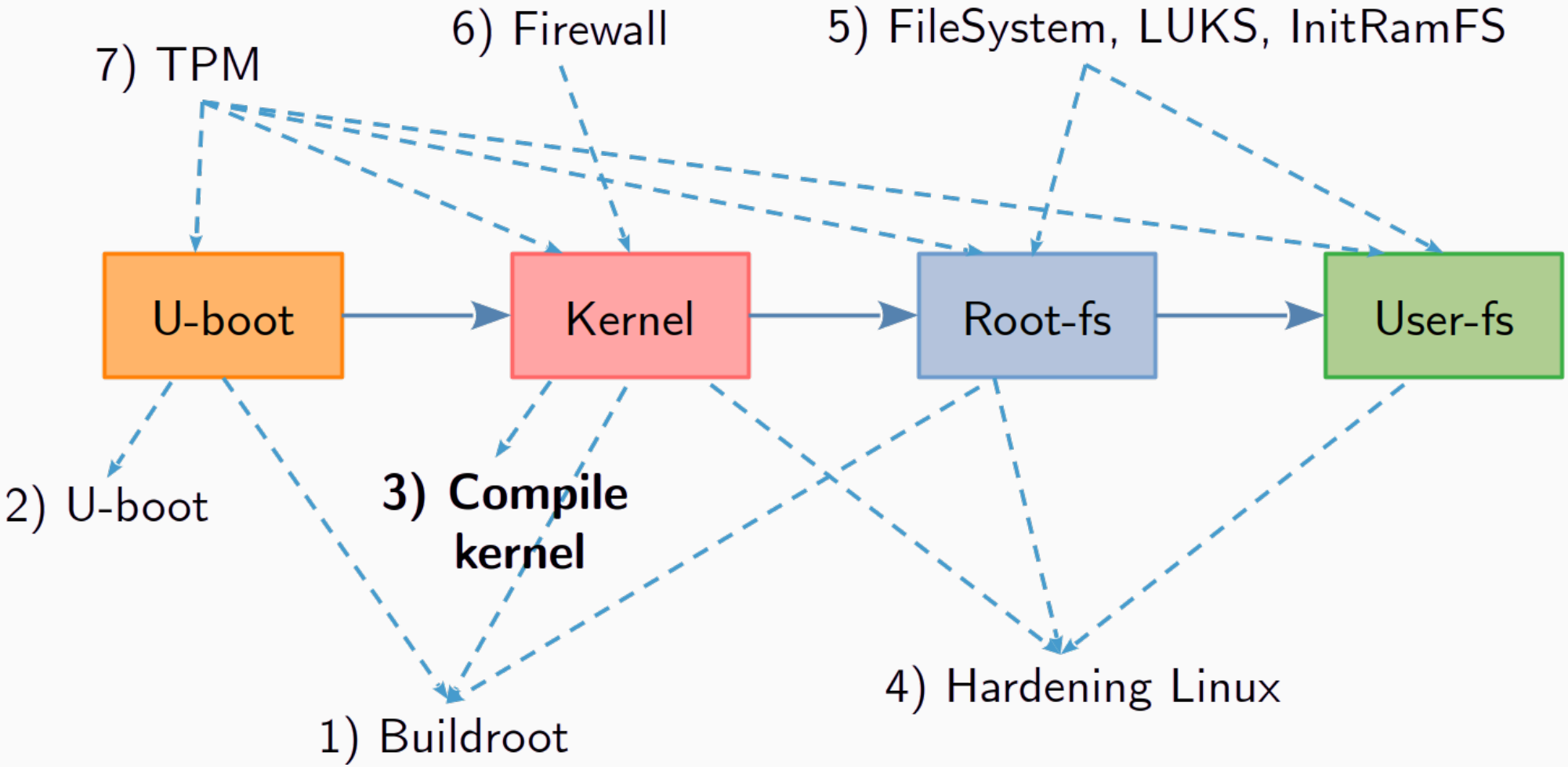
MSE | MASTER OF SCIENCE IN ENGINEERING

Linux kernel configuration and hardening

Florent Glück, Jean-Roland Schuler

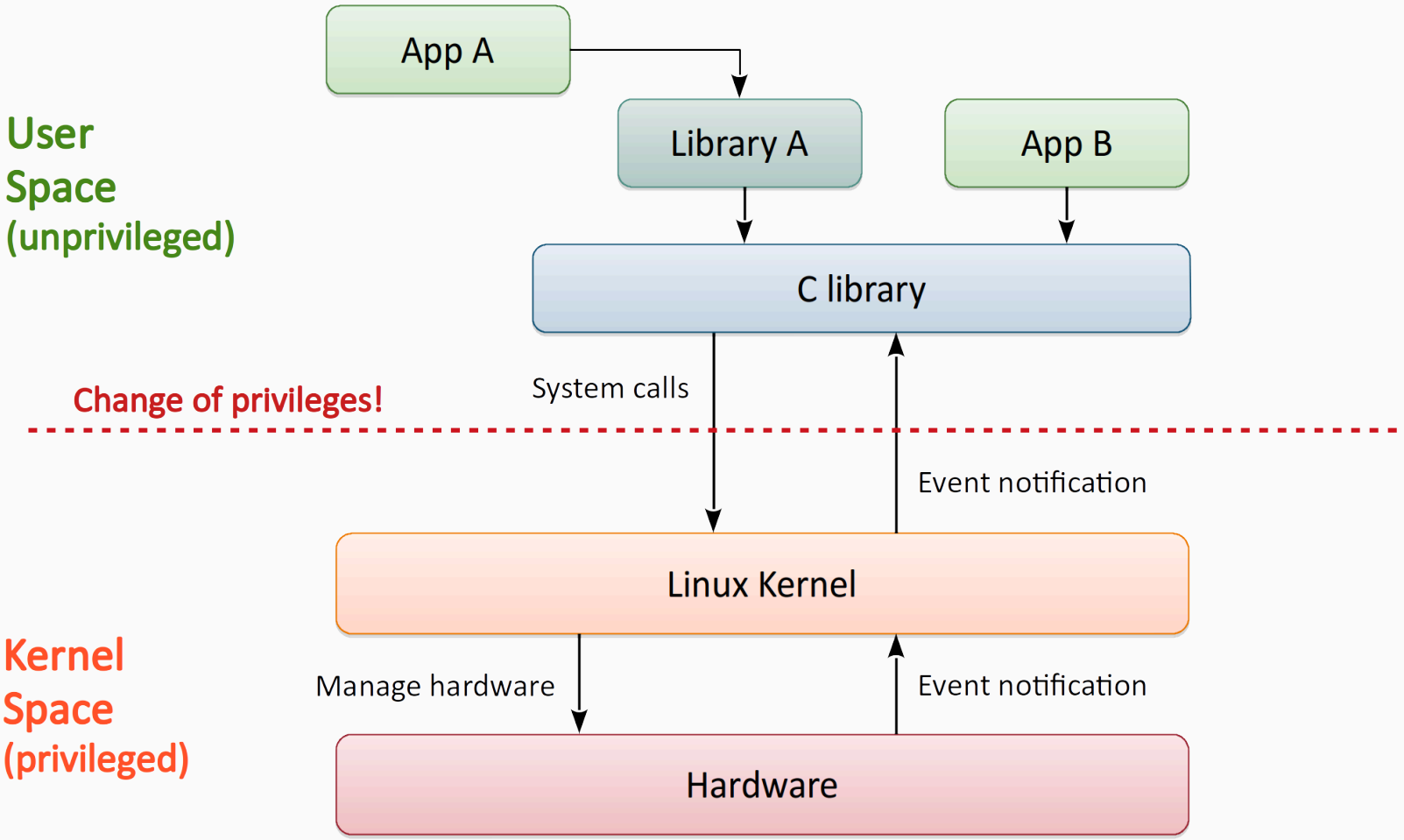
October 20, 2025

Overview


7) TPM
6) Firewall
5) FileSystem, LUKS, InitRamFS
U-boot
Kernel
Root-fs
User-fs
2) U-boot
3) Compile kernel
4) Hardening Linux
1) Buildroot

Reminder

CPU execution privilege: kernel and userspace

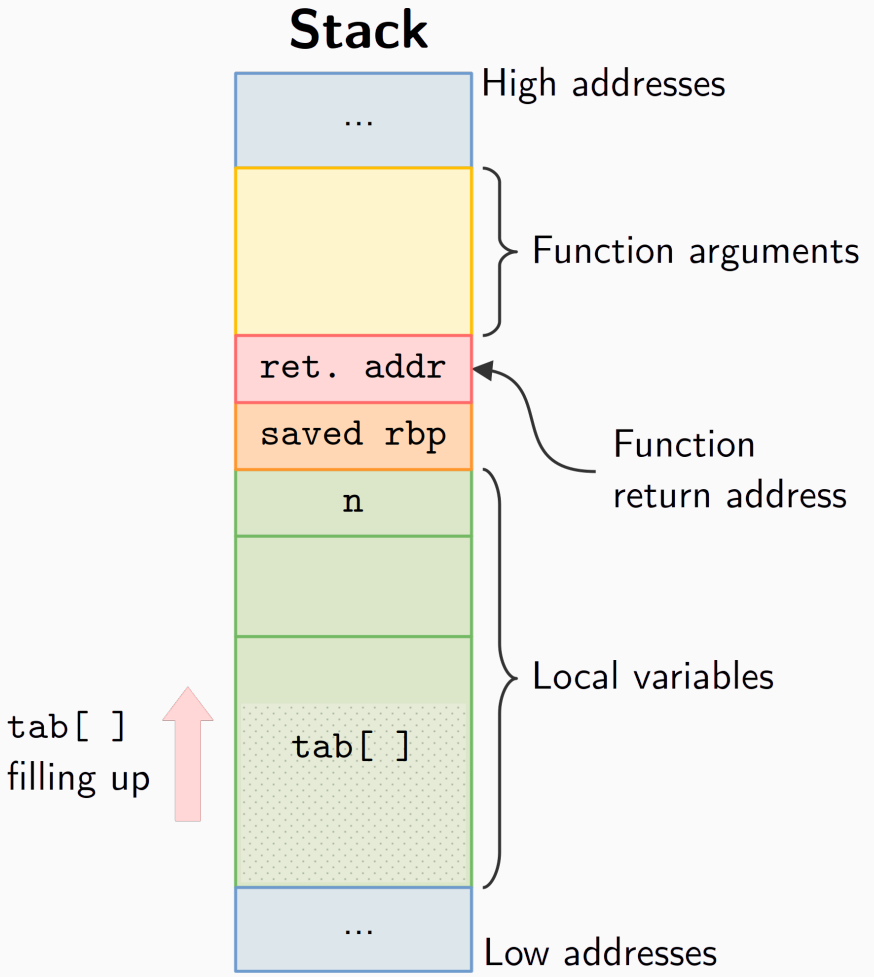


Common attacks and protections techniques

Buffer overflow

- Code writes (or reads) outside of the buffer's memory area
- Example:

```
void main () {
    char buffer[4];
    // "buffer" is the same
    // as "&buffer[0]"
    strcpy (buffer, “123456”);
}
```

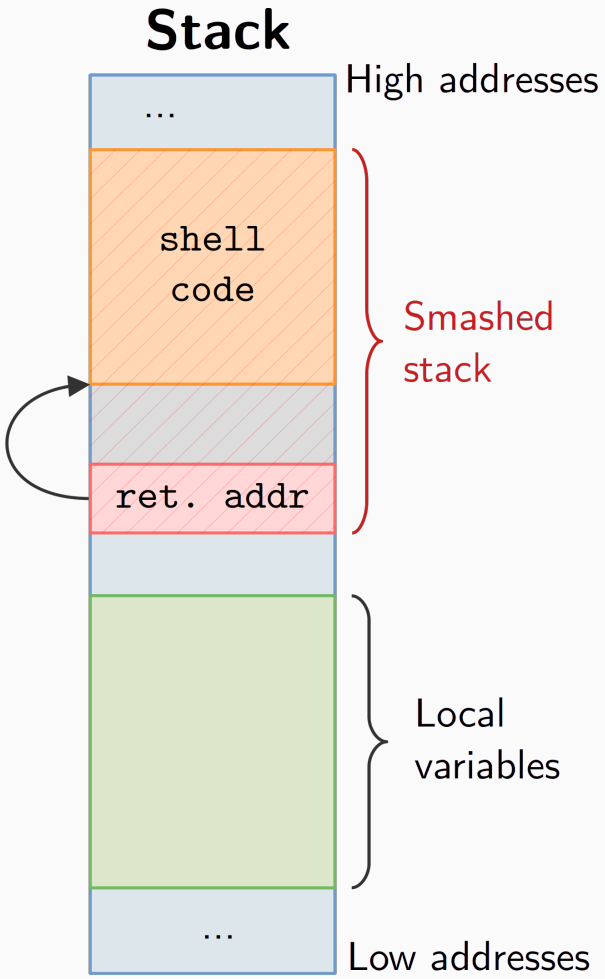


Buffer overflow attack

In an buffer overflow attack, the attacker **overwrites (smash) the stack with some malicious code**, e.g.:

- Overwrites the stack with the code of a shell
- Overwrites the function's return address with the address of the attacker' shell

ⓘ Only works if the stack is executable!



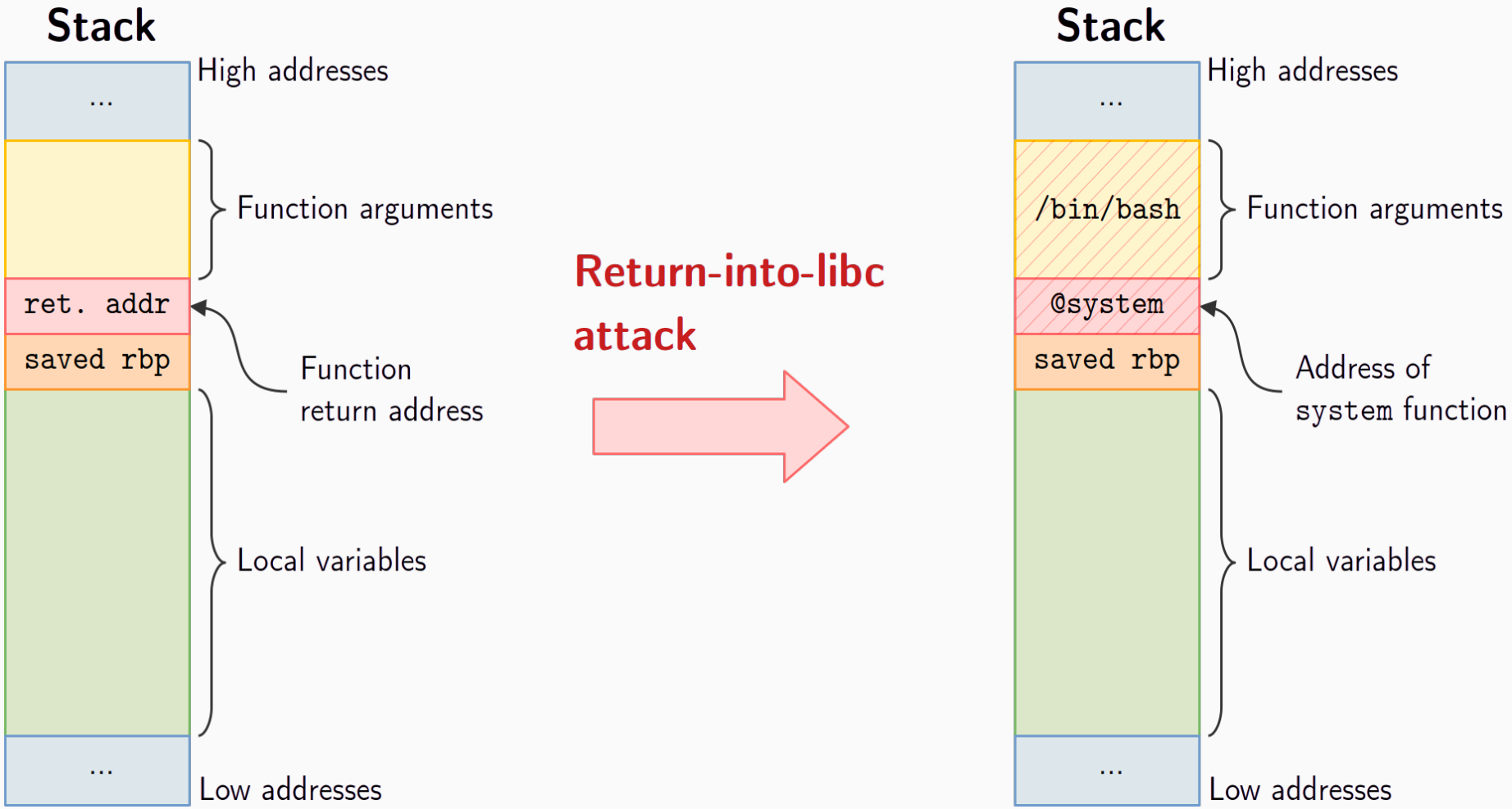
Buffer overflow attack: protection

- The compiler/linker/OS loader can **mark the program' stack as non-executable** (stack pages marked as non-executable)
- In this case, the buffer overflow attack described previously doesn't work since the memory area for the stack is not executable
- The attacker can no longer execute the shell code from the vulnerable program's stack

Return-into-libc attack

- This attack can be used to bypass the non-executable stack protection
- The return address is replaced by the **address of a function that is already present in the process executable memory**
  - Typically, on systems using shared libraries, the system library (e.g. libc on Linux) is always mapped into the process' address space
- Overwrites the stack with arguments suitable to the function
- Typical example:
  - Overwrite the return address with the address of the `system` function
  - Arguments are overwritten with the path to a shell, e.g. `/bin/bash`

Return-into-libc attack: example



Borrowed code chunks attack

- This attack uses **chunks of library functions**, instead of entire functions themselves, to exploit buffer overflows
- Looks for functions that contain instruction sequences that pop values from the stack into registers
- Careful selection of these code sequences allows an attacker to put suitable values into the proper registers to perform a function call
- The rest of the attack proceeds as a return-into-libc attack

Return-oriented programming (ROP) attack

- ROP extends the borrowed code chunks attack to provide Turing-complete functionality to the attacker, including loops and conditional branches
- Provides a fully functional "language" to make a compromised machine perform any operation desired
  - Superior to other buffer overflow attacks, both in expressive power and in resistance to defensive measures
- First published by Hovav Shacham in 2007[1]

[1] "The geometry of innocent flesh on the bone: return-into-libc without function calls"

Position Independent Executable (PIE)

- PIE are executable binaries made entirely from position-independent code
- Compile-time technique applied to the binary itself
  - **The program is compiled in a way allowing it to be loaded at any address without modification**
  - References to code and data use relative addresses, not fixed ones
  - Makes it possible for the loader to relocate the executable anywhere in memory

Address Space Layout Randomization (ASLR)

- Technique to prevent exploitation of memory corruption vulnerabilities
- **The OS loader randomly chooses, at runtime, the addresses of a process' key data areas**:
  - Executable base address (requires PIE)
  - Stack address
  - Heap address
  - Shared libraries addresses
  - Memory-mapped regions (mmap) addresses
- When applied to the kernel, this technique is called Kernel Address Space Layout Randomization (KASLR)

ASLR vs PIE

	ASLR	**PIE**
Function	Randomizes memory layout	Enables executables to be relocatable
Level	Runtime	Compile-time
Controls	OS loader	Compiler/linker
Interaction	Only fully exploitable with PIE binaries	PIE provides relocatability, no randomness

ASLR + PIE protection

- An operating system configured with ASLR and PIE applications offers effective protection against buffer overflow attacks (return-into-libc, borrowed code chunks, and ROP)
- Strongly advised to:
  - Enable ASLR at the system level
  - Compile programs with PIE enabled

Linux kernel configuration

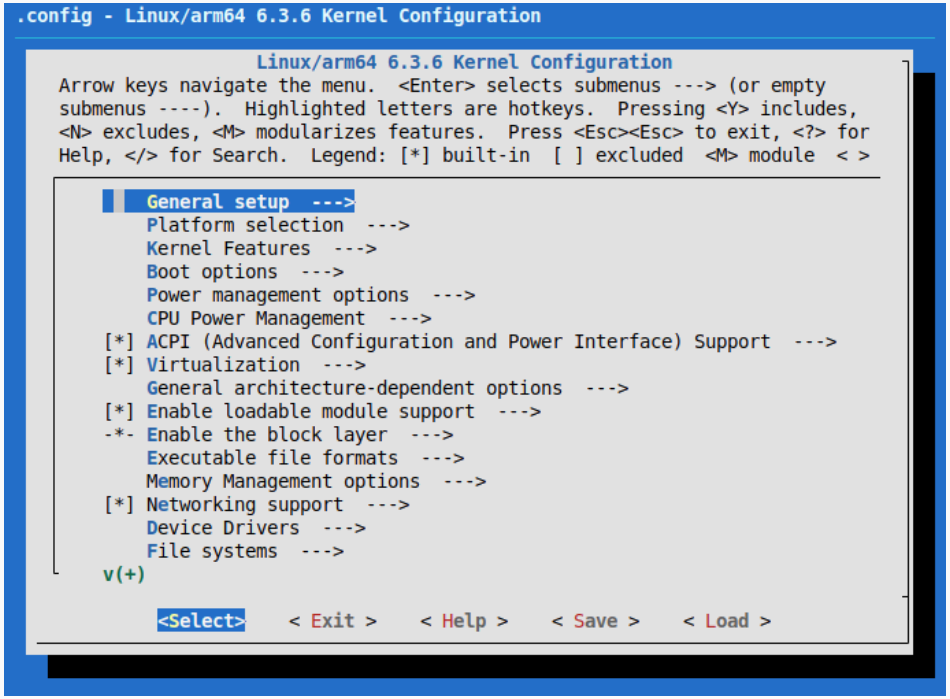
Linux kernel configuration & compilation from Buildroot

- To configure the Linux kernel (from Buildroot's root directory):

```
make linux-menuconfig
```

- To only build the kernel:

```
make linux-rebuild
```



- The configuration and build process is almost identical to U-Boot

Linux kernel sources

- In Buildroot, the Linux kernel sources are located in: `output/build/linux-xx/`

File	Description
`vmlinux`	Kernel in the ELF format
`.config`	Kernel configuration
`.config.old`	Previous kernel configuration
`Kconfig`	Root kernel configuration file (for `menuconfig`)
`Makefile`	Makefile allowing to configure and build the kernel

Linux kernel sources: main directories

Directory	Description
`arch`	Hardware dependent code
`block`	Generic functions for block devices
`crypto`	Crypto. algorithms
`Documen...`	Kernel documentation
`drivers`	Device drivers
`fs`	Filesystems
`include`	Include files
`init`	Initialization code (function `start_kernel`)

Directory	Description
`ipc`	Interprocess communications
`kernel`	Scheduler, concurrency, etc.
`lib`	Kernel libraries
`mm`	Memory management
`net`	Network protocols
`samples`	Code examples
`security`	Cryptographic keys, SELinux, etc.
`sound`	Sound drivers
`virt`	Virtualization layer (KVM)

Linux kernel entry point

- During boot, first kernel code to be executed is the startup entry point in:

```
arch/arm64/kernel/head.S    # hardware dependent code
```

  Early initialization of CPU, MMU, and retrieves the arguments passed by the bootloader (`bootargs`)

- Next, calls the `void start_kernel()` function implemented in:

```
init/main.c    # hardware independent code
```

  Initialization of CPUs (SMP), MMU, boot args parsing, interrupts, scheduler, timers, ramdisk, keys, security, etc.

Linux kernel hardening configuration

Kernel configuration embedded in the kernel

- The `IKCONFIG` option enables the complete Linux kernel `.config` file contents to be saved in the kernel
- `General setup ---> < > Kernel .config support`
- It is exposed to unprivileged users in `/proc/config.gz` and can easily be read, e.g.:

```
zcat /proc/config.gz
```

- Kernel configuration provides information that can be exploited by attackers

/dev/mem access security risks

- `/dev/mem` is a device file that provides raw access to the physical memory of the entire system
- Read/write to it allows direct access to any location in RAM, as well as memory-mapped I/O (device registers)
- Bypasses memory protection mechanisms provided by the kernel!
- Allowing unrestricted access to `/dev/mem` can lead to **huge security risks!**

Disabling /dev/mem entirely

- `/dev/mem` is configured through the `DEVMEM` option
- Deselecting `DEVMEM` disables `/dev/mem` entirely
-
```
Device Drivers --->
    Character devices --->
        [ ] /dev/mem virtual device support
```

Restricting access to /dev/mem

- The Linux kernel features two other options related to `/dev/mem`: `STRICT_DEVMEM` and `IO_STRICT_DEVMEM`
- If `STRICT_DEVMEM` is on and `IO_STRICT_DEVMEM` is off:
  - Only allows userspace access to PCI space and the BIOS code and data regions
- If both `STRICT_DEVMEM` and `IO_STRICT_DEVMEM` are on:
  - Only allows userspace access to idle io-memory ranges (`/proc/iomem`)
  - ⚠ Might break applications using `/dev/mem` (dosemu, legacy X, etc.)
- ```
  Kernel hacking --->
      [*] Filter access to /dev/mem
      [*]   Filter I/O access to /dev/mem
  ```
  ```

Kernel buffer overflow detection

- The Linux kernel can be compiled with stack overflow checks
  - Only protects against stack smashing within the kernel (not user processes)
- ```
  General architecture-dependent options --->
      [*] Stack Protector buffer overflow detection
      [*]   Strong Stack Protector
  ```
- If stack smashing is detected, the kernel will die with a kernel panic
- On x86, `STACKPROTECTOR` add canary checks to ~3% of kernel functions
  - Increases kernel code size by ~0.3%
- On x86, `STACKPROTECTOR_STRONG` add canary checks to ~20% of kernel functions
  - Increases kernel code size by ~2%
  ```

Address Space Layout Randomization (ASLR)

- ASLR is usually enabled by default
- To check its state:

```
cat /proc/sys/kernel/randomize_va_space
```

  - 0 ⟶ disabled
  - 1 ⟶ addresses of base and stack are randomized, and libraries are loaded at random addresses
  - 2 ⟶ in addition, addresses in the heap are randomized

- To change ASLR configuration:

```
sysctl -w kernel.randomize_va_space=2
```

Kernel Address Space Layout Randomization (KASLR)

- The `RANDOMIZE_BASE` option randomizes the virtual address at which the kernel image is loaded
- Security feature that deters exploit attempts relying on knowledge of the location of kernel internals
- The bootloader (U-Boot) must provide entropy, by passing a random value in `/chosen/kaslr-seed` at kernel boot
  - `/chosen/kaslr-seed` is a property in the device tree
- ```
  Kernel Features --->
      [*] Randomize the address of the kernel image
  ```
  ```

Linux kernel Random Number Generator (RNG)

- The kernel built-in RNG produces cryptographically secure pseudo-random data
- It works by collecting entropy from various hardware sources (hardware RNGs, interrupts, CPU-based jitterentropy, etc.)
- No single entropy source is relied on exclusively
- Entropy is extracted using the BLAKE2s cryptographic hash function and used to seed a set of ChaCha CRNGs[1] that provide the actual random data
- Entropy continues to be collected, and the CRNGs are periodically reseeded, while the kernel is running

[1] Cryptographic Random Number Generators

Accessing the Linux kernel RNG from userspace

- How to access the kernel RNG from userspace?
  - `getrandom` system call (see `man getrandom`)
  - `/dev/random` or `/dev/urandom`
- Since Linux 5.6 (March 2020), `/dev/random` and `/dev/urandom` are functionally identical and they never block[1]
- On kernels < 5.6:
  - Entropy source[2] decreases when reading random numbers
    - when it reaches 0, `/dev/random` blocks (but not `urandom`)

---

[1] See `man 4 random`

[2] See `/proc/sys/kernel/random/entropy_avail`

Hardware random number generator

- Often, platforms feature hardware random generators which provide a secure source of random numbers
  - Good random number generator important for cryptographic functions ⟶ security
- Options `HW_RANDOM` and `HW_RANDOM_BCM2835` enable hardware RNG support and specific driver for the Broadcom BCM2835/BCM63xx hardware
- ```
  Device Drivers ---> Character devices --->
      {M} Hardware Random Number Generator Core support  --->
      <M>   Broadcom BCM2835/BCM63xx Random Number Generator support
  ```
- Hardware RNG exposed through `/dev/hwrng` device file
- If `HW_RANDOM` selected as module ⟶ `rng_core` module name
  ```

Restrict unprivileged access to kernel system logs

- The `SECURITY_DMESG_RESTRICT` option enforces restrictions on unprivileged users reading the kernel system logs (syslog) via `dmesg`
  - Only root can access the kernel system logs
- ```
  Security options --->
      [*] Restrict unprivileged access to the kernel syslog
  ```
  ```

Heap memory zeroing

- The `INIT_ON_ALLOC_DEFAULT_ON` option zeroes heap memory when allocated, eliminating many memory flaws, especially heap content exposures
  - Performance impact varies (up to 7%), but most cases see <1% impact
- The `INIT_ON_FREE_DEFAULT_ON` option zeroes heap memory when freed, eliminating many memory flaws, especially heap content exposures
  - Anything freed is wiped immediately → makes live forensics or cold boot memory attacks unable to recover freed memory contents
  - Performance impact varies (up to 8%), but most cases see 3-5% impact
- ```
  Security options ---> Kernel hardening options  --->
      Memory initialization  --->
          [*] Enable heap memory zeroing on allocation by default
          [*] Enable heap memory zeroing on free by default
  ```
  ```

Check memory copies between kernel and userspace

- Device drivers often copy memory from userspace to kernel space (and conversely)
- The `HARDENED_USERCOPY` option checks for obviously wrong memory regions when copying memory to/from the kernel (via `copy_to_user` and `copy_from_user` functions)
  - Prevents entire classes of heap overflow exploits
- ```
  Security options --->
      [*] Harden memory copies between kernel and userspace
  ```
  ```

Check string and memory functions for buffer overflows

- The `FORTIFY_SOURCE` option detects overflows of buffers in common string and memory functions where the compiler can determine and validate the buffer sizes
-
```
Security options --->
    [*] Harden common str/mem functions against buffer
overflows
```

kernel-hardening-checker

- The kernel-hardening-checker is a tool for checking the security hardening options of the Linux kernel
- The security hardening recommendations are based on:
  - Kernel Self-Protection Project (KSPP) recommended settings
  - Direct feedback from the Linux kernel maintainers
  - Kernel options disabled by grsecurity to cut attack surface
  - CLIP OS kernel configuration
  - GrapheneOS recommendations
  - `SECURITY_LOCKDOWN_LSM` patchset
  - CIS Benchmark
- Supported architectures: `X86_64`, `IA-32`, `ARM64`, `ARM`, `RISC-V`

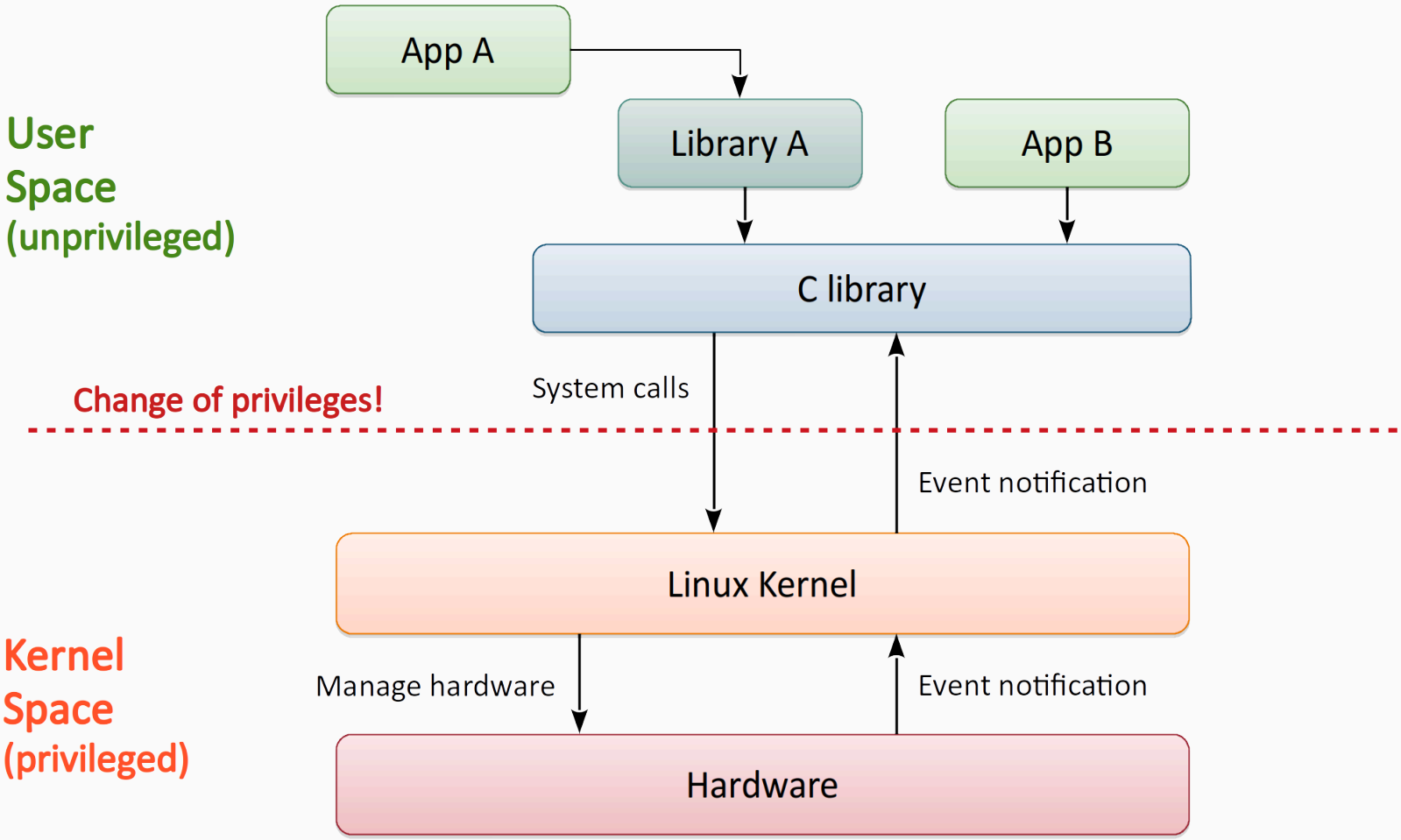
Miscellaneous Linux kernel configuration

Optimize for performance or size

- The kernel can either be optimized for:
  - Performance (config option `CC_OPTIMIZE_FOR_PERFORMANCE`)
  - Size (config option `CC_OPTIMIZE_FOR_SIZE`)
-
```
General setup --->
    Compiler optimization level (Optimize for xxx)
```
  - where `xxx` is either `performance (-O2)` or `size (-Os)`
- Kernel targeting embedded systems are usually optimized for size

Userspace protections

CPU execution privilege: kernel and userspace



Kernel vs userspace: why?

- Kernel space is critical and is not accessible by userspace
  - hardware enforces it through different CPU privilege levels
- **Applications run in userspace for security reasons**
- However:
  - `dmesg`, `/dev/mem`, `/proc`, and `/sys` allow data exchanges between user and kernel spaces

dmesg security risks

- `dmesg` reads and displays kernel messages (in-RAM ring buffer)
- Useful for system monitoring and debugging
- However, messages can contain **sensitive information**:
  - Kernel addresses (can weaken kernel ASLR)
  - Application stack traces/dumps (leak code paths, memory contents)
  - Driver logs (firmware versions, hardware serials, MACs, paths)
  - Kernel crash stack traces (snippets of in-kernel data)
- Allowing unrestricted access to `dmesg` can lead to **security risks!**

dmesg example

```
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[    0.000000] Linux version 6.3.6 (root@1cd494c45efe) (aarch64-buildroot-linux-gnu-gcc.br_real (Buildroot 2022.08.3) 11.3.0,
GNU ld (GNU Binutils) 2.37) #1 SMP PREEMPT Mon Oct 13 18:52:24 CEST 2025
[    0.000000] Machine model: FriendlyARM NanoPi NEO Plus2
[    0.000000] efi: UEFI not found.
[    0.000000] [Firmware Bug]: Kernel image misaligned at boot, please fix your bootloader!
[    0.000000] NUMA: No NUMA configuration found
[    0.000000] NUMA: Faking a node at [mem 0x0000000040000000-0x000000005fffffff]
[    0.000000] NUMA: NODE_DATA [mem 0x5fede9c0-0x5fee0fff]
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000040000000-0x000000005fffffff]
[    0.000000]   DMA32    empty
[    0.000000]   Normal   empty
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000040000000-0x000000005fffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000040000000-0x000000005fffffff]
[    0.000000] cma: Reserved 32 MiB at 0x000000005d600000
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv0.2 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: Trusted OS migration not required
[    0.000000] percpu: Embedded 22 pages/cpu s50088 r8192 d31832 u90112
[    0.000000] pcpu-alloc: s50088 r8192 d31832 u90112 alloc=22*4096
[    0.000000] pcpu-alloc: [0] 0 [0] 1 [0] 2 [0] 3
[    0.000000] Detected VIPT I-cache on CPU0
[    0.000000] CPU features: detected: ARM erratum 845719
...
```

/proc security risks (1/2)

- `/proc` is a virtual filesystem that exposes to userspace:
  - Process info
  - Kernel parameters
  - System hardware and configuration data
- Readable by unprivileged users by default, which can leak:
  - Command-line arguments of other processes
  - Environment variables (might include credentials or API keys)
  - Open file descriptors
  - Network connections
  - Memory mappings, revealing ASLR layout

/proc security risks (2/2)

- If writable to unprivileged users, they can:
  - Disable certain kernel security protections (randomize VA space, etc.)
  - Enable kernel debug interfaces
  - Change network stack behavior
- Potential container breakouts if mounted without isolation:
  - Container could access the host's process list
  - Container could kill or trace host processes
  - Container could modify host kernel parameters

/sys security risks (1/2)

- `/sys` is a virtual filesystem that exposes to userspace:
  - Live view and control interface to devices, drivers, firmwares, and buses
- Used by `systemd` and `udev` to:
  - Enumerate hardware devices
  - Bind/unbind kernel drivers
  - Change kernel settings
  - Load/unload drivers
  - Adjust power management, CPU governors, thermal zones, etc.

/sys security risks (2/2)

- Read-only access exposes sensitive system data:
  - Hardware IDs, serial numbers, and manufacturer info
  - Kernel versions and configuration
  - Memory, CPU, and NUMA layout (useful for side-channel attacks)
- If writable to unprivileged users, they can:
  - Change hardware states (fan speeds, CPU frequency, etc.)
  - Modify kernel parameters (disable AppArmor, SELinux, etc.)
  - Load custom firmwares (firmware attacks)
- Container escape risk if mounted directly from the host:
  - Container can see the host hardware
  - Possible tamper with host's cgroups or security modules