Initial notes papers

Gilles De Borger

October 10, 2020

1 SoK: Eternal War in Memory

1.1 Abstract

we describe attacks that succeed on today's systems.We systematize the current knowledge about various protection techniques by setting up a general model for memory corrup- tion attacks. Using this model we show what policies can stop which attacks. The model identifies weaknesses of currently deployed techniques, as well as other proposed protections enforcing stricter policies.

We analyze the reasons why protection mechanisms imple- menting stricter polices are not deployed. Especially important is performance as experience shows that only solutions whose overhead is in reasonable bounds get deployed

1.2 Introduction

According to the MITRE ranking [1], memory corruption bugs are considered one of the top three most dangerous software errors.

A multitude of defense mechanisms have been proposed to overcome one or more of the possible attack vectors. Yet most of them are not used in practice, due to one or more of the following factors: the performance overhead of the approach outweighs the potential protection, the approach is not compatible with all currently used features (e.g., in legacy programs), the approach is not robust and the offered protection is not complete, or the approach depends on changes in the compiler toolchain or in the source-code while the toolchain is not publicly available.

The motivation for this paper is to systematize and evaluate previously proposed approaches. The systematization is done by setting up a general model for memory corruption vulnerabilities and exploitation techniques. The evaluation is based on robustness, performance and compatibility.

1.3 Attacks

(verwijs naar figuur 1 in de paper)

1.3.1 Memory corruption

The root cause of vulnerabilities discussed in this paper are memory corruption. First step is to trigger memory error. To do so, first you make a pointer invalid, and then you dereference this pointer. A pointer can become invalid by going out of bounds of its pointed object, or by deallocating it (*dangling pointer*). Dereferencing an out-of-bounds pointer causes a so called spatial error, while dereferencing a dangling pointer causes a temporal error. Forcing a pointer out of bounds:

- trigger *allocation failure* that is left unchecked -¿ result in null pointer
- incrementing or decrementing an array pointer in a loop without proper bound checking -¿ a buffer overflow/underflow
- By causing indexing bugs where an attacker has control over the index into an array, but the bounds check is missing or incomplete -¿ the pointer might be pointed to any address.

- Lastly, pointers can be corrupted using the memory errors under discussion. This is represented by the backward loop in Figure 1

Making a pointer "dangling"

- (for instance) exploiting an incorrect exception handler, which deallocates an object, but does not reinitialize the pointers to it.
-

Then, we exploit an out-of-bound or dangling pointer to execute any third step in the exploitation model by reading or writing it. When a value is read from memory into a register by dereferencing a pointer controlled by the attacker, the value can be corrupted. It can also leak information (as with *printf*)

If an attacker controlled pointer is used to write the memory, then any variable, including other pointers or even code, can be overwritten. (including other pointers -¿ backwards loop in figure)

1.4 Attacks

Code corruption attack: Use abovementioned bugs to overwrite program code in memory.

Control-flow hijack attack: take control over the program by diverting its control-flow by corrupting a *code pointer*. Can be done (for example) by modifying return address due to buffer overflow. A substep is to know the correct target value (address of payload, code we want to execute). Suppose the code pointer (e.g. function pointer) has been corrupted. Fifth step is to load it into instruction pointer. This can only be done indirectly, by executing indirect control-flow transfer function (e.g. indirect function call, indirect jump, or function return instruction). Finally, execute the attacker specified malicious code (present in data). A combination of Non-executable Data and Code Integrity results in the Write XOR Execute. This is cheap and easily applicable except for the case of JIT compilation or self modifying-code

To bypass the non-executable data policy, attackers can reuse existing code in memory.For example, return-to-libc. Libc contains usefull calls, such as system calls. Hijack the control-flow to execute some system call to get access to the system or whatnot. Reused code can also be small instruction sequences (gadgets) found anywhere in the code that can be chained together (Return oriented programming). To bypass the non-executable data policy, attackers can reuse existing code in memory.

There is no policy which can stop the attack at this point, since the execution of valid and already existing code cannot be prevented.

We classify a control-flow hijacking attack successful as soon as attacker-specified code starts to execute. There are higher level policies that can restrict attacker (file permissions, sandbox), but the focus of this paper is preventing the compromise of trusted programs, so there are not covered in this paper.

Data-only attack The target of the corruption can be any security critical data in memory, e.g., configuration data, the representation of the user identity, or keys. (setting bool isAdmin to true). Same steps as before, except the target of corruption is a variable, not a code pointer. Requires knowledge of the variable you want to overwrite (where it is, what value it needs to be). Similar to code pointer corruption, data-only attacks will succeed as soon as the corrupted variable is used.

information leak We showed that any type of memory error might be exploited to leak memory contents, which would otherwise be excluded from the output. This is typically used to circumvent probabilistic defenses based on randomization and secrets.

1.5 Currently used protections and real world exploits

Most widely deployed protection mechanisms are stack smashing protection, DEP/Write XOR Read and ASLR. Stack smashing protection detects buffer overflows of local stack-based buffers. Place a random value (canary) between return adress and the local buffers and check integrity before returning. Example of a Control-flow integrity solution. These techniques provide the weakest protection: they place checks only before a small ubset of indirect jumps, focusing checking the integrity of only some specific code pointers, namely saved return addresses and exception handler pointers (in case of SafeSEH and SEHOP) on the stack.

1.6 approaches and evaluation criteria

Two main categories: probabilistic and deterministic protection. Probabilistic solutions, e.g., Instruction Set Randomization, Address Space Randomization, or Data Space Randomization, build on randomization or encryption. All other approaches enforce a deterministic safety policy by implementing a low-level reference monitor which observes the program execution and halts it whenever it is about to violate the given policy. Reference monitors enforcing lower level policies, e.g., Memory Safety or Control-flow Integrity, can be implemented efficiently in two ways: in hardware or by embedding the reference monitor into the code.

Since adding new features to the hardware is unrealistic, from this point we focus only on solutions which transform existing programs to enforce various policies

Dynamic (binary) instrumentation can be used to dynamically insert safety checks into unsafe binaries at run- time. Dynamic binary instrumentation supports arbitrary transformations but introduces some additional slowdown due to the dynamic translation process. Simple reference monitors, however, can be implemented with low overhead. Static instrumentation inlines reference monitors statically. This can be done by the compiler or by static binary rewriting. Inline reference monitors can implement any safety policy and are usually more efficient than dynamic solutions, since the instrumentation is not carried out at run-time.

1.6.1 Properties, requirements

- enforced policy
- false negatives
- false positives
- performance overhead
- memory overhead
- source compatibility. Most experts from the industry consider solutions which require porting or annotating the source code impractical.
- binary compatibility. Binary compatibility allows compatibility with unmodified binary modules
- modularity support. Support for modularity means that individual modules (e.g. libraries) are handled separately.

groot stuk van de rest van de paper is een beschrijving van defences tegen verschillende attacks. Voor complete beschrijving, lees paper of Google **probabilistic methods**

- Address space randomization
- Data space randomization

Memory safety Our focus is transforming existing unsafe code to enforce similar policies by embedding low-level reference monitors.

- Spatial safety with pointer bounds. SoftBound addresses the compatibility problem by splitting the metadata (contains pointer bounds) from the pointer, thus the pointer representation remains unchanged
- Spatial safety with object bounds. associate bounds information with the object instead of the pointer. object based techniques focus on pointer arithmetic instead of dereferences. One problem is that pointers can legitimately go out of bounds as long as they are not dereferenced. For example a pointer typically goes off the array by one,but it is not dereferenced.

temporal safety

1. Special allocators. The naive (wasteful) approach to protect against use-after-free exploits would be to never reuse the same virtual memory area. Special memory allocators, like Cling, ... Cling is a replacement for malloc, which allows address space reuse only among objects of the same type and alignment

2. Object based approaches. These tools try to detect use-after-free bugs by marking locations which were de-allocated in a shadow memory space. Accessing a newly de-allocated location can be detected this way. ??

3. Pointer based approaches.

Data integrity

1.7 Generic attack defenses

bla bla bla

2 What you corrupt is not what you crash: challenges in fuzzing embedded devices

2.1 Introduction

1. fuzz-testing, or "fuzzing", describes the process of automatically generating and sending malformed input to the software under test, while monitoring its behavior for anomalies [51]. Anomalies themselves are visible ramifications of fault states, often resulting in crashes

2. embedded devices often lack such mechanisms (to detect and analyze faulty states) because of their limited I/O capabilities, constrained cost, and limited computing power. As a result, silent memory corruptions occur more frequently on embedded devices than on traditional computer systems, creating a significant challenge for conducting fuzzing sessions on embedded systems software.

3. Indeed, the only way left to identify a successful memory corruption is to monitor the [emdedded] device to detect signs of an incorrect behavior

4. In this paper, we show that these "liveness" checks are insufficient to detect many classes of vulnerabilities because it is often difficult to detect in a blackbox manner when the internal memory of the embedded device has been corrupted.

2.2 Fuzzing embedded systems

1. Most of the theory behind fuzzing and most of the available fuzzing tools were designed to test software running on desktop PCs. As we will discuss later in this paper, there are a number of relevant differences that makes fuzzing embedded system particularly challenging

2. **classes of embedded devices**

   (a) General purpose OS-based devices: general purpose OS, retrofitted to suit embedded system. For example Linux OS, coupled with lightweight user space environments.

   (b) Embedded OS-based devices: custom operating systems for embedded devices. a logical seperation between kernel and application code is present

   (c) Devices without an OS-Abstraction: "monolithic firmware" - operation is typically based on a single control loop and interrupts triggered from the peripherals in order to handle events from the outer world

3. **Main challenges of fuzzing embedded devices**

3 Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic

3.1 Introduction

1. today, software components (of different stakeholders) are isolated with help of a sizeable privileged software layer (part of the OS?). This can be vulnerable to bugs
2. in response: Protected Module Architectures (PMA). They safeguard *enclaves* (security sensitive application components) by enforcing isolation and attestation primitives directly in hardware or in a small hypervisor (software that creates/runs VMs)
3. the untrusted OS is prevented from accesing enclave code, data
4. PMA has black box view – kernel-level attacker should only be able to observe input and output.
5. **enclave-internal behavior may still leak through the CPU's underlying architectural state** – side-channel attacks exploit this weakness
6. *controlled-channel attack* when PMA's are targeted, the OS itself has become an untrusted agent, two major consequences (see paper)
7. *We abuse the key microarchitectural property that hardware interrupts/faults are only served upon instruction retirement, after the currently executing instruction has completed, which can take a variable amount of CPU cycles depending on the instruction type and the microarchitectural state of the processor*
8. *delaying interrupt handling until instruction retirement introduces a subtle timing difference that by itself reveals side-channel information about the interrupted instruction and the microarchitectural state when the interrupt request arrived*
9. *an untrusted operating system can exploit this timing measurement when interrupting enclaved instructions to differentiate between secret-dependent program branches, or to extract information for different side-channel analyses*

3.2 BACKGROUND AND BASIC ATTACK

3.2.1 Attacker Model and Assumptions

1. The adversary's goal is to derive information regarding the internal state of an enclaved application
2. we consider an adversary with (i) access to the (compiled) source code of the victim application, and (ii) full control over the Operating System (OS) and unprotected application parts.
3. we assume the untrusted OS can securely interrupt and resume enclaves
4. In this paper we focus exclusively on hardware-level security monitors
5. We assume that enclaves can be interrupted repeatedly within the same run
6. Nemesis-type interrupt timing attacks only assume a generic stored program computer with a multi-cycle instruction set, where each individual instruction is uninterruptible (i.e., executes to completion)

3.2.2 Fetch-Decode-Execute Operation

uitleg over hoe die werkt – zie paper

3.2.3 Basic Nemesis Attack

1. We consider processors that serve interrupts after the execute stage has completed
2. Our attacks are based on the key observation that an IRQ during a multi-cycle instruction increases the interrupt latency with the number of cycles left to execute – where interrupt latency is defined as the number of clock cycles between arrival of the hardware IRQ and execution of the first instruction in the software ISR
3. When interrupt arrival time is known (e.g., generated by a timer), untrusted system software can infer the duration of the interrupted instruction from a timestamp obtained on ISR entry.
4. figure 2 example: attack for an enclaved execution that branches on a secret – trigger interrupt request and measure the interrupt latency to infer the number of clock cycles needed by the instruction after the conditional jump. This breaks the black box view on protected modules by leaking a piece of the CPU's microarchitectural state (in this case the instruction opcode)
5. Nemesis-type interrupt timing attacks exploit secret-dependent control flow
6. The main difficulty for a successful attack lies in determining a suitable timer value so as to interrupt the instruction of interest (the one after the conditional jump in the example)

4 Static Detection of Side Channels in MSP430 Programs

4.1 Introduction

1.