lDOS boot documentation

This document has been compiled on 2024-01-02.

Contents

Section 1: lDOS boot protocols

1.1 Sector to iniload protocol

The iniload kernel is loaded to an arbitrary segment. The segment must be at least 60h. Common choices are 60h, 70h, and 200h. At least 1536 bytes of the file must be loaded. Current loaders will load at least 8192 bytes if the file is as large or larger than that. The entrypoint is found by applying no segment adjustment (0) and choosing the offset 400h (1024).

1.1.1 File properties

The file must be at least 4096 bytes long. This is now required, beyond the former lower bound of 1536 bytes, to support an optimisation of the FAT12 and FAT16 boot sector loaders. The lDebug loader and the FAT32+FSIBOOT loader currently retain the 1536 bytes limit.

The file may allow multi-use as a flat .COM format executable, flat .SYS format device driver, or MZ .EXE format executable and/or device driver. It is also valid to append arbitrary sized data such as a .ZIP archive.

The file needs to be placed in the root directory for the boot sector loaders. The lDebug loader allows to load a file from any subdirectory and this is also allowed. The file may be fragmented in any part. The file data may be located anywhere in the file system. The supported cluster sizes should be between 32 Bytes and 2 MiB, inclusive. The sector size should be between 32 Bytes and 8 KiB, inclusive.

1.1.2 Signatures

At offset 1020 (3FCh) there is the signature ‘`lD`’. Behind that there are two bytes with printable non-blank ASCII codepoints. Currently the following signatures are defined:

‘`lDOS`’

lDOS kernel (not yet in use)

‘`lDRx`’

RxDOS kernel

‘`lDFD`’

FreeDOS kernel in fdkernpl stage

‘`lDDR`’

Enhanced DR-DOS single-file kernel in drkernpl stage ‘`lDeb`’

lDebug

‘`lDDb`’

lDDebug (debuggable lDebug)

‘`lDbC`’

lCDebug (conditionally debuggable lDebug)

‘`lDTP`’

lDOS test payload kernel (testpl.asm)

‘`lDTW`’

lDOS test result writer kernel (testwrit.asm)

1.1.3 Load Stack Variables (LSV)

Under this protocol, the pointer ‘`ss:bp`’ is passed. It points to a boot sector with (E)BPB. ‘`bp`’ must be even for compatibility with older iniload (before 2023-March). The stack pointer must be at most ‘`bp - 10h`’. Below the pointed to location there live the Load Stack Variables. These follow this structure:

```
        struc LOADSTACKVARS, -10h
lsvFirstCluster:        resd 1
lsvFATSector:           resd 1
lsvFATSeg:              resw 1
lsvLoadSeg:             resw 1
lsvDataStart:           resd 1
        endstruc
```

lsvFirstCluster

(FAT12, FAT16) Low word gives starting cluster of file. High word uninitialised.

(FAT32) Dword gives starting cluster of file.

(else) Should be zero.

lsvFATSector

(FAT16) Low word gives loaded sector-in-FAT. -1 if none loaded yet. High word uninitialised.

(FAT32) Dword gives loaded sector-in-FAT. -1 if none loaded yet.

(FAT12, else) Unused.

lsvFATSeg

(FAT16, FAT32) Word gives segment of FAT buffer if word/dword [lsvFATSector] != -1.

(FAT12) Word gives segment of FAT buffer. Zero if none. Otherwise, buffer holds entire FAT data, up to 6 KiB.

lsvLoadSeg

Word points to segment beyond last loaded paragraph. Allows iniload to determine how much of it is already loaded.

lsvDataStart

Dword gives sector-in-partition of first cluster's data.

An LSV extension allows to pass a command line to the kernel. The base pointer must be at least ‘`114h`’ then. The stack pointer must be at most ‘`bp - 114h`’ then. This follows the structure like this:

```
lsvclSignature          equ "CL"
lsvclBufferLength       equ 256

        struc LSVCMDLINE, LOADSTACKVARS - lsvclBufferLength - 4
lsvCommandLine:
.start:         resb lsvclBufferLength
.signature:     resw 1
lsvExtra:       resw 1
        endstruc
```

lsvCommandLine.start

Command line buffer. Contains zero-terminated command line string.

lsvCommandLine.signature

Contains the signature value ‘`CL`’ if command line is given.

lsvExtra

Used internally by iniload. Space for this must be reserved when passing a command line.

If no command line is passed then either the stack pointer must be ‘`bp - 10h`’, or ‘`bp - 12h`’, or the word in the lsvCommandLine.signature variable (`word [ss:bp - 14h]`) must not equal the string ‘`CL`’.

- dosemu2's RxDOS.3 support sets ‘`sp = bp - 10h`’
- ldosboot boot.asm (FAT12/FAT16) loader makes sure not to pass the variable with the content "CL". Refer to placeholder and DIRSEARCHSTACK_CL_FIRST uses in the source.
- ldosboot boot32.asm (FAT32) loader uses the variable for an ‘entries per sector’ value which is always a power of two and always below-or-equal 100h.
- lDebug with protocol options `cmdline=0 push_dpt=0` sets ‘`sp = bp - 10h`’

1.1.4 Memory map

The initial loader part that is loaded must be loaded at above or equal to linear 00600h. The FAT buffer segment (if used) must also be stored at above or equal to linear 00600h. The stack (which should extend at least 512 bytes below ‘`ss:bp`’) and boot sector (pointed to by ‘`ss:bp`’, at least 512 bytes length) should also be stored at above or equal to linear 00600h.

There is an additional memory area, the Low Memory Area top reservation, which should be unused by the load protocol at handoff time but be at least 20 KiB in size. It is located below the usable Low Memory Area top. That is, directly below the EBDA, RPL-reserved memory, video memory, or otherwise UMA. This area is reserved in order to facilitate initial loader operation.

None of the memory areas may overlap. This does not include the FAT buffer in case it is uninitialised.

1.1.5 Load filename in the boot sector

The boot sector may be expected to contain a valid 8.3 format (blank-padded FCB) filename in the area of the boot sector starting behind the (E)BPB, extending up to below the boot sector signature word with value AA55h (at offset 510 in the boot sector). This name should not contain blanks other than trailing in the file name portion or trailing in the file extension portion. It should consist of printable ASCII codepoints. That is, byte values between 20h and 7Eh inclusive. It should not consist of eleven times the same byte value. Additional FAT Short File Name restrictions may be assumed.

Although a loader should not depend on this for crucial operation, it may want to detect the kernel name it was presumably loaded from for informational or optional purposes. The canonical implementation of this is currently the function ‘`findname`’ in the `testpl.asm` test payload kernel. It is found within the ldosboot repo. This handling is based on the function of the same name in the instsect application.

1.1.6 Query patch support

The ldosboot repo includes a patch Script for lDebug (.sld) file which allows to patch the initial loader stage. The patches concern handling of the CHS geometry detection, and whether LBA or CHS access is used. There are several legacy patch sites in which patch.sld can directly patch the initial loader's code.

However, the preferred way is to find the query patch sequence. It should appear within the first 1536 bytes, that is within the part of the initial loader that must be loaded. This is the sequence:

```
8A5640  mov dl, byte [bp + 40h]
B8xxyy  mov ax, yyxxh
84D2    test dl, dl
7902    jns @F
86C4    xchg al, ah
@@:
```

The immediate word of the `mov ax` instruction is to be patched. The sequence should be scanned for without regard as to what the current contents of this word are.

The following flag values are used:

- 01h Force CHS access, do not detect LBA support with 13.41

- 02h Force LBA access, do not detect LBA support with 13.41
- 04h Force use of BPB's CHS geometry, do not detect with 13.08
- 80h Used by lDebug. If this value is set for the load unit, then lDebug will make use of the other flags set up for that unit. The corresponding flags will be saved in lDebug's load_unit_flags. This affects only the load unit (LD in lDebug terminology), which suffices to pass commands in the startup Script for lDebug.

The flag 01h takes precedence over 02h if both are set.

The low byte (xxh) is used in case the loader loads from a diskette unit, that is a unit number below 80h. The high byte (yyh) is used otherwise, in case the loader loads from a hard disk unit, that is a unit number above-or-equal 80h.

1.2 Iniload to payload protocol

The payload is loaded to an arbitrary segment. The segment must be at least 60h. The entire payload must be loaded. The size of the payload is determined at iniload build time. The entrypoint is found by applying a segment adjustment and choosing an offset. The segment adjustment is specified at iniload build time by the numeric define `_EXEC_SEGMENT` (default 0), and the offset by the define `_EXEC_OFFSET` (default 0).

1.2.1 Stack

The stack must allocate at least 256 bytes of space below `ss:sp`. This space must not underflow, ie `sp` is at least 256 (100h). `sp` should not be higher than 8192 (2000h). `bp` is typically equal to `sp + 120h`. `bp` must be below or equal 7000h. The stack starting at `ss:0` up to below `ss:bp + 21Ch` should be in the area between the paragraphs indicated by the load top lower limit and memory top upper limit. (Refer to LOADDATA fields ldLoadTop and ldMemoryTop.) Subsequent stages like inicomp, fdkernpl, nullpl, and kernels like lDebug and testpl may depend on these characteristics.

1.2.2 Extended BIO Parameter Block (EBPB)

Above the LSV, `ss:bp` points to an EBPB and surrrounding boot sector. Note that this is always a FAT32-style EBPB. If the filesystem that is loaded from is not FAT32, and is therefore FAT16 or FAT12, then the FAT16/FAT12 BPBN structure is moved up. It is placed where the FAT32 BPBN is usually expected. In this case, the entire boot sector contents behind the BPBN are also moved up by the size of the FAT32-specific fields. The FAT32-specific fields are filled with zeros, except for the FAT32 ‘sectors per FAT’ field. It is filled with the contents of the FAT16/FAT12 ‘sectors per FAT’ field.

Note that a FAT32 file system is determined by whether the FAT12/FAT16 ‘sectors per FAT’ BPB field is zero. If it is zero, then the EBPB is used unchanged from the boot sector. If it is nonzero, the FAT12/FAT16 BPB is expanded into an EBPB as described here. To do the expansion, typically the BPB New fields of the FAT12/FAT16 BPB as well as the trailing boot sector contents (up to below offset 512) are moved up the 1Ch bytes to open the gap for use by the FAT32-specific EBPB fields. That means the boot sector trailer reaches up to below offset 21Ch (540).

1.2.3 Load Stack Variables (LSV)

Refer to section 1.1.3.

1.2.4 Load Data 1 (LD)

Below the LSV, iniload passes the LOADDATA (1) structure.

```
        struc LOADDATA, LOADSTACKVARS - 10h
ldMemoryTop:    resw 1
ldLoadTop:      resw 1
ldSectorSeg:    resw 1
ldFATType:      resb 1
ldHasLBA:       resb 1
ldClusterSize:  resw 1
ldParaPerSector:resw 1
ldLoadingSeg:
ldQueryPatchValue:
                resw 1
ldLoadUntilSeg: resw 1
        endstruc
```

ldMemoryTop

Word. Segment pointer to behind usable memory. Points at the first of the EBDA, RPL-reserved memory, or video memory or otherwise UMA. Indicates how much memory may be used by a typical kernel. (lDebug detects the EBDA to move that below where it installs.)

ldLoadTop

Word. Segment pointer to lowest lDOS boot memory in use. All memory between linear 600h and the segment indicated here is usable by the payload. Only the payload itself is stored in this area. The other buffers, stack, and structures passed by iniload must live above this segment.

ldSectorSeg

Word. Segment pointer to an 8 KiB transfer buffer. It is insured that this buffer does not cross a 64 KiB boundary. This may be needed by some disk units. The buffer is not initialised to anything generally.

ldFATType

Byte. Indicates length of FAT entry in bits. 12 indicates FAT12, 16 FAT16, 32 FAT32. It is planned to allow zero for non-FAT filesystems.

ldHasLBA

Byte. Only least significant bit used. Bit on indicates LBA extensions available for the load disk unit. Bit off indicates LBA extensions not available.

ldClusterSize

Word. Contains amount of sectors per cluster. Unlike the byte field for the same purpose in the BPB, this field can encode 256 (EDR-DOS compatible) without any masking. May be given as zero for non-FAT filesystems.

ldParaPerSector

Word. Contains amount of paragraphs per sector. Must be a power of two between 2 (32 B/s) and 200h (8192 B/s). May be given as zero for non-FAT filesystems.

ldLoadingSeg

Word. Internally used by iniload. Available for re-use by payload. However, ldQueryPatchValue re-uses the same field.

ldQueryPatchValue

Word. Passes the query patch value from the initial loader. This provides an opportunity to patch a well-known site in the initial loader to change its behaviour in some ways. Near the end of its operation, the initial loader passes along this value in this variable for the next stage to use.

ldLoadUntilSeg

Word. Internally used by iniload. Available for re-use by payload.

1.2.5 Load Command Line (LCL)

Below the LOADDATA structure, iniload passes the LOADCMDLINE structure.

```
lsvclBufferLength       equ 256

        struc LOADCMDLINE, LOADDATA - lsvclBufferLength
ldCommandLine:
.start:         resb lsvclBufferLength
        endstruc
```

This buffer is always initialised to an ASCIZ string. At most 255 bytes may be initialised to string data. At most the 256th byte is a zero.

If the first word of the buffer is equal to 0FF00h, that is there is an empty command line the terminator of which is followed by a byte with the value 0FFh, then no command line was passed to iniload. Currently lDebug can pass a command line to iniload when loading with its lDOS, RxDOS.2, RxDOS.3, or FreeDOS protocols. When iniload is loaded as a Multiboot1 or Multiboot2 specification kernel, it is also assumed that a command line can be passed.

1.3 Application mode protocol

An iniload payload can be loadable as a DOS application. In this case the iniload MZ executable header will point to the payload. The _IMAGE_EXE define to iniload will indicate support for this if it is enabled.

The entire payload will be loaded as a program image just behind the PSP. The entrypoint is determined by the _IMAGE_EXE_CS and _IMAGE_EXE_IP defines. These default to a -16:256 + 64 entrypoint. Minus 16 means the code segment will be equal to the PSP, just like for flat .COM style executables. The 256 in the instruction pointer calculation skips past the PSP. Thus, the default _IMAGE_EXE_IP will enter the payload at offset 64 counting within the payload.

Likewise _IMAGE_EXE_SS and _IMAGE_EXE_SP specify the initial stack. They default to -16:0FFFEh, mimicking flat .COM style load somewhat. (However, iniload doesn't get a chance to push a zero word onto this stack before the first payload is run. Adding a nullpl, checkpl, or inicomp stage can be used to provide this.)

_IMAGE_EXE_AUTO_STACK defaults to zero. If it is nonzero it indicates to use the automatic stack placement. Its value, which defaults to 2048 if defined as empty, determines how large the stack should be (at least). The explicit _IMAGE_EXE_SS and _IMAGE_EXE_SP are ignored if the automatic stack placement is enabled. The automatic stack is placed into its own segment, stretching from ss:0 to ss:sp. The stack pointer will match the value of _IMAGE_EXE_AUTO_STACK.

_IMAGE_EXE_MIN indicates how much memory to allocate to the process at least. It defaults to 65536 (64 KiB). It is actually used in the _IMAGE_EXE_MIN_CALC define to calculate the minimum allocation value. _IMAGE_EXE_MIN_CALC is not a numdef, but can be overridden from the assembler command line. This is the default content of this define:

```
%define _IMAGE_EXE_MIN_CALC \
(((_IMAGE_EXE_MIN \
- (payload.actual_end - payload) \
- 256 \
+ _IMAGE_EXE_AUTO_STACK) + 15) & ~15)
```

_IMAGE_EXE_MAX is a define that defaults to 0FFFFh. If it is nonzero it is used directly as the maximum allocation value. 0FFFFh indicates to allocate the largest memory block to the process, just like for flat .COM style load. If it is zero, then the maximum allocation field is set to equal the minimum allocation field.

Most of the stages of iniload payloads can pass through the execution control flow from being loaded as a payload to hand off to their payload. This includes the nullpl, checkpl, and inicomp stages. Each stage must be configured appropriately, enabling the _IMAGE_EXE define and setting up the other defines appropriately. These three stages expect -16:256 + 64 or -16:256 entrypoints as input for application mode, but their output entrypoints can be configured using the defines listed above.

The registers ds, es, and ax are passed through from iniload, nullpl, checkpl, or inicomp to each one's payload. The two segment registers will point to the PSP. The ax register contains information on the validity of the drives of the two default FCBs.

The iniload MZ executable header's relocation table is currently always empty. The application has to implement relocations by itself.

_SECOND_PAYLOAD_EXE allows to include a different payload that is to be used as the application mode program image. In this case the main iniload payload (_PAYLOAD_FILE) is used only for boot loaded mode. A different file can be specified with _SECOND_PAYLOAD_FILE. Each _IMAGE_EXE_* define has an equivalent define named according to the _SECOND_PAYLOAD_EXE_* scheme.

1.4 Device mode protocol

The device mode load of the payload shares its program image with the application mode. The very beginning of the image must contain a device header. (The first device header, if more than one exist.)

The inicomp, nullpl, and checkpl stages expect certain offsets for the entrypoints: 18 for the strategy, and 22 for the interrupt. Each passthrough stage must be configured with the appropriate device name and device attributes contents, and with the _DEVICE define enabled. (These defines are not needed for iniload. However, _IMAGE_EXE is required.)

The inicomp stage uses the request header's break address as passed by DOS to determine how much memory is available to the depacker. This requires MS-DOS version 5 compatibility.

Section 2: DRLoad boot protocols

The drload stage is a replacement for lDOS's iniload initial loader. It does away with most of the entrypoints supported by iniload, including the native lDOS load protocol. This allows it to drop most of the file system and disk read related code that is included in iniload.

The motivation for drload was to minimise the overhead for a compressed Enhanced DR-DOS single-file kernel, hence the name. This allows the single-file kernel to be smaller than the sum of the two files of the prior build of a double-file kernel.

2.1 Sector to drload protocol

The sector to drload protocol is either the FreeDOS load protocol or the original Enhanced DR-DOS load protocol (minus the need for a second file). Both of these share the fact that the prior loader has to load the entire file of the drload stage, so that no temporary file system support is needed in the drload stage or later in the subsequent kernel stages.

That does imply that the kernel file may not carry appended data of arbitrary size, as the entire file has to fit in the space within the Low Memory Area allocated by the prior loader.

2.1.1 Overview

The FreeDOS and Enhanced DR-DOS load protocols share:

- The entire file is loaded to a segment boundary
- The stack must not overlap the loaded file data
- The load unit can also be found in a BPB New field of the passed boot sector
- The hidden sectors can be found in a BPB field of the passed boot sector

They differ in the following details:

- FreeDOS:
  - File loaded at linear address 600h, entered at 60h:0
  - SS:BP -> boot sector with BPB (FAT12/FAT16) or EBPB (FAT32)
  - BL = load unit
- Enhanced DR-DOS:
  - File loaded at linear address 700h, entered at 70h:0
  - DS:BP -> boot sector with BPB (FAT12/FAT16) or EBPB (FAT32)
  - DL = load unit

2.2 DRLoad to payload protocol

The drload to payload protocol only sets up:

- The boot sector with FAT32 EBPB or expanded FAT32-style FAT12/FAT16 BPB
- The stack at the high end of the Low Memory Area
- The Load Command Line
- The following Load Data 1 fields:
    - ldMemoryTop
    - ldLoadTop
    - ldQueryPatchValue
- The entire next stage's payload image at a paragraph boundary

This happens to be enough to run a kernel-only inicomp stage, as well as the drkernpl stage that sets up the DRBIO and DRDOS modules and runs the former.

Source Control Revision ID

hg 85a55eda605d, from commit on at 2024-01-02 22:09:11 +0100

If this is in ecm's repository, you can find it at https://hg.pushbx.org/ecm/ldosboot/rev/85a55eda605d