PETE'S BLOG

We're full of IT!

2014-05-29

Compiling and installing Grub2 for standalone USB boot

SEARCH

Search

TOTAL PAGEVIEWS

637,244

The goal here, is to produce the necessary set of files, **to be written to an USB Flash Drive using dd** (rather than using the Grub installer), so that it will boot through Grub 2.x and be able to process an existing `grub.cfg` that sits there.

As usual, we start from nothing. I'll also assume that you know nothing about the intricacies of Grub 2 with regards to the creation of a bootable USB, so let me start with a couple of primers:

1. For a BIOS/USB boot, Grub 2 basically works on the principle of a standard MBR (`boot.img`), that calls a custom second stage (`core.img`), which usually sits right after the MBR (sector 1, or `0x200` on the UFD) and which is a flat compressed image containing the Grub 2 kernel plus a user hand-picked set of modules (`.mod`).
   These modules, which get added to the base kernel, should usually limit themselves to the ones required to access the set of file systems you want Grub to be able to read a config file from and load more individual modules (some of which need to be loaded to parse the config, such as `normal.mod` or `terminal.mod`).
   As you may expect, the modules you embed with the Grub kernel and the modules you load from the target filesystem are exactly the same, so you have some choice on whether to add them to the core image or load them from the filesystem.

2. You most certainly do NOT want to use the automated Grub installer in order to boot an UFD. This is because the Grub installer is designed to try to boot the OS it is running from, rather than try to boot a random target in generic fashion. Thus, if you try to follow the myriad of quick Grub 2 guides you'll find floating around, you'll end up nowhere in terms of booting a FAT or NTFS USB Flash Drive, that should be isolated of everything else.

With the above in mind, it's time to get our hands dirty. Today, I'm going to use Linux, because my attempts to try to build the latest Grub 2 using either MinGW32 or cygwin failed miserably (crypto compilation issue for MinGW, Python issue for cygwin on top of the usual CRLF annoyances for shell scripts due to the lack of a .gitattributes). I sure wish I had the time to produce a set of fixes for Grub guys, but right now, that ain't gonna happen ⇒ Linux is is.

First step is to pick up the latest source, and, since we like living on the edge, we'll be using git rather than a release tarball:

```
git clone git://git.savannah.gnu.org/grub.git
```

Then, we bootstrap and attempt to configure for the smallest image size possible, by disabling NLS (which I had hoped would remove anything gettext but turns out not to be the case - see below).

```
cd grub
./autogen.sh
./configure --disable-nls
make -j2
```

After a few minutes, your compilation should succeed, and you should find that in the `grub-core/` directory, you have a `boot.img`, `kernel.img` as well as a bunch of modules (`.mod`).

As explained above, `boot.img` is really our MBR, so that's good, but we're still missing the bunch of sectors we need to write right after that, that are meant to come from a `core.img` file.

The reason we don't have a `core.img` yet is because it is generated dynamically, and we need to tell Grub exactly what modules we want in there, as well as the disk location we want the kernel to look for additional modules and config files. To do just that, we need to use the Grub utility `grub-mkimage`.

Now that last part (telling grub that it should look at the USB generically and in isolation, and not give a damn about our current OS or disk setup) is what nobody on the Internet seems to have the foggiest clue about, so here goes: We'll want to tell Grub to use BIOS/MBR mode (not UEFI/GPT) and that we'll have one MBR partition on our UFD containing the boot data that's not included in `boot.img`/`core.img` and that it may need to proceed. And with BIOS setting our bootable UFD as the first disk (whatever gets booted is usually the first disk BIOS will list), we should tell Grub that our disk target is `hd0`. Furthermore, the first MBR partition on this drive will be identified as `msdos1` (Grub calls MBR-like partitions `msdos#`, and GPT partitions `gpt#`, with the index starting at `1`, rather than `0` as is the case for disks).

Thus, if we want to tell Grub that it needs to look for the first MBR partition on our bootable UFD device, we must specify `(hd0,msdos1)` as the root for our target.
With this being sorted, the only hard part remaining is figure out the basic modules we need, so that Grub has the ability to actually identify and read stuff on a partition that may be FAT, NTFS or exFAT. To cut a long story short, you'll need at least `biosdisk` and `part_msdos`, and then a module for each type of filesystem you want to be able to access. Hence the complete command:

```
cd grub-core/
../grub-mkimage -v -O i386-pc -d. -p\(hd0,msdos1\)/boot/grub biosdisk part_msdos fat ntfs exfat -o core.i
```

NB: If you want to know what the other options are for, just run `../grub-mkimage --help`
Obviously, you could go crazy adding more file systems, but the one thing you want to pay attention is the size of `core.img`. That's because if you want to keep it safe and stay compatible with the largest choice of disk partitioning tools, you sure want to have `core.img` below 32KB - 512 bytes. The reason is there still exists a bunch of partitioning utilities out there that default to creating their first partition on the second "track" of the disk. And for most modern disks, including flash drives, a track will be exactly 64 sectors. What this all means is, if you don't want to harbour the possibility of overflowing `core.img` onto your partition data, you really don't want it to be larger than `32256` or `0x7E00` bytes.
OK, so now that we have `core.img`, it's probably a good idea to create a single partition on our UFD (May I suggest using Rufus to do just that? ;)) and format it to either FAT/FAT32, NTFS or exFAT.

Once this is done, we can flat-copy both the MBR, a.k.a. `boot.img`, and `core.img` onto those first sectors. The one thing you want to pay attention to here is, while copying `core.img` is no sweat, because we can just use a regular 512 byte sector size, for the MBR, you need to make sure that **only** the first 446 bytes of `boot.img` are copied, so as not to overwrite the partition data that

LABELS

Windows USB linux BIOS Rufus VMWare AVR Boot driver EFI MSVC MinGW UEFI Android DOS LUFA Nexus 7 VMWare-Player Visual Studio Windows 7 serial AVR-Studio Atmel Authenticode Debug Formatting FreeDOS Grub NEC NTFS QEMU Renesas UAC UBRX USB-3.0 WCID Windows 8 Windows XP bash bootloader certificate clang git installation kernel libusb modules panic-room pki rEFInd shell ssh x64 80080003 Apache Arabic CAR CAT CLSID_FileOpenDialog COFF Certum DDK DOS bootable DUET Debian Device Manager DeviceIOControl EDK2 Extended-Capabilities Flash Fresco Logic GPL GlobalSign Grub4DOS HD HP HTML HyperTerminal IFileOpenDialog L2-Cache LGPO LNK2005 Laserjet MBR Makefile Microchip MinGW-w64 OTG Open Source PD720200 PD720200A PHP PIC PnPUtil Program Compatibility Assistant RTF RealTek Reverse Engineering Rich Text Format SATA SMC-1211 SSL Storage Space UEFI:SIMPLE USB_DOS Unicode VIA Labs VMWare-Tools Vista Visual Studio Express WDK WYGIWER Watcom Windows 2003 access denied afxres.h assembly attachments automake autotools backup bad blocks badfile-number bin2coff binary binutils bootable busybox cacls cathash certificate chkdsk code signing com0com compile cpio cross-compiler custom development digital-signatures downgrade eSATA enumeration firmware flash drive fmifs formatx gPXE gcc gettext git-daemon github hash hidden-devices hotplug https inf inline issuer statement kfifo libtool libusbx libwdi link local-group-policies localization logging macros module msys-git multilib null-modem objdump option-ROM patches preprocessor putty recovery remote repository restore-point review rooting rtl8139.exe rss2email sample setup sha1 signed-driver static-analyzer sysinternals system32 thunderbird trailing-whitespace troubleshooting udev unmkbootimg vmdk vmware-mount website x86 also resides in the MBR and that has already been filled. So please pay close attention to the `bs` values below:

```
dd if=boot.img of=/dev/sdb bs=446 count=1
dd if=core.img of=/dev/sdb bs=512 seek=1 # seek=1 skips the first block (MBR)
```

Side note: Of course, instead of using plain old `dd`, one could have used Grub's custom `grub-bios-setup` like this:

```
../grub-bios-setup -d. -b ./boot.img -c ./core.img /dev/sdb
```

However, the whole point of this little post is to figure out a way to add Grub 2 support to Rufus, in which we'll have to do the copying of the `img` files without being able to rely on external tools. Thus I'd rather demonstrate that a `dd` copy works just as good as the Grub tool for this.
After having run the above, you may think that all that's left is copying a `grub.cfg` to `/boot/grub/` onto your USB device, and watch the magic happen... but you'll be wrong.

Before you can even think about loading a `grub.cfg`, and at the very least, Grub **MUST** have loaded the following modules (which you'll find in your `grub-core/` directory and that need to be copied on the target into a `/boot/grub/`**`i386-pc`**`/` folder):

- boot.mod
- bufio.mod
- crypto.mod
- extcmd.mod
- gettext.mod
- normal.mod
- terminal.mod

As to why the heck we still need `gettext.mod`, when we made sure we disabled NLS, and also why we must have `crypto`, when most usages of Grub don't care about it, your guess is as good as mine...

Finally, to confirm that everything works, you can add `echo.mod` to the list above, and create a `/boot/grub/grub.cfg` on your target with the following:

```
insmod echo
set timeout=5

menuentry "test" {
    echo "hello"
}
```

Try it, and you should find that your Grub 2 config is executing at long last, whether your target filesystem in FAT, NTFS or exFAT, and you can now build custom bootable Grub 2 USBs on top of that. Isn't that nice?

FINAL NOTE: In case you're using this to try boot an existing Grub 2 based ISO from USB (say Aros), be mindful that, since we are using the very latest Grub code, there is a chance that the modules from the ISO and the kernel we use in core may have some incompatibility. Especially, you may run into the obnoxious:

```
error: symbol 'grub_isprint' not found.
```

What this basically means is that there is a mismatch between your Grub 2 kernel version and Grub 2 module. To fix that you will need to use kernel and modules from the same source.

POSTED BY PETE AT 02:29

LABELS: BOOT, BOOTLOADER, GRUB, LINUX, MODULES, RUFUS, USB, WINDOWS

NO COMMENTS:

POST A COMMENT

Enter your comment...

Comment as: Select profile...

Publish Preview

LINKS TO THIS POST

Create a Link

Newer Post Home Older Post

Download PC Repair Tool

Network Management Software & IT Tools for Free. Download Now!

LINKS

Rufus
libusb
libwdi
efifs
Bled
ubrx
UEFI:SIMPLE
cecd / libcec

BLOG ARCHIVE

January (1)
November (3)
October (1)
September (1)
June (1)
May (3)
January (1)
November (1)
October (2)
September (1)
August (1)
July (1)
April (2)
December (5)
November (6)
October (4)
August (5)
July (5)
June (2)
May (1)

Subscribe to: Post Comments (Atom)

April (3)
March (9)
February (1)
January (3)
December (2)
September (1)
August (2)
July (3)
June (3)
May (2)
April (2)
March (4)
January (1)

SUBSCRIBE TO

Posts

Comments