Internet of Pwnable Things

Challenges in Embedded Binary Security

JOS WETZELS

Jos Wetzels is a Research Assistant with the Distributed and Embedded Security (DIES) Research Group at the University of Twente in The Netherlands. He currently works on projects aimed at hardening embedded systems used in critical infrastructure, where he focuses on binary security in general and exploit development and mitigation in particular, and has been involved in research regarding on-the-fly detection and containment of unknown malware and advanced persistent threats. He has assisted teaching hands-on offensive security classes for graduate students at the Dutch Kerckhoffs Institute for several years.
a.l.g.m.wetzels@gmail.com

Embedded systems are everywhere, from consumer electronics to critical infrastructure, and with the rise of the Internet of Things (IoT), such systems are set to proliferate throughout all aspects of everyday life. Due to their ubiquitous and often critical nature, such systems have myriad security and privacy concerns, but proper attention to these aspects in the embedded world is often sorely lacking. In this article I will discuss how embedded binary security in particular tends to lag behind what is commonly expected of modern general purpose systems, why bridging this gap is non-trivial, and offer some suggestions for promising defensive research directions.

Embedded Systems Security

Because embedded systems are so diverse, the threat landscape is equally varied, ranging from life-threatening sabotage of cyber-physical systems (e.g., electrical blackouts, smart-car crashes, insulin pump tampering) to economic (e.g., cable TV piracy, smart meter fraud) and privacy (e.g., smart-home surveillance) threats. Embedded security priorities also differ from those in the *general purpose* (GP) world. Whereas the latter tend to be mostly concerned about threats to confidentiality, embedded systems tend to prioritize availability and integrity. You want nuclear reactors to operate safely and automotive braking and flight control systems to function properly at all times.

Compared to GP systems, attention to embedded security is relatively recent, something that is especially visible in the industrial control systems (ICS), which form the technological backbone of electric grids, water supplies, and manufacturing environments. These systems were never designed to be connected to untrusted networks in the first place but, over the years, have steadily become more and more networked and exposed. As a result, these systems do not have corresponding security improvements. And concerns here are far from hypothetical as high-profile attacks have damaged nuclear facilities in Iran, caused blackouts on the Ukrainian power grid, and physically damaged a German steel mill.

This situation is compounded by the challenges of embedded patch deployment. Whereas in the GP world, patch management is often centralized and automated, the embedded world is faced by a myriad of problems (absence of hot-patching capabilities, safety recertification upon introduction of new code, extreme availability requirements, long device lifespans exceeding vendor support, etc.) complicating such an approach. This creates a situation of prolonged vulnerability exposure and exploits with long shelf-life capable of targeting millions of vulnerable, unpatched, and connected embedded devices.

Memory Corruption, Safe Languages, and Exploit Mitigations

When it comes to embedded systems, memory corruption issues (e.g., buffer overflows) consistently rank among the most prevalent categories of vulnerabilities as exemplified by a 2016 Kaspersky study of ICS vulnerabilities [1]. This prevalence is largely due to the dominance of unsafe languages such as C++ and assembly in embedded software development. As

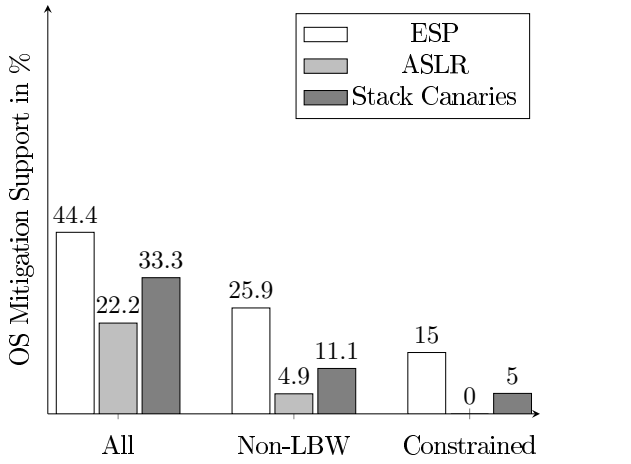

Figure 1: Exploit mitigation support among 36 popular embedded OSes. Non-LBW means Non-Linux, BSD, and Windows-based OSes, and Constrained indicates those tiny, minimalistic OSes designed for so-called deeply embedded systems.

someone once said: "C is a terse and unforgiving abstraction of silicon." Ideally, this problem would be mitigated by widespread adoption of safe languages, and while some are currently used (e.g., Ada, which is used in civilian and military avionics and aerospace systems) or show potential (e.g., Rust, which provides memory safety without garbage collection) for future adoption in the embedded world, there are some serious limitations.

First of all, the "close to metal" nature of C makes it well-suited for writing similarly bare-metal software (e.g., OSes or firmware) in a way that almost all safe languages are not. Note that Rust seems promising in this regard as shown by the intermezzOS and Tock [2, 8] OSes. Secondly, there's the issue of portability as there are billions of lines of legacy code written in unsafe languages, and there already are C toolchains for nearly every platform out there. Hence, even if the ideal embedded safe language existed right now, it would still take quite a while for an industry-wide shift in development practices to take off, never mind what to do with all that legacy code. So safe languages are a long-term solution at best, and we live in a short-term world that needs short-term solutions.

Exploit mitigations are just such a short-term solution since they seek to complicate exploitation of existing vulnerabilities rather than prevent their introduction in the first place. Exploit development can be conceptualized as the programming of so-called "weird machines" [3] through composition of "exploit primitives" into a chain. Complicating this chain means *making each link harder to forge* by making mitigations harder to overcome and *lengthening the chain* by crafting mitigations for various steps of the exploitation process in order to raise attacker cost and eliminate practical exploitability of certain vulnerabilities altogether.

Ever since memory corruption vulnerabilities started getting widespread attention with Aleph One's 1996 *Phrack* article "Smashing the Stack for Fun and Profit," various exploit mitigations have been proposed, implemented, broken, and improved until we've arrived at the present-day situation, where exploiting a stack buffer overflow on a modern GP system often requires you to at least either find an information leak to bypass stack canaries or overwrite a function pointer, find an information leak to bypass ASLR, craft a ROP (return-oriented programming) chain to bypass non-executable memory, and find a sandbox escape: that's two to three additional bugs (though less if one has a flexible enough vulnerability) on top of the actual vulnerability itself to achieve arbitrary code execution.

Embedded Exploitation: Blast from the Past

Compared to the GP world, embedded exploitation often feels like it's stuck somewhere in the '90s. Consider, for example, the Shadow Brokers incident [4] last year, where an unknown threat actor managed to obtain exploit and implant code used by the top-tier, probably state-sponsored, Equation Group threat actor and published part of the plunder online. This included exploits targeting enterprise firewalls used in very sensitive environments; what stood out here is that none of the exploits needed bypasses for any mitigation whatsoever.

In order to get an idea of what the situation with respect to embedded mitigation adoption looks like, I surveyed 36 popular embedded operating systems (ranging from high-end Linux-based ones to tiny proprietary real-time microkernels) for support of the "bread & butter" baseline of mitigations: Executable Space Protection (ESP, also known as DEP, NX, or W^X memory), Address Space Layout Randomization (ASLR), and stack canaries (also known as stack cookies or stack smashing protection). Briefly put: ESP forces attackers to use code-reuse payloads (such as ROP chains) by making data memory non-executable, while ASLR complements this by ensuring memory layout secrecy in order to prevent attackers from constructing such code-reuse payloads. Stack canaries are orthogonal to the former mitigations and work by inserting a randomized secret value, between stackframe metadata and local variables, that is checked for integrity when a function returns in order to detect whether it has been overwritten as part of a stack-smashing attack.

As you can see in Figure 1, only a minority supports these mitigations, and this becomes a negligibly small minority once you discard the Linux-, BSD-, and Windows-based OSes or only consider the most constrained OSes. And note that this survey was an optimistic one: if a mitigation is supported by an OS for even a single platform, no matter implementation quality, it was marked as supported. It's pretty safe to say embedded binary security lags behind the GP world significantly.

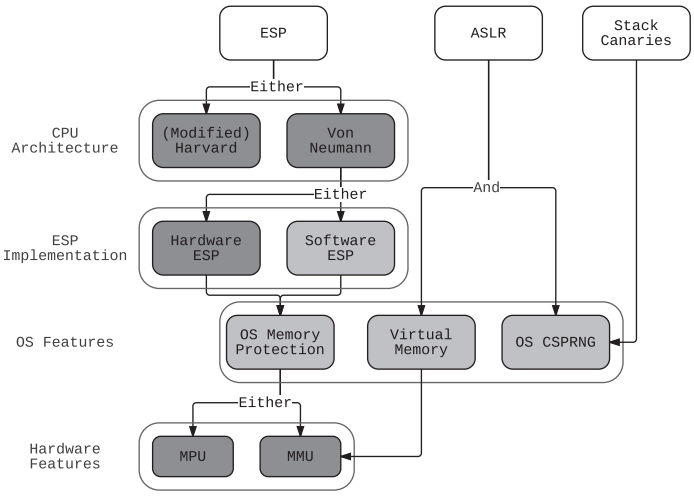

Figure 2: Exploit mitigation hardware and OS feature dependencies

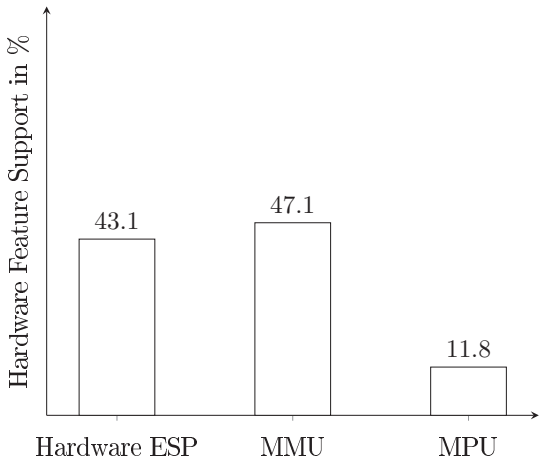

Figure 3: Hardware feature support among 51 popular von Neumann embedded core families

Dependencies, Constraints, and Possible Solutions

So what's the reason for this adoption gap? Well, it turns out that if you map out the hardware and software dependencies of these mitigations (Figure 2), there's some serious constraints that complicate adoption. Embedded devices are designed for a specific task and tend to have limited resources as well as often being headless and diskless. The hardware is often simple and lacking in advanced features, and the software is tailored for such constraints. And on top of all that there are often real-time and safety-critical requirements.

In order to get an idea of the state of mitigation dependency support among common embedded hardware and OSes, I surveyed 51 popular von Neumann-style embedded core families (Figure 3) and mapped out OS feature dependency support (Figure 4) among the 36 previously surveyed OSes. As shown in these figures, widespread support for key dependencies is lacking, which presents a significant hurdle to mitigation adoption. To see why these dependencies are so crucial and to provide some suggestions for research directions that can potentially overcome existing limitations, let's take a look at each mitigation in our baseline in detail.

Stack Canaries and Embedded Random Number Generators (RNGs)

Stack canary mechanisms are implemented as a compiler feature but require some sort of (secure) random number generator to be present on the target OS to generate the master canary value when the binary in question is loaded. This is best left to the cryptographically secure pseudo-random number generator (CSPRNG) provided by the OS itself (e.g., */dev/urandom* on UNIX-like systems), but as Figure 4 shows, only 41.7% of surveyed embedded OSes provide a system CSPRNG, and this number drops to 22.2% if you eliminate Linux-, BSD-, and Windows-based ones and becomes negligible altogether if you only consider the most constrained operating systems.

I've discussed the issues with embedded OS CSPRNGs in more detail in my recent 33C3 talk "Wheel of Fortune: Analyzing Embedded OS Random Number Generators." To put it briefly, it's not trivial to port existing designs from the GP world, mainly because of a combination of resource constraints in terms of processing speed, memory and power consumption, and a general low entropy environment. These systems are designed for limited, specific tasks, often in a machine-to-machine setting without human activity, and are designed to perform those tasks in a reliable, predictable fashion. This is a major stumbling block because PRNGs need sources with some external entropy in order to stretch their output into longer sequences of pseudo-random output.

On GP systems common sources for entropy are user input devices like the mouse, keyboard, or disk activity, but since many embedded systems are headless and/or diskless this is not an option. Depending on the embedded device in question, potentially suitable entropy sources might be available from sensor values, radio measurements, accelerometer data, etc., but from an OS designer's point of view these sources cannot be assumed to be universally present on all devices the OS is to be deployed on. This problem would ideally be solved by having omnipresent on-chip high-throughput true random number generators (TRNGs), but this is quite unrealistic considering accompanying cost increases. In addition, it doesn't help with existing legacy systems.

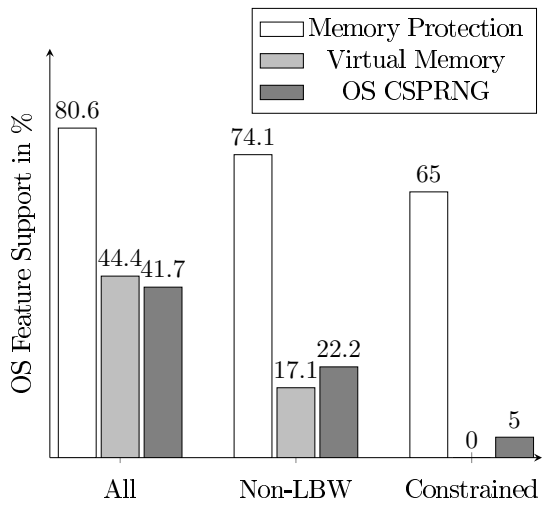

Figure 4: OS feature support among 36 popular embedded OSes

Two promising research directions upon which embedded OS CSPRNG designers could draw are advances in lightweight cryptography and investigation of omnipresent entropy sources. The former encompasses various cryptographic primitives designed for highly constrained systems. Initiatives such as the ACRYPT project [5] have produced a "zoo" of IoT-oriented lightweight primitives, with accompanying implementation footprint information (in terms of code size, memory usage, and execution time), which can serve as building blocks in a larger OS CSPRNG design. The embedded entropy problem is a more fundamental one and doesn't lend itself well to a one size fits all solution, but a thorough exploration of the suitability of potential entropy sources, which are virtually omnipresent in embedded systems, such as startup values of on-chip SRAM, clock jitter, and so on, would definitely be worthwhile.

Executable Space Protection (ESP)

Essentially there are two main CPU architectural styles: Harvard and von Neumann. The former has physically separate code and data memories, while the latter has a single memory holding both code and data. There are many possible nuances to these "pure" styles, but when it comes to the goals of ESP the only thing that matters is that memory can't be both writable and executable so that attackers can't easily inject malicious shellcode payloads into memory. As such, Harvard architectures trivially provide ESP, but for von Neumann-style CPUs, ESP will have to be implemented either in a hardware-assisted way or through software emulation. The former case is implemented in the form of a dedicated hardware feature (x86 NX bit, ARM XN bit, etc.), usually as part of the memory management unit (MMU) regulating memory executability at a certain granularity level on a per-page basis. In the case of software emulation, there are multiple approaches all outside the scope of this article, the most famous of them probably being the PaX project's implementation [6], but all of them incur at least some overhead and tend to be architecture-specific.

As shown in Figure 3, 43.1% of surveyed core families have hardware ESP support, something you need to consider in light of the fact that software emulation-based approaches to ESP only exist for a limited number of OS and architecture combinations (e.g., Linux on x86). Both ESP implementations require memory protection support (and as such an MMU or more lightweight memory protection unit (MPU)) on the part of the OS to allow for memory permission management. And while most embedded OSes offer memory protection support, we can see in Figure 3 that only 47.1% of all surveyed core families have MMU support and only 11.8% have MPU support, leaving 41.1% unable to accommodate memory protection. Now some microcontrollers might offer (limited) memory permission management capabilities without featuring an MPU/MMU, and for some processors there are external MMUs available, like the Motorola 68851, but apart from these edge cases, there's a significant "gap segment" of embedded systems without support for the core ESP dependencies.

Ideally, embedded systems designers would start consciously using either Harvard CPUs (AVR, 8051, PIC, etc.) or von Neumann ones with hardware ESP support (ARMv6+, MIPS32r3+, x86, etc.), but for those systems where this is not an option we will need widespread embedded OS adoption of a multi-architecture, low-overhead software emulation ESP approach. This does, however, still leave us with the open problem of how to deal with MPU-/MMU-less systems that cannot offer any form of memory protection to begin with.

Address Space Layout Randomization (ASLR)

In order to craft the code-reuse payloads used to bypass ESP, attackers will have to know the addresses of particular code fragments (so-called "gadgets") to incorporate into their payload. ASLR aims to complicate this by ensuring memory layout secrecy through randomization, which is done by placing various different memory objects—the stack, heap, main program image, loaded libraries—at randomized addresses. In order to do this, ASLR has three key dependencies: a CSPRNG, OS virtual memory support, and hardware with an MMU. The ASLR randomization takes place at load-time and draws upon an OS CSPRNG, as we've seen earlier, and is far from omnipresently available in all embedded operating systems.

Virtual memory provides memory isolation between different tasks/processes and thus prevents shared memory conflicts that might otherwise arise from ASLR's memory object randomization. If we look at Figure 4, however, we can see that only 44.4% of all surveyed embedded operating systems provide virtual

memory support, and this number drops to a mere 17.1% if we eliminate the Linux-, BSD-, and Windows-based OSes. Even worse are the most constrained operating systems, none of which support virtual memory for various reasons, such as being designed for MMU-less and diskless targets or having conflicting hard real-time requirements.

This widespread lack of embedded virtual memory and MMU support are two major obstacles to widespread ASLR adoption that are not going away anytime soon, which means that we need an embedded alternative to ASLR. ASLR's dependency on virtual memory arises from the fact that it is a *load-time software diversification* technique [7]. This dependency does not apply, however, to diversification techniques operating at earlier points in the software life cycle such as at compile or install time. In these cases either a compiler feature or a dedicated transformation program produce diversified binaries by randomizing code layout and/or individual instruction sequences. Such approaches achieve a similar effect to ASLR by randomizing the addresses (and nature) of code-reuse gadgets but have the downside of being less effective since they only diversify between different software builds or individual device instances rather than between individual boots or program runs as well as only diversifying code memory. There are currently no mature, widely adopted implementations of such schemes that I know of, nor has their applicability to the embedded world been covered, but they seem to be a promising embedded ASLR alternative.

A Call to Action

So where do we go from here? First of all, security researchers should continue to demonstrate the urgency and impact of embedded vulnerabilities to drive the point home that embedded systems cannot afford to keep lagging behind when they are becoming increasingly ubiquitous and interconnected. Secondly, work on short-term solutions (researchers addressing the challenges outlined in this article working together with OS developers to push for embedded exploit mitigation adoption) should be conducted alongside work on more long-term solutions such as embedded safe language research and development of secure embedded patching and updating mechanisms. And, finally, with the rise of the Internet of Things there is a real need for IoT standardization, policy, and regulation that focuses on *security by design* rather than leaving it as an afterthought or something that has to be retrofitted after the first vulnerabilities are discovered due to a vendor focus on novel features and time-to-market.

References

[1] O. Andreeva, S. Gordeychik, G. Gritsai, O. Kochetova, E. Potseluevskaya, S. I. Sidorov, and A. A. Timorin, "Industrial Control Systems Vulnerabilities Statistics," Kaspersky Lab, 2016: https://kasperskycontenthub.com/securelist/files/2016/07/KL_REPORT_ICS_Statistic_vulnerabilities.pdf.

[2] http://intermezzos.github.io/.

[3] S. Bratus, S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming: From Buffer Overflows to 'Weird Machines' and Theory of Computation," *;login:* vol . 36, no. 6 (December 2011): https://www.usenix.org/system/files/login/articles/105516-Bratus.pdf.

[4] M. Al-Bassam, Equation Group Firewall Operations Catalog, 2016.

[5] A. Biryukov, D. Dinu, J. Großschädl, D. Khovratovich, Y. Le Corre, L. Perrin, "The ACRYPT Project: Lightweight Cryptography for the Internet of Things," CRYPTO 2015 Rump Session, 2015.

[6] NOEXEC, PaX project Documentation, 2003.

[7] P. Larsen, A. Homescu, S. Brunthaler, M. Franz, "SoK: Automated Software Diversity," 2014 IEEE Symposium on Security and Privacy: https://www.ics.uci.edu/~perl/automated_software_diversity.pdf.

[8] https://www.tockos.org/.