Efficient Reverse Engineering of Automotive Firmware

Alyssa Milburn and Niek Timmers
{milburn,timmers}@riscure.com

Abstract

The firmware executed by components found in a car provide a starting point for adversaries to obtain confidential information and discover potential vulnerabilities. However, the process of reverse engineering a specific component is typically considered a complex and time-consuming task. In this paper we discuss several techniques which we used to significantly increase the efficiency of reverse engineering the firmware of an instrument cluster. Using this example target, we demonstrate it is fairly easy to implement an emulator which is capable of emulating the target's firmware entirely without the need for the original hardware, including many essential components of the target such as the EEPROM, display controller and CAN bus. Our implementation allows standard Linux tooling to be used to send CAN messages to the target. Using this emulator we were able to efficiently understand the target's functionality, recover secrets (e.g. UDS keys) and perform fuzzing to identify vulnerabilities.

1 Introduction

Reverse engineering exists since humankind invented engineering. It is the process of deconstructing a man-made object in order to extract knowledge from it; in particular, it allows the architecture and implementation of products to be reconstructed, including confidential aspects. Reverse engineering is applicable to all forms of engineering, and the automotive industry is no exception.

In principle, the entire manufacturing process of an entire car can be reversed which potentially allows reconstruction of the entire vehicle without the need for access to any restricted design information. Obviously, reversing an entire car requires a significant amount of time, skill and expert tooling not available to most. However, reversing a specific component is certainly something that is within the capabilities of a determined adversary; the example target we present in this work is an instrument cluster from a modern car.

To reconstruct the entire target, an adversary is required to reverse both the target's hardware and software. This research focuses on reverse engineering the functionality implemented in software, which we typically refer to as the target's firmware. Reverse engineering of the target's firmware enables an adversary to learn about the design of the target, discover the functionality used by and exposed by the device, and extract secrets stored in the firmware. It also potentially allows exploitable vulnerabilities to be identified.

Reverse engineering a large complex code base is often considered to be a time-consuming task. Our research shows that reverse engineering the functionality of many automotive devices can be performed efficiently by emulating the entire firmware. More concretely, we show that emulation is an effective method to understand the available functionality, extract secrets and analyze the firmware for vulnerabilities.

2 Extracting firmware

Obtaining the firmware itself is a prerequisite for any analysis or emulation. Although this is outside the scope of this work, publicly available research has shown that there are numerous ways to extract this firmware:

- **Protections are not enabled**: The processors used in modern cars include functionality to prevent firmware extraction, in particular disabling debug facilities (e.g. JTAG). These protection mechanisms are typically turned off during development and often not enabled

until late in the manufacturing process. If these protections remain disabled on hardware which is available to the public, the firmware can simply be read using these interfaces.

- **Firmware is stored externally**: Firmware stored on an external storage chip (e.g. EEPROM/flash) may simply be readable by an adversary. If the firmware is stored in encrypted form, so that the plain-text firmware is not directly available to an adversary, software vulnerabilities or side-channel attacks may allow the decryption key to be obtained, allowing an adversary to decrypt the firmware.
- **Firmware is extracted using software attacks**: The firmware may include vulnerabilities that can be used to extract the firmware itself, or may simply allow the firmware to be read using a standard communication interface (e.g. CAN). All software includes bugs, and some of these bugs will inevitably result in exploitable vulnerabilities.
- **Firmware is extracted using hardware/physical attacks**: Firmware stored in internal memory inside a processor can potentially be extracted using a physical attack where the micro-controller's package is opened using chemicals in order to access the micro-controller's internals directly. Alternatively, hardware attacks such as fault injection could be used to re-enable debug functionality which can be used to obtain the firmware.
- **Firmware is leaked**: Finally, the firmware might be obtained independently of the deployed products; for example, an employee of the manufacturer might leak the firmware, or it might be obtained by a hacker that penetrated the manufacturer's network.

Once the firmware of a target device has been extracted, further analysis can be performed on the firmware rather than using the target, so the attacks described above only need to be performed once for each target.

3 Case study: analysis of an instrument cluster

The target firmware used to demonstrate our research is from an instrument cluster, which is designed around a chip implementing a 32-bit RISC CPU architecture that is commonly used in the automotive industry. For reasons of confidentiality, we will not identify the specific processor architecture, nor details of the specific cluster we use as an example here; our experience is that the analysis approach described here can be performed on a wide variety of processors and automotive devices.

The processor in our target is connected to several instruments (e.g. gauges, lights and a segmented display) as well a variety of sensors and the immobilizer, and communicates with other components within the car using the CAN bus. An external non-volatile memory chip (EEPROM) is used to store data. The firmware itself is stored in internal non-volatile memory, inside the processor package. We found no evidence that the external memory is used to store code or any other parts of the firmware.

The chip's architecture is supported by commercial reverse engineering tools (e.g. IDA [4]) and open source reverse engineering tools (e.g. Radare2 [5]), although we needed to add support for some more obscure and/or recent instructions. We initially attempted to perform static analysis of the firmware, but it became clear that important parts of the data and code for the device were not stored in plain text. Some of the functionality identified in the firmware hinted to compression and/or encryption. Manual reverse engineering of this code was error-prone and time-consuming; we started investigating more efficient methods.

Analysis of similar instrument clusters has revealed the presence of information which might make static analysis easier for an adversary, such as partition/section offsets and debugging strings; the lack of such information in our target firmware increased the difficulty of static analysis significantly. An adversary who obtained the firmware using debug interfaces or by performing attacks may instead be able to make a copy of the contents of memory after the target has booted; however, this approach also has limitations and importantly may provide an inaccurate or incomplete view (e.g. if hardware configuration is performed as part of this boot phase, or if different code/data is loaded in other situations).

We concluded that dynamic analysis would be required for efficient reverse engineering of the instrument cluster firmware. If an adversary has access to debug functionality and/or development boards for the relevant processor, it may be possible to perform dynamic analysis without emulation; however, our experience is that emulation provides a more effective tool for dynamic analysis where possible. In our case, development of a customized emulator was necessary, since no standard tooling was available to emulate the firmware – typical emulation tools (e.g. qemu [6] and unicorn [7]) do not support the target's processor architecture.

4 Implementing our own emulator

We implemented the minimum hardware support required for complete emulation of the firmware of the instrument cluster. In this section, we provide a high-level summary of our implementation of the emulator itself, followed by a discussion of the dynamic analysis functionality it provides and the advantages this brings to the reverse-engineering process.

4.1 The emulator

We spent approximately 10 man days of time writing the necessary software, including implementing the dynamic analysis features described below and debugging the code so it correctly emulated the target's firmware. The core functionality of the emulator (including the peripherals/interfaces described below) is implemented in approximately 3000 lines of C. We did not attempt to accurately emulate features which were not necessary for our target firmware, such as accurate emulation of the CPU pipeline and caches. Such functionality could be implemented if it were found to be necessary for specific firmware to function correctly.

We used publicly-available data sheets that describe the functionality of the target processor in detail. Depending on the target, this type of information might not be available to an adversary and writing an emulator would be significantly more complicated. However, our experience is that the necessary information can often be found in leaked documentation, leaked software or by investing (significantly) more time into the reverse-engineering process.

4.1.1 Core peripherals

The target's firmware uses a large number of hardware registers, but many can be ignored without any apparent adverse effect on the functioning of the firmware (for example, power management registers) and others were successfully implemented using *stubs* which ignore writes and always return a fixed value. We bootstrapped the emulator by outputting warning messages when unknown reads/writes were performed, and adding the necessary functionality only as required. However, an implementation of both the interrupt controller, as well several different timer peripherals, were required for the target firmware to successfully execute the boot process.

4.1.2 I2C interface

The target's processor communicates with two other chips that are populated on the PCB using its I2C interface: the EEPROM and the display controller. We emulated the functionality of the EEPROM and display controller which were necessary for our target's firmware. We needed to provide a reasonably complete emulation of the EEPROM, including both reads and writes. However, only a limited implementation of the display controller was necessary; it appears to be sufficient to simply ACK the writes over the I2C interface. Since the I2C interface is external to the processor, we were able to use a logic analyzer on the instrument cluster to confirm that the bus traffic was identical for the real hardware and the emulator – this approach can also be used for debugging during development.

4.1.3 CAN interface

We implemented emulation of the CAN controller peripheral used by our target, including reception and transmission of all supported message frames. This emulated controller can connect to Linux's

SocketCAN, using either a virtual CAN bus or a physical CAN dongle. This allows us to use the same tools to communicate with both the emulated device and the actual target itself, and would allow an adversary to replace the original instrument cluster with a flexible and controllable emulator-driven instance of the target, which simplifies reverse engineering of other components (e.g. gateways).

4.2 Dynamic analysis

An adversary who is able to emulate the target's firmware is provided with several advantages compared to static analysis of the firmware. In this section we introduce these advantages in further detail, by discussing the relevant functionality which we implemented in our emulator.

4.2.1 Execution tracing

The most obvious benefit of dynamic analysis is the ability to capture *traces* of the execution of the firmware. This allows us to observe which exactly instructions are executed, and which data is present in registers and memory, at each point in time. We captured a baseline trace of execution and colored the relevant instructions in IDA Pro, allowing us to instantly visually observe which instructions were executed during normal operation. We can then perform a second, *different* execution, where we make a change to the execution such as sending a specific CAN message or changing the contents of memory. This produces a different execution trace, which we can compare against the original trace; this makes it easy to identify which portions of the firmware are responsible for handling specific tasks, and which regions of memory store the relevant data. These differences can then in turn also be visualized in a static analysis tool, or used as input for further analysis (e.g. fuzzing).

4.2.2 Taint analysis

We identified that minor changes of input data sometimes result in large changes in the contents of memory, making it more difficult to interpret the differences between memory traces. Despite this, we needed to determine the flow of data for the "interesting" parts of the firmware. This data can both originate from user input (e.g. CAN bus message handling) or specific locations in (external) memory. A technique typically referred to as *Taint Analysis* [1] is capable of doing exactly that. It runs code and observes which computations are affected by a predefined taint source. It does this by *tainting* the source of the data with a specific tag, and keeping track of the tags associated with every memory location and/or register. When program flow depends on a tainted value, such as a conditional branch which uses a value which was calculated based on tainted data, then this branch can be marked as having a dependency on the input data.

We implemented taint analysis in our emulator, which we found to be very helpful for quick understanding of data flow. However, practical problems mean that some manual guidance can be necessary for this process. For example, tainting individual bits of input (e.g. reads from the EEPROM) requires a large number of different tags (slowing the emulator), but tainting at a coarser level (e.g. byte granularity) is insufficient for following the flow of some data. The completeness of the taint propagation – for example, how taint values are combined for some arithmetic operations, or whether taint is propagated when a load is performed from a tainted pointer – can be customized as necessary for each situation, which we found to be necessary for avoiding tainted values from spreading too far in some situations.

4.2.3 State rewinding

Since the entire state of the firmware is contained within the emulator, we added support for saving and restoring the state of the firmware. This allows us to save the state of the firmware once it has already booted, and *rewind* it back to the previous state after performing every test, without the risk that we change the state in an unexpected way or trigger a timeout mechanism, and without having to wait for the device to boot.

In particular, this allows us to quickly perform brute-force attacks, such as modifying the contents of ranges of memory to determine which are important to a specific function, or perform

penetration testing on an external interface (such as fuzzing the CAN bus, see below). An attempt of even a complex test can typically be performed within a few milliseconds (timer manipulation can be applied if necessary), and such testing scales well since the runs can trivially be parallelized.

4.2.4 Fuzzing

Execution traces and taint analysis give us the tools we need to observe whether a given input has resulted in an *interesting* change to the execution of the firmware. However, significant manual effort is still required to discover the inputs themselves, despite the advantages provided by our emulation approach. Modern fuzzing tools such as American Fuzzy Lop (AFL) [8] provide a way to discover these inputs in a more efficient manner than brute-forcing potential inputs, modifying input in random ways in an attempt to reach new paths within the firmware. Since taint analysis allows us to directly determine which comparisons are made on the input, in some cases we can automatically create mutated input which *passes* a given comparison (i.e. causes a given branch to be taken, or not to be taken).

We developed a simple fuzzer which used these approaches to fuzz inputs such as the EEPROM contents and incoming messages on the CAN bus. However, human interaction is still currently required to interpret the results of the fuzzing; inputs which result in new/unexpected register writes or the execution of new regions of code are highlighted for manual investigation. As such, we generally found the less automated dynamic analysis described above to be a more efficient method for reverse engineering the functionality of the target; the use of directed fuzzing tools which understand the functionality of the target allow us to discover the relevant code.

5 Firmware analysis

We captured a full memory dump of the contents of the emulated memory after the boot was complete, and loaded it into the IDA Pro disassembly software to allow us to analyze the firmware statically. If required, it would be possible to obtain dumps at many different stages of execution (keeping track of memory changes as discussed below); however, this was not necessary for our target firmware. The remainder of this section describes the different avenues that were explored using the emulator and related functionality.

5.1 Analysis of the EEPROM usage

The information stored in the EEPROM includes static vehicle-specific information such as the *Vehicle Identification Number* (VIN), as well as persistent data such as the vehicle mileage and security state. From an adversary's point of view it is interesting to identify what is stored where and how different data is stored. By tracing the flow of execution before and after the EEPROM is accessed, we were able to discover where certain data is stored. Data read from EEPROM was spread across multiple locations and obfuscation was applied which we will not publicly discuss.

Applying taint analysis again to the memory locations used to store the data after this reading process was complete, we observed code which iterated over entire 'block' of EEPROM data and producing a single value, which appears to be the checksum algorithm used to verify the data's integrity. Rather than investigating this code in further detail, we simply determined which functions were responsible for reading and writing EEPROM data by examining the runtime behavior of the firmware. Then we were able to successfully use the relevant code from the firmware directly ('lifting' the functions outside their context) to read and write to the EEPROM directly.

5.2 Analysis of the CAN communication

The target firmware sends and receives a variety of CAN packets; by observing the CAN ID filters applied by the firmware, we could easily determine which ranges of packet IDs were potentially relevant and could be interesting to analyze further. Rather than brute-forcing these ID ranges, we tainted the bytes representing the CAN IDs of packets by tainting the relevant reads from the CAN controller registers. Combined with execution tracing to discover when new paths were taken, this allowed us to observe which specific IDs were of interest.

We ran open source CAN penetration testing tools (e.g. 'Caring Caribou' for UDS [9]) against a virtual CAN bus connected to the emulator, and observed that packets sent using these IDs influenced the behavior of the device. For example, many changed I/O outputs (e.g. corresponding to turn indicators or the speedometer) of the processor, or produced a CAN packet in response. Other packets only changed the internal state (e.g. memory) of the emulated device, without producing an externally visible result. Using the dynamic analysis techniques described above allowed us to discover which of these responses and/or changes were relevant to our analysis; as an example of how we approached this problem, we will discuss the UDS handling of the target firmware.

5.3 Analysis of UDS communication

The target supports the commonly used Unified Diagnostic Services (UDS) communication protocol. The relevant IDs were correctly discovered by running 'Caring Caribou' (via SocketCAN) on the emulated firmware. Again, taint analysis was applied to efficiently determine the 'interesting' values for diagnostic sessions and security access IDs. Note that such results must be taken with caution, since the code paths may change when the device is in different states such as before/after security authentication.

Our first challenge was to discover the correct keys (responses) needed for the UDS security access service. The simplest approach was simply to taint the input values for the relevant (key) bytes of the CAN packet; the taint analysis would then find the conditional which was used to check whether this matched the correct response, and we could then just restore the original state and provide the correct response value instead, allowing us to obtain access to restricted UDS services. Since we control all sources of entropy (such as the timing, sensor inputs and EEPROM contents) of the emulated device, we only needed to do this once, since the seed (challenge) was always identical on all other execution runs.

We also confirmed that other approaches were possible. Static analysis of the region of code identified by the taint analysis revealed the function which calculated the response, which allowed us to re-implement the algorithm ourselves. We could also replace the seed in memory with a different value (such as that requested by a real device) and use the emulated firmware to calculate the correct response, without needing to perform any such analysis.

Finally, we also discovered by observation that the timeout (of several minutes) after incorrect authentication attempts would only be enforced if we allowed the firmware to write to the EEPROM to indicate that an attempt had failed; blocking EEPROM writes allows us to attempt to brute-force the correct value without needing to wait for the timeout period after each attempt. This attack can also be successfully performed on the original device, by blocking the EEPROM writes on the exposed I2C bus.

5.4 Analysis of the display controller

Communication with the display controller appears to be one-way, with messages being sent over the I2C interface containing control messages for display segments. We used dynamic analysis to discover (by searching for the memory containing the transmitted buffer, and then observing which code wrote to that memory), and then performed manual static analysis to decode the algorithm and structures used for these messages. Our emulator decodes the messages sent on the I2C interface by referencing the same structures; during the dynamic analysis phase we also hooked the relevant display functions in the firmware directly.

Although not strictly necessary for our analysis, and not part of our emulator itself, this was the missing piece we needed to develop a complete simulator for the instrument cluster. We can send CAN messages to the emulator and visually observe the effect of the output on both the display and other components such as the speedometer, and compare the results when sending the same messages to the real dashboard device.

6 Conclusion

We conclude that, with limited effort and publicly available tooling and skills, we have demonstrated that it is possible to emulate the entire firmware of automotive devices. We have presented a case study of the emulation and reverse engineering of a instrument cluster. Emulation enables an adversary to analyze the firmware dynamically, which speeds up the reverse engineering process significantly. An adversary can "peek and poke" the firmware arbitrarily, since the emulation is done entirely independently of the original target device. This approach enables an adversary to significantly speed up the process of analyzing the implementation of modern automotive firmware.

Many of the techniques we have described are already widely known to the public. These techniques are used by adversaries when reverse engineering, fuzzing or analyzing both embedded or non-embedded software. A variety of research is available which could also be applied to automotive devices. Techniques such as symbolic execution [2] allow interfaces and other code to be more efficiently understood, which may be particularly interesting for adversaries who are interested in discovering security-related vulnerabilities. Combining these methods has even more promising results; for example, combining taint tracking and symbolic execution has been used to 'guide' fuzzing [3]. Researchers have developed many automated methods for deobfuscation of code, which may be useful for reverse engineering obfuscated portions of automotive firmware.

These efficient analysis techniques are only possible for adversaries that have full control of the firmware executed by the target device. Therefore, we recommend manufacturers operating in the automotive industry to increase the difficulty of doing so. For example, it is possible to increase the complexity of extracting the firmware if the software and hardware of automotive products is hardened against clever adversaries. Experience has shown that even then, these defenses are not sufficient to prevent dedicated attackers from obtaining plain text firmware, which means that reverse engineering of firmware may be inevitable. Rather than relying on confidentiality of firmware, we recommend using dedicated hardware security (e.g. cryptographic engines, secure storage, etc.) to protect a target's most sensitive secrets, such as cryptographic keys; the necessary functionality is provided by many modern system-on-chips (SoCs) intended for automotive purposes.

Acknowledgments

We thank Eloi Sanfelix Gonzalez, Ramiro Pareja Veredas and Santiago Córdoba Pellicer for their help and contributions.

References

[1] Edward J. Schwartz, Thanassis Avgerinos and David Brumley; All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask), Proceedings of the IEEE Symposium on Security and Privacy (2010)

[2] King, James C; Symbolic execution and program testing, Communications of the ACM (1976)

[3] Istvan Haller, Asia Slowinska et al.; Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations, USENIX Security Symposium (2013)

[4] IDA Pro `https://www.hex-rays.com/products/ida/`

[5] Radare2 `https://rada.re/r/`

[6] QEMU, the FAST! processor emulator `https://www.qemu.org/`

[7] Unicorn – The Ultimate CPU emulator `http://www.unicorn-engine.org/`

[8] american fuzzy lop (2.52b) `http://lcamtuf.coredump.cx/afl/`

[9] Caring Caribou – A friendly car security exploration tool `https://github.com/CaringCaribou/caringcaribou`