Co-funded by the Erasmus+ Programme of the European Union

ISSES – Information Security Services Education in Serbia

Supported by the Erasmus+ Capacity Building in the field of Higher Education (CBHE) grant N° 586474-EPP-1-2017-1-RS-EPPKA2-CBHE-JP

MEMORY MANAGEMENT FUNDAMENTALS

Computer Security

Lecture 8

ISSES 2017-2020
Novi Sad, Serbia

Information Security Services Education in Serbia (ISSES)

8.1 INTRODUCTION

Purpose of lecture

- The purpose of this lecture is to introduce a sequence of deep-dive lectures on attacks based on memory.
- We have already touched on memory many times so far: memory is one of the main things a computer security system protects and one of the main targets of attack.
- This, however, will be a methodic introduction into the field and the types of attacks and counters in context of operating systems.
- The purpose of this first lecture is to first understand, in detail, how managing memory works in two of the main modern architectures.

Contents

- We first cover, in sections 8.2 the memory organization and management modes of the x86 and x86_64 architectures paying attention to both current modes of operation and historic modes of operation where needed.
- Then we'll look, in section 8.3 at how ARM does it.
- Section 8.4 uses what was covered in sections 8.2 and 8.3 to introduce the first sort of common memory attack: the stack attack. The stack attacks is discussed on a real example.

Links to previous material

- You will need what you learned in lecture 2, specifically 2.7.
- It’s there where we first introduced memory translation rules and operating modes.
- We will be relying on this information now, but expanding it and including information about ARM.
- Certain technical information will be repeated from the 2.7 section in order to provide context, but you are expected to have seen it already and for it to be familiar to you.

Links to upcoming material

- Our entire memory deep-dive will rely on work here.
- Lecture 10 on memory attack mitigation will also rely heavily on foundations from this lecture as it discusses largely alternations to Level-4 paging which produce additional security.

Sources

- The chief reference source for x86 information is the official Intel® manual on the architecture accessible on the official site. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html
- The chief reference source for ARM information is the official ARM architecture manual for the ARMv8 and Armv8-A architecture profiles available on the official site: https://developer.arm.com/documentation/ddi0487/latest/

Information Security Services Education in Serbia (ISSES)

8.2 x86 Memory Management

Terminological note

- We will be periodically discussing subtypes of modern PC architecture and we'll need terms.
- So far we've used the commonly used 'x86' and 'x86_64' to indicate the 32-bit and 64-bit modern PC architectures.
- This is, strictly speaking, incorrect.
- The 64-bit version of the architecture is indeed x86_64, but the 32-bit version is most correctly referred to as IA-32.
- It should be noted that IA-64 **does not** refer to x86_64. It refers to the Itanium architecture which is entirely distinct and is being sunsetted in 2021.
- X86 does not, strictly, refer to x86, as this is a broader term and applies to, e.g. earlier 16-bit architectures.
- We will be following Intel and using **IA-32** and **x86_64**.

Further terminological note

- Another important distinction is between IA-32 and IA-32e.
- IA-32 has been explained already.
- IA-32e is the mode used by most modern CPUs.
- Given the propensity for still publishing 32-bit code and the use of legacy code, a CPU may need to execute both 32bit and 64bit code one after another, in, say, a modern operating system.
- IA-32e enables that and can be in 32-bit mode and 64-bit mode.
- We will mostly use x86_64 mode and IA-32e mode interchangeably, as the subtle distinction is largely not important for our purposes, however, occasionally for the sake of correctness IA-32e will be mentioned.

Basics

- The memory addressed on the system bus is called physical memory.
- Physical memory is a sequence of bytes each of which is uniquely addressable using its **physical address.**
- A physical address on IA-32 is in the range of 0 to $2^{36} – 1$, meaning a maximum addressable memory of 64 GB. Note that this is only possible to use through the paging mechanism through Extended Physical Addressing.
- A physical address on an x86_64 is in a range *at most* of 0 to $2^{64} – 1$. In practice, it is less and dependent on the implementation. Modern CPUs have either 48 or 52 bits implemented. To work, addresses must be 64-bit but in canonical form, i.e. with bits above the highest implementation-provided set to the value of the highest.

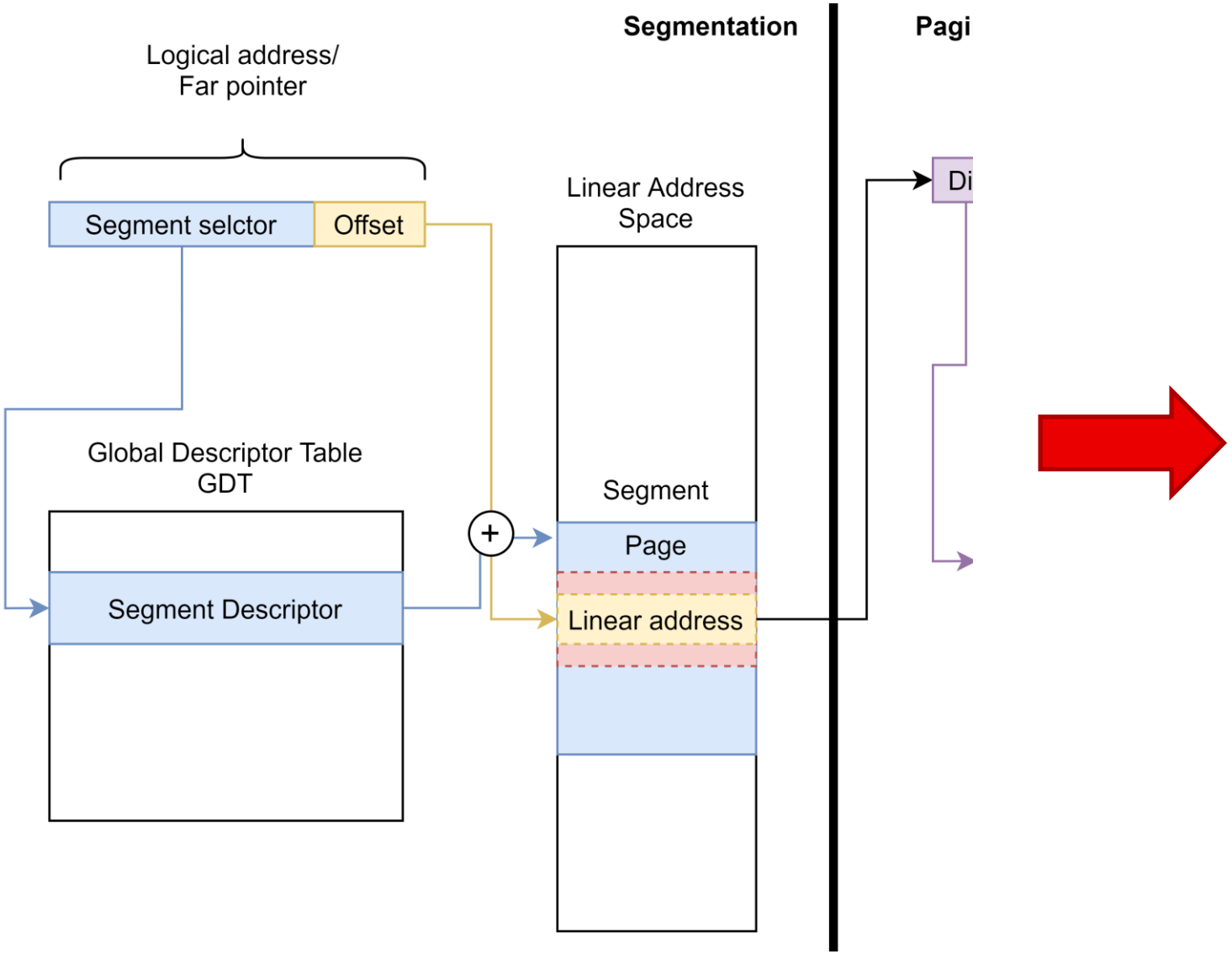
Canonical addresses

Address	Canonical form
1011 1010 1100 0110 1111 0000 1001 0111 0110 1010 0011 1001	1111 1111 1111 1111 1011 1010 1100 0110 1111 0000 1001 0111 0110 1010 0011 1001
0011 1010 1100 0110 1111 0000 1001 0111 0110 1010 0011 1001	0000 0000 0000 0000 0011 1010 1100 0110 1111 0000 1001 0111 0110 1010 0011 1001
1100 0011 1010 1100 0110 1111 0000 1001 0111 0110 1010 0011 1001	1111 1111 1111 1100 0011 1010 1100 0110 1111 0000 1001 0111 0110 1010 0011 1001

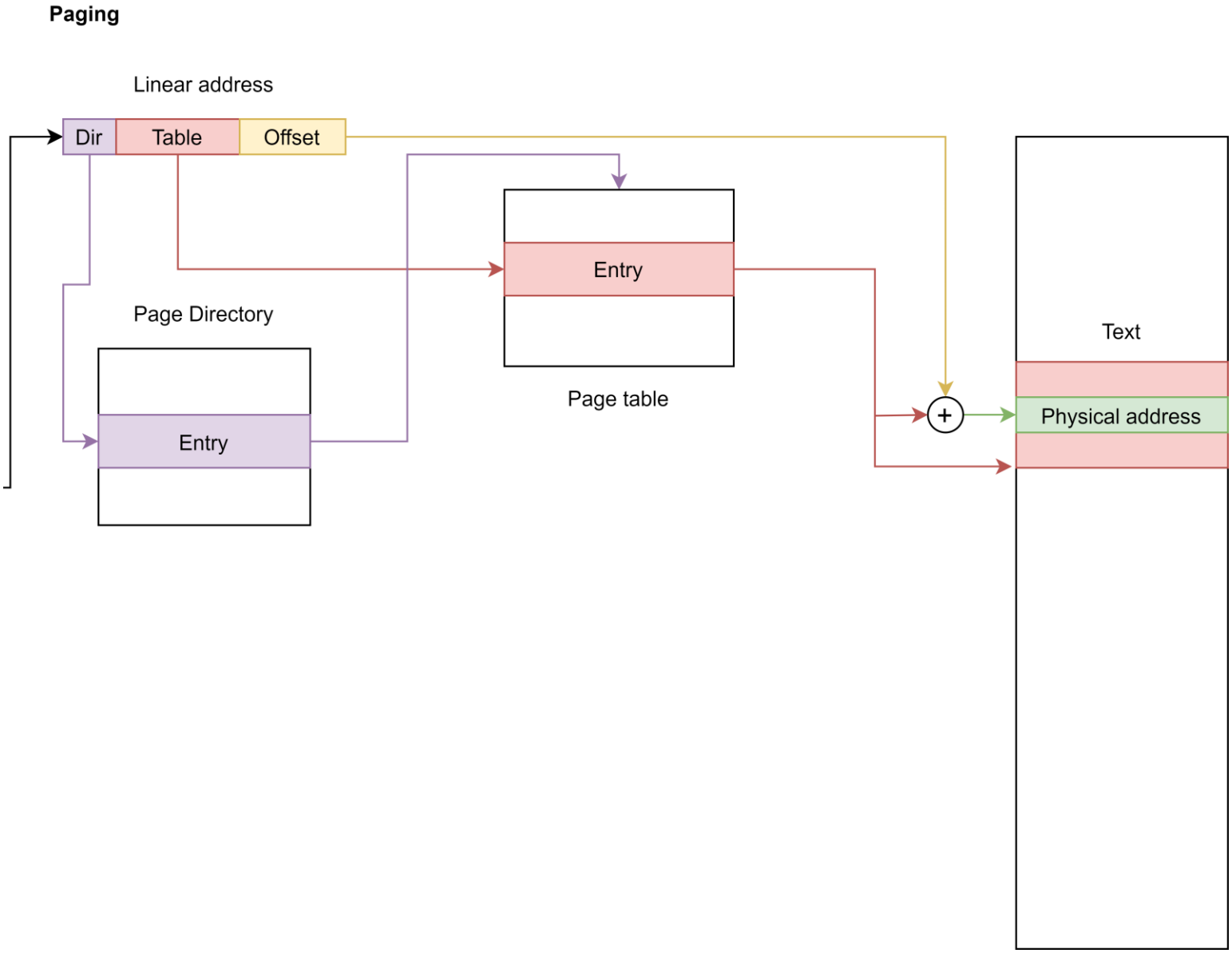
Types of address

- **Logical** addresses are what appear in code when addressing something in a separate segment. Depending on the *mode* the CPU is in, these addresses can have different parts, typically a segment index and a logical offset, i.e. which segment is being worked with and where in the segment the requested data is.
- **Linear** addresses are addresses that were transformed from various types of segmented modes to take a segment into account. For most modes, the linear is also the physical, however. Linear addresses belong to the addressable range of a processor.
- **Physical** addresses are equal to linear addresses unless paging is used. In the case of paging, a physical address is made by using the first 20bits of an address as page index.

Addressing schema



Addressing Schema



Global Descriptor Table

- The global descriptor table is where segments (among other things) are stored.
- Segments are still used and are relevant for security, but are not used for segmented addressing anymore.
- Segmentation table entries may define segments, but also specialized other entries including:
  - Task-State segments contain the state of a task being switched to. This can be used to simplify task switching by just using a CALL or a JMP with a segment selector set to a TSS. This only works in IA-32 mode, not in IA-32e mode.
  - LDTs are local descriptor tables and contain much the same as the GDT, but for local use.

Entry of the GDT

Field	Bits	Purpose
Base1	63:56	The first part of the base value of the segment.
G	55	Limit in bytes or 4kb units?
D/B	54	Compatibility; 1 iff 32bit code
L	53	Compatibility; 1 iff 64bit code
AVL	52	Reserved
Limit	51:48	Limit (size) of the segment
P	47	Validity of entry; whether the segment is there
DPL	46:45	0-3, privilege level, mentioned before.
S	44	System segment?
Type	43:40	Type specifier
Base2	39:32	Second part of the base.
Base3	31:16	Final part of the base.
Limit	15:0	Final part of the limit.

On type

- Depending on the state of the S bit the type can refer to a type of an application segment or a type of a system segment.
- Application segments are ‘plain old’ segments
- System segments are all the ‘other stuff’ that can be in a GDT.
- 16 separate types exist: for app segments we are told whether it is a data or a code segment (bit 3).
- Data segments allow us to mark read-only vs. read-write segments (bit 1) and also to track whether the segment has been accessed (bit 2), and the direction of expansion (bit 0).
- Expansion direction controls the direction in which a segment grows, if need be.

On type

- Code segments allow us to mark, aside from access which is the same, whether the code is execute-only or execute-read(bit 1) and whether the code is *conforming* (bit 0).
- Note that that no code segment is ever writeable, and no data segment is ever executable.
- Conforming code can be called out of code of a lower privilege level.
- Nonconforming code cannot and will create a general protection exception if called from lesser-privilege code.
- Conforming code has been invented for the purpose of system code which need not be protected. Intel suggests things like divide error exception handlers.
- Needless to say, this is a potential vulnerability.

System types

- As stated before, we can put ‘other things’ into descriptor tables. These include:
  - LDT segment descriptors.
  - TSS descriptors.
  - Call-gate descriptors.
  - Interrupt-gate descriptors
  - Trap-gate descriptors.
  - Task-gate descriptors.
- Where the GDT is, is defined by special register (GDTR register) which points to the first element at pos. 0 which is always reserved.
- Where the local table is, is also held in a register but *that* data changes and must come from somewhere: the LDT entry of the GDT.

Briefly on gates

- Gates are a mechanism, now largely superceded, for launching code which is in a different privilege level in a structured way.
- Specialized system call instructions (which we have covered already) have mostly replaced them.
- If you have been taught the old method of initiating a syscall from Linux assembly through ‘int 0x80’ you have been unwittingly using a gate, specifically a *trap* gate.
- It should be noted that call gates, and task gates, for instance, are supported *but aren’t used*
- In much the same way, hardware task switching is something Intel implemented (and we are stuck with, forever) but which isn’t in use anymore.
- These unused features are often where vulnerabilities lie.

Segmentation in IA-32e

- Compatibility mode (the normal 32bit kind) treats segmentation the same way normal 32-bit protected mode treats it.
- 64-bit mode, however, nearly completely ignores segmentation. The GDT still exists and can be used but generally the logical address is instantly converted to the linear address since all the segment registers are forced to zero.
- The GDT can still be used for ‘other things’ and generally speaking, is equipped with a full set of entries to support flat mode addressing where all of the userland data is always mapped and all of the kernel data is also always mapped (or was: see former lecture).

Paging

- The use of segment descriptors allows us to get to a linear address.
- The question is how to convert it into the real deal?
- The solution is the use of pages. All linear space is divided into fixed-size pages (typically 4096 bytes in size) and so is all of physical memory.
- A linear address is treated as consisting of a page address and an offset within that page. Given that we know the dimensions at play, the offset has a known number of bits (12), and the page address can be everything else.
- Physical memory is much smaller than possible linear addresses. Therefore *virtualization* is typically used: most pages are on disk and only a subset is in memory at any given time: the OS handles this behavior.

Types of paging

- x86_64 processors support three types of paging: 32-bit paging, PAE paging, and 4-level paging.
- The one we care about is 4-level paging as it is what's currently most likely to be used.
- 4-level paging also has some novel security features.

Mode of paging	Linear addr. width	Phys. Addr. width	Page sizes
None	32	32	/
32-bit	32	Up to 40	4KB 4MB
PAE	32	Up to 52	4KB 2MB
4-level	48	Up to 52	4KB 2MB 1GB

Hierarchical paging structures

- The way the information on all pages is kept is through *paging structures*
- In its base form a paging structure is a 512-entry table, with each entry being 64 bits long and a total structure being 4KB which is the pagesize.
- Aside from bookkeeping information, the entry always contains a *physical address* which will either be the address of a page frame (a region in physical memory to which a page in linear address space will map) or the address of another paging structure.
- We can imagine these as directories containing directories eventually containing files.

Decoding a linear addres with 4-level paging

Bits	Purpose
63:48	Not used – While addresses are 64-bit, only the lower 48 are used.
47:39	First paging structure entry index.
38:30	Second paging structure entry index
29:21	Third paging structure entry index
20:12	Fourth paging structure entry index
11:0	Offset within the page

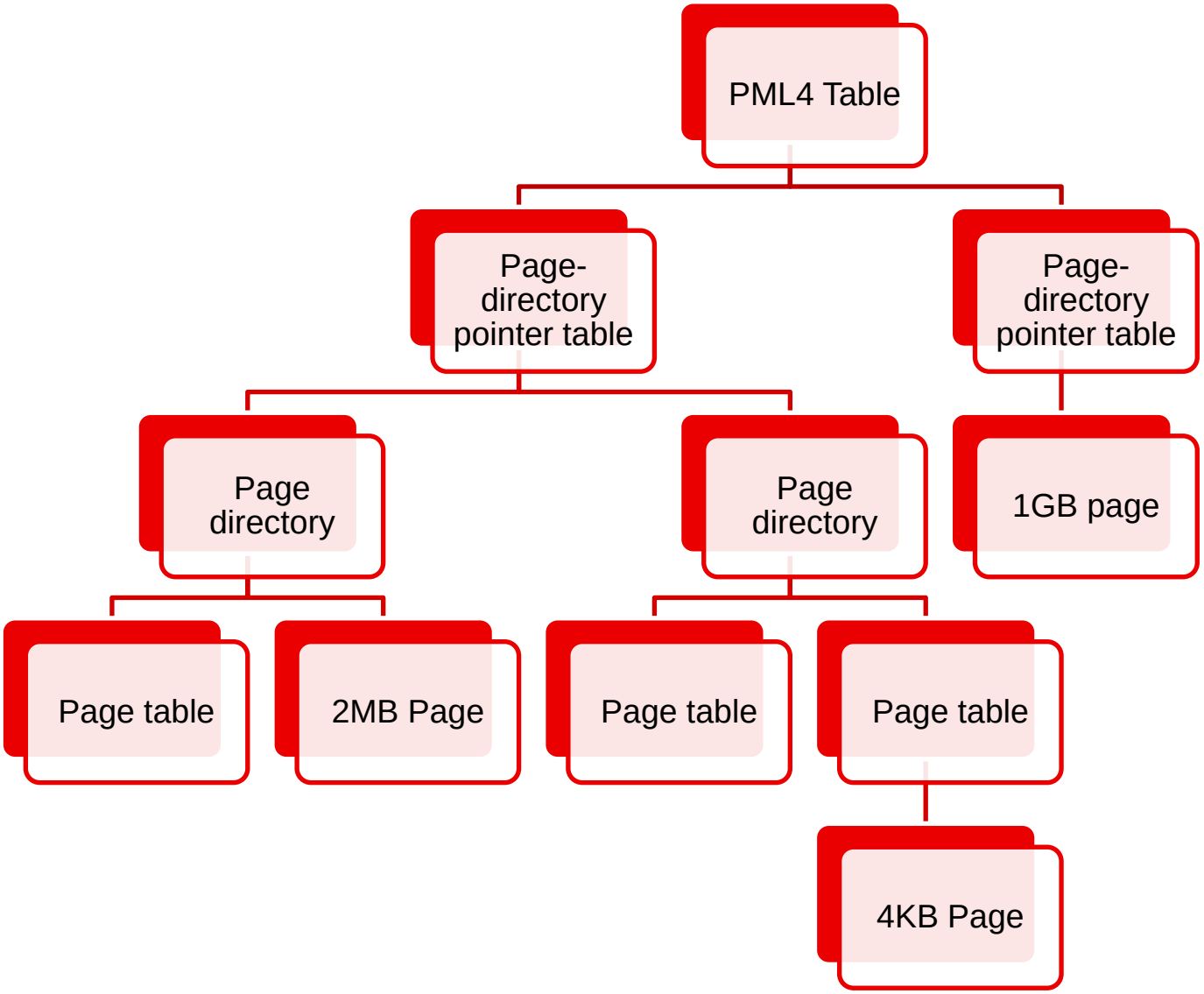
Translation procedure

- Normally, the root entry is located using the CR3 register (the OS is responsible for preparing the memory and setting this up).
- Then each entry is found using the index and followed until the page frame is found.
- However, the procedure may terminate early if the ‘not present’ bit is set.
- This generates a *page fault.*
- This is not a problem: the OS expects page faults, normally. That’s how it does memory virtualization.
- Not all translations take four steps to terminate. You can terminate earlier because of larger pages.

Larger pages

- If only 12 bits remain in the linear address being mapped then whatever is mapped to, if it is present, is a page frame for a 4kb page.
- However, if more than 12 bits remain and the PS bit in what we've called 'bookkeeping information' is set to '1' then whatever we are mapping to references not a paging structure, but to a *larger page.*
- How large? Well in case of a termination after only 3 layers of indirection, there's 9+12=21 bits of address remaining, leading to a page of 2^{21} bytes, or 2MB.
- In the case of termination after only two layers of indirection, there's 18+12=30 bits of address remaining, enough for 2^{30} bytes or 1GB.
- No other mapping is permitted.

Four-level structure



Security

- We will discuss the full sophisticated nature of the security mechanisms that protect memory in modern systems in Lecture 10.
- However, before we reach that point, we will only cover the bare minimum: each page has security information the most important of which is information which divides all pages into those that are *supervisor mode* and those which are *user mode.*
- This is similar to the privilege levels we've discussed both in lecture 2 and just now when discussing segments, however it collapses to just two levels after, in practice, nobody used levels 1 and 2 for anything much.

Information Security Services Education in Serbia (ISSES)

8.3 ARM Memory Management

ARM approach to memory

- ARM is a platform with a much wider intended field of application than IA-32 or x86_64.
- ARM CPUs power servers, laptops, phones, tablets, medical gear, and SoCs.
- To handle this it can be configured in three profiles: A, R, and M.

	Meaning	Purpose
A	Application	High-performance devices that run Operating Systems like Android, iOS, Linux, and Windows. Supports VMSA
R	Real time	Medical devices, avionics, anywhere with a lot of safety.
M	Microcontroller	Microcontrollers. Supports PMSA

ARM Approach to memory

- This course is mostly interested in the security of general-purpose operating systems operating largely in isolation.
- Microcontroler mode and Real-Time mode are, therefore, not relevant to our interests. Real-Time OS security is very important but involves a different approach than the one presented here.
- Therefore, given the nature of GPOSs, we focus on the A profile and how its MMU operates.
- This MMU is based on something ARM calls the Virtual Memory System Architecture.
- If you wish to follow along in the ARM reference manual, dection D5.1 is where you want to start looking.

ARM approach to memory

- We assume here that ARM is configured in a way which works very similarly to the way x86 paging worked in the previous section.
- This is a moderately safe assumption: it will help us understand the differences between architectures without being bogged down in detail.
- We will, therefore, focus in this sections on interesting *differences* between ARM and x86.
- All things not mentioned as different can be safely (for the purposes of this orientation) be assumed to be the same.

ARM approach to memory

- The ARM paging systems is very complex and flexible.
- A lot of features are meant to be implementation-dependent or configurable for a variety of uses.
- Therefore, it's not as easy to talk about what *does* happen as it's more of a case of what *can* happen.
- This configurability means that when dealing with ARM and when wanting to examine a vulnerability at that level a key question to ask is how *precisely* the chip is configured, as that may drastically effect chip behavior.
- It's best to treat whole categories of device with an ARM chip on them as devices of their own and not just ARM devices.

ARM memory terminology

- ARM does not rely on backwards compatibility as much as IA-32 and x86_64.
- This simplifies the translation procedure somewhat as there is no segmentation to worry about at all.
- As far as ARM is considered the only translation it does is from virtual addresses to physical addresses.
- VAs are either 48 or 52 bits long and are used whenever an address is used in code.
- Physical addresses are self-explanatory, but ARM also recognizes the term IPA—intermediate physical addresses.
- This is the intermediate result when doing a two-stage translation.

Granule size

- A granule size is the number that defines the granularity at which lookups operate in an ARM system.
- It affects page size, among other factors.
- Note the much smaller granularities than in 4-level paging.

Granule	Max number of entries in trans. table	Page size	Bits resolved in one lookup step
4KB	512	4KB	9
16KB	2048	16KB	11
64KB	8192	64KB	13

Granule size

- Setting the size of a granule to a value ahead of time means that the lookup procedure is known in advance.
- The bigger the granule the more of the output address bits come directly from the least significant bits of the input address.
- Meanwhile the most significant bits represent successive addresses into tables of various levels, at most four.
- The change to both the data and where the metadata is kept means that 64kb pages also imply correspondingly huge tables.

Block size

- An entry in a translation table holds in its second bit whether it describes a block.
- When the second bit is a zero, the entry describes a block: which is to say what we called a ‘large page’ in x86.
- Not all levels support this: blocks can only happen on levels 1 and 2 for 4KB granules, level 2 for 16KB granules, and levels 1 and 2 for 64KB granules.
- 4KB granules act just like x86: 2MB and 1GB blocks are formed when we stop resolution at the second and first levels of indirection, respectively.
- 16KB granules can lead to 32MB blocks.
- 64KB granules can make 512MB and 4TB pages on levels two and one. Needless to say, not a lot of systems support 4TB pages.

Information Security Services Education in Serbia (ISSES)

8.3 Stack Attacks

Introduction

- We introduce stack attacks here as a particularly simple form of attack to whet the appetite for memory-based attacks.
- Stack attacks are based on exploiting the fact that the stack contains sensitive app-relevant information and general data which may come from an outside, untrusted and malicious source.
- The weakness they exploit is the fact that data which governs execution and user data share the same space and can interact in dangerous ways.
- More on this sort of attack later.

Layout of the stack

- The system stack should be well known to any experienced programmer
- It's a region of memory that's a part of the process image (alongside the data and the 'text' i.e. the code) and is used for a few crucial things:
  - Storing local variables
  - Storing parameters
  - Storing information needed for calling subroutines including:
    - Previous frame pointers
    - **Return addresses**
    - Return values
- Stacks in most architectures are so designed that they grow towards negative addresses, i.e. for a stack 'ahead' is in minus values, and behind is in positive ones.

The return addresses

- Why is the return address on the stack in the first place?
- If we call a subroutine, we want some method of coming back from where we called it.
- Worse yet, we want this to be stored in a way which permits subroutines to call subroutines, recursively if needed.
- Luckily, this fits perfectly well with the stack structure as it fits recursion perfectly well.
- However, this means that information on *what code will be executed next* is stored on the same stack as, say, user input.
- This is the ultimate cause of the stack vulnerability. Why?

Simple stack smashing

- The simple attack isn't particularly dangerous, but it allows for an illustration on how the return address is powerful.
- To that end, we'll smash our own stack, and show how manipulation with the return value can change the behavior of our program, even though it really shouldn't.
- Once we've mastered that we'll cover the theoretical basics in this lecture, and you'll get to do it all in the lab.
- Note for people who want to try it home: none of these examples will work unless you follow the lab results.
- This is because modern software's too sophisticated for simple stack smashing. Lecture 10 will explain which defenses in particular I had to cut out to get the desired effect.

Stack layout

- Consider the following main function of a trivial program:

```
int main(){
    int x;
    x = 0;
    function(1,2,3);
    x=1;
    printf("%d\n", x);
}
```

Stack layout

- If we disassemble this code we get:

```
0x000000000000117b <+0>:     endbr64
0x000000000000117f <+4>:     push   %rbp
0x0000000000001180 <+5>:     mov    %rsp,%rbp
0x0000000000001183 <+8>:     sub    $0x10,%rsp
0x0000000000001187 <+12>:    movl   $0x0,-0x4(%rbp)
0x000000000000118e <+19>:    mov    $0x3,%edx
0x0000000000001193 <+24>:    mov    $0x2,%esi
0x0000000000001198 <+29>:    mov    $0x1,%edi
0x000000000000119d <+34>:    callq  0x1149 <function>
0x00000000000011a2 <+39>:    movl   $0x1,-0x4(%rbp)
0x00000000000011a9 <+46>:    mov    -0x4(%rbp),%eax
0x00000000000011ac <+49>:    mov    %eax,%esi
0x00000000000011ae <+51>:    lea    0xe4f(%rip),%rdi
0x00000000000011b5 <+58>:    mov    $0x0,%eax
0x00000000000011ba <+63>:    callq  0x1050 <printf@plt>
0x00000000000011bf <+68>:    mov    $0x0,%eax
0x00000000000011c4 <+73>:    leaveq
0x00000000000011c5 <+74>:    retq
```

Stack layout

- The circled code is how a function gets called.
- If you've had assembly, you may have had it in a different architecture, possibly IA-32, so you might be used to an older calling convention for functions.
- However in x86_64 which this is compiled for, note that we are sending all parameters not through the stack but through registers.
- We still manipulate the stack, preserving our stack frame, and crucially expanding it to store additional data.
- Sub-ing 0x10 from the %rsp means that we increase the stack by 16 bytes. We'll see why later.
- This is the standard preamble to calling functions and happens every time.

Stack layout

- Consider this function

```
void function(int a, int b, int c){
    char buffer1[5];
    char buffer2[10];
}
```

Ignore for a moment the fact that it doesn't *do* anything useful, it still has code. So what does this code look like?

Stack layout

```
0x0000000000001149 <+0>:     endbr64
0x000000000000114d <+4>:     push   %rbp
0x000000000000114e <+5>:     mov    %rsp,%rbp
0x0000000000001151 <+8>:     mov    %edi,-0x14(%rbp)
0x0000000000001154 <+11>:    mov    %esi,-0x18(%rbp)
0x0000000000001157 <+14>:    mov    %edx,-0x1c(%rbp)
0x000000000000115a <+17>:    nop
0x000000000000115b <+18>:    pop    %rbp
0x000000000000115c <+19>:    retq
```

Note saving our own frame and also saving registers into local variables. We don’t have to make for the two buffers, as that’s been done for us. The code between 0x00 and 0x10 in relation to the rbp is ours already.
But is there anything else on the stack of any use?

Stack layout

Negative							Positive
Variable:	c	b	a	buffer2	buffer1	ebp	Return
Bytes:	4	4	4	16 bytes, with padding because of word alignment.		8	8
Purpose	/	/	/	/	/	Prev. frame	The return address

This is what we expect the stack to be like, but have we any proof? We know stacks like to pad extensively to make sure everything's a multiple of the principal word of the system for the sake of efficiency and performance. How to be sure? Answer: experiment.

Reading stack layout

```
(gdb) break function
Breakpoint 1 at 0x1149
(gdb) run
Starting program: /home/veljko/prg/sec/stack/s01

Breakpoint 1, 0x000055555555149 in function ()
(gdb) info frame
Stack level 0, frame at 0x7fffffffe2e0:
 rip = 0x555555555149 in function; saved rip = 0x55555555518b
 called by frame at 0x7fffffffe300
 Arglist at 0x7fffffffe2d0, args:
 Locals at 0x7fffffffe2d0, Previous frame's sp is 0x7fffffffe2e0
 Saved registers:
  rip at 0x7fffffffe2d8
```

This handy set of commands gave us the precise location of the saved program counter, i.e. the return address. And by simply printing out the value of buffer1 in gdb code, we can get the address 0x7fffffffe2c3, too.

Planning our attack

- Once we've determined these two values, the offset between them is trivial to compute: 21.
- This means that to get from the buffer1 address to the return values we need to move in the positive direction (back for stacks) 21 bytes.
- Once there we can write whatever we wish.
- In this instance, we want the code to write '0' instead of '1' as it always must, just to show we can later the control flow from inside a function.
- How to do this? Let's look at the disassembly of main once again:

Disassembly of main

```
0x000055555555519d <+34>:    callq  0x555555555149 <function>
0x00005555555551a2 <+39>:    movl   $0x1,-0x4(%rbp)
0x00005555555551a9 <+46>:    mov    -0x4(%rbp),%eax
0x00005555555551ac <+49>:    mov    %eax,%esi
0x00005555555551ae <+51>:    lea    0xe4f(%rip),%rdi        # 0x555555556004
0x00005555555551b5 <+58>:    mov    $0x0,%eax
0x00005555555551ba <+63>:    callq  0x555555555050 <printf@plt>
0x00005555555551bf <+68>:    mov    $0x0,%eax
0x00005555555551c4 <+73>:    leaveq
0x00005555555551c5 <+74>:    retq
```

The function is called at offset 34, at te top. The placing of 1 into X is done immediately after, at 39. This is our return value: this is where the function will return to once it finishes. So let's make it skip it. All we have to do is alter the return value by the difference between 46 and 39.

This is seven.

The code

```c
#include <stdio.h>

void function(int a, int b, int c){
    char buffer1[5];
    char buffer2[10];
    long *ret;

    ret = (long*)((long)buffer1 + 21);
    (*ret) += 7;
}

int main(){
    int x;
    x = 0;
    function(1,2,3);
    x=1;
    printf("%d\n", x);
}
```

The result

- Reading the code the only possible outcome is for this code to print 1.
- Why? Because we set one and immediately print the result. What else could it be?
- Despite that, in the correct environment it will print 0, as the stack manipulation will make the return address point at the command *after* setting x to 1.
- Normally, functions can’t affect things outside, but this one *can.*

Practical use?

- It is impressive that we can ruin control flow from inside a supposedly encapsulated function, but it can be improved upon.
- The central idea is actually deceptively simple.
- We have control over the stack if someone foolishly reads user input in a way which can cause a *buffer overflow.*
- Let’s say that strcpy or gets are used which just read into a buffer until they hit a null sign.
- This means they can easily continue writing past a buffer’s end and this is exactly what we want.