Preventing Memory Errors in Networked Vehicle Services Through Diversification

Héctor Marco, Juan-Carlos Ruiz, David de Andrés, Ismael Ripoll

Preventing Memory Errors in Networked Vehicle Services Through Diversification

Héctor Marco[‡], Juan Carlos Ruiz[†], David de Andrés[†], and Ismael Ripoll[*]

[†] Instituto de Aplicaciones de las TIC Avanzadas (ITACA),
[*] Instituto Tecnológico de Informática (ITI)
[‡] Departamento de Informática de sistemas y computadores (DISCA)
Univ. Politècnica de València (UPV), Campus de Vera s/n, 46022, Valencia, Spain
Phone: +34 96 3877007 Ext {75774, 75752, 79707}, Fax: +34 96 3877579
{hecmargi, iripoll, jcruizg, ddandres}@disca.upv.es

Abstract. Car-to-X communication stands for the communication of different vehicles (vehicle-to-vehicle) as well as for the communication of vehicles and infrastructure (vehicle-to-infrastructure). The development of these technologies promotes the emergence of new car infotainment and telematic services of added value for users. The side effect is the exposure of vehicles to a number of new threats, such as memory errors. Among other consequences, the exploitation of memory errors may lead to code-reuse attacks, where intruders reuse existing non-malicious code with malicious purposes, such as gaining complete car control. Since memory error exploits usually rely on highly specific processor characteristics, the same exploit rarely works on different hardware architectures. This paper proposes a strategy to thwart memory error exploitation by combining the diversification of HW through processor emulation with the creation of Service variants using off-the-shelf cross-compilation suites.

Keywords: Car-to-X communications, memory errors, HW virtualization, Cross-compilation

1 Introduction

Modern automobiles are pervasively computerized and hence potentially vulnerable to attack. As reported in [1], presupposing an attacker's ability to physically connect to a car's internal computer network to gain car control is unrealistic. This statement of prime importance with the advent of new vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) systems. These systems rely on the use on wireless networks (sometimes also called vehicular ad-hoc networks, or VANETs) to offer added value networked services to users. However, the expand at the same time the conventional attack surface to be considered and raise a number of security issues, like derived from memory errors, for *future* transportation solutions [2].

Memory errors [3] have been around for over 30 years and, despite research and development efforts carried out by academia and industry, they are still included in the CWE SANS top 25 list of the most dangerous software errors [4]. Today, the exploitation of these errors has evolved towards code-reuse attacks, where no malicious code is injected to enable the activation and reused of legitimate code for malicious purposes [5]. Most of approaches proposed so far to eradicate or mitigate memory errors, such as countermeasures to prevent overwriting memory locations, detect code injections at early stagesor prevent attackers from finding, using, or executing injected code , lead protected systems or applications to crash in case of detecting a memory error. In addition, the effectiveness of these mechanisms against brute force attacks is quite limited.

In fact, very limited actions are usually taken when brute force attacks are detected in automotive systems: either the affected service is shutdown, with the subsequent unavailability effect, or it keeps running and an alert is issued to the car user, who may not be able to react fast enough to prevent a successful intrusion. The proposal presented in this paper builds on the principle that the exploitation of memory errors relies on highly specific processor characteristics, so the same procedure rarely works on different hardware architectures. Obviously, diversifying the hardware also means diversifying the considered software. Software diversification, i.e. the production of server variants, will be obtained using off-the-shelf cross-compilation suites, whereas hardware diversification relies on the emulation of different processor architectures. In this way, some vulnerabilities, which manifest on a given architecture, could be removed just by changing the execution platform to another particular architecture where existing software faults do not constitute a vulnerability anymore. So, basically, a variant replacement policy is deployed upon detecting a process crash issued from memory errors. The approach can be seamlessly combined with existing protection techniques to complement a highly secure mechanism against memory errors exploitation. The rest of this paper describes in detail this proposal.

2 Memory Errors: Prevention and Brute Force Attacks

Memory errors usually derive from the exploitation of vulnerabilities existing in a given application due to software faults introduced during its implementation. The most common software faults leading to memory errors include off-by-one, integer, and buffer overflowvulnerabilities [3] .

Most effective protection techniques known today to fight against memory errors include, although they are not limited to, Address-Space Layout Randomisation (ASLR [6]), Stack-Smashing Protection (SSP [7]) and Non-Executable Bit (NX [8]). These mechanisms are effective against multitude of attacks, like those relying on the precise knowledge of the absolute address of a library function, like `ret2libc`, those trying to overwrite the saved return address that has been stored for a function in the stack, or those trying to execute code in only writeable memory regions.

In all the cases, the aforementioned protection mechanisms abort the execution of the affected service.The decision of aborting a networked service, or more precisely the process running it, to hinder an attack can be however put into question, in particular in the case of automotive systems. On one hand, assuming the intervention of a system administrator is unrealistic in these systems. On the other hand, the system can be easily compromised in the presence of brute force attacks. This happens because depending on the protection technique and how the targeted application is internally architected, the number of tries or guesses required to find the secret key information of the target networked service varies but in many cases, it is not very high. For instance, it takes only 216 seconds to bypass a memory protection mechanism, such as Address Space Layout Randomization, a feature included today in most unix-based operating systems to manage processes memory [9].

In summary, although several techniques have been proposed so far to prevent the successful exploitation of memory errors, the truth is that all these mechanisms can be bypassed one way or another. So, additional effort is required to complement them with features to cope with service crashes and enable memory error tolerance.

3 Security Strategy

The core idea consists of having the same application compiled for different processors and replace the executable process when an error is detected. Each variant is executed in sequential order on the same host by a fast processor emulator. In the case of a malicious attack, since code execution is highly processor dependant, changing the processor that runs the application greatly hinders attack success. The proposed architecture has the following elements:

1. A set of cross-compiler suites for creating the set of variants.
2. A set of emulators for running the variants.
3. An error detection mechanism, which triggers the variant replacement.
4. The recovery strategy, which selects the variant that will be used once an error has been detected.

The approach ensures the service continuity while trying to fix the fault, in the case that the fault does not manifest in one of the variants.

Figure 1 illustrates this approach as a simple mechanical device: i) the source code, which presents software faults, is cross-compiled for different architectures to build the pool of variants (note that some variants do not manifest the error), ii) the existing protection mechanisms (the grid), like those introduced in the previous section, prevent the successful exploitation of memory errors by crashing the affected process, iii) the monitor, which is an external program, detects the crash events and activates the variant replacement mechanism (the pulleys), according to a established security policy.

The rest of this section details each one of the aforementioned aspects.

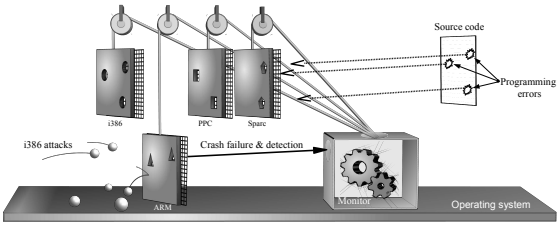

Fig. 1. High-Level view of the security approach

- **SW diversification**: The production of the SW variants relies on a on the use of off-the-self compiles, thus incurring in a reduced investment. ust by compiling the application source code for different target processors, the particular architecture of each processor will provide variants with different i) *endianness* and *instruction set*, so raw data and machine code injected by attackers will be differently interpreted, ii) *register set*, thus changing the stack layout (on non-orthogonal architectures), iii) *data and code alignment*, so unaligned instructions and word data type will raise an exception, iv) *address layout*, which results in different positions for functions and main data structures according to resulting code size and data layout, and v) *compiler optimisations*, some generic and some processor specific, resulting in register allocation, instruction reordering, or function reordering. Furthermore, the libraries liked to services during compilation can be also diversified whenever several alternatives are available. This will i) increase the degree of diversification among them, and let the resulting variant to ii) get rid of specific software faults that are not present in some libraries. This form of binary diversification preserves the semantic behaviour on each variant, it is easy to implement because of the reuse of widely available and tested software, and it provides a strong differentiation between resulting binaries.
- **HW diversification**: Providing a proper execution environment for resulting variants, including the operating system API, system calls convention, processor instruction set, and the executable file format, is not a so difficult task. In fact, the native variant, the one compiled for the physical processor and operating system hosting the server, will run on the native execution environment. However, as the rest of variants have been built for different processors, it is necessary to create a virtual execution environment to run them all.

  Nowadays, there are two different virtualisation solutions (see Figures **??**) to build a complete execution environment: **i) platform emulation**, where the emulator provides a virtual hardware to execute the guest operating system managing the guest application, and **ii) user mode emulation**, where the emulator provides both, processor virtualisation and operating system services, translating guest system calls into host system calls that are forwarded to the host operating system. User-mode emulation is a less common form of emulation but offers better performance since the operating

system code is directly executed by the host processor. This is basically why it is the one adopted in our proposal.

- **Memory error detection**: The proposed approach relies on the existing protection mechanisms (SSP, ASLR, etc.) for memory error detection. As previously explained, that detection leads to the crash of the compromised process. A monitor will be in charge of detecting these crash-related events and triggering the established variants replacement strategy according to the defined security policy.
  It must be noted that, although those techniques were initially developed to face malicious faults, they also provide a good coverage for accidental faults, like wild pointers. Accordingly, the accidental activation of software faults leading to memory errors will also crash the process, and give the system a chance to deal with them.
  The precise diagnosis of whether the problem is related to an accidental or malicious fault and its precise origin (kind of attack), to define a more specific reaction, is still an issue for further research.
- **Selection of variants**: The widely used multi-process architecture of the networking servers provides an ideal scenario for deploying different security policies for variants replacement upon detection of memory errors. The proposed policy enables the service to run in either a normal or a degraded mode, thus identifying the following states (see Figure 2):
  1. *High performance service*, where the service is natively executed at the maximum speed.
  2. *Fault avoidance*, where the system commutes only when an attack or a fault leading to a crash is detected. After a while, the native variant is placed again in execution, thus limiting the penalty induced by the approach. If the attack persist we move to the following state.
  3. *Confuse the attacker*, where for each memory error detection the variant is changed. After a while, if no additional error detection is detected the native varian is placed again into execution.

Fig. 2. Variants replacements policy

4 Conclusions and future work

The increasing communication capabilities embedded in new generation of vehicles are promising the proliferation of networked services of added value for user.

The side effect is that transportation solutions will be more and more exposed to many different threats, such as memory errors.

Nowadays, memory errors keep ranking among the top dangerous software errors despite vast research efforts from academia and industry. Although existing protection mechanisms work quite well in most of the cases, their inability to ensure a complete protection of in-car systems leads to an unsafety situation claiming for complementary mitigation solutions. The ongoing work presented in this paper, relies on diversification to complements existing protection mechanisms in situation when they simply crash affected vehicle services.

Contrarily to most automatic diversification techniques which customise the compiler or even the resulting executable binary, the use of cross-toolchains provides a simple and powerful solution for software diversification, while the required processor diversification can be done in an efficient way thanks to current advances on processor emulation techniques.

The approach is promising and a first implementation of the strategy is currently under implementation.

References

1. 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 conference on Security. SEC'11, Berkeley, CA, USA, USENIX Association (2011) 6–6
2. Rouf, I., Miller, R., Mustafa, H., Taylor, T., Oh, S., Xu, W., Gruteser, M., Trappe, W., Seskar, I.: Security and privacy vulnerabilities of in-car wireless networks: a tire pressure monitoring system case study. In: Proceedings of the 19th USENIX conference on Security. USENIX Security'10, Berkeley, CA, USA, USENIX Association (2010) 21–21
3. van der Veen, V., dutt Sharma, N., Cavallaro, L., Bos, H.: Memory errors: The past, the present, and the future. In: In the Proceedings of the 15th International Symposium on Research in Attacks Intrusions and Defenses (RAID). (September 2012)
4. CWE/SANS: Top 25 most dangerous software errors (2011)
5. 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 conference on Recent Advances in Intrusion Detection. RAID'11, Berlin, Heidelberg, Springer-Verlag (2011) 121–141
6. Pax Team: PaX address space layout randomization (ASLR) (2003)
7. Cowan, C., Pu, C., Maier, D., Hintongif, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proc. of the 7th USENIX Security Symposium. (Jan 1998) 63–78
8. Paulson, L.D.: New chips stop buffer overflow attacks. Computer **37**(10) (2004) 28–30
9. 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