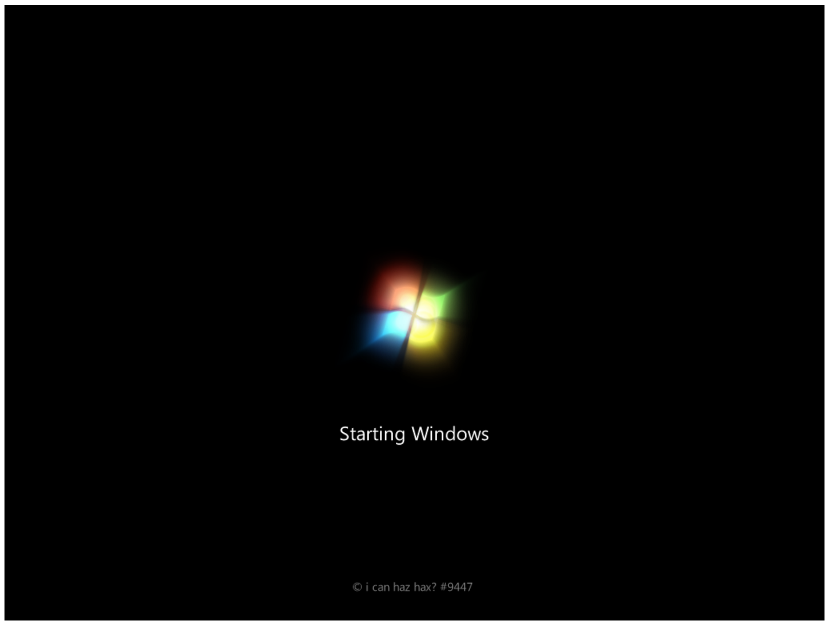


Fig. 3: Running unsigned `winload.exe`; note the non-standard copyright text

Steps 3 and 4 did cause some issues. For starters, the codecave was in a different segment, so one cannot just use a `JMP` instruction. There were two options here: either use a `LJMP` (long jump) to jump across the segment, or load the address pointer into a register and jump to that address.

The second option was taken, merely because it was easier to code. This had its own issues though. Initially, running the code in virtual machine used for testing would cause the CPU to be reset. Because it was difficult to debug the issue, boot mode debugging was enabled in the BCD and Windbg attached to the virtual machine over a virtual COM port.

Windbg isn't the most easy to use debugger. It has a poorly designed user interface and a high learning curve. However, after spending some TLC with it, the problem was discovered. The issue was that code execution would jump to an incorrect place in memory. However, what was confusing is that disassembling with IDA Pro showed that the modified executable loads in the correct addresses to the code cave and in theory, should be jumping correctly.

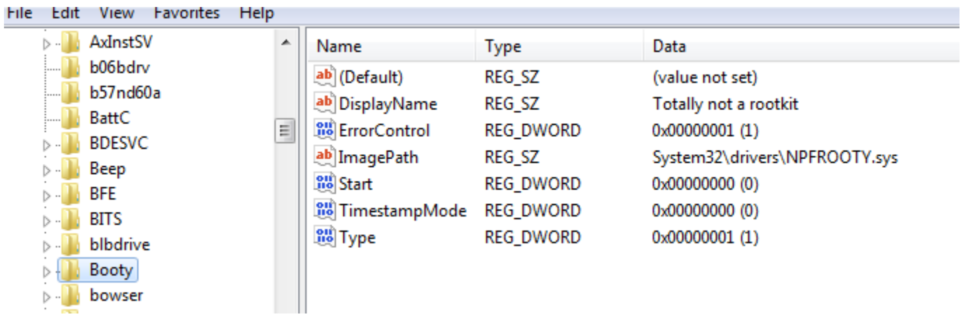
It turns out that `winload.exe` is loaded into memory, but not at the expected virtual address. `bootmgr` relocates the `winload.exe` image after it loads it by running `LdrRelocateImageWithBias`. Additionally, `bootmgr`, after loading winload, does not appear to do rebasing on the address we load into the register to jump to. As such, it needs to be done manually. `winload.exe` always appears to loaded 0x116000 bytes before its preferred base address and so we subtract this value from the pointers. Suddenly the code jumps to the correct place!

Unfortunately, this did not cause the kernel module to be loaded. In hindsight, this made sense. `LoadImageEx` merely loaded the driver into memory. The kernel does not know about the existence of the driver - it is not running until the I/O manager in the Windows Kernel initalises it (Phase 1 of the NT boot process).

4.3.2 Attempt 2

After spending some time researching how the kernel, boot loader and service control manager all interact with each other, and further analysis of winload, an easier way to load the driver at boot time became apparent.

After loading the base system modules, `NTOSKRNL.EXE`, `HAL.DLL`, the boot debugger(s), and then resolving their imports, `winload` iterates over the keys located in `HKEY_LOCAL_MACHINE\CurrentControlSet\Control\Services` and finds those with the `Order` DWORD set to `0` (those with order 1 or higher are loaded by `NTOSKRNL`). These are then sorted by the `GroupOrderList` registry keys, and then, as mentioned previously, iterated over and `LoadImageEx` called on each driver and its dependencies. This can be seen in figure 4.



Name	Type	Data
(Default)	REG_SZ	(value not set)
DisplayName	REG_SZ	Totally not a rootkit
ErrorControl	REG_DWORD	0x00000001 (1)
ImagePath	REG_SZ	System32\drivers\NPFROOTY.sys
Start	REG_DWORD	0x00000000 (0)
TimestampMode	REG_DWORD	0x00000000 (0)
Type	REG_DWORD	0x00000001 (1)

Fig. 4: 'Rooty' boot driver registry entry

Simply by adding a registry key with `Order` set to 0 will cause the driver to load at boot time. In a stealth situation, this registry key could be created before the list iterated and then deleted in the kernel after the driver had been loaded (see section §5).

Of course, being unsigned, means by default the driver would not load, unless the specified option was selected by hitting F8 on boot and choosing Load unsigned drivers. To work around this, a similar patch was developed for bootmgr (see 4.2). This tells bootmgr to always ignore any signing and verification checks that would normally occur. The patch is listed verbaitum below:

```
; intended to replace BlImgQueryCodeIntegrity
; offset 0x30230

mov [rsp+0x10], rbx
```

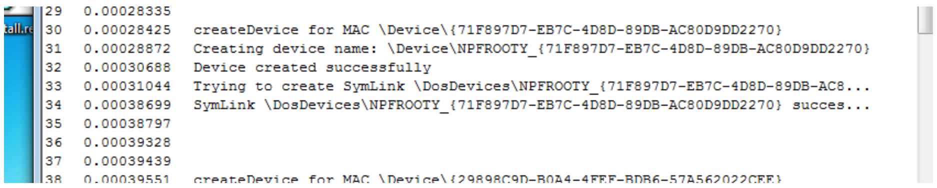
```
push rdi
sub rsp, 0x20              ; setup stack
mov r9b, 1

mov [rsp+0x30], r9b        ; argument 1
mov [rsp+0x38], r9b        ; argument 2
mov rbx, 0                 ; success?

add rsp, 0x20
pop rdi
retn
```

Finally, the driver loads (see figure 5)! Note that disabling PatchGuard is *not* required.

While the NDIS code works, we cannot perform keylogging, as the keyboard class driver and HID drivers are initialised later on in the boot process. This means it is necessary to increase the Order setting for the rootkit. Since increasing the boot order means the driver is loaded via the Service Control Manager, it must be patched to also ignore unsigned drivers.



```
29   0.00028335
30   0.00028425   createDevice for MAC \Device\{71F897D7-EB7C-4D8D-89DB-AC80D9DD2270}
31   0.00028872   Creating device name: \Device\NPFROOTY_{71F897D7-EB7C-4D8D-89DB-AC80D9DD2270}
32   0.00030688   Device created successfully
33   0.00031044   Trying to create SymLink \DosDevices\NPFROOTY_{71F897D7-EB7C-4D8D-89DB-AC8...
34   0.00038699   SymLink \DosDevices\NPFROOTY_{71F897D7-EB7C-4D8D-89DB-AC80D9DD2270} succes...
35   0.00038797
36   0.00039328
37   0.00039439
38   0.00039551   createDevice for MAC \Device\{29898C9D-B0A4-4FEF-BDB6-57A562022CEE}
```

Fig. 5: Success!

If I was running 32-bit Windows 7, this would not be a problem, as there is no driver signing enforcement.

4.3.3 Attempt 3

Unfortunately, time did not allow to disable PatchGuard. However, this has been done by several others in the past, so the hard work in this area is already complete. Perhaps with more time and/or better time management this may have been possible.

To some extent, circumventing this is *beyond the idea of a bootkit* and instead part of a rootkit. However, this is only a technicality, as to fulfil all intended requirements by being able to steal users' data, the code must be able to do both.

5 Bootkit Detection

This section deals with the detection of the bootkit only, not

5.1 Current state

There are several ways in which the current bootkit can be detected:

- finding the driver in the current list of loaded kernel modules
- checking the registry for the Service entry (see 4.3.2)
- verifying the checksum and MD5 of modified files on the filesystem
- scanning the memory to check for the existence of the payload driver

5.2 Theoretical

In theory, it is possible to create a rootkit that is almost impossible to detect.

The kernel could be patched after loading into memory to add code to remove the driver from the kernel's list of loaded modules.

Similarly, the Service registry entry could be added during boot so that the driver is loaded and removed after the kernel has intialised the driver.

The two remaining issues, file changes and memory scanning require a slightly more involved solution to get around.

5.2.1 Filesystem changes

As it stands, the system boot files can simply be hashed and checked against their shipped values to see if the files have changed. To work around this, either an NTFS filter driver or intercepts added to the NTFS file system driver. This is possible now because boot system files are verified if they are signed at boot time.

The best way would be to create a copy of winload.exe and bootmgr, and funnel any changes made to winload.exe and bootmgr to these copies using this modified NTFS driver. This way, it would appear from the operating system that both these files are the originals. They would have the original MD5 hash since they are 'read' from the originals, and writes made by updates only update the copies, not the originals that the system boots from. This also prevents updates from removing the bootkit.

The **only way** the bootkit can be detected is from offline disk inspection (ie mounting the disk from another operating system and scanning/verifying file checksums). Note however, that the this is a very unlikely scenario, and in most cases will never occur (especially on desktop systems).

5.2.2 Memory scanning

Since the driver is (or can be) intialised before any antivirus tools, it is possible to enable hardware breakpoints on memory access to the region where the rootkit driver lies. If the breakpoint is fired, the code could move itself to another kernel page that it has left for itself and hide temporarily.

6 Future

Windows 8, being a continuation of the NT6 series, continues to use to winload and either bootmgr.exe or its UEFI equivilent.

Unfortunately, I was unable to get `bmzip` to decompress bootmgr. A cursory look shows that one of the three executables have been removed, leaving only the 16-bit initialisation code and the architecture dependent PE binary. Otherwise, the organisation of the file looks exactly the same; the main executable is compressed, and it's imagined it would function largely in the same way, apart from small additions of new code.

More importantly, however, is the addition of a 'chain of trust' into the boot process. Note that this applies only to UEFI machines. The Windows 8 boot loader will be signed, and the UEFI chipset will verify that the bootloader is signed with a key that matches the built in keys[4]. OEMs (those conforming to the Windows 8 logo program, anyway) are additionally required that this signing be shipped enabled, although it's likely support for disabling this will exist from firmware[5].

How self-signing (for example, Linux would require this) and key management would work has not been disclosed, but will certainly prove interesting.

More than likely this will mean that bootkits will need to target yet another layer to get a foothold of the system, although their complexity may mean that this can all be done inside the operating system if an exploit is found in calling out to the boot firwmare, or alternatively if a certificate is compromised, this would make bootkits much easier to distribute, as keys cannot be revoked.

7 Other operating systems

A brief comparison to other major operating systems, and any challenges to creating a bootkit for those platforms.

7.1 Mac OS X

Mac OS X has supported UEFI since 2005, with the introduction of Intel CPUs at the core of their system designs. The first few iterations used the Tiano Core implementation (Intel's reference implementation).

However, Apple decided to stop developing on this platform and use their own implementation. It is up to the reader to decide whether or not this was because of ideas about the unsightly nature of the Tiano codebase, or merely because Apple's boot team were suffereing from Not Invented Here syndrome.

In any case, the current implementation of the platform is very nonstandard. By default, the UEFI boot application is loaded off the filesystem from `/System/Library/CoreServices/boot.efi`, unless a different path is specified in the NVRAM, the user holds one of the boot keys (ie holding Option to select

[4] http://video.ch9.ms/build/2011/slides/HW-457T_van_der_Hoeven.pptx
[5] http://mjg59.dreamwidth.org/5552.html

a different boot device), or an attached bootable device is detected (such as USB key).

This file can easily be modified on the filesystem, and the firmware will happily boot this file.[6] Additionally, unlike Windows NT6, no verification or signing is performed on the bootloader by any of Apple's personal computer offerings, and the file is also written entirely in native code.

Creating a bootkit would therefore be trivially easy.

7.2 Linux

Since the source code to the bootloader is available, malicious code can be added to the GRUB source code, recompiled and then installed on the target system. Optionally, the kernel image can be replaced.

Once again, creating a bootkit would be trivially easy.

[6] A special checksum algorithm is used on later Apple TV models to make sure the bootloader is 'signed' by Apple. Since it is only a checksum algorithm, the code could easily be reference engineered or even bruteforced to produce a potentially valid bootloader. It is also worth noting that iPhones do NOT use this bootloader.