What the Stack? On Memory Exploitation and Protection in Resource Constrained Automotive Systems

What the Stack? On Memory Exploitation and Protection in Resource Constrained Automotive Systems

Aljoscha Lautenbach, Magnus Almgren, Tomas Olovsson

Chalmers University of Technology, Gothenburg, Sweden,
aljoscha@chalmers.se, magnus.almgren@chalmers.se,
tomas.olovsson@chalmers.se

Abstract. This is the authors' version of this paper. The final authenticated version is available online at https://www.doi.org/10.1007/978-3-319-99843-5_17.
The increased connectivity of road vehicles poses significant challenges for transportation security, and automotive security has rapidly gained attention in recent years. One of the most dangerous kinds of security relevant software bugs are related to memory corruption, since their successful exploitation would grant the attacker a high degree of influence over the compromised system. Such vulnerabilities and the corresponding mitigation techniques have been widely studied for regular IT systems, but we identified a gap with respect to resource constrained automotive systems.
In this paper, we discuss how the hardware architecture of resource constrained automotive systems impacts memory exploitation techniques and their implications for memory protection. Currently deployed systems have little to no protection from memory exploitation. However, based on our analysis we find that the simple and well-known measures like stack canaries, non-executable RAM, and to a limited extent memory layout randomization can also be deployed in this domain to significantly raise the bar for successful exploitation.

Keywords: Embedded System Security, Electronic Control Unit, Resource Constraints, Memory Exploitation, Memory Protection

1 Introduction

In the automotive domain, considerations of safety have a long tradition in vehicle development. Security considerations on the other hand, often called "cybersecurity" to distinguish from physical security, are still relatively new.

Several recent developments necessitate the introduction of security measures. One such development is the gradual introduction of "Intelligent Transport Systems (ITS)" which aims to improve traffic flow and safety through real-time information exchange between traffic participants and traffic infrastructure. Incidentally, with the advent of self-driving cars, the correct functioning of these

"Intelligent Transport Systems" will be crucial. Another factor is the emergence of the "Internet of Things" which opens the door to machine-to-machine communication and interoperability to facilitate completely new types of services, including automotive services such as remote diagnostics or smartphone apps to control vehicle functions. This offers a large attack surface to potential attackers.

Newly manufactured vehicles have around 50 - 100 electronic control units (ECUs), which are specialized microcontrollers of differing complexity, connected in smaller networks, forming one large internal network. The vulnerability of the in-vehicle network and their connected systems has consistently been highlighted and demonstrated by security researchers for several years [1–4]. The work by Miller and Valasek was particularly media-effective [5–9].

Meanwhile, in the world of desktop computers and servers, an arms race between memory exploit writers and memory protection developers has been going on for decades [10, 11]. Ever more advanced defensive techniques inspire ever more creative and complicated attacks. Due to the limited hardware capabilities and limited connectivity, automotive systems and other embedded systems have not been primary targets for such exploits. But with increased connectivity and computing capabilities, this is slowly changing. Therefore it is important to understand how effective such exploits could be in the automotive domain, and what trade-offs are required to deploy known mitigation techniques.

2 Resource Constrained Microcontrollers

Due to cost, power and size constraints, microcontrollers have limited capabilities and very particular architectures. In the following we will sketch the most typical hardware and processor architecture [12–16]. The specifications of a typical resource constrained microcontroller used for safety-critical automotive applications are listed in Table 1.

Table 1. Highlights of typical ranges of resource constrained microcontroller configurations

Hardware	Specification	Most Common
RAM	4 KB - 500 KB	40 KB
Flash Memory	256 KB - 6 MB	1 MB
Processor Speed	16 - 150 MHz	80 MHz

All modern microcontrollers (MCUs) support at least two different execution modes: privileged and unprivileged mode [16]. Applications generally execute in unprivileged mode, whereas the operating system executes in privileged mode.

There are several different types of memory: flash memory, for the boot loader, OS and other program code; data flash, for permanent data storage; RAM, for dynamic state information; and often there is memory-mapped I/O. The different kinds of memory are commonly mapped into a single, linear 32-bit address space. The concrete mapping is configurable. For instance, the flash

memory could be mapped into `0x00000000 - 0x009FFFFF`, and the RAM into `0x00A00000 - 0x00AFFFFF`, as depicted in figure 1.

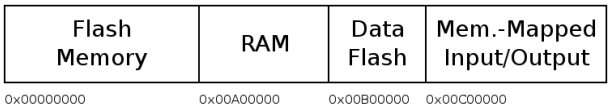

Fig. 1. An example of a linear memory address space mapping

The amount of available RAM is very limited. Every task has a *statically assigned memory region* dedicated to it; there is *no virtual memory* [16]. For instance, the first OS task could have its dynamic memory in the address range `0x00AFFFFF - 0x00AFFD01`, while the second task would have range `0x00AFFD00 - 0x00AFFC01`. This mapping is ordinarily *enforced by a memory protection unit (MPU)*, which is available on most microcontrollers. This unit keeps track which memory regions are readable, writable or executable. If, for instance, Task 2 tries to access the stack of Task 1, and they were configured to be in separate memory regions, the MPU will not allow the access. This is illustrated in figure 2. Note that the stack grows downwards, as usual.

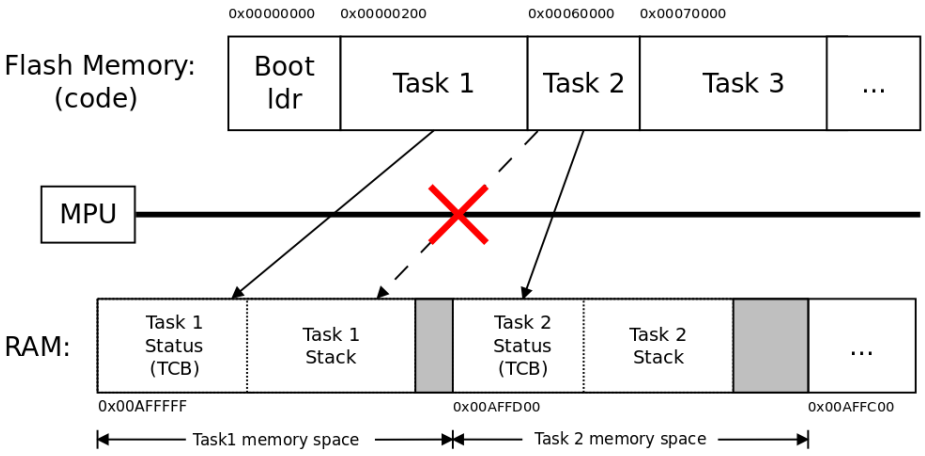

Fig. 2. Static task memory mapping into RAM

Two data structures are relevant for running tasks and thus kept in RAM: (1) the task control block (TCB), which keeps status information, and (2) the call stack. The common microcontroller architectures are link-register based, i.e., they keep track of the next return address in a special CPU register (the link-register `lr`) [16]. Before a function call, the return address (`lr`) is pushed on the

stack, just like on x86. There are some minor differences in stack-handling among the architectures: on ARM, for instance, parameters are loaded from registers rather than from the stack. Regularly, RAM is executable; this can be necessary to reprogram the flash memory outside a workshop, e.g., for unsupervised firmware upgrades.

The flash memory contains all program code: the boot loader, the operating system and application code. Since the RAM is too small to contain the operating system code and the code of the programs, everything is executed directly from flash. A context switch from one task to another then simply requires that the registers are stored for later retrieval, that the task status is changed and that the current execution mode is updated. Finally the program counter (`pc`) is changed to point to the task to be executed next.

To summarize the main differences with x86 type architectures in desktops and servers: there is no heap, there are no dynamically loaded libraries or shared objects and there is no virtual memory. Furthermore, the code is executed directly from flash, and *RAM is statically mapped*.

3 Exploiting Memory-Related Software Bugs and Protection Mechanisms

Given the architecture outlined in the previous section, we will now look into possible memory corruption bugs, how they can be exploited, and possible protection mechanisms.

3.1 Stack-based Buffer Overflows and Stack Canaries

During the design and development of automotive systems, great care is taken not to introduce memory corruption bugs due to their safety implications, for instance by enforcing the MISRA C guidelines which help to reduce software bugs. However, memory corruption bugs still occur occasionally, and when they do, they also pose a security risk. Using dynamic memory during run-time is generally forbidden in automotive software; all memory must be statically assigned. Therefore, most memory corruption bugs commonly found on regular PCs, such as heap-based memory corruption bugs or string-formatting errors, are not a priority. However, the most common type of memory corruption bug, stack-based buffer overflows [11], can occur.

An important observation is that *the principles of exploiting stack-based buffer overflows are the same for all common architectures*. As long as the return address is written to the stack, buffer overflows can be exploited by an attacker. Since the available RAM is rather limited, the attack code needs to be fairly compact, but this will not stop an attacker from finding useful exploits. Even an extremely short exploit which crashes the targeted task can be dangerous.

Stack canaries, also known as stack cookies or stack-guards [17], can be used to detect and prevent buffer overflow exploits. They write a random canary value before the return address on the stack, and before a function returns, the

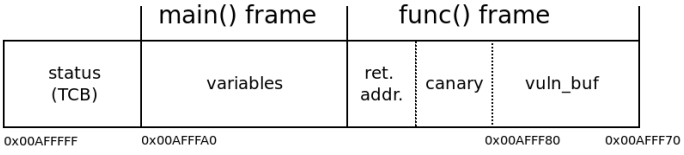

Fig. 3. Memory layout of a vulnerable task using a canary

canary value is validated. If the value changed, program integrity can no longer be guaranteed, and a memory exception is triggered. Figure 3 depicts a simple stack using a canary: if the vulnerable buffer `vuln_buf` in `func()` overflows, the canary in front of the saved return address will be overwritten, and the program will be aborted before the potential attack code written to `vuln_buf` can be executed. This comes at the expense of performance, but it is an effective safeguard. There are techniques to circumvent canaries [11], but they make successful exploitation of stack-based memory corruption bugs harder.

3.2 Non-executable RAM and Return Oriented Programming

The straight-forward exploitation of a buffer overflow only works if the RAM is executable [18]. As mentioned earlier, an executable RAM is often necessary during firmware upgrades, but since all code is stored in flash memory, most of the time RAM can, and should, be non-executable.

To prevent regular buffer overflow exploitation, it is common practice in desktop operating systems to make the data section non-executable; all major processor architectures have hardware support for this. In response, attackers have found ways to circumvent non-executable stacks with so called code-reuse attacks. One of the first attacks of this kind is called "return-into-libc" [19], and it was shown to be Turing complete: arbitrary computations can be achieved with this technique [20]. There is an even more fine-grained code-reuse technique called return oriented programming (ROP) [21]. Instead of using an entire function, the attacker uses *gadgets*, trailing function code snippets before a return instruction. The attack is then composed of a chain of fake stack frames pointing to gadgets. A key point for code-reuse attacks to work is that the gadget locations in memory are known.

Applying these techniques to the world of constrained automotive systems, one will find many similarities and a few differences. Since the microcontrollers we consider have no virtual memory, the entire memory space is addressable, within the confines of the MPU. Therefore, it should be possible to construct a working exploit using ROP, since all memory addresses are known a priori. Moreover, since everything is statically compiled, an exploit which works for a single ECU will work for all ECUs of the same type, implying that a resourceful attacker can study one vehicle to write an exploit which works across the fleet.

Successful construction of a ROP attack requires a substantial number of gadgets within the process or task address space. On regular PCs, these gadgets

are relatively easy to find in shared libraries. Since there are no shared libraries in resource constrained automotive systems, and the code base of the executing tasks is typically small, it is not certain that a sufficient number of gadgets can be found. Furthermore, ROP attacks require more space than a regular attack since every gadget, each containing a small number of instructions, requires its own stack frame. When RAM is very limited, this additional space may be a problem for successful exploitation.

The practical construction of ROP exploits on resource constrained automotive systems should be thoroughly studied by the security community to explore their feasibility.

3.3 Compile-time memory layout randomization

On desktop systems, the answer to return oriented programming is memory layout randomization: the stack and the shared libraries are loaded at random locations in virtual memory, rather than in predictable ones. This makes successful exploitation much harder.

Due to the static memory in constrained automotive systems, this technique can not be applied directly. However, it can still be utilized by randomizing the program memory mappings at compile-time. This would make them different for every ECU, and consequently, writing consistent exploits would be harder. The downside is that it would require custom images for every ECU.

Another caveat, apart from the potential production difficulties, is that the entropy is very low. Layout randomization has been shown to be relatively ineffective on regular 32-bit systems, due to low entropy of the memory locations [22]. However, this is only true under certain conditions, and it should be investigated if the same conditions also hold in resource constrained automotive systems.

4 Discussion

In this paper, we have discussed several simple measures that could be implemented to improve security, but further investigations are necessary to judge the cost and effectiveness of these mitigation techniques.

The first measure would be to add stack canaries to detect and prevent the exploitation of buffer overflows. The main concern with this technique is the associated performance degradation since automotive systems have strict real-time requirements and any impact on performance must be carefully evaluated. Nevertheless, the simplicity of canaries and their long use in desktop operating systems make them a very good candidate for immediate adoption, and the costs should be manageable.

The second measure would be to ensure that RAM is generally non-executable. RAM should only be executable when necessary, e.g., during (authenticated) firmware upgrades. Since virtually all microcontrollers include a memory protection unit, this should be relatively straight-forward and cheap to implement.

Non-executable RAM raises the bar for successful exploitation significantly, so this is another good candidate for immediate adoption.

The third measure, compile-time memory layout randomization, may help to combat attacks via return oriented programming. However, this technique has several practical difficulties. Changing the production process so that every ECU uses a slightly different image may be very costly. Given the cost, the added level of protection may be too small, and it aims to protect against complex attacks which currently are not likely to occur since simpler attacks will work. Therefore, this technique is unlikely to be adopted soon, but in anticipation of future ROP attacks on ECUs, it should be further investigated and improved.

There is also a new trend in the automotive industry to consolidate several smaller ECUs onto a single, more powerful ECU, as a result of efforts to reduce weight, costs, power and fuel consumption. The ECU architecture will resemble regular PCs, including virtual memory and complex operating systems. Nevertheless, it is unlikely that the light-weight ECUs as described in this paper will disappear quickly, and they still need to be secured.

5 Conclusion

With the advent of intelligent transport systems, the internet of things and self-driving cars, the protection of vehicles against malicious manipulation becomes ever more important. We have discussed three memory protection mechanisms to hinder successful exploitation of memory corruption bugs: stack canaries, non-executable RAM and compile-time memory layout randomization.

Since code typically executes directly from flash memory, and since memory protection units are in wide-spread use, we highly recommend to make RAM non-executable. When necessary, executing from RAM can be allowed for specific events such as firmware upgrades. Similarly, stack canaries have a proven track record, and their performance impact can be measured and accounted for, so we also highly recommend to adopt them in automotive systems. The idea of compile-time memory layout randomization on the other hand has serious flaws at this point, and it requires more work to be a viable protection mechanism. The analysis in this paper clearly shows that additional protection mechanisms are urgently needed, and that the techniques are readily available.

Finally, it should also be noted that the issues discussed in this paper are not necessarily unique to the automotive domain, and probably apply to a wide number of embedded systems.

Acknowledgments

We would like to thank all anonymous reviewers for their valuable feedback. The research leading to these results has been partially supported by the HoliSec project (2015-06894) funded by VINNOVA, the Swedish Governmental Agency for Innovation Systems, and by the Swedish Civil Contingencies Agency (MSB) through the project “RICS”.

References

1. Wolf, M., Weimerskirch, A., Paar, C.: Security in automotive bus systems. In: Workshop on Embedded Security in Cars. (2004)
2. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S.: Experimental security analysis of a modern automobile. In: 2010 IEEE Symposium on Security and Privacy (SP), IEEE (2010) 447–462
3. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T.: Comprehensive experimental analyses of automotive attack surfaces. In: Proceedings of the 20th USENIX Security Symposium, San Francisco, CA, USA (August 2011) 77–92
4. Kleberger, P., Olovsson, T., Jonsson, E.: Security aspects of the in-vehicle network in the connected car. In: 2011 IEEE Intelligent Vehicles Symposium (IV). (June 2011) 528–533
5. Greenberg, A.: Hackers remotely kill a jeep on the highway—with me in it. Accessed: 2017-06-01. Wired.com (2015) https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.
6. Greenberg, A.: Hackers remotely kill a jeep on the highway—with me in it. Accessed: 2017-06-01. Wired.com (2016) https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks/.
7. Valasek, C., Miller, C.: Adventures in Automotive Networks and Control Units. Technical report, Defcon 21 (August 2013) http://www.ioactive.com/pdfs/ IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf.
8. Miller, C., Valasek, C.: A survey of remote automotive attack surfaces. Technical report, Defcon 22 (August 2014) http://blog.hackthecar.com/wp-content/uploads/2014/08/236073361-Survey-of-Remote-Attack-Surfaces.pdf.
9. Miller, C., Valasek, C.: Remote Exploitation of an Unaltered Passenger Vehicle. Technical report, Defcon 23 (August 2015) http://illmatics.com/Remote%20Car%20Hacking.pdf.
10. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: Eternal War in Memory. In: 2013 IEEE Symposium on Security and Privacy (SP). (May 2013) 48–62
11. Van der Veen, V., dutt-Sharma, N., Cavallaro, L., Bos, H.: Memory errors: the past, the present, and the future. In: Proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses, Springer (2012) 86–106
12. Quigley, C.P., McMurran, R., Jones, R.P., Faithfull, P.T.: An investigation into cost modelling for design of distributed automotive electrical architectures. In: 2007 3rd Institution of Engineering and Technology Conference on Automotive Electronics. (June 2007) 1–9
13. Mayer, A., Hellwig, F.: System performance optimization methodology for Infineon's 32-bit automotive microcontroller architecture. In: Proceedings of the Conference on Design, Automation and Test in Europe. DATE '08, New York, NY, USA, ACM (2008) 962–966
14. Erjavec, J., Thompson, R.: Automotive technology: a systems approach. Cengage Learning (2014)
15. Gai, P., Violante, M.: Automotive embedded software architecture in the multi-core age. In: 2016 21st IEEE European Test Symposium (ETS). (May 2016) 1–8
16. ARM: ARMv7-M architecture reference manual. Technical report (December 2014)

17. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security. Volume 98. (1998) 63–78
18. Aleph One: Smashing the stack for fun and profit. Phrack magazine **7**(49) (1996) 14–16
19. Solar Designer: Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63 (August 1997)
20. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the expressiveness of return-into-libc attacks. In: Proceedings of the 14th International Symposium on Research in Attacks, Intrusions, and Defenses, Springer (2011) 121–141
21. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, ACM (2007) 552–561
22. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security. CCS '04, New York, NY, USA, ACM (2004) 298–307