There Will Be Glitches: Extracting and Analyzing Automotive Firmware Efficiently

Alyssa Milburn
Vrije Universiteit Amsterdam
a.a.milburn@vu.nl

Niek Timmers
Riscure
timmers@riscure.com

Nils Wiersma
Riscure
wiersma@riscure.com

Ramiro Pareja
Riscure
pareja@riscure.com

Santiago Cordoba
Riscure
cordoba@riscure.com

Abstract

Automotive security is a hot topic, and hacking modern cars is cool. These cars are suffering the growing pains seen in many embedded devices: security is a work-in-progress, and in the meantime we see some fun and impressive hacks. Perhaps the most well-known examples are the Jeep and Tesla hacks. But, we know that the automotive industry is paying attention. Consider a bright future where secure boot methods have been universally implemented, without obvious bugs; adversaries no longer have access to plain-text firmware, ECUs refuse to run any unsigned code, and we feel safe again. Will automotive exploitation be *mission impossible*, or do hackers still have a few tricks up their sleeve?

In this paper we discuss hardware attacks, like fault injection, which can be used to efficiently extract automotive firmware from secured ECUs. These attacks do not rely on an exploitable software vulnerability. Access to the plain-text firmware allows an attacker to understand the ECU's functionality, extract the ECU's secrets and identify exploitable software vulnerabilities. We describe multiple techniques in order to analyze binary firmware efficiently. We use an instrument cluster from a modern car to demonstrate the practicality of the described techniques on a real ECU. Finally, we explain the real-world impact of these issues, how they lead to scalable attacks, and what can be done to secure the cars of today and tomorrow.

1 Introduction

Cars consist of Electronic Control Units (ECU). Therefore, to hack a car, you need to hack one or more ECUs. A typical attack that compromises the security of an Electronic Control Unit (ECU) consists of three phases: *Understanding*, *Vulnerability Identification* and *Exploitation*. The goal during the *Understanding* phase is to understand just enough to move on to the *Vulnerability Identification* phase. Additionally, the *Understanding* phase allows attackers to extract information and/or secrets from the firmware. In this paper we focus on performing the *Understanding* phase efficiently. The *Vulnerability Identification* and *Exploitation* phase are something for another day.

We bought a wide variety of ECUs from *Ebay* from modern cars in order to analyze their security. The ECUs that are sold through *Ebay* are often from crashed cars and therefore we can buy them for a fraction of the retail price. One of those ECUs, an instrument cluster, is the target we use to demonstrate the techniques described in this paper. We perform several fault injection attacks in order extract its firmware and then we analyzed its firmware efficiently using emulation. Today's standard embedded technology is not resilient against hardware fault injection attacks. This includes the embedded technology used by the automotive industry. These attacks alter the intended behavior of chips by manipulating the chip's environmental conditions. Often this is accomplished by manipulating the voltage supplied to the chip. However, more advanced techniques use electromagnetic or optical pulses. Fault injection attacks are nowadays performed by the masses as tooling and knowledge is available in abundance. In Section 3.1 these attacks described in more detail.

In Section 2 with a description of the target that we used to demonstrate the practicality of the techniques described in this paper. We describe an efficient technique to extract firmware from secured ECUs in Section 3. We describe efficient techniques to analyze the extracted firmware in Section 4. We discuss the scalability of hardware attacks in Section 6 and provide insights into hardening ECUs in Section 5. We finalize the paper with an overall conclusion in Section 7.

2 The Target

We use an instrument cluster from a modern cars to demonstrate the practicality of the techniques presented in this paper. The instrument cluster of a car provides the driver with information about the current state of the car (e.g. speed, mileage, etc.). The information publicly available about this instrument cluster is limited. We did not find much more than the pin-out of the instrument cluster's connector to the car's harness. Nonetheless, after opening the instrument cluster, we identified the main microcontroller (MCU) and found its datasheet to be leaked on the *Internet*. The instrument cluster is designed around a MCU that implements a processor with a 32-bit RISC architecture which is commonly used by the automotive industry. The instrument cluster communicates with the rest of the car using its CAN bus. An external EEPROM chip is used to store data. The instrument cluster's firmware is stored inside the MCU on internal flash. This instrument cluster will from now on be called the *target* throughout this paper.

3 Firmware Extraction

The target's firmware is not available using official channels. The available firmware upgrades are encrypted and we did not find the firmware available elsewhere. The firmware is stored inside the MCU and therefore standard manufacturer tooling can be used to extract it. However, this debug interface is secured and therefore could not be used to extract the firmware. Therefore, we started exploring other paths in order to extract the firmware from the target for which the results are described in the remainder of this section.

3.1 Fault Injection

We used fault injection attacks to extract the firmware from the target. These attacks alter a target's intended behavior by manipulating its environmental conditions [13]. This can be accomplished using different fault injection techniques such as Voltage Fault Injection [7] [10], Electromagnetic Fault Injection [7] [11] and Optical Fault injection [7] [9]. In this paper we focus primarily on Voltage Fault Injection. The manipulation of the target's environmental conditions is from now on referred to as the *glitch*. The behavioral alteration in the target is from now on referred to as the *fault*.

Specialized tooling is required in order to perform fault injection. This tooling is nowadays available to the masses due to affordable open source initiatives [8]. We used commercially available tooling that allows better control of the injected glitches [12]. We use a FPGA based glitcher that is capable of setting an arbitrary voltage signal and a CAN bus communication device that allows us to set a trigger signal once we send out a command. The command we send depends on the functionality we want to access. By reverse engineering the PCB, and the leaked datasheet that we downloaded, we were able to identify the power pins and reset pin of the target's MCU. We removed several components in order to be able to control the voltage supplied to the MCU arbitrarily. Additionally, the decoupling capacitors are removed, as they could flatten the voltage glitches. A logical representation of the setup is shown in Figure 1.

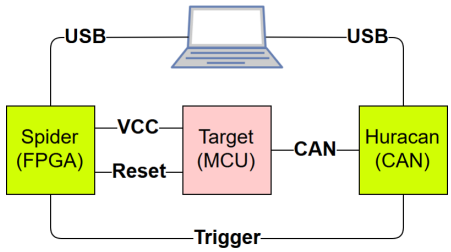

Figure 1: Voltage Fault Injection setup.

There are several parameters important when performing *Voltage Fault Injection*. These parameters are visualized in Figure 2. In green the *Voltage* supplied to the target is shown. The dip in this signal is the *Glitch*. The amplitude of the dip and the width of the dip determine the strength of the glitch. We time glitch using a *Trigger* signal that is set once the command is send out on the CAN bus which is shown in blue. The timing of the *Glitch* is commonly referred to as the *Glitch Delay*. The command and response are shown in red.

Figure 2: Voltage glitch.

Typically a divide-and-conquer strategy is used to find glitch parameters that result in successful glitches. Optimally these glitch parameters are identified by performing a characterization test where we inject glitches into a fully controlled test application. Unfortunately we did not have this luxury and therefore we were required to identify successful glitch parameters in the dark. We used behavioral indicators in order to see the light in the dark. For example, we used alterations in the responses and alterations in the power consumption to create this light.

3.2 Attacking UDS

The UDS protocol is a diagnostics protocol standardized in ISO 14229 [6] and progressively adopted by the automotive industry. It is implemented in most ECUs found in a modern car. The UDS standard describes functionality that is interesting for hackers like accessing the internal memories of an ECU. Therefore, the UDS stack on an ECU is a perfect entry point for hackers. If the UDS implementation complies with the standard, access to the memory and firmware of the ECU is protected. The client must established privileged session in order to use security critical services. The *SecurityAccess* service implements a seed-key authentication scheme where the client requests a seed from the ECU to derive a key using an algorithm and a secret. The algorithm and secret are known to both the client and the ECU. A logical representation of this algorithm is shown in Figure 3.

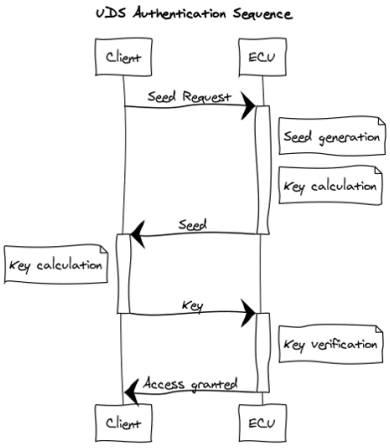

Figure 3: SecurityAccess check seed-key algorithm.

The UDS standard describes the functionality of UDS but it does not provide recommendations on the strength of the *Security Access* check. Therefore the security level of the *Security Access* checks used by the automotive industry depends on the implementers themselves. This results in a wide variety of different implementations. We have identified implementations where attackers were able to brute force the key due to bad practices. Better implementations are secure and do not include vulnerabilities that can be exploited logically. Nonetheless, even though there are no software vulnerabilities, these implementation can be attacked using fault injection.

3.2.1 Attacking SecurityAccess

We initially set out to bypass the *SecurityAccess* check using fault injection. However, we found that a practical hurdle prevent us from performing this attack efficiently. It is essential for fault injection to be able to perform a huge number of experiments. However, as the UDS standard demands, a 10 minute timeout is implemented after three incorrect key guesses. This slows us down significantly as we can do only 3 key guesses within 10 minutes. Therefore, we started exploring the possibility to attack the UDS stack at another location for which the results are described in Section 3.2.2.

3.2.2 Attacking ReadMemoryByAddress

The target implements the *ReadMemoryByAddress* service which potentially allows us to read out its internal memory. No 10 minute delay is implemented for accessing this service. Other interesting commands that do not implement this delay are: *WriteMemoryByAddress*, *RequestUpload*, *RequestDownload* and *TransferData*. Whenever this functionality is accessed, the target will respond with an error code as we are not properly authenticated using the *SecurityAccess* check. We will focus on *ReadMemoryByAddress*, but a similar approach can be taken for the other commands.

The *ReadMemoryByAddress* service uses two parameters: *address* and *length*. Whenever this service is accessed, it will check first if the tester is properly authenticated using the SecurityAccess check. Then, if the tester is proven to be valid, the requested memory is returned. The maximum amount of bytes that can be returned is 0x40. This restriction is not enforced by the UDS standard and will therefore be different between different targets. Whenever this service is accessed without the right privilege level, an error code (NRC) is sent back to the tester by the server.

We performed an attack on the target where we aim to bypass the privilege level check using a voltage glitch. We do not know exactly when this check is implemented, but we do know it must be between the command is sent and the response comes back. The time between the command is sent via the CAN bus and the response is sent back via the CAN bus is 1700 microseconds. We focused our initial glitch campaign on this window. After performing 45,000 experiments, which took roughly one day, we observed several successful glitches. By narrowing down the glitch parameters, we achieved a maximum success rate of 3%.

We are able to extract 0x40 bytes per successful glitch and therefore we needed in total 8192 successful glitches to extract the target's 512 KB internal flash. After approximately 3 days and roughly 300,000 glitches, we extracted the entire firmware from the target. Now we are read to start analyzing the target's firmware for which the results are described in Section 4. A more efficient fault injection attack to extract the firmware is described in Section 3.3.

3.3 Attacking debug interfaces

Most MCUs include debug interfaces that can be used to access the their internals, including firmware. These debug interfaces are often protected from malicious by a password or simply by disabling them.

A logical representation of standard methods to implement debug interfaces on MCUs is shown in Figure 4.

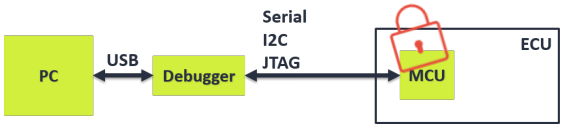

Figure 4: Typical debug interface.

Often a hardware fuse is used to enable the protection forever. The enforcement of this protection can be implemented in hardware and software. The authors of [14] already identified that software is used to enforce this protection. We have shown that software can easily be altered using fault injection attacks. Therefore, the protection of debug interfaces can be subverted using fault injection.

We used publicly available information in order to enable the target's debug interface by setting a specific external strap-pin pattern. Once the debug interface is enabled, we used the tooling available from the MCU manufacturer in order to figure out what commands to send to the debug interface in order to access the MCU's internals. Using Voltage Fault Injection, we were able to bypass the protections of the debug interface that were enabled. This allowed us to extract the target's firmware efficiently in the order of hours instead of days.

4 Firmware analysis

Firmware analysis is often revered to as reverse engineering. The target's processor's architecture is partially supported by commercial and open source reverse engineering tools (e.g. IDA Pro [2] and Radare2 [4]). Several instructions were not implemented and we were required to implement our own in order to analyze the firmware statically. For other, more common processor architectures, this tooling is better and less work is required before the analysis can start. We found the firmware to be very complex as it consists of multiple different code parts like operating system code, custom code and generated code. Additionally, the target's firmware is properly stripped from debug information which increases the analysis difficulty significantly. We concluded quickly after starting the static analysis of the firmware that we needed a more efficient technique. Analyzing the firmware statically is simply too time consuming and too error prone.

A better method for analyzing code is dynamic analysis which requires full control of the code executed on the target. On embedded systems, like ECUs, this is typically done using the debug interfaces. However, these debug interfaces are protected on the target and are not functional for us. Therefore, we started investigating the possibilities for emulating the firmware without needing the target's hardware itself. For our target, development of a customized emulator was necessary, since no standard emulation tooling (e.g. qemu [3] and unicorn [5]) was available for the target's processor architecture.

4.1 Firmware emulation

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. 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.

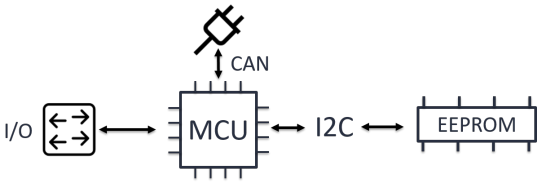

Figure 5: Firmware emulation.

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.

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.

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. 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.

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.

4.2 Advanced 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 describe several advanced features we implemented to speed up the analysis of the taret's firwmare.

4.2.1 SocketCAN

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).

We ran open source CAN penetration testing tools (e.g. *Caring Caribou* for UDS [1]) 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.

4.2.2 Debugging

We implemented a GNU Debugger (GDB) stub in order to implement debugging facilities using standard tooling. This allowed us to set software break points and single step through the code. Debugging is a standard tool or developers to analyze bugs in their code. However, in a similar fashion, this can be used by attackers analyze binary firmware.

4.2.3 Execution tracing

One limitation of analyzing binary code statically is that it takes significant effort in order to determine if certain branches are taken or not. Using the emulator, we can simply keep track of the locations in the code that are executed. This allows us to separate the code that is executed from the code that is not executed. This prevents us from analyzing code that is not executed.

4.2.4 Taint analysis

We implemented a technique, often referred to as taint analysis, in order to identify specific functionality within the target's firmware. For example, using this technique it is easy for us to identify what parts of the code are responsible for operating on the data of a CAN bus message. Simply said, we keep track on the reads and writes on the data of the CAN bus message. A logical representation of taint tracking is provided in Figure 6.

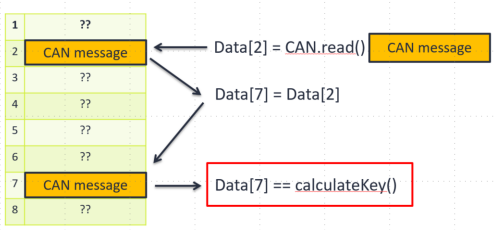

Figure 6: Taint analysis on incoming UDS command.

In this example we tracked the SecurityAccess key in order to identify the functionality responsible for deriving the key from the seed. 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.

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.

5 Scalability

A single successful fault injection attack does not scale well to multiple cars. The scalability is often brought up as an argument when the impact of fault injection attacks is discussed. It is a valid argument. Fault injection attacks do not scale well. However, one must realize that the results of fault injection attacks do scale. In Figure 7 we visualize how the extraction of firmware using fault injection results in scalable attacks.

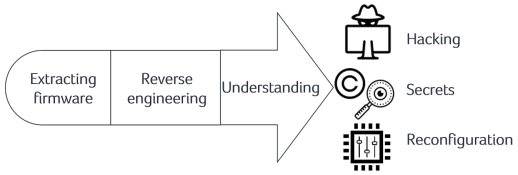

Figure 7: Scalable attacks

The firmware extracted using fault injection allows attackers to analyze the firmware in order to perform the *Understanding* phase of a full attack. This understanding can lead to the identification of exploitable vulnerabilities, extraction of secrets or knowledge required for reconfiguration. Fault injection attacks are really stepping-stone for attackers to be able to perform attacks that do scale.

6 Hardening ECUs

Is all hope lost? No! ECUs can be hardened against sophisticated attacks like fault injection. You should not rely on the secrecy of the firmware. It is essential that no secrets are embedded in the firmware. Secrets, like keys, should be properly stored securely using hardware security modules (HSM). These keys should be diversified between ECUs as well in order to minimize the impact when a key is extracted by an attacker. Authentication mechanisms should be implemented using asymmetric cryptography which means no secret is stored in the ECU at all. If there is no secret in the ECU, an attacker cannot extract that secret. Most importantly, the threat model of embedded devices, including ECUs, should be adjusted to incorporate hardware hackers. It is simply not true nowadays that an ECU is secure when there are no exploitable software vulnerabilities. Hardware attacks are a real threat and are performed by the masses.

7 Conclusion

We showed that sophisticated hardware attacks, like fault injection, are feasible, effective and not difficult. We used fault injection in order to alter the intended behavior secure software in order to completely subvert the software security model on which most ECUs rely. We used the result of a fault injection attack as a stepping-stone to perform several scalable attacks. We extracted the firmware of a secured ECU which allowed us to start performing the *Understanding* phase required to pull of a full attack.

We analyzed the extracted firmware efficiently without using the ECU's hardware. We used a custom emulator as the ECU's processor architecture is not supported by publicly available tooling. Our emulator allowed us to speed up the *Understanding* phase significantly. We are able to understand, extract secrets from, and identify specific functionality in the firmware efficiently.

Our recommendation for the automotive industry is: include hardware attacks in your threat model. Do not not rely on the secrecy of firmware, do not expose secrets to software and make use of hardware security where possible. Hackers will go to great lengths in order to extract the firmware of secure devices. We would like to finalize with a simple statement that has shown to hold up through time: *defense in depth* is key.

Acknowledgment

We thank all our friends and colleagues for their help. You all know who you are! :*.

References

[1] Caring caribou – a friendly car security exploration tool. `https://github.com/CaringCaribou/caringcaribou/`. [Online; accessed 2018].

[2] Ida pro. https://www.hex-rays.com/products/ida/. [Online; accessed 2018].

[3] Qemu, the fast! processor emulator. https://www.qemu.org/. [Online; accessed 2018].

[4] Radare2. https://rada.re/r/. [Online; accessed 2018].

[5] Unicorn – the ultimate cpu emulator. http://www.unicorn-engine.org/. [Online; accessed 2018].

[6] ISO 14229-1:2013. Iso. https://www.iso.org/standard/55283.html. [Online; accessed 2018].

[7] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *IACR Cryptology ePrint Archive*, 2004:100, 2004.

[8] NewAE Technology Inc. Chipwhisperer. https://newae.com/tools/chipwhisperer/. [Online; accessed 2018].

[9] Federico Menarini Jasper G. J. van Woudenberg, Marc F. Witteman. Practical optical fault injection on secure microcontrollers. *FDTC*, 2011.

[10] Colin O'Flynn. Fault injection using crowbars on embedded systems. *IACR Cryptology ePrint Archive*, 2016:810, 2016.

[11] Robert Van Spyk Rajesh Velegalati and Jasper van Woudenberg. Electro magnetic fault injection in practice. *International Cryptographic Module Conference (ICMC)*, 2013.

[12] Riscure. Inspector fault injection (fi). https://www.riscure.com/security-tools/inspector-fi/. [Online; accessed 2018].

[13] Niek Timmers and Cristofaro Mune. Escalating privileges in linux using voltage fault injection. *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 1–8, 2017.

[14] N. Wiersma and R. Pareja. Safety != security: On the resilience of asil-d certified microcontrollers against fault injection attacks. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 9–16, Sept 2017.