Anatomy of Memory Corruption Attacks and Mitigations in Embedded Systems

Nektarios Georgios Tsoutsos, *Student Member, IEEE,* and Michail Maniatakos, *Senior Member, IEEE*

***Abstract*—For more than two decades, memory safety violations and control-flow integrity attacks have been a prominent threat to the security of computer systems. Contrary to regular systems that are updated regularly, application-constrained devices typically run monolithic firmware that may not be updated in the lifetime of the device after being deployed in the field. Hence, the need for protections against memory corruption becomes even more prominent. In this article, we survey memory safety in the context of embedded processors, and describe different attacks that can subvert the legitimate control flow, with a special focus on Return Oriented Programming. Based on common attack trends, we formulate the anatomy of typical memory corruption attacks and discuss powerful mitigation techniques that have been reported in the literature.**

***Index Terms*—Memory safety violations, control-flow integrity protections, buffer overflows, return oriented programming.**

I. Introduction

THE proliferation of Internet of Things (IoT) and ubiquitous computation has raised the demand for secure and high-performance embedded computers. Embedded processors are application-specific computers that are typically used to perform special functions within a constrained computing environment. Contrary to general-purpose x86 CPUs, embedded processors consume very little energy, have specific instruction set architecture (ISA) as well as additional peripherals on the same silicon die.

In several scenarios, embedded processors support monolithic applications and the executed programs are not expected to change dramatically during the lifetime of the underlying system. In this case, the embedded processors run their own firmware, which may be updated a limited number of times, or not updated at all for some constrained IoT applications. Interestingly, the latter scenario outlines a threat model that motivates the use of runtime security controls against memory corruption vulnerabilities, since patching such vulnerabilities after the embedded processor has been deployed in the field may not be possible.

Moreover, embedded processor firmware may be used in early boot environments that are responsible for initializing an underlying system and loading a tiny operating system. In such scenarios, in the interest of saving memory, a frequent practice is to reuse parts of the boot code during later execution stages (after the system boots), when external inputs are also processed. Thus, embedded processors can operate under conditions that *allow potential memory corruption vulnerabilities to be exploited*, and this is emphasizes the need to deploy additional protection mechanisms. In this article, we review memory corruption threats to embedded processors and summarize contemporary mitigation techniques.

N. G. Tsoutsos and M. Maniatakos are with New York University, Abu Dhabi, UAE. E-mail: {nektarios.tsoutsos, michail.maniatakos}@nyu.edu.

Memory Safety. In embedded system programming, *memory corruption* remains a major class of memory safety violations that could enable attackers to subvert the legitimate program behavior. Using programming languages that provide unlimited freedom on how to data are handled, such violations can be attributed to programmer errors or improper input validation [1]. Consequently, memory safety violations comprise a broad range of software problems that could enable unauthorized access to sensitive information resources, escalate privileges, modify the program control flow or allow execution of arbitrary code [2]. Notable examples of memory safety violations are (a) *buffer overflows*, including stack smashing, heap smashing and return oriented programming (ROP), where buffer data can overwrite adjacent data structures; (b) *uncontrolled format strings*, where print functions can disclose or overwrite other memory values; (c) *pointer violations* such as *null* pointer dereference, double-free, use-after-free and pointer arithmetic errors, which can cause arbitrary memory reads/writes or control flow redirections; and (d) *integer overflows*, such as signedness errors, where a value may become negative or unexpectedly small [1], [2].

Harvard Architectures. An important property of embedded processors that is relevant to memory corruption threats is the underlying memory architecture. Contrary to modern x86 processors that use the *von Neumann* model, most embedded processors follow the *modified Harvard* architecture [3]. The latter implements two different address spaces for data and instructions, so injecting arbitrary code (e.g., using a buffer overflow) is not straightforward as in the von Neumann model. Nevertheless, since the instruction memory of a modified Harvard system could still be updated by invoking special Store to Program Memory (SPM) instructions, exploiting a memory corruption vulnerability to trigger a ROP attack can enable persistent execution of attacker-controlled code [4]. In fact, ROP remains a major threat to modified Harvard embedded processors, as it can bypass the address space separation between instructions and data by writing executable code to instruction memory.

The rest of the article is organized as follows: In Section II we elaborate on major classes of memory corruption attacks, while in Section III we focus on ROP attacks that could impact embedded systems. A summary of exploitation strategies and existing mitigation methods is discussed in Section IV, and our concluding remarks are presented in Section V.

II. Memory Corruption Attacks

For more than two decades, memory corruption has been the most popular security vulnerability affecting computer systems [5]. In this case, attackers exploit program deficiencies to write past the end of a data buffer and overwrite neighboring date structures, such as active pointers, which can cause the program to behave arbitrarily [6], [7].

Stack Smashing Attacks are launched by overflowing a data buffer stored on the stack area of a program. Attacker-provided data overwrite adjacent control data, such as the return address pushed by the *caller* subroutine or the frame pointer. Modifying the return address would subvert the nominal control flow as soon as the *called* function terminates. Likewise, overwriting the frame pointer would enable indirect control of the return address [8]. Having control of the return address, an attacker can then branch to injected instructions masqueraded as data already on the stack (i.e., shellcode).

Heap Smashing Attacks exploit dynamic memory allocators (e.g., `malloc`) by corrupting the control structures defining the heap itself. By overflowing a heap block, attackers could overwrite adjacent heap headers that chain different heap blocks, and eventually cause the dynamic memory allocator to modify arbitrary memory locations as soon as a heap `free` operation is executed. The malicious payload can also be generated on-the-fly: for example, by exploiting Just-In-Time (JIT) compilation, assembled code can be written on the heap [9].

Arc Injection Attacks enable payload execution by reusing existing instruction sequences in instruction memory. This attack is possible assuming that the control flow has already been subverted and the adversary controls the instruction pointer. Moreover, this attack is applicable when the overflown buffer lies in non-executable memory, so the payload cannot be provided through a buffer or JIT compilation. For example, an adversary can call an existing library function after pushing malicious arguments (such as "`\bin\sh`") via stack smashing [8].

Function Pointer Clobbering allows modifying a function pointer with an address selected by the adversary. The malicious target can be either injected code in data memory or existing code in instruction memory, and the attack is triggered as soon as the program calls the function whose pointer has been modified. For example, the adversary can modify pointers in the Global Offset Table (GOT) or the Procedure Linkage Table (PLT) [1], [10], and subvert calls to innocuous functions, such as `printf` or `_exit()` [7], [8].

Data Pointer Modification allows the adversary to overwrite a data pointer that is subsequently used as the target of an assignment instruction, and eventually perform an *arbitrary memory write*. In effect, the adversary can modify chosen memory locations that directly or indirectly affect the control flow (such as function pointers, return addresses or indirect branch targets). For example, if an overflown buffer modifies an adjacent pointer variable p and an adjacent numeric variable val, then any existing program statement that assigns val to the dereference of p (e.g., $*p = val$) would allow the adversary to modify arbitrary memory values (e.g., privilege level, authentication status or program state variables), or bypass runtime assertions and avoid detection.

Exception Handler Hijacking entails overflowing a buffer into the exception handling control structures stored on the stack. Such control structures are typically defined as linked lists with function pointers to exception handling code that is invoked when a specific exception occurs. An adversary can cause arbitrary code execution by overwriting an exception handling pointer and triggering a runtime exception.

Longjmp Buffer Modification exploits the `setjmp-longjmp` checkpoint functionality available in the C language, to modify the stored instruction pointer [6], [11]. When the program state relies on `setjmp-longjmp` pairs, the instruction pointer and other checkpoint data are stored by the `setjmp` command within a `jmp_buf` data structure. If the adversary can overflow an adjacent buffer and corrupt the stored instruction pointer stored in `jmp_buf`, the legitimate control flow would be hijacked as soon as the `longjmp` command is invoked by the program.

III. Arc Injection & Return Oriented Programming

Return Oriented Programming (ROP) is a powerful form of Arc Injection that enables attackers to chain short sequences of existing instructions (dubbed *ROP gadgets*) based on a sequence of return addresses popped from the stack [12]. Each ROP gadget terminates with a return instruction that causes an indirect branch to the next gadget pointed by each return address on the stack. This method enables arbitrary code execution in the vulnerable process, where the stack pointer acts as the instruction pointer; hence, using gadgets to load/store memory values, modify memory and unconditional branching, an attacker can achieve universal computation [12]. Moreover, ROP exploits can bypass traditional mitigations against the injection of arbitrary code in executable memory, such as *NX-bit* memory segregation [9], [11].

Even though most gadgets terminate with return instructions, it is possible to implement exploits using gadgets comprising register-indirect branches to a reusable *pop & indirect jump* sequence (dubbed *trampoline*). Another variant is *Jump Oriented Programming (JOP)* that is based on register-indirect branches to chain gadgets through a special *dispatcher* instruction sequence [13]. Likewise, *Call Oriented Programming (COP)* uses gadgets terminating with memory-indirect call instructions, which eliminates the need for dispatcher code: in this case, gadgets can be chained using pre-initialized memory locations, each one indirectly pointing to the next gadget in sequence [13].

One extension to the trampoline approach is the *borrowed code chunks* technique, which combines calls to existing library functions (using malicious arguments pushed on the stack) with existing gadgets (dubbed code *chunks* that end with a return instruction or a pop & jump combination. These code chunks are required to "glue" together the sequences of library function calls [10]. If an adversary controls the stack through a buffer overflow vulnerability, both return addresses and library function arguments can be chosen by the adversary; however,

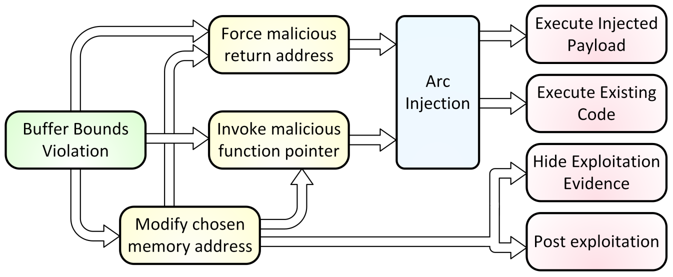
Buffer Bounds Violation
Force malicious return address
Invoke malicious function pointer
Modify chosen memory address
Arc Injection
Execute Injected Payload
Execute Existing Code
Hide Exploitation Evidence
Post exploitation

Fig. 1. Exploiting buffer overflow vulnerabilities.

it may not always be possible to correctly chain the desired library function calls, unless some special state is set between them. By judiciously injecting stack values, the adversary could return from a library function into an appropriate code chunk and then return control back to the next library function. In effect, the borrowed code chunks paradigm allows short return-terminated gadgets to interconnect library function calls more efficiently and enable arbitrary code execution for the adversary.

Arc Injection attacks are also possible when the adversary can overflow the heap instead of the stack, and eventually cause an arbitrary function to be called through *heap smashing* [9]. In this case, the adversary transfers control to a special return-terminated gadget that creates a *stack pivot* in order to build a fake stack on the heap. In particular, assuming that the adversary controls both the instruction pointer (by modifying a function pointer) and the heap buffer contents (used for both heap smashing but also for hosting a fake stack), the stack pivot gadget would assign a chosen heap address to the stack pointer. The return instruction that terminates the stack pivot gadget transfers the control flow to the adversary-controlled address popped from the fake, heap-based stack. Thus, using the stack pivot, it is possible to modify the stack pointer to pop malicious data from the heap, which is masqueraded as the stack.

IV. Attack Strategies & Mitigations

A. Exploitation Anatomy

In order to exploit a memory corruption vulnerability and subvert the legitimate control-flow of a program, an adversary would have to perform two steps [12]: (a) exercise an entry point to hijack the control flow for the first time, and (b) introduce arbitrary behavior in the program using that entry point. As illustrated in Fig. 1, the first step requires corrupting a data buffer by writing beyond its boundaries, which is possible by abusing an underlying vulnerability. At this point, the adversary may implement three alternative tactics:

1) Force a malicious return value as a branch target, which could be achieved by overwriting a return address already pushed on the stack, overwriting the pushed frame pointer, or by overwriting the instruction pointer stored within a `jmp_buf` buffer (invoked by a `longjmp` statement).
2) Call arbitrary functions using malicious pointers, which is possible by overwriting function pointers stored on the stack or the heap, or by overwriting pointers responsible for exception handling.
3) Modify arbitrary memory locations, which is possible by tampering with pointer targets and values on the stack before executing assignments like `*p=val;`, or by corrupting the linked-list pointers of heap blocks before executing `free` statements. This tactic allows modifying return addresses and function pointers (Fig. 1), as well as pointers in GOT or PLT tables, and pointers in `fnlist` and destructor `.dtors` entries, which also enables hijacking the execution flow.

As soon as an entry point is established, the adversary can proceed with one of the following actions:

1) Perform Arc Injection to execute injected or existing code. This is possible by redirecting the control-flow to buffer data in executable memory (which are interpreted as instructions), or by executing library functions and ROP sequences using malicious arguments from the stack, as mentioned in Section III.
2) Modify judiciously selected data to mask the attack, or perform post-exploitation tasks such as privilege escalation. For example, the adversary may restore corrupted values (e.g., canaries) to keep the attack undetected, or modify a state variable (e.g., authentication status or privilege level) and to force an alternative control flow already defined for the modified state.

B. Attack Countermeasures

As it is evident from the previous discussion, an adversary may hijack the execution flow by modifying return targets, function pointers or arbitrary memory locations. This enables arbitrary execution of injected or existing code, as well as modification of critical state data and bypassing of detection mechanisms. To mitigate these common attack vectors, a diverse set of mitigation techniques has been proposed in the literature:

Non-Executable Memory prevents execution of code injected into the system as data, making a memory range either writable or executable (but not both). In particular, defining memory regions as "data-only" ensures that these locations are never treated as instructions, which makes code-injection ineffective. This security policy can be enforced either through hardware support (e.g., NX bit) or through software emulation (e.g., Exec Shield) [9].

Stack Canaries can detect corruption of control data on the stack by checking the integrity of a special ("canary") value inserted at a memory location before such data [14]. The fundamental assumption is that overwriting the control data requires overwriting the canary value first, and thus it suffices to verify the integrity of the canary as a condition for the integrity of the actual control data. If the canary value is corrupted, the program execution is usually terminated to prevent further exploitation of the system.

Secondary Return Stack provides integrity guarantees to a function return address by maintaining a golden copy in a user-inaccessible memory range [8]. The redundant stack does not contain any buffers that could be overflown and thus cannot be corrupted through stack smashing, as it is the case with the regular stack of the program. As soon as a return instruction is executed, the address stored in the secondary stack is compared against the address in the original stack; if they are equal, it is safe to branch to that address, otherwise the return address on the original stack has been corrupted.

Obfuscating Return Addresses and Pointers prevents meaningful modification of branch targets to addresses chosen by the adversary. For example, this is possible by XORing a control value with a secret random key [15]. In this case, the control values are obfuscated as soon as they are created or modified by the program, and de-obfuscated immediately before each use. A fundamental assumption is that adversaries cannot leak the obfuscated values from memory (e.g., through memory disclosure vulnerabilities), as this allows making intelligible guesses about the secret key when the effective return address or pointer value are predictable.

Code Rewriting is a compilation-time mitigation that modifies the generated program to eliminate potential gadgets and minimize the risk of ROP attacks. In effect, this technique removes instances of unintended *free branch* instructions (i.e., misaligned byte sequences that could be interpreted as branches) by adding *alignment sleds*, by reallocating registers used as operands, by replacing instructions sequences with safe equivalents, and by modifying branch offsets, immediates and displacements [16].

Address Space Layout Randomization (ASLR) entails random relocation of critical parts of the program memory, such as the base addresses of the heap, the stack, as well as the base of the executable and shared libraries [17]. Since most Arc Injection exploitations require absolute memory offsets and predictable memory layouts, randomizing the offsets of critical addresses would cause the program to crash instead of executing arbitrary code chosen by the adversary [2].

Inline Reference Monitors insert security checks within the program code to verify a predefined security policy [18]. The corresponding policy should depend only on past events (i.e., future assumptions are not required), and as soon as it is violated for the first time, it cannot be restored [19]. One prominent example includes Control Flow Integrity (CFI) [18].

Buffer Bounds Checking implements a dynamic verification of each buffer write operation to ensure it does not cause an overflow. This method can be very effective, since it prevents exercising an entry point to begin exploiting the system. Nevertheless, since every buffer access is mediated, this technique can incur prohibitive overheads that range from 2.3–30$\times$, depending on the aggressiveness of the implemented checks [20].

Instruction Set Randomization (ISR) is a hardware-enforced protection that uses a secret randomization key to obfuscate the instruction stream stored in memory; instructions are only de-randomized during the *instruction fetch* stage of the pipeline. Even if a malicious payload is injected into a buffer, it would still be impossible to execute it without access to the randomization key (i.e., the injected instructions would be "de-randomized" into a non-meaningful stream) [21].

V. Conclusion

Memory safety remains a major concern for application-constrained devices that are not updated after their initial deployment. The embedded processors installed in such systems may become vulnerable to memory corruption attacks that can subvert the intended control-flow and allow execution of arbitrary code. In particular, one important attack vector is return oriented programming, which combines existing code sequences with injected data on the stack to execute malicious operations. In this paper, we summarize different types of memory corruption attacks, we analyze the anatomy of typical exploitations, and we survey applicable countermeasures that have been reported in the literature.

Acknowledgment

The Embedded Security Challenge was funded in part by NSF grant 1513130.

References

[1] C. Kil *et al.*, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Annual Computer Security Applications Conference*. IEEE, 2006, pp. 339–348.
[2] J. Hiser *et al.*, "ILR: Where'd my gadgets go?" in *Symposium on Security and Privacy*. IEEE, 2012, pp. 571–585.
[3] K. Watts and P. Oman, "Stack-based buffer overflows in harvard class embedded systems," in *Critical Infrastructure Protection*. Springer, 2009, pp. 185–197.
[4] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Computer and Communications Security*. ACM, 2008, pp. 15–26.
[5] Y. Younan, "25 years of vulnerabilities: 1988–2012," Tech. Rep., 2013.
[6] C. Cowan *et al.*, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition*, vol. 2. IEEE, 2000, pp. 119–129.
[7] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *Security & Privacy*, vol. 2, no. 4, pp. 20–27, 2004.
[8] G. Richarte, "Four different tricks to bypass stackshield and stackguard protection," Core Security Technologies, Tech. Rep., 2002.
[9] K. Z. Snow *et al.*, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Symposium on Security and Privacy*. IEEE, 2013, pp. 574–588.
[10] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," SuSE, Tech. Rep., 2005.
[11] S. Checkoway *et al.*, "Return-oriented programming without returns," in *Computer and Communications Security*. ACM, 2010, pp. 559–572.
[12] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Computer and Communications Security*. ACM, 2007, pp. 552–561.
[13] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *USENIX Security*, 2014, pp. 385–399.
[14] C. Cowan *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security*, 1998, pp. 63–78.
[15] ——, "Pointguard: protecting pointers from buffer overflow vulnerabilities," in *USENIX Security*, vol. 12, 2003, pp. 91–104.
[16] K. Onarlioglu *et al.*, "G-Free: defeating return-oriented programming through gadget-less binaries," in *Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58.
[17] H. Bojinov *et al.*, "Address space randomization for mobile devices," in *Wireless Network Security*. ACM, 2011, pp. 127–138.
[18] M. Abadi *et al.*, "Control-flow integrity," in *Computer and Communications Security*. ACM, 2005, pp. 340–353.
[19] F. B. Schneider, "Enforceable security policies," *Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.
[20] G. E. Suh *et al.*, "Secure program execution via dynamic information flow tracking," in *ASPLOS*. ACM, 2004, pp. 85–96.
[21] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Computer and Communications Security*. ACM, 2003, pp. 272–280.