

Compromising the IC_204 ECU

Flashing arbitrary, unsigned firmware

JinGen Lim

████@████

June 2021

Abstract

Automotive ECUs employ signature checks in their firmware update mechanism to prevent unauthorized software modifications. An implementation flaw is present in the flash loader of the IC204 ECU (of Mercedes-Benz vehicles, circa 2011), where an adversary may bypass the firmware verification and load any arbitrary firmware.

1 Background

The IC204 is a series of instrument clusters that are manufactured by VDO, and are frequently found in MB vehicles of around 2007-2013. There are several variants, with key visual differences in the number of dial faces and display type (monochrome and full color). Internally, the main microcontroller typically uses a NEC (V850) processor.

These devices are designed to be tamper resistant as they store sensitive content such as odometer data. These protections also have a side effect of impeding repairs, as used devices cannot normally be paired with another compatible vehicle, and is therefore a popular target for reverse engineers.

1.1 Target Device

The vulnerable target in this document is a IC212_IC_204_Mid.Line_AJ10 (identifying as 0x1307) from a MB C300 (2011), and may be relevant for other IC204 variants that include the same processor.

ECU	IC_204
Variant Name	IC212_IC_204_Mid_Line_AJ10
Variant ID	0x1307
Connection	HSCAN_UDS_500
Boot Version	09/31.00
Hardware	2049013800 (09/44.00)
Software #0	2049022702 (10/16.00)
Software #1	2049022802 (10/16.00)

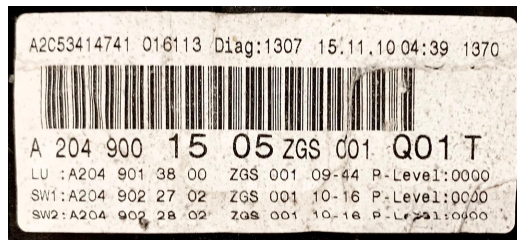


Figure 1: MB Part Number of the device under test

If you are familiar with the IC_204 architecture and flash process, you may wish to skip straight to the "Flashing Unsigned Firmware" section.

1.2 Architecture

The heart of this specific cluster is a NEC μ PD70F3426GJ¹, with a RISC V850E architecture. The processor includes 2048 kbytes of internal flash, 84kbytes of internal RAM, and a bunch of helpful memory-mapped peripherals such as I2C, CAN, stepper drivers and LCD interfaces.

The IC204 also has several protection features enabled to prevent unauthorized code-readout:

- External flash read/write is disabled.
- External flash blocks cannot be individually erased².
- The debug port is protected by a 10-byte password which can only be written with an external programmer, and cannot be read out at runtime.

¹Datasheet search keywords: "UPD70F3426GJ" and "RNCCS17265-1"

²Precludes the possibility of erasing a region of interest (e.g. clearing security flags)

1.3 Memory Layout

The memory layout for this μC is linear, with its ROM from $0\text{x}0$ to $0\text{x}200000$, and RAM starting at $0\text{x}3\text{FF}0000$. By default, its reset vector points to $0\text{x}0$. For this specific device, two software blocks are available for flashing with official CFF flash files:

- Block #0: 2049022702, spanning from $0\text{x}9000$ to $0\text{x}\text{F}2000$, containing code and sometimes referred to as APPL (Application).
- Block #1: 2049022802, spanning from $0\text{x}\text{F}2000$ to $0\text{x}200000$, containing non-executable data.

The region from $0\text{x}0$ to $0\text{x}9000$ is conspicuously absent in the flash files, and cannot be written by the internal flash programmer. This region is never modified for the device's lifetime after it is programmed in the factory, and essentially behaves like a bootrom.

1.4 Logical Blocks

Within each software block, the data is further partitioned into logical blocks, which appear to have their own header and footer structure. Most blocks do not have public names, so these are guessed in relation to their behavior:

```

001F4F80 5A 5A 5A 5A FF FF FF FF 00 00 00 00 00 00 00 00  ZZZZyyyy.....
001F4F90 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..
001F4FA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..
001F4FB0 00 00 00 00 00 00 00 00 00 00 00 00 00 04 70 00  .....p.
001F4FC0 31 30 31 35 36 35 39 33 41 46 00 00 FF FF FF FF  10156593AF..yyy
001F4FD0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  yyyyyyyyyyyyyyyy
001F4FE0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  yyyyyyyyyyyyyyyy
001F4FF0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  yyyyyyyyyyyyyyyy
001F5000 6B 0D 00 00 3C 50 1F 00 50 50 1F 00 40 50 1F 00  k...<P..PP..@P..
001F5010 54 50 1F 00 00 00 00 00 00 00 00 00 2C 50 1F 00  TP.....,P..
001F5020 00 02 02 00 00 00 00 00 02 00 02 00 53 6F 75 6E  .....Soun
001F5030 64 56 65 72 73 69 6F 6E 30 00 FF FF 6B 0D 00 00  dVersion0.yyk...

```

Figure 2: Boundary of a logical block

- BOOTROM: First stage that starts from processor reset vector. This stage is technically writable, but never appears to be updated.
- BOOT: Second stage loader, likely capable of executing from RAM³. Appears to be part of software block #0.

³Requirement for self-programming

- APPL: Executable application code running cluster-specific tasks.
- DATA/BITMAP/SOUND: Multiple data blocks storing non-executable data (e.g. strings)

1.5 Boot Procedure

On reset, the processor starts executing from its reset vector at 0x0, which immediately jumps into the first stage where basic processor peripherals such as timers are initialized. Execution is subsequently transferred to later stages after they are verified.

1.5.1 Trust Chain

Each stage verifies the integrity of its subsequent stage by checking specific memory regions for the presence of a verification flag.

These flags are only written after clearing a signature check in the final stage of the flash process, after the firmware verification routine is completed. The absence of this flag indicates a failed or interrupted flash process, or an invalid firmware checksum. In such a situation, the ECU will fail to boot normally.

As an example, when the verification flag is invalid for the DATA section only, the ECU will behave normally on the CAN bus and is capable of entering an extended session. However, the LCD will only show a visual error such as "Reprogramming not correct!". For failure in more critical stages, the ECU will remain in a DFU mode (Boot_Variante), which responds to a minimum command set that is required to flash a working firmware.

1.5.2 Firmware Verification Routine

During the flash process, untrusted firmware data is directly written to the non-volatile flash of the uPD70F3426 as it is being received. The verification routine steps in at the end to ensure that the device can only boot with signed and unmodified firmware. Upon successful verification, a 4-byte flag is written to the headers of specific blocks to mark it as trusted.

The verification routine takes in a checksum+signature pair. The checksum ensures firmware integrity, and the signature is checked with an embedded public key to ensure its authenticity.

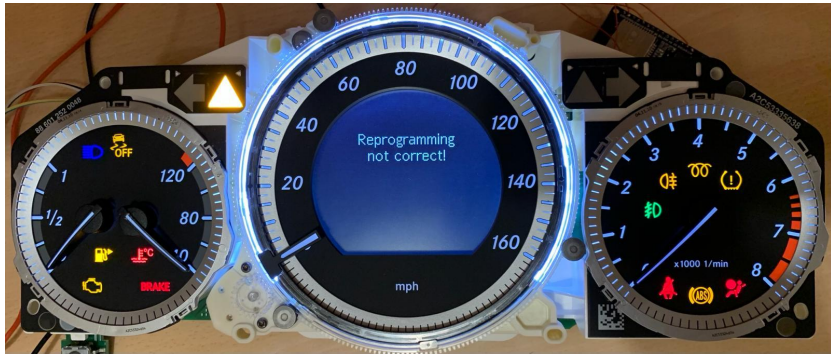


Figure 3: Data verification error on the IC204

1.6 Standard Flash Procedure

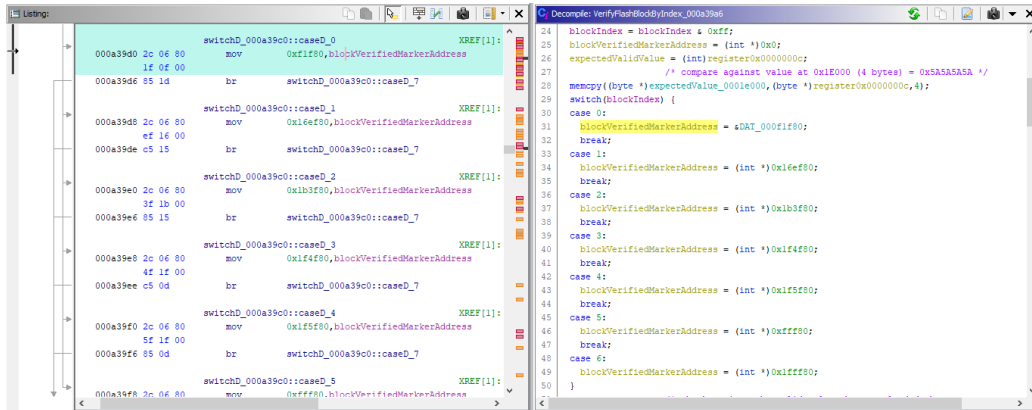
This is an overview of a typical flash process:

1. Enter boot session
1002
2. Authenticate with seed/key to enable flash commands
2705, 2706
3. Write fingerprint, select active block, then erase block
2EF15A, 3101FF00
4. Configure connection for flashing (longer timeout, no tester-present)
5. For each block..
 - (a) Request to download to a specific memory address: 3400
 - (b) Transfer the data blocks: 36
6. Call verification routine with checksum and signature
3101FF01
7. Restore original connection configuration
8. Hard reset: 1101

As ECU flash processes are not fully standardized, every CBF and SMR-D diagnostics file that supports flashing capabilities will always include a flash script that is tailored for its target.

2 Flashing Unsigned Firmware

For this specific IC204 ECU/firmware, there are 7 logical blocks that require a valid verification value; should this value be empty or incorrect, the ECU will refuse to boot normally. These values are always erased when starting the flash process, and are written only after a successful verification routine.



```
Listing:
switchD_000a39c0::case_0      XREF[1]:
000a39d0 2c 06 80      mov     0x1f1f80,blockVerifiedMarkerAddress
000a39d6 1f 0f 00      lf 0f 00
000a39de 85 1d          br     switchD_000a39c0::case_7

switchD_000a39c0::case_1      XREF[1]:
000a39dc 2c 06 80      mov     0x16ef80,blockVerifiedMarkerAddress
000a39e2 ef 16 00      ef 16 00
000a39de c5 15          br     switchD_000a39c0::case_7

switchD_000a39c0::case_2      XREF[1]:
000a39e0 2c 06 80      mov     0x1b3f80,blockVerifiedMarkerAddress
000a39e8 3f 1b 00      3f 1b 00
000a39de c5 15          br     switchD_000a39c0::case_7

switchD_000a39c0::case_3      XREF[1]:
000a39e8 2c 06 80      mov     0x1f4f80,blockVerifiedMarkerAddress
000a39ee 4e 1f 00      4e 1f 00
000a39de c5 0d          br     switchD_000a39c0::case_7

switchD_000a39c0::case_4      XREF[1]:
000a39f0 2c 06 80      mov     0x1f5f80,blockVerifiedMarkerAddress
000a39f8 5f 1f 00      5f 1f 00
000a39de 85 0d          br     switchD_000a39c0::case_7

switchD_000a39c0::case_5      XREF[1]:
000a39f8 2c 06 80      mov     0xffff0,blockVerifiedMarkerAddress

Decompile: VerifyFlashBlockByIndex_000a39ac
24  blockIndex = blockIndex + 0xff;
25  blockVerifiedMarkerAddress = (int *)0x0;
26  expectedValidValue = (int)register0x0000000c;
27  memcopy(byte *)expectedValue_0001e000,(byte *)register0x0000000c,4);
28  switch(blockIndex) {
29  case 0:
30  blockVerifiedMarkerAddress = *DAT_000f1f80;
31  break;
32  case 1:
33  blockVerifiedMarkerAddress = (int *)0x16ef80;
34  break;
35  case 2:
36  blockVerifiedMarkerAddress = (int *)0x1b3f80;
37  break;
38  case 3:
39  blockVerifiedMarkerAddress = (int *)0x1f4f80;
40  break;
41  case 4:
42  blockVerifiedMarkerAddress = (int *)0x1f5f80;
43  break;
44  case 5:
45  blockVerifiedMarkerAddress = (int *)0xffff0;
46  break;
47  case 6:
48  blockVerifiedMarkerAddress = (int *)0x1fff80;
49  }
50 }
```

Figure 4: Disassembly of the block verification function.

In this example, the block verification reads 4 bytes at the addresses 0x0F1F80, 0x16EF80, 0x1B3F80, 0x1F4F80, 0x1F5F80, 0x0FFF80 and 0x1FFF80. The result is then checked against the expected value at 0x1E000, which is a constant value of 0x5A5A5A5A.

2.1 Vulnerability

Due to an implementation flaw, it is possible to write the firmware verification flag directly onto the flash and completely bypass the verification routine, as write attempts to sensitive memory regions are not adequately checked during the flash process.

2.2 Exploit

A flashing tool can be modified to use this exploit by writing the expected flag value (in this case, 0x5A5A5A5A) into the 7 check addresses. This will replicate the behavior of a successful verification routine and skip the 3101FF01

request which is no longer required. This exploit requires some prerequisite conditions to work correctly:

- ECU must be in boot session and authenticated for security level 3
- The write command must be within the block's boundary;
For example, attempting to write in DATA (e.g. 0x1F5F80) while flashing an APPL block will fail.

One possible approach is to merge the 0x5A5A5A5A constants into the existing flash bytes, then flash the new content onto the ECU. This technique is used by the proof-of-concept, which uses a custom flasher to deliver the flash payload together with the exploit.

3 Proof of Concept



Figure 5: Successfully running unsigned firmware on the IC204

As a proof-of-concept, a modified flash file has been prepared and flashed into a vulnerable IC204 with the above technique. To demonstrate execution of modified code, authentication for security levels 5 and 7 have been altered to accept any key [Figure 6]. With this privilege escalation, the external EEPROM can be accessed with read and write capabilities. Some bitmaps for fault warnings and gear state have also been replaced to differentiate itself as an unsigned firmware. [Figure 5].

3.1 Impact

- Compromise of update mechanism
(Flashing an ECU with arbitrary firmware)⁴
- Broken Authentication
(PoC: security level 7, maximum for IC204)
- Unlocking/Modifying vehicle functions

As this is a flash-related vulnerability, *physical access is required* and there is *no immediate safety risk* to existing owners of MB vehicles equipped with an IC204 ECU.

⁴This section is mostly paraphrased from <https://www.daimler.com/whitehat/>

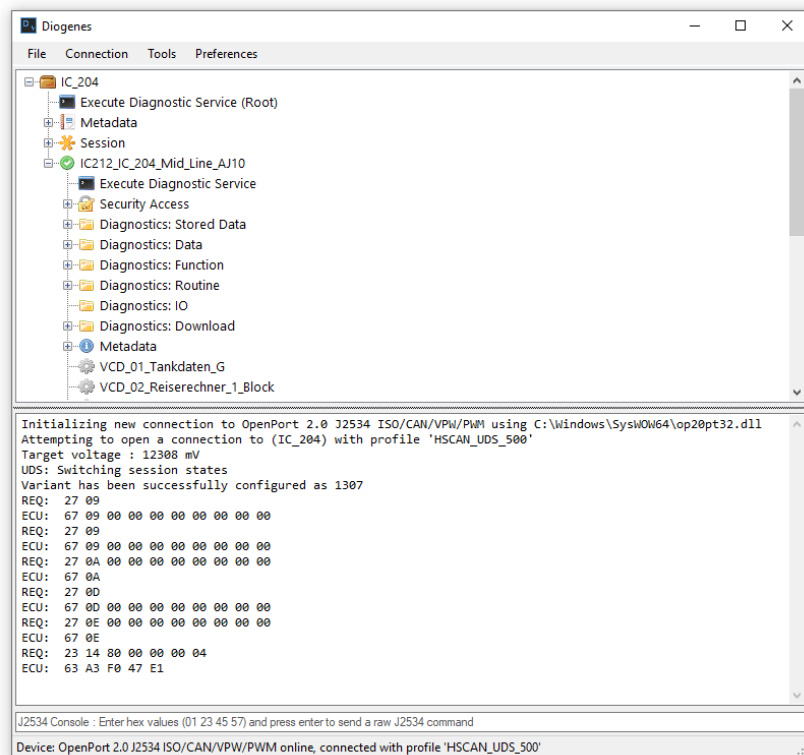


Figure 6: Privilege escalation to level 7 and reading the onboard EEPROM

3.2 Mitigation

The flash loader should be updated to track and reject write attempts into sensitive memory regions such as the verification flags. This may be applied through a future firmware update.

3.3 Conclusion

The IC204 flash loader generally observes good security practices, with a minimal command set which reduces its attack surface. Most common checks are properly implemented, e.g. ensuring that writes are rejected unless the ECU is erased to prevent flipping specific flash bits upwards. There are also bounds checks to ensure that writes are contained within the erased block, which is a documented exploit for Volkswagen ECUs [1].

The signature checking strategy can be improved with an additional check at flash time, to ensure that the verification addresses are protected. MB/VDO may also wish to review other ECUs of similar architecture to check for the presence of this class of vulnerabilities.

References

- [1] Brian Ledbetter. VW Flashing Tools over ISO-TP / UDS. URL: https://github.com/bri3d/VW_Flash/blob/master/docs/docs.md.