Brief Guide to Creating Patched Bootloaders

The essence of the patch is that instead of the command 11 handler (which normally displays the useless message 'poweroff not supported'), we introduce our code into the bootloader's address space, allowing us to execute arbitrary code (up to 2K in size). The patch consists of three components:

- The code for the command handler.
- A chipset identification block placed immediately after the command handler.
- Another chipset identification block added at the end of the bootloader.

The code for the handler is a short program in machine code, and you can find its source in the 'loaders/pexec_*.asm' file. There are versions of the program for translation into ARM or THUMB-2 code, depending on the mode used in the bootloader's command handlers.

The first identification block is intended to determine the chipset type for all tools in the qtools suite (except qdload) that interact with the bootloader. This eliminates the need to use the '-k' flag in the command line. The block has the following format:

```
  .word  0xdeadbeef  ; Signature for programmatic block searching
  .byte  id          ; 1-byte chipset code assigned to it in the chipset.cfg config file.
```

The second identification block is only used by the qdload program. It allows the program to determine the loading parameters and chipset type directly from the bootloader file, eliminating the need to specify the '-k' and '-a' flags. This block is created using the 'setident' utility located in the 'loaders/' directory.

So, the patching process begins with determining the load and command 11 handler addresses. For this purpose, the 'qblinfo' program is used. Run it, specifying the bootloader file name in the command line, and you will get something like this:

```
$ ../qblinfo NPRG9x35p.bin
** NPRG9x35p.bin: 95704 bytes

 Download address: 02a00000
 Code Start Address: 02a00028
 CMD 01 = 02a00c41
 CMD 03 = 02a00d95
 CMD 05 = 02a00e67
 CMD 07 = 02a00e75
 CMD 09 = 02a00ef5
 CMD 0b = 02a00e23
 CMD 11 = 02a00f2d
 CMD 13 = 02a00f71
 CMD 15 = 02a00fc1
 CMD 17 = 02a011fd
 CMD 19 = 02a0122b
 CMD 1b = 02a012a1
 CMD 1d = 02a00f47
 CMD 20 = 02a01399
 CMD 30 = 02a0132d
 CMD table offset: 16a38
 Invalid CMD handler: 02a00c33

 Unsigned code or no HW_ID value in Subject field
```

From all this magnificence, we are interested in three addresses: the load address (02a00000), the start of the code address (02a00028), and the address of command 11 handler (02a00f2d). Now we load our file into IDA starting from address 2a000000, making sure to specify the ARM Little Endian processor type. If auto-analysis doesn't start automatically, initiate it manually and indicate that the code begins at the code's starting address (02a00028).

Next, navigate to the address of the command 11 handler (02a00f2d). Here's an approximate fragment of what you can expect to see around the address of interest:

```
CODE_ALL:02A00EF4 ; =============== S U B R O U T I N E =======================================
CODE_ALL:02A00EF4
CODE_ALL:02A00EF4
CODE_ALL:02A00EF4                 EXPORT handle_sync
CODE_ALL:02A00EF4 cmd_07_sync                             ; DATA XREF: APP_RAM:02A16A60#o
CODE_ALL:02A00EF4                 PUSH            {R4,LR}
```

```
CODE_ALL:02A00EF6                 ADDS            R1, R0, #6
CODE_ALL:02A00EF8                 LDRH            R0, [R0,#4]
CODE_ALL:02A00EFA                 SUBS            R0, R0, #1
CODE_ALL:02A00EFC                 UXTH            R4, R0
CODE_ALL:02A00EFE                 LDR             R0, =reply_buffer
CODE_ALL:02A00F00                 MOVS            R2, #0xA
CODE_ALL:02A00F02                 STRB            R2, [R0]
CODE_ALL:02A00F04                 MOVW            R2, #0x3FD
CODE_ALL:02A00F08                 CMP             R4, R2
CODE_ALL:02A00F0A                 BLS             loc_2A00F10
CODE_ALL:02A00F0C                 BL              buffer_overflow_error
CODE_ALL:02A00F10 ; ---------------------------------------------------------------------------
CODE_ALL:02A00F10
CODE_ALL:02A00F10 loc_2A00F10                             ; CODE XREF: handle_sync+16#j
CODE_ALL:02A00F10                 ADDS            R1, R1, #1
CODE_ALL:02A00F12                 ADDS            R0, R0, #1
CODE_ALL:02A00F14                 MOV             R2, R4
CODE_ALL:02A00F16                 BL              hostdl_memcpy
CODE_ALL:02A00F1A                 LDR             R1, =escape_state
CODE_ALL:02A00F1C                 ADDS            R0, R4, #1
CODE_ALL:02A00F1E                 STRH            R0, [R1,#8]
CODE_ALL:02A00F20                 BL              compute_reply_crc
CODE_ALL:02A00F24                 BL              force_xmit_reply
CODE_ALL:02A00F28                 MOVS            R0, #0
CODE_ALL:02A00F2A                 POP             {R4,PC}
CODE_ALL:02A00F2A ; End of function handle_sync
CODE_ALL:02A00F2A
CODE_ALL:02A00F2C
CODE_ALL:02A00F2C ; =============== S U B R O U T I N E =======================================
CODE_ALL:02A00F2C
CODE_ALL:02A00F2C
CODE_ALL:02A00F2C                 EXPORT handle_power_off
CODE_ALL:02A00F2C cmd_11_power_off                        ; DATA XREF: APP_RAM:02A16A80#o
CODE_ALL:02A00F2C                 PUSH            {R4,LR}
CODE_ALL:02A00F2E                 MOVS            R0, #0x12
CODE_ALL:02A00F30                 LDR             R1, =reply_buffer
CODE_ALL:02A00F32                 STRB            R0, [R1]
CODE_ALL:02A00F34                 MOVS            R0, #1
CODE_ALL:02A00F36                 LDR             R1, =escape_state
CODE_ALL:02A00F38                 STRH            R0, [R1,#8]
CODE_ALL:02A00F3A                 MOVS            R0, #0x15
CODE_ALL:02A00F3C                 ADR             R1, aPowerOffNotSup ; "Power off not supported"
CODE_ALL:02A00F3E                 BL              transmit_error
CODE_ALL:02A00F42                 MOVS            R0, #0
CODE_ALL:02A00F44                 POP             {R4,PC}
CODE_ALL:02A00F44 ; End of function handle_power_off
```

Here is the code for not only the command 11 handler but also the preceding command 09 (sync) handler. In this handler, we need to find two important addresses:
- reply_buffer: This is where the image of the packet sent to the host at the end of command processing is written.
- escape_state: This is the area for parameters of the response packet, and we only need the field where the length of the response packet is recorded. So, here's where:

The length of the response packet is written here. So, from here:

```
CODE_ALL:02A00EFE                 LDR             R0, =reply_buffer
CODE_ALL:02A00F00                 MOVS            R2, #0xA
CODE_ALL:02A00F02                 STRB            R2, [R0]
```

From here, we take the address of reply_buffer. And from here:

```
CODE_ALL:02A00F1A                 LDR             R1, =escape_state
CODE_ALL:02A00F1C                 ADDS            R0, R4, #1
CODE_ALL:02A00F1E                 STRH            R0, [R1,#8]
```

we take the address of escape_state, along with the offset where the packet length is stored (in this case, 8). But we don't need the addresses in their raw form. We need to find the block in memory that these literals point to and determine its address. We go to the address of reply_buffer and look for all references to this address. Some references will be of the form DCD reply_buffer. It's this DCD located in the address space after the command 07 and 11 handlers that we need. It looks like this:

```
CODE_ALL:02A01098 off_2A01098     DCD escape_state        ; DATA XREF: handle_hello:loc_2A00CA0#r
CODE_ALL:02A01098                                         ; handle_simple_read:loc_2A00E12#r ...
CODE_ALL:02A0109C off_2A0109C     DCD reply_buffer        ; DATA XREF: handle_hello+6E#r
CODE_ALL:02A0109C                                         ; handle_simple_read:loc_2A00DDA#r ...
```

Here, you can find consecutive addresses for escape_state and reply_buffer.
Finally, we need to determine the exit address for the handler. For this, we use the code of the command 07 handler, which is located above the command 11 handler, specifically this code:

```
leavecmd:
CODE_ALL:02A00F20                BL              compute_reply_crc
CODE_ALL:02A00F24                BL              force_xmit_reply
CODE_ALL:02A00F28                MOVS            R0, #0
CODE_ALL:02A00F2A                POP             {R4,PC}
```

We'll designate the address of this code for clarity as leavecmd.
Now we're ready to obtain the binary image of our patch. Open the file "pexec_arm.asm" or "pexec_thumb.asm" in your editor, depending on whether the processor in the bootloader's command handlers operates in ARM or THUMB-2 mode. Next, insert our addresses into the file:

```
pkt_data_len_off=8                          ; Insert the packet length offset here
                .ORG            0x02a00f20  ; Insert the "leavecmd" address here
leavecmd:
                .ORG            0x02a00f2c  ; Insert the command 11 handler address here
```

Now, add the chipset code into the identification block:

```
        .byte       8       ; Insert the chipset code
```

And, finally, specify the address of the escape_state and reply_buf literals:

```
                .ORG  0x02a01098
reply_buf_ptr:    .word  0
escape_state_ptr: .word  0
```

That's it. Now, assemble our assembly source using any version of the ARM assembler (AS) for ARM architecture.

```
$ arm-none-androideabi-as -o /dev/null -f=pexec_thumb.lst pexec_thumb.asm
```

Open the "pexec_thumb.asm" listing and insert all the code for your command 11 handler directly into the IDA database (Edit → Patch Program → Change Byte). Now, export the database to the resulting output file (File → Produce File → Create EXE File). To distinguish the patched bootloader from the unpatched one, add the letter "p" to the end of the file name, for example, "NPRG9x35p.bin."

The final step is to insert the second identification block into the bootloader:

```
$ ./setident NPRG9x35p.bin 8 02a00000
```

Here, we specify the chipset code (8) assigned to it in the "chipset.cfg" file and the load address (02a00000). That's it! The process of creating a patched bootloader is now complete.