System Security

Jürgen Schönwälder

Constructor University

May 18, 2024

https://cnds.constructor.university/courses/sysec-2024/

Content and Educational Aims

- Attacks on Computer Systems
- Isolation Mechanisms
- Trusted Execution Environments
- Malware Analysis
- Confidential Computing
- Authentication and Authorization
- Attack Detection and Response

Intended Learning Outcomes

By the end of this module, students will be able to

- describe microarchitectural attacks on computer components and suitable counter measures
- illustrate trusted execution environments and how they can be used to bootstrap security
- compare the isolation achieved by hypervisors and operating system mechanisms
- assess application layer isolation and sandboxing mechanisms
- explain how systems can identify misbehaving code and protect themselves against malware
- outline how protected data storage can be implemented
- recommend authentication methods suitable for different kinds of applications
- compose authorization mechanisms to define effective security policies

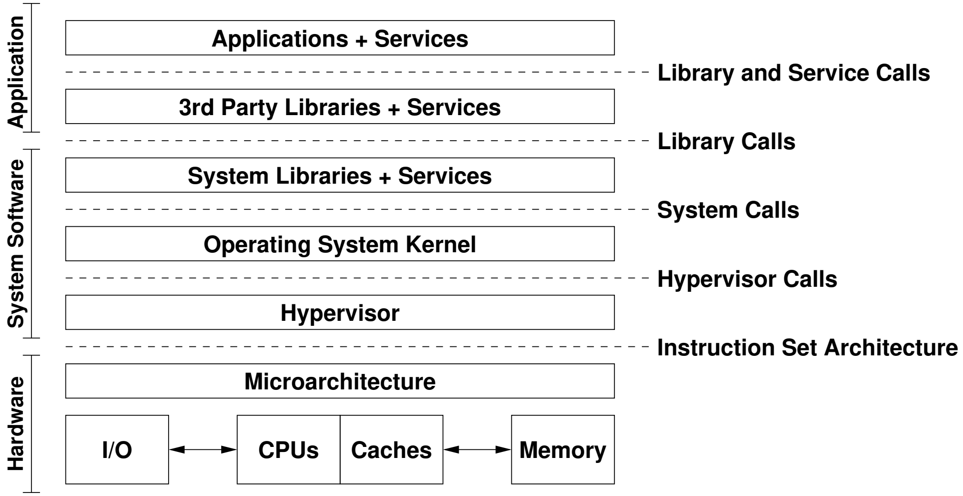
Study Material and Forums

- There is no required textbook.
- The slides and notes are available on the course web page.
  `https://cnds.constructor.university/courses/sysec-2024/`
- We will be using Moodle and it hosts a forum for this course.
- General questions should be asked on the Moodle forum.
  - Faster responses since many people can answer
  - Better responses since people can collaborate on the answer
- For individual questions, see me at my office (or talk to me after class).

Part: Introduction

Fundamental Concepts

Layered Model of a Computing System



Security Requirements

Definition (security requirements)

The *security requirements* are all requirements of an information system that ensure confidentiality, integrity, and availability of information being processed, stored, or transmitted.

- Security requirements may originate from laws, directives, policies, standards, regulations, procedures, or mission/business cases.

Threat Models

Definition (threat model)

A *threat model* defines what an adversary can be expected to do.

- A system can only be secure against a given threat model.
- Typical aspects to consider:
  - Has the adversary physical access to a system?
  - Has the adversary an account on a system?
  - Has the adversary the capability to install software on a system?
- Do not confuse this term with threat modeling, which is the process of modeling potential threats of a system.

Security Mechanisms

Definition (security mechanism)

A *security mechanism* is a method, tool, or procedure for enforcing a security policy.

- Given a security policy (the security requirements) and a set of concrete security mechanisms, it should be possible to reason whether the system can be secured given a certain threat model.
- Fewer sufficiently flexible security mechanisms are often preferred over many complex security mechanisms that may interact in ways that are difficult to analyze.

Ethical Considerations

Computer Ethics

Definition (computer ethics)

The term *computer ethics* refers to the ethical principles and guidelines governing the behaviour and decisions of individuals and organizations in the field of computing and information technology.

- Examples of ethical principles:
  - Respecting user privacy
  - Avoiding unauthorized access to computer systems
  - Respecting intellectual property rights
  - Providing accurate information
  - Considering societal implications of technology
- There can be ethical dilemmas, e.g., striking a balance between protecting systems and respecting individual privacy.

Ethical Hacking and Penetration Testing

Definition (penetration testing)

Ethical hacking or *penetration testing* requires that the ethical hacker or the pentester has obtained the permission to perform the security tests from an authorized party.

- Penetration testing is an effective mechanism to identify technical weaknesses and to remind people of the importance of executing proper security procedures.
- Penetration testing often includes tests used to determine whether employees implement security best practices.
- Exploiting the nature of humans is often a cheap and effective way to circumvent technical security mechanisms.

Responsible Disclosure

Definition (responsible disclosure)

A *responsible disclosure* is a vulnerability disclosure where a vulnerability is disclosed to the public only after the responsible parties have been allowed sufficient time to remedy the vulnerability.

- The goal is to report vulnerabilities is such a way that they can be fixed before they may be exploited by malicious actors.
- The time given to the responsible parties should be aligned with the complexity and potential severity of a detected vulnerability.

Part: Attacks on Computer Systems

Attacks on Hardware

Counterfeit Hardware Attacks

Definition (counterfeit hardware attack)

A *counterfeit hardware attack* involved the distribution of malicious or tampered hardware components that compromise the security of devices.

Examples:

- Hardware trojans describe hardware components that carry a hidden function such as providing backdoors
- Supply chain attacks aim at compromising hardware components during manufacturing or distribution processes
- Dropping USB sticks in public spaces that emulate other USB devices such as keyboards to gain access by injecting keystrokes

Hardware Fault Injection Attacks

Definition (hardware fault injection attack)

A *hardware fault injection attack* alters the operation of a device by physically tampering with hardware components.

Examples:

- Injecting voltage glitches to change the execution of CPU instructions
- Injecting clock glitches to change the execution of CPU instructions
- Injecting faults through brief electromagnetic pulses in a specific clock cycle

Hardware Side-Channel Attacks

Definition (hardware side-channel attack)

A *hardware side-channel attack* exploits unintended information leakage from a system's physical implementation.

Examples:

- Monitoring power consumption during cryptographic operations to extract properties of cryptographic keys (power analysis side-channel attacks)
- Exploiting timing variations to obtain information about data being processed by a system (timing side-channel attacks)
- Intercepting electromagnetic radiation of wires to obtain potentially sensitive information (electromagnetic side-channel attacks)
- Capture sound emissions using microphones to reveal information about data received or processed by a system (acoustic side-channel attacks)

Microarchitectural Attacks

Definition (microarchitectural-attack)

A *microarchitectural-attack* targets the implementation of a system's microarchitecture.

Examples:

- Speculative execution attacks to read arbitrary memory via a timing side-channel attack (Spectre, Meltdown)
- Exploiting vulnerabilities of dynamic random-access memory (DRAM) potentially leading to privilege escalation (Rowhammer)

Spectre: Vulnerability of the Year 2018

```
#define PAGESIZE 4096
unsigned char array1[16]                 /* base array */
unsigned int array1_size = 16;           /* size of the base array */
int x;                                   /* the out of bounds index */
unsigned char array2[256 * PAGESIZE];    /* instrument for timing channel */
unsigned char y;                         /* does not really matter much */

// ...

if (x < array1_size) {
    y = array2[array1[x] * PAGESIZE];
}
```

- Is the code shown above a vulnerability?

Spectre: Main Memory and CPU Memory Caches

- Memory in modern computing systems is layered
- Main memory is large but relatively slow compared to the speed of the CPUs
- CPUs have several internal layers of memory caches, each layer faster but smaller
- CPU memory caches are not accessible from outside of the CPU
- When a CPU instruction needs data that is in the main memory but not in the caches, then the CPU has to wait quite a while...

Spectre: Speculative Execution

- In a situation where a CPU would have to wait for slow memory, simply guess a value and continue execution speculatively; be prepared to rollback the speculative computation if the guess later turns out to be wrong; if the guess was correct, commit the speculative computation and move on.
- Speculative execution is in particular interesting for branch instructions that depend on memory cell content that is not found in the CPU memory caches
- Some CPUs collect statistics about past branching behavior in order to do an informed guess. This means we can train the CPUs to make a certain guess.
- Cache state is not restored during the rollback of a speculative execution.

Spectre: Reading Arbitrary Memory

- Algorithm:
  1. create a small array `array1`
  2. choose an index `x` such that `array1[x]` is out of bounds
  3. trick the CPU into speculative execution (make it read `array1_size` from slow memory and guess wrongly)
  4. create another uncached memory array called `array2` and read `array2[array1[x]]` to load this cell into the cache
  5. read the entire `array2` and observe the timing; it will reveal what the value of `array1[x]` was
- This could be done with JavaScript running in your web browser; the first easy "fix" was to make the JavaScript time API less precise, thereby killing the timing side channel. (Obviously, this is a hack and not a fix.)

Attacks on Software

Control Flow Attacks

Definition (control flow attack)

A *control flow attack* diverts the intended control flow of a program to direct the execution to malicious code.

Examples:

- Stack smashing attacks redirecting control flow to execute malicious code (often called shellcode) injected on the stack
- Return-oriented programming attacks construct malicious code by sequencing existing code fragments (gadgets) via function returns
- Attacks on virtual function tables change the control flow of an application written in certain programming languages supporting late binding and the overriding of functions

Call Stacks and Stack Frames

Definition (call stack and stack frames)

A *call stack* holds a stack frame for every active function call. A *stack frame* provides memory space to store

1. the return address to load into the program counter when the function returns,
2. local variables that exist during an active function call,
3. arguments that are passed into the function or results returned by the function.

To support function calls, a CPU needs to provide

- a register pointing to the top of the call stack (stack pointer),
- a register pointing to the start of the current stack frame (frame pointer),
- a mechanism to call a function (allocating a new stack frame on the call stack),
- an mechanism to return from a function (deleting a stack frame from the stack).

Function call prologue and epilogue

Definition (function call prologue)

A *function call prologue* is a sequence of machine instructions at the beginning of a function that prepare the stack and the registers for use in the function.

Definition (function call epilogue)

A *function call epilogue* is a sequence of machine instructions at the end of a function that restores the stack and CPU registers to the state they were in before the function was called.

Definition (stack and frame pointer)

The *stack pointer* is a register pointing to the beginning of the function call stack. The *frame pointer* is a register pointing to the beginning of the current stack frame.

Stacks (Intel x86_64)

```
                   : .... .... .... .... :
                   |---------------------|
0x00007fffffffe318 | .... .... .... .... | ] return address
0x00007fffffffe310 | .... .... .... .... | ] saved rbp
                   |---------------------| <- rbp (frame pointer)
0x00007fffffffe308 | .... .... .... .... | \
0x00007fffffffe300 | .... .... .... .... |  |
0x00007fffffffe2f8 | .... .... .... .... |  |
0x00007fffffffe2f0 | .... .... .... .... |  | char name[64]
0x00007fffffffe2e8 | .... .... .... .... |  |
0x00007fffffffe2e0 | .... .... .... .... |  |
0x00007fffffffe2d8 | .... .... .... .... |  |
0x00007fffffffe2d0 | .... .... .... .... | /
                   '---------------------' <- rsp (stack pointer)
```

Shellcode (Intel x86_64)

```
                   : .... .... .... .... :
                   |---------------------|
0x00007fffffffe318 | d0e2 ffff ff7f 0000 | ] return address -.
0x00007fffffffe310 | 0000 0000 0000 0000 | ] saved rbp        |
                   |---------------------| <- rbp              |
0x00007fffffffe308 | 0000 0000 0000 0000 | \                   |
0x00007fffffffe300 | 0000 0000 0000 0000 |  |                  |
0x00007fffffffe2f8 | 0000 0000 0000 0000 |  |                  |
0x00007fffffffe2f0 | 0000 0000 0000 0000 |  | char name[64]    |
0x00007fffffffe2e8 | 6e2f 7368 00ef bead |  |                  |
0x00007fffffffe2e0 | e8ed ffff ff2f 6269 |  |                  |
0x00007fffffffe2d8 | 4831 f648 31d2 0f05 |  |                  |
0x00007fffffffe2d0 | eb0e 5f48 31c0 b03b | /                   |
                   '---------------------' <- rsp <----------'
```

Shellcode (Intel x86_64) Improvements

- We have to know the exact start address of the `name` buffer on the stack. This can be relaxed by prefixing the shellcode with a sequence of `nop` instructions that act as a landing area.
- We have to know where precisely the return address is located on the stack. This can be relaxed by filling a whole range of the stack space with our jump address.
- Systems with memory management units often randomize the memory layout, i.e., the stack is placed randomly in the logical address space whenever a program is started.
- Systems with memory management units often disable the execute bit for the stack pages and hence our attack essentially leads to a memory access failure.
- Compilers may insert bit pattern (stack canary) that can be checked to detect memory overwrites.

Return Oriented Programming (Intel x86_64)

```
                   : ....  ....  ....  .... :
0x00007fffffffe328 | c0c9 e3f7 ff7f 0000 | ] return to system =>
0x00007fffffffe320 | d0e2 ffff ff7f 0000 | ] char *command ------.
                   +---------------------+                      |
0x00007fffffffe318 | 5fba e1f7 ff7f 0000 | ] return to gadget => |
0x00007fffffffe310 | 0000 0000 0000 0000 | ] saved rbp           |
                   |---------------------| <- rbp               |
0x00007fffffffe308 | 0000 0000 0000 0000 | \                    |
0x00007fffffffe300 | 0000 0000 0000 0000 |  |                   |
0x00007fffffffe2f8 | 0000 0000 0000 0000 |  |                   |
0x00007fffffffe2f0 | 0000 0000 0000 0000 |  | char name[64]     |
0x00007fffffffe2e8 | 0000 0000 0000 0000 |  |                   |
0x00007fffffffe2e0 | 0000 0000 0000 0000 |  |                   |
0x00007fffffffe2d8 | 0000 0000 0000 0000 |  |                   |
0x00007fffffffe2d0 | 2f62 696e 2f73 6800 | /                    |
                   '---------------------' <- rsp <--------------'
```

C Format Strings

`%s`	interpret the next argument as a pointer to a null-terminated string
`%x`	interpret the next argument as an integer and print the value in hexadecimal
`%#lx`	interpret the next argument as a long integer and print the value in hexadecimal prefixed with `0x`
`%#018lx`	interpret the next argument as a long integer and print the value in hexadecimal prefixed with `0x` and 0-padded filling 18 characters
`%n`	interpret the next argument as a pointer to an integer and write the number of characters printed so far to the integer pointed to
`%4$s`	interpret the fourth argument as a pointer to a null-terminated string

- The classic C format string can do many fancy things...
- We focus here on the subset most relevant / convenient for exploits.

Format String Attacks (Intel x86_64)

```
                   : .... .... .... .... :
                   | 0000 0000 0000 0001 | long i
                   | cccc cccc cccc cccc | long c
                   | 0000 0002 5555 5060 | ??
                   | 0000 7fff ffff e328 | ??
                   |---------------------|
0x00007fffffffe218 | .... .... .... .... | ] return address
0x00007fffffffe210 | 0000 7fff ffff e240 | ] saved rbp
                   |---------------------|
0x00007fffffffe208 | .... .... .... .... | ] char *s
0x00007fffffffe200 | aaaa aaaa aaaa aaaa | ] long a
0x00007fffffffe1f8 | bbbb bbbb bbbb bbbb | ] long b
                   |---------------------|
                   : .... .... .... .... :
```

Heap Overflows and Use After Free

- Memory regions dynamically allocated on the heap can be overrun or underrun.
- Dangling pointers can lead to use after free situations.
- Heap smashing problems are a bit more challenging to exploit.
- General idea:
  - Target function pointers stored on the heap.
  - Overwrite function pointers to change the control flow.
  - Function pointers are easily found in virtual function tables.

Code Injection Attacks

Definition (code injection attack)

A *code injection attack* is an attack on a system where input is passed to a program (the generator), which generates executable code to be executed by another program (the executor). A code injecting attack exploits that input can inject instructions into the generated code.

```
         .-----------.                    .-----------.
         |           |  generated code    |           |
-------> | generator | -----------------> | executor  |
 input   |           |      (output)      |           |
         '___________'                    '___________'
```

SQL Injection Attacks

Definition (sql injection attack)

An *sql injection attack* is a code injection attack where an attacker sends input to an application with the goal to modify SQL queries made by the application in order to gain access to additional information or to modify database content.

- SQL injection attacks are often made possible by careless construction of queries. Here is an example in C:

```
snprintf(buffer, size,
         "SELECT user, balance FROM account WHERE user='%s'", name);
```

- Prepared statements provide a safe way to construct SQL queries, ensuring that parameters remains data and do not accidentally become code.

Command Injection Attacks

Definition (command injection attack)

A *command injection attack* is a code injection attack where an attacker sends input to an application with the goal to modify commands passed to a command interpreter in order to execute commands injected by the attacker.

- Command injection attacks are often the result of careless construction of system level commands. Here is an example in C:

```
char cmd[256] = "/usr/bin/cat ";
strncat(cmd, argv[i], sizeof(cmd) - strlen(cmd) - 1);
system(cmd);
```

- If `argv[i]` contains the string `/dev/null;reboot`, the command `/usr/bin/cat /dev/null;reboot` will be executed.

Cross-Site-Scripting Attacks

Definition (cross-site scripting attack)

A *cross-site scripting attack* is a code injection attack where an attacker injects code (scripts) into web pages such that the injected code (scripts) are delivered for execution to browsers run by visitors of the web page.

- A simple cross-site scripting attack would be to submit some JavaScript to a web form, e.g.:

  ```
  <script type="text/javascript">alert("XSS");</script>
  ```

- If the browser does not check the content, it may deliver the script to other users.
- The script running in the browser of other users can then do malicious things such as collecting information or displaying phishing dialogues.

Cross-Site-Request-Forgery Attacks

Definition (cross-site request forgery attack)

A *cross-site request forgery attack* is an attempt to invoke actions on a web application where malicous commands are injected into the context of an existing web session.

- The attack requires that a user has a valid session with a web application.
- A URL injected into the user's web browser leads a request within the existing session to the web application.

First and Second Order Code Injection Attacks

Definition (first order code injection attacks)

First order code injection attacks are caused by inputs that directly cause modified code to be generated and executed.

Definition (second order code injection attacks)

Second order code injection attacks are caused by data that is stored in the system and causes system components to execute modified code when the data is processed.

- The injection of attack data and the execution of the attack are often decoupled in second order attacks, making it harder to track down the origin of the attack data.

Time-of-Check-to-Time-of-Use Attacks

Definition (time-of-check-to-time-of-use attack)

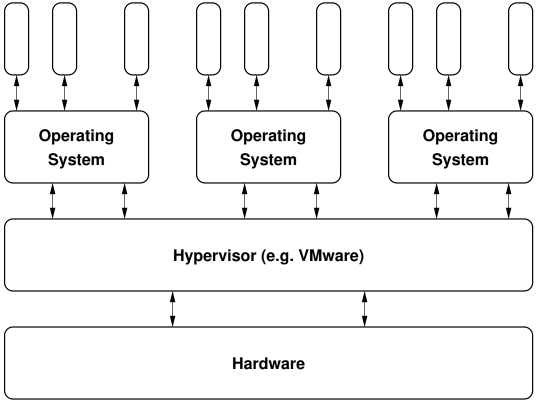
A *Time-of-Check-to-Time-of-Use attack* exploits a race condition between the check of a condition and the use of the result.

Examples:

- Operations on file systems are problematic if the file system can change between the check of file properties (or permissions) and the subsequent action
- Operations on databases when there is a delay between the authorization of a transaction and the execution of the transaction

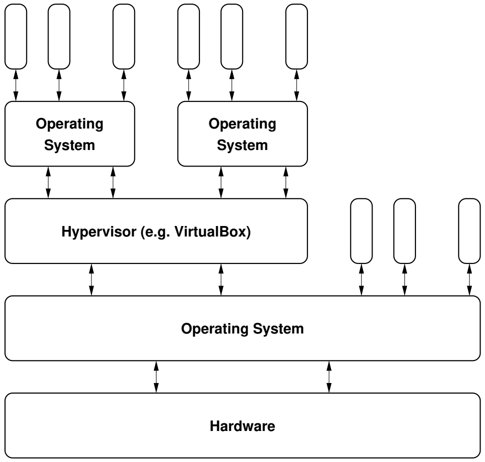
Attacks on Hypervisors

Bare Metal Virtualization (Type I)



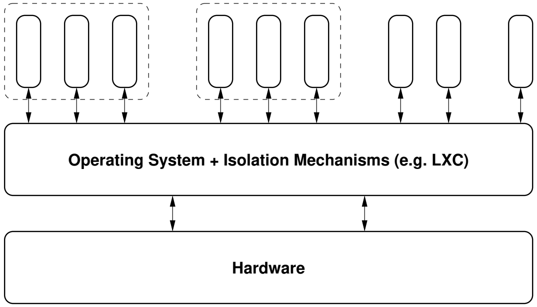
- Running on hardware
- Multiple operating systems
- Separation and isolation
- Targeting server systems
- Server consolidation
- VMware, KVM, ...

Hosted Virtualization (Type II)



- Running on operating system
- Multiple operating systems
- Separation and isolation
- Targeting desktop systems
- VirtualBox, Parallels, ...

OS-Level Virtualization (Container)



- Separation of processes
- Small performance overhead
- Single operating system ABI
- Linux Container (LXC), . . .

Hyperjacking Attacks

Definition (hyperjacking attack)

A *hyperjacking attack* aims at replacing a hypervisor with a malicious hypervisor.

- Hyperjacking can be achieved by
  1. injecting a malicious hypervisor below the legitimate hypervisor
  2. replacing the legitimate hypervisor
  3. injecting a malicious hypervisor above the legitimate hypervisor
- The presence of a malicious hypervisor is difficult to detect from the operating systems

Virtual Machine Escape Attacks

Definition (virtual machine escape attack)

A *virtual machine escape attack* is a program breaking out of a virtual machine in order to interact with the underlying hypervisor or host operating system.

- Virtual machine escape attacks break the isolation provided by the hypervisor
- Vulnerabilities of this kind have been found in many popular hypervisors (e.g., Xen, KVM, VMware, Hyper-V, VirtualBox)

Virtual Machine Rootkit Attacks

Definition (virtual machine rootkit attack)

A *virtual machine rootkit attack* is an attempt to install a rootkit running underneath an operating system.

- A virtual machine rootkit attack can be seen as a special case of a hyperjacking attack.
- A virtual machine rootkit may launch and run malware in separate invisible virtual machines that are hard to detect.

Attacks on Operating System Kernels

Loadable Kernel Module Attacks

Definition (loadable kernel module attack)

A *loadable kernel module attack* uses loadable kernel modules to injects malicious code into an operating system kernel.

- Operating systems supporting many different hardware configurations often load drivers dynamically into the kernel
- Loadable kernel modules can also extend the functionality of the kernel by implementing additional file systems, network protocols etc.
- Malicious kernel modules have almost unlimited access to programs on monolithic kernels

Kernel Rootkit Attacks

Definition (kernel rootkit attack)

A *kernel rootkit attack* is an attempt to install a rootkit within an operating system kernel.

- A kernel rootkit can be installed as a loadable kernel module.
- Privilege escalation attacks often precede kernel rootkit attacks in order to obtain the necessary permissions.
- Persistent kernel rootkits surviving reboots
- Volatile kernel rootkits disappearing after a reboot
- Kernel rootkits are difficult to detect and hence a complete re-installation is usually necessary if there are indicators for the existence of a kernel rootkit.

Kernel Privilege Escalation Attacks

Definition (kernel privilege escalation attack)

A *kernel privilege escalation attack* is an attempt to gain access to resources that are normally protected from an application or user.

- Operating system kernels usually execute at higher privilege levels and protect resources against access from less privileged processes.
- Kernel privilege escalation attacks cause less privileged processes to gain higher privileges, on Unix systems typically root privileges.
- Some operating systems put application processes into restricted execution environments called jails and kernel privilege escalation attacks may be used to escape from the jail (jailbreaking).

Kernel Resource Exhaustion Attacks

Definition (kernel resource exhaustion attack)

A *kernel resource exhaustion attack* is an attempt to consume resources managed by the operating system kernel with the goal to make the system unusable.

- Such attacks are also called denial of service attacks.
- Example: A program recursively creating processes as fast as possible

Attacks on System Software

Dynamic Linking Attacks

Definition (dynamic linking attack)

A *dynamic linking attack* tries to manipulate the dynamic linker to link malicious libraries to a program, usually at program startup time.

- On Linux systems, the `LD_PRELOAD` environment variable can specify libraries that are loaded before any of the other system libraries are loaded.
- On MacOS systems, the `DYLD_INSERT_LIBRARIES` environment variable can specify libraries that are loaded before any of the other system libraries are loaded.

Package Management Attacks

Definition (package management attack)

A *package management attack* is an attempt to modify a software package manage to accept malicious software packages.

Examples:

- Configure the package manager to accept packages from malicious sources.
- Configure the package manager to use malicious keys to validate packages.
- Create malicious packages signed with value keys.
- Create malicious packages exploiting bugs of a package manager.
- Instruct the user to manipulate the package manager's configuration.

Development Tool and Software Supply Chain Attacks

Definition (development tool attack)

A *development tool attack* is an attack on software development tools in order to create malicious executable software or malicious configurations of executable software.

Definition (software supply chain attack)

A *software supply chain attack* is a malicious attempt to gain access to a target by exploiting a weakness in the systems or processes of a third-party vendor or partner.

System Logging and Auditing Attacks

Definition (system logging and auditing attack)

A *system logging and auditing attack* is an attempt to (1) obtain additional information about a target or (2) manipulate system logging and auditing functions in order eliminate or modify information about a security incident.

Ransomware Attacks

Definition (ransomware attack)

A *ransomware attack* is a cryptovirology attack permanently blocking access to the victim's data unless a ransom is paid.

- Ransomware attacks usually encrypt files using a random key.
- The victim can buy the corresponding decryption key after paying a ransom.
- Advanced ransomware also targets backups to increase the pressure to pay ransom.
- More advanced ransomware attacks steal data before encrypting it with the thread to publish the data if no ransom is paid.
- Some advanced ransomware attacks also try to encrypt backups in order to make it difficult to rollback to a time before the ransomware attack started.

Name Resolution and Routing Attacks

Definition (name resolution attack)

A *name resolution attack* attempts to modify name resolution services to attack users of name resolution services.

Definition (routing attack)

A *routing attack* attempts to modify the flow of information in order to obtain access to information or to install a man-in-the-middle.

Attacks on Application Software

Malicious Macro Attacks

Definition

A *malicious macro attack* (also called a macro virus) uses a macro embedded in some innocent document to install malware on systems opening the document and executing the macro without supervising its execution.

Examples

- Malicious macros distributed with office documents have been used for years to break into systems.
- More recent versions of office software disables the execution of macros by default.

Phishing Attacks

Definition

A *phishing attack* attempts to steal sensitive information by masquerading a malicious resource as a reputable resource.

Examples:

- Email phishing attacks tricking individuals into giving away sensitive information
- Spear phishing attacks use contextual information to hide the phishing attack
- Whaling attacks use spear phishing techniques to target senior executives
- Voice phishing attacks make automated phone calls to large numbers of people
- Calendar phishing attacks send fake calendar invitations with phishing links
- QR code phishing attacks use QR codes to direct users to phishing sites

Social Engineering Attacks

Definition (social engineering attack)

A *social engineering attack* is the psychological manipulation of people into performing actions or divulging confidential information.

Examples:

- *A*n attacker sends a document that appears to be legitimate in order to attract the victim to a fraudulent web page requesting access codes (phishing).
- An attacker pretends to be another person with the goal of gaining access physically to a system or building (impersonation).
- An attacker drops devices that contain malware and look like USB sticks in spaces visited by a victim (USB drop).

Part: Security by Design

Security in the Software Lifecycle

Security by Design

Definition (Security by Design)

Secure by design, in software engineering, means that software products and capabilities have been designed to be foundationally secure.

- The goal of security by design is to ensure that security is an inherent part of a product, rather than being added on as an afterthought.

Software Life Cycle Model

1. Beginning of Life
   1.1 Idea
   1.2 Concept
   1.3 Development
   1.4 Prototype
   1.5 Launch
   1.6 Manufacture
2. Middle of Life
   2.1 Distribution
   2.2 Use
   2.3 Service
3. End of Life
   3.1 Recycle

- Security by design stresses the importance to consider security aspects in all phases of a software life cycle.
- Retrofitting security is complicated and costly and often leads to solutions that are complex and thus hard to fully understand.

Ten Security by Design Principles

Principle 1: Least Privilege

Definition (least privilege)

Limit privileges to the minimum necessary for a given context.

Motivation:

- Broad privileges allow malicious or accidental access to protected resources

Example:

- Execute server processes with exactly the privileges they require and not more:
    - Execute the processes using a restricted account
    - Limit access to the filesystem
    - Limit access to network resources
    - Limit the system calls that can be used

Principle 2: Separate Responsibilities

Definition (separate responsibilities)

Separate and compartmentalise responsibilities and privileges.

Motivation:

- Limit the impact of successful attacks and achieve control and accountability

Example:

- A software module responsible for payments should not also be responsible for placing orders

Principle 3: Trust Cautiously

Definition (trust cautiously)

Assume unknown entities are untrusted and have a clear process to establish trust.

Motivation:

- Security problems are often caused by inserting malicious intermediaries in communication paths.

Examples:

- Do not accept network connections from untrusted endpoints.
- Verify the identity of unknown people (do not get fooled by how they look)
- Do not plug USB devices of unknown origin into a computer.
- Do not execute code originating from an unknown source.

Principle 4: Simplest Solution Possible

Definition (simplest solution possible)

Actively design for simplicity, avoiding complex failure modes, implicit behaviour, unnecessary features, ...

Motivation:

- Complex systems are hard to analyze and they may have complex failure modes or implicit behaviour.

Examples:

- A logging system should not have a need to dynamically load code.
- A configuration language does not need to be Turing complete.
- A document format does not need to include executable code.

Principle 5: Audit Sensitive Events

Definition (audit sensitive events)

Record all security significant events in a tamper-resistant store.

Motivation:

- Auditing logs are useful for monitoring the operation of a system and reconstructing the past.
- It may be necessary to be able to proof legally that an auditing is complete and authentic.

Examples:

- Logging auditing information on write-only storage systems.
- Duplicating auditing information to increase availability.

Principle 6: Secure Defaults and Fail Securely

Definition (secure defaults and fail securely)

Force changes to security sensitive parameters (never use default values) and think through failures to make them secure but recoverable.

Motivation:

- Devices with default credentials (passwords) still are a problem.
- Security mechanisms may be downgrading through failures.
- Failures during privileged operation may leave systems in a vulnerable state.

Examples:

- Home routers shipped with a default administrative password.
- Temporary files left behind after a failure leaking information.

Principle 7: Never Rely on Obscurity

Definition (never rely on obscurity)

Assume attacker with perfect knowledge since this forces secure system design.

Motivation:

- Sooner or later someone will accidentally or on purpose find hidding things.

Examples:

- Using simple substitution rules to create passwords from ordinary words
- Using port knocking sequences to open administrative remote access to systems.
- Kerckhoff's principle states that security of a cryptographic system should not rely on the secrecy of the algorithm.

Principle 8: Defense in Depth

Definition (defense in depth)

Do not rely on single point of security, secure every level, stop failures at one level from propagating.

Motivation:

- Systems do get attacked and humans make mistakes, hence it is necessary to plan for such situations and to minimize the impact.

Examples:

- Multiple levels of access control (with different granularity):
  - Access control within a database backend
  - Access control at the file system level
  - Access control at the virtual machine level

Principle 9: Never Invent Security Technology

Definition (never invent security technology)

Do not create your own security technology, always use proven technology and components.

Motivation:

- Creating good security technology is difficult and requires the cooperation of many people to eliminate design flaws.

Examples:

- Wireless security technology has a long history of failed attempts
- Implementing crypto algorithms is hard (e.g., prevention of side channels)

Principle 10: Secure the Weakest Link

Definition (secure the weakest link)

Find the weakest link in the security chain, strengthen it, and then repeat.

Motivation:

- Perfect security does not exist, security a continuous process.
- By constantly improving the weakest link, the security of a system improves efficiently.

Examples:

- Encrypted communication but cleartexts stored in a database.
- Security access from the outside while leaving the doors wide open inside.
- Providing access to accounts without a plan how to terminate access.

Part: Control Flow Integrity

Control Flow Integrity

Control Flow

Definition (control flow)

The *control flow* of an imperative program is the order in which instructions or function calls are executed.

- The control flow of a program can be visualized in a control flow graph.
- At the machine instruction level, we have (i) sequences of instructions (nodes) and (ii) jumps between them (edges).
- Some jumps are conditional (so called branching points).
- Some jumps are function calls, i.e., the program eventually returns and continues.
- The control flow graph of a program is usually known at compilation time.

Control Flow Integrity

Definition (control flow integrity)

Techniques ensuring *control flow integrity* prevent attacks diverting a program's execution from the programs intended control flow graph.

- Defense against attacks overwriting return addresses
- Defense against attacks overwriting function pointers
- No defense against attacks that do not change a program's control flow
- Jumps are edges between sequences of machine code that do not return.
- Calls are edges between functions where the callee normally returns to the callsite.

Control Flow Integrity Details

Definition (forward edge cfi integrity)

Forward edge control flow integrity verifies jumps and function calls.

Definition (backward edge cfi integrity)

Backward edge CFI verifies the returns of function calls.

Definition (direct and indirect jumps)

A control transfer where the destination is directly known is a *direct jump*. Control transfers that follow addresses (pointers) stored in memory are called *indirect jumps*.

Control Flow Guad

Control Flow Guard

- Control flow guard is a forward edge cfi mechanism.
- Check a bitmask whether the destination is a valid jump/call target.
- The bitmask of valid jump/call targets is created when a program is compiled.
- The granularity of the bitmask is a concern (every address is pretty costly).
- The bitmask must be kept protected.

Pointer Authentication Codes

Pointer Authentication Codes

- Not all bits of an address are used and these bits can carry a pointer signature.
- A pointer (address) is signed with a key and the resulting authentication code is stored in “unused” bits.
- Authentication codes are verified at runtime.
- Attacker would need the secrect key to create valid pointers (so store the key in a protected register)
- Guessing authentication codes is possible, feasibility depends on the number of bits available

Shadow Stacks

Shadow Stack

- Shadow stacks are a backward edge cfi mechanism.
- Copy return addresses into a protected shadow stack
- Validate return address against the shadow stack on function return
- Shadow stack must be protected
    - placing the shadow stack between guard pages
    - using a special hardware enforced memory protection mechanism

Part: Isolation Mechanisms

Authentication, Authorization, Auditing

Isolation

- Isolation is a fundamental technique to increase the robustness of computing systems and to reduce their attack surface.
- Isolation can be achieved in many different layers of a computing system:
  - Physical (e.g., preventing physical access to compute clouds)
  - Hardware (e.g., memory management and protection units)
  - Virtualization (e.g., virtual machines, containers)
  - Operating System (e.g., processes, file systems)
  - Network (e.g., virtual LANs, virtual private networks)
  - Applications (e.g., transaction isolation in databases)
- Isolation should be a concern of every system design.
- Isolation also concerns the deployment of computing systems.

Authentication, Authorization, Auditing

Definition (authentication)

Authentication is the act of proving an assertion, such as the identity of a computer system user.

Definition (authorization)

Authorization is a function deciding whether a principal can access a certain resource in a certain way.

Definition (auditing)

Auditing is the process of collecting and storing evidence about authentication and authorization decisions.

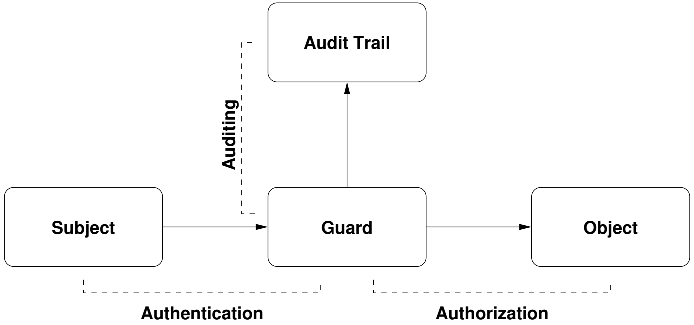
Authentication

Definition (authentication)

Authentication is the process of verifying a claim that a system entity or system resource has a certain attribute value.

- An authentication process consists of two basic steps:
    1. Identification step: Presenting the claimed attribute value (e.g., a user identifier) to the authentication subsystem.
    2. Verification step: Presenting or generating authentication information (e.g., a value signed with a private key) that acts as evidence to prove the binding between the attribute and that for which it is claimed.
- Security services frequently depend on authentication of the identity of users, but authentication may involve any type of attribute that is recognized by a system.

Authorization Lampson Model



- This basic model works well for modeling static access control systems.
- Dynamic access control systems allowing dynamic changes to the access control policy are difficult to model with this approach.

Subjects, Objects, Rights

- Subjects (S): set of active objects
    - processes, users, ...
- Objects (O): set of protected entities
    - files, directories, ...
    - memory, devices, sockets, ...
    - processes, memory, ...
- Rights (R): set of operations a subject can perform on an object
    - create, read, write, delete ...
    - execute ...

Lampson's Access Control Matrix

Definition (access control matrix)

An *access control matrix* M consists of subjects $s_i \in S$, which are row headings, and objects $o_j \in O$, which are column headings. The access rights $r_{i,j} \in R^*$ of subject s_i when accessing object o_j are given by the value in the cell $r_{i,j} = M[s_i, o_j]$.

- Another way to look at access control rights is that the access rights $r \in R^*$ are defined by a function $M : (S \times O) \to R^*$.
- Since the access control matrix can be huge, it is necessary to find ways to express it in a format that is lowering the cost for maintaining it.

Access Control Lists

Definition (access control list)

An access control list represents a column of the access control matrix. Given a set of subjects S and a set of rights R, an access control list of an object $o \in O$ is a set of tuples of $S \times R^*$.

- Example: The inode of a traditional Unix file system (the object) stores the information whether a user or a group or all users (the subject(s)) have read/write/execute permissions (the rights).
- Example: A database system stores for each database (the object) information about which operations (the rights) users (the subjects) can perform on the database.

Capabilities

Definition (capabilities)

A capability represents a row of the access control matrix. Given a set of objects o and a set of rights R, a capability of a subject s is a set of tuples of $O \times R^*$.

- Example: An open Unix file descriptor can be seen as a capability. Once opened, the open file can be used regardless whether the file is deleted or whether access rights of the file are changed. The capability (the open file descriptor) can be transferred to child processes. (Note that passing capabilities to child processes is not meaningful for all capabilities.)
- Example: The Linux system has pre-defined capabilities like `CAP_SYS_TIME` or `CAP_CHOWN` that partition the rights of the root user into more manageable smaller capabilities.

Access Control Lists versus Capabilities

- Both are theoretically equivalent (since both at the end can represent the same access control matrix).
- Capabilities tend to be more efficient if the common question is “Given a subject, what objects can it access and how?”.
- Access control lists tend to be more efficient if the common question is “Given an object, what subjects can access it and how?”.
- Access control lists tend to be more popular because they are more efficient when an authorization decision needs to be made.
- Systems often use a mixture of both approaches.

Discretionary, Mandatory, Role-based Access Control

- Discretionary Access Control (DAC)
  - Subjects with certain permissions (e.g., ownership of an object) can define access control rules to allow or deny (other) subjects access to an object.
  - It is at the subject's discretion to decide which rights to give to other subjects concerning certain objects.
- Mandatory Access Control (MAC)
  - System mechanisms control access to objects and an individual subject cannot alter the access rights.
  - What is allowed is mandated by the security policy implemented by the security administrator of a system.
- Role-based Access Control (RAC)
  - Subjects are first mapped to a set of roles that they have.
  - Mandatory access control rules are defined for roles instead of subjects.

Fine-grained Operating System Security Profiles

Basic Operating System Isolation Services

- Basic isolation services provided by operating system kernels:
  - Isolation of process memory via virtual memory
  - Isolation of storage devices via filesystems
  - Isolation of network devices via sockets
  - Isolation of keyboard, pointer, display devices
  - Isolation of pluggable devices
- These are coarse grained isolation mechanisms
- More fine grained isolation mechanisms are system specific

POSIX User and Group IDs

Definition (user ID, group ID)

A *user ID* (UID) is a positive integer assigned to a user of a POSIX system. A *group ID* (GID) is a positive integer assigned to a group on a POSIX system.

- A user refers to an account on the system and accounts may represent software services.
- Every user is a member of one or more groups.
- The UID 0 has a special meaning and refers to the root user, an account with special and typically unlimited privileges.
- The UID 65534 and the GID 65534 are commonly reserved for nobody, a user and a group with no system privileges.

POSIX Process User and Group IDs

Definition (real user ID, real group ID)

The *real user ID* (ruid) and the *real group ID* (rgid) of a process is the UID and the GID of the user who invoked a program.

Definition (effective user ID, effective group ID)

The *effective user ID* (euid) and the *effective group ID* (egid) of a process are used to check permissions.

Definition (saved user ID, saved group ID)

The *saved user ID* (suid) and the *saved group ID* (sgid) of a process are a UID and GID pair that are (temporarily) not used to check permissions.

POSIX File System Owner and Group IDs

Definition (owner, group, other)

Every file system object has an associated *owner* (a UID) and *group* (a GID). Separate permissions (read, write, execute, ...) are associated with the owner of a file system object, the members of a group of a file system object, and all others.

Definition (effective permissions)

The *effective permissions* are determined based on the first match in the order of user, group then others properties of a file system object.

- The owner of a file system object has the right to set the permissions.
- More fine grained access control lists are defined in POSIX 1003.1e.

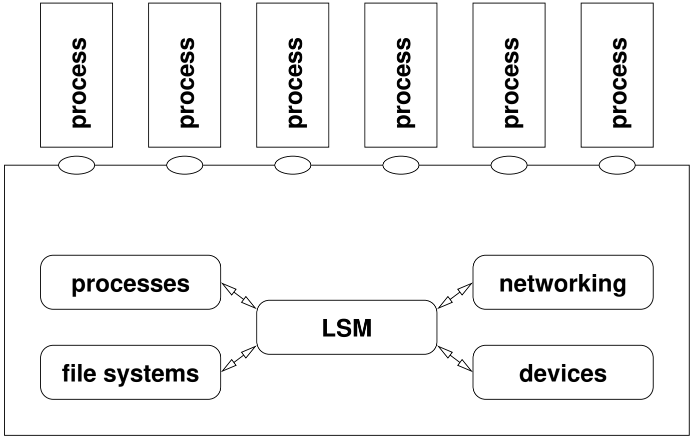
POSIX Permission Model

- The kernel via its system calls acts as a guard checking which process (euid,egid) has access permissions against which objects.
- For file systems, the access control lists are stored as part of the file system objects.
- Access control lists are discretionary, i.e., the owner of a file system object controls the access rights.
- The powerful root user can overwrite and change all permissions.
- The permission model is coarse grained, applications often have permissions that they do not need to do their work.

Linux Security Modules

- There is wide agreement that mandatory access control is necessary in certain deployment scenarios.
- There is no agreement on a single solution to express mandatory access control policies.
- As a consequence, the Linux kernel provides an internal API enabling kernel programmers to write different security modules without requiring major changes across the entire kernel.
- Specific security modules are built into the kernel and not loaded into the kernel.
- Users have to choose between special security modules (they do not compose).

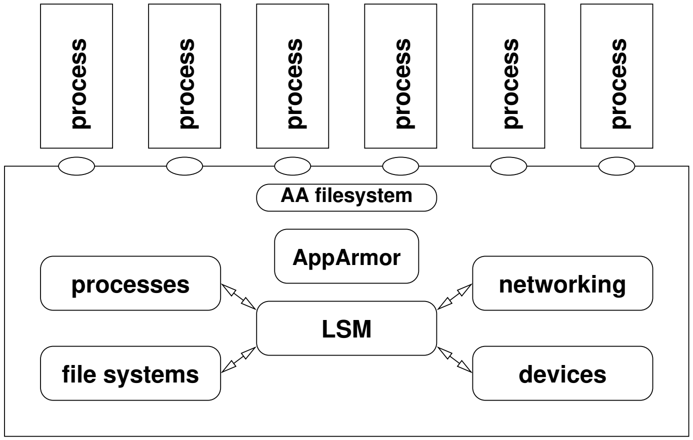
Linux Security Modules Kernel View



Application Armor (AppAmor)

- AppAmor restricts permissions of programs via security profiles.
- Security profiles are loaded into the kernel.
- Profiles are automatically applied to programs.
- Profiles implement mandatory access control.
- AppAmor is enabled by default on Ubuntu and Debian systems.

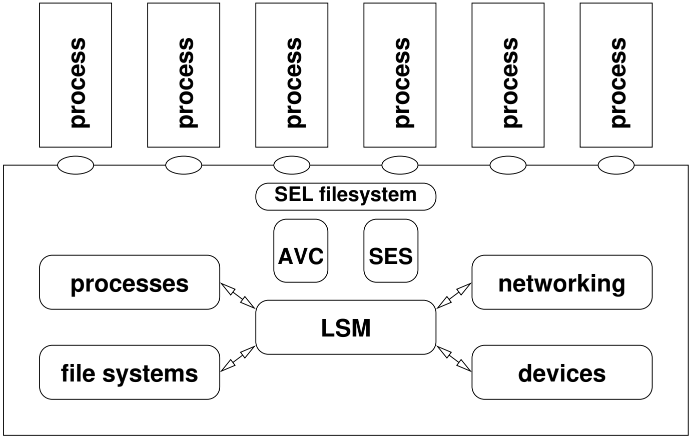
AppArmor Kernel View



Security Enhanced Linux (SELinux)

- SELinux was originally created by the NSA
- Part of the Linux kernel since 2003 (kernel version 2.6)
- Fedora was an early adopter of SELinux
- Android uses SELinux to enforce mandatory access control rules

SELinux Kernel View



SELinux Security Contexts

- All subjects (processes) and objects (files, directories, sockets, network ports, ...) have a security context associated with them.
- The security context of a process is sometimes called a domain.
- A security context has three mandatory elements (the level is optional):

```
user:role:type[:level]
```

- The security context of processes and file system objects can be obtained using the `-Z` option on some command line tools:

```
$ id -Z
$ ps -Z
$ ls -Z
```

- The security context of file system objects is stored in file system attributes.

SELinux Allow Rules (Type Enforcement)

- SELinux is default deny, all access needs to be explicitely granted.
- An allow rule has four elements:
  1. The type of the subject (the type of the security context of a process)
  2. The type of the object being accessed (the type of the security context of an object)
  3. The class of the object being accessed (file, directory, ...)
  4. The permissions, i.e. the kind of access allowed to the object
- Example: A process with a security context type of `user_t` can read, execute, or get attributes for a file object with a type of `bin_t`.

```
allow user_t bin_t: file { read execute getattr };
```

SELinux Domain Transitions

- A user shell with type user_t needs to transition to the type passwd_t when the user runs the program /bin/passwd.
- This is allowed by domain transition rules:

```
allow user_t passwd_exec_t: file { getattr, execute };
allow passwd_t passwd_exec_t: file entrypoint;
allow user_t passwd_t: process transition;
```

- The first rule allows user_t to execute programs with type passwd_exec_t.
- The second rule states that passwd_t has entry point permissions on programs with type passwd_exec_t. Entrypoint permissions mean that the security context passwd_t may start with this file.
- The third rule says that a user_t process may transition into a passwd_t process.

SELinux Users and Roles

- The following definitions define the role `user_r` and associate the type `passwd_t` with it.

  ```
  role user_r;
  role user_r types passwd_t;
  ```

  This definition specifies that `passwd_t` can exist in a context with the role `user_r`.
- A user may have multiple roles and roles may have multiple types.

  ```
  user full_u roles { mgr_r, cashier_r };
  role cashier_r types { cashier_t, cashier_register_t };
  ```
- While users and roles are an essential part of an SELinux security context, the access control decision is based on the type enforcement logic.

Sandboxing Applications and Privilege Separation

Berkeley Packet Filter (BPF)

- Human readable filter expressions are translated into (optimized) BPF programs using a tiny compiler.
- Compiled BPF programs are loaded into the kernel.
- BPF programs are invoked on each received packet to decide whether to move them for further analysis to user-space.
- The original BPF machine has the following components:
  - An accumulator for all calculations
  - An index register (x) allowing access to data relative to a certain position
  - Memory for storing intermediate results
- All registers and memory locations are 32-bit wide

Extended Berkeley Packet Filter (eBPF)

- Main changes compared to the classic BPF (cBPF):
  - 64-bit registers instead of 32-bit registers
  - 10 general purpose registers plus a read-only frame pointer register
  - different jump semantics
  - call instruction and corresponding register passing conventions
  - several new instructions and atomic operations
  - different instruction encoding
- Application of eBPF:
  - network packet filtering (the traditional use case)
  - network policy processing (e.g., load balancing)
  - kernel monitoring and tracing
  - security monitoring and security policy enforcement

Secure Computing (seccomp and seccomp-bpf)

- The Linux Secure Computing feature enables processes to voluntarily reduce the system calls they can use.
- The motivation is that many server processes go through some initialization and afterwards only needs very few system calls.
- The basic idea was extended extended in 2005 to support flexible filtering of system calls via BPF filters.
- This simple but effective mechanism has been picked up by several crucial server and system components.

Linux firejail

- The firejail program can run unmodified programs in a sandbox
- Firejail uses Linux namespaces and seccomp-bpf to create sandboxes
- Program specific profiles are relatively easy to create
- Firejail project started in 2014 (GPL) and is gaining traction
- A downside of firejail is that it requires root privileges

Privilege Separation

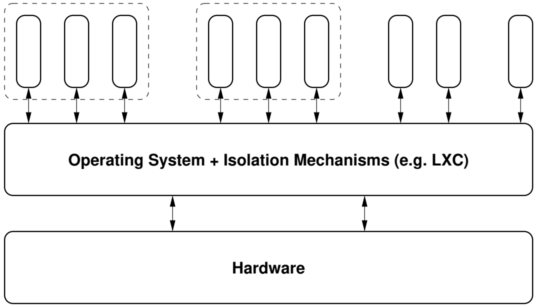
Definition (privilege separation)

Privilege separation divides a complex program into smaller components, each running with least privileges.

- Privilege separation aims at limiting the damage caused by a compromised program.
- Requires efficient communication between the smaller components.
- Good software architecture makes privilege separation easier.
- Library isolation is a form of privilege separation aiming at isolating libraries.

Container and Container Security

OS-Level Virtualization (Container)



- Separation of processes
- Small performance overhead
- Single operating system ABI
- Linux Container (LXC)
- BSD Jails
- Solaris Zones
- ...

Container Terminology

- An *image* is a read-only template of a container. An image consists of layers stacked on each other. Images are portable and can be stored in repositories.
- A *layer* is a part of an image, it may consist of a command or files that are added to an image.
- A *repository* is a collection of (versioned) images.
- A *registry* (like Docker Hub) manages repositories.
- A *container* is an active (running) instance of an image. An image can have many concurrently running container.
- A *runtime* turns and image into a running container.
- A *host operating system* executes containers and the runtime.

Linux Namespaces

Name	Description
cgroup	separates control groups used to manage resources
ipc	separates interprocess communication facilities
mnt	separates file systems
net	separates network stacks
pid	separates processes
time	separates system time and timezones
user	separates user identities
uts	separates host and domain names

- Linux namespaces isolate global operating system resources.

Linux Control Groups

A control group (cgroup) is a collection of processes under a set of resource limits. Control groups are hierarchical and control resources such as memory, CPU, block I/O, or network usage. Controller (subsystems) have been implemented for these resources:

- cpu scheduling and accounting
- cpu pinning (assigning specific CPUs to specific tasks)
- suspending or resuming tasks
- memory limits
- block I/O
- network packet tagging setting network traffic priorities
- namespaces
- performance analysis data collection

Container Security: Images

Images

Images bundle the software and data accessible to a running container. Images are typically constructed by building on other images, using overlay filesystems.

Recommendations:

- keep images updated
- understand image dependencies
- limit permissions on images
- sign images to ensure that their integrity can be verified

Container Security: Image Repositories

Image Repositories

Images are stored in an image repository from where they can be pulled and started.

Recommendations:

- keep your image repository private
- monitor your image repository for changes
- harden the server hosting images

Container Security: Container Runtimes

Container Runtimes

A container runtime is responsible for starting and managing containers according to some configuration using a given host operating system. Container runtimes can be minimal or coming with many features (recall that complexity is the enemy of security).

Recommendations:

- keep the container runtime updated
- monitor the container runtime

Container Security: Container Orchestrator

Container Orchestrator

A container orchestrator provisions, deploys, scales, and manages containerized applications consisting of multiple containers.

Recommendations:

- configure access controls
- monitor orchestrator

Container Security: Host Operating System

Host Operating System

The host operating system executes the processes of the containers and the container runtime.

Recommendations:

- minimal host operating systems are easier to harden
- keep the host operating system and container runtime updated
- use mandatory access control (e.g., SELinux)
- monitor the host operating system

Podman vs. Docker Security

- Docker appeared in 2013 (Docker Inc. and others)
- Podman appeared in 2018 (Red Hat and others)
- Docker does its work through a service daemon
- Podman does not require a service daemon
- Podman is rootless, it can run container from a regular user account
- Podman supports user id separation
- Podman can also run pods, which aggregate related container into a single unit (called a pod)
- Podman integrates with secure computing and SE Linux

Part: Trusted and Confidential Computing

Trusted Computing Base

Definition (trusted computing base)

The *trusted computing base* of a computer system is the set of hard- and software components that are critical to achieve the systems' security properties.

- The components of a trusted computing base are designed such that when other parts of a system are attacked, the device will not misbehave.
- Trusted computing bases should be small to be able to verify their correctness.
- Trusted computing bases should be tamper-resistant.
- Trusted computing bases typically involve special hardware components.

Trusted Computing Security Goals

- *Isolation*: Separation of essential security critical functions and associated data (keys) from the general computing system.
- *Attestation*: Proving to an authorized party that a specific component is in a certain state.
- *Sealing*: Wrapping of code and data such that it can only be unwrapped and used under certain circumstances.
- *Code Confidentiality*: Ensures that sensitive code and static data cannot be obtained by untrusted hardware or software.
- *Side-Channel Resistance*: Ensures that untrusted components are not able to deduce information about the internal state of a trusted computing component.
- *Memory Protection*: Protects the integrity and authenticity of data sent over system buses or stored in (external) memory from physical attacks.

Trusted Platform Modules

Trusted Platform Module (TPM)

- A Trusted Platform Module (TPM) is a dedicated micro-controller designed to secure hardware through integrated cryptographic operations and key storage.
- The TPM 1.2 specification was published in 2011:
  - Co-processor capable of generating good random numbers, storing keys, performing cryptographic operations, and providing the basis for attestation.
  - Limited protection against physical attacks.
- The TPM 2.0 specification was published in 2014.
  - Support of a larger set of cryptographic algorithms and more storage space for attestation purposes.
- The TPM specifications have been created by the Trusted Computing Group (a consortium of vendors with large influence of Microsoft on TPM 2.0).

Trusted Platform Module Version 2.0

TPM 2.0 implementations can come in various forms:

- Discrete TPMs are dedicated chips implementing TPM functionality in their own tamper resistant semiconductor package.
- Integrated TPMs are part of another chip.
- Firmware TPMs are software-only solutions that run in a CPU's trusted execution environment.
- Software TPMs are software emulators of TPMs that run with no more protection than a regular program gets within an operating system.
- Virtual TPMs are provided by a hypervisor and rely on the hypervisor to provide them with an isolated execution environment.

Trusted Execution Environments

Trusted Execution Environment (TEE)

Definition (trusted and rich execution environment)

A *trusted execution environment* (TEE) is a secure area of a processor providing isolated execution, integrity of trusted applications, as well as confidentiality of trusted application resources. A *rich execution environment* (REE) is the non-secure area of a processor where an untrusted operating system executes.

- REE resources are accessible from the TEE
- TEE resources are accessible from the REE only if explicitly allowed.
- The TEE specifications have been created by the GlobalPlatform (another industry consortium).

TrustZone Cortex-A (ARM)

- The ARM processor architecture has an internal communication interface called the Advanced eXtensible Interface (AXI).
- ARM's TrustZone extends the AXI bus with a Non-Secure (NS) bit.
- The NS bit conveys whether the processor works in secure mode or in normal mode.
- The processor is normally executing in either secure or normal mode.
- To perform a context switch (between modes), the processor transits through a monitor mode.
- The monitor mode saves the state of the current world and restores the state of the world being switched to.
- Interrupts may trap the processor into monitor mode if the interrupt needs to be handled in a different mode.

TrustZone Cortex-M (ARM)

- The Cortex-M design follows the Cortex-A design by having the processor execute in either secure or normal mode.
- Instructions read from secure memory will be executed in the secure mode of the processor and instructions read from non-secure memory will be executed in normal mode.
- Cortex-M replaces the monitor mode of the Cortex-A design with a faster mechanism to call secure code via multiple secure function entry points (supported by the machine instructions `SG`, `BXNS`, `BLXNS`).
- The Cortex-M design supports multiple separate call stacks and the memory space is separated into secure and non-secure sections.
- Interrupts can be configured to be handled in secure or non-secure mode.

Security Guard Extension (SGX, Intel)

- SGX places the protected parts of an application in so called enclaves that can be seen as a protected module within the address space of a user space process.
- SGX enabled CPUs ensure that non-enclaved code, including the operating system and potentially the hypervisor, cannot access enclave pages.
- A memory region called the Processor Reserved Memory (PRM) contains the Enclave Page Cache (EPC) and is protected by the CPU against non-enclave accesses.
- The content of enclaves is loaded when enclaves are created and measurements are taken to ensure that the content loaded is correct.
- The measurement result obtained during enclave creation may be used for (remote) attestation purposes.
- Entering an enclave is realized like a system call and supported by special machine instructions (`EENTER`, `EEXIT`, `ERESUME`).

Virtual Machine Memory Encryption

Secure Boot

Secure Boot, Measured Boot, Remote Attestation

Definition (secure boot)

Secure boot refers to mechanisms that verify signatures of all software components (e.g., firmware, bootloader, kernel) involved in the boot process of a computing system.

Definition (measured boot)

Measured boot refers to mechanisms that take measurements during the boot process of a computing system. The cryptographically protected measurements can be compared against expected baseline values to detect whether a system has been compromised.

Definition (remote attestation)

Remote attestation refers to mechanism that enable a remote verifier to obtain trustworty evidence about the integrity and security properties of the attester.

Remote Attestation

Confidential Computing

Part: Malware Analysis and Detection

Malicious Software

Definition (malicious software)

Software designed to cause disruption to a computing system or to gain unauthorized access to data or to interfere in any other way with computer security and data privacy is called *malicious software* (or short malware).

- A specific piece of malware under investigation is called a malware sample.
- The term malware (sample) is often used even if the is not yet evidence that the software under investigation has any malicious behavior.

Malware Signatures

Definition (malware signatures)

Malware properties that can be used to identify specific items of malware are called *malware signatures*.

- Some signatures, like whole-file hashes, are very precise but more fragile regarding changes of the malware.
- Some signatures, like specific byte sequences embedded in an executable, are less precise but less fragile against changes of the malware.
- The analysis of malware typically leads to signatures that can be used to scan for instances of malware efficiently.

Indicators of Compromise

Definition (indicators of compromise)

An *indicators of compromise* (IoC) is an artefact observed on a computer system indicating with high confidence a computer intrusion.

- IoCs generally do not provide evidence of an infection.
- It is challenging to keep track of the many known IoCs and their relevance.
- Thread intelligence platforms track IoCs to produce security insights and reports.
- Malware signatures can server as IoCs but not all IoCs are malware signatures.

Pyramid of Pain

```
                          /\                              MORE PAIN
                        /    \                           LESS FRAGILE
                      /  TTPs  \                         LESS PRECISE
                    /------------\                            |
                  /      Tools     \                          |
                /--------------------\                        |
              / Network/Host Artefacts \                      |
            /----------------------------\                    |
          /          Domain Names          \                  |
        /------------------------------------\                |
      /              IP Addresses              \              |
    /--------------------------------------------\        LESS PAIN
  /                   Hash Values                  \     MORE FRAGILE
/----------------------------------------------------\   MORE PRECISE
```

Malware Analysis

Malware Analysis Process

Definition (maleware analysis process)

A *malware analysis process* normally consists of the following steps:

1. Collecting and safely archiving malware samples
2. Documenting malware meta-data and contextual information
3. Preliminary analysis to determine subsequent steps
4. Technical analysis (static, dynamic, hybrid)
5. Producing an analysis report documenting the analysis process, the findings, and providing recommendations for detection and mitigation

- It is essential to document all findings appropriately.

Approaches to Malware Analysis

Definition (static malware analysis)

Static malware analysis investigates a malware sample without executing it.

Definition (dynamic malware analysis)

Dynamic malware analysis investigates a malware sample by executing it in a controlled environment and observing its behavior.

Definition (activate malware analysis)

Active malware analysis is a form of dynamic malware analysis where user input is generated to stimulate a malware sample to expose its malicious behaviour.

Precautions to Malware Analysis

- Use an offline virtual machine for malware analysis
- If feasible use separate hardware that can be completely wiped
- Frequent snapshots can speed up the process and provide evidence

Reverse Engineering Tool Example: Radare2

- Radare2 is a framework for reverse-engineering and analyzing binaries.
- Radare2 features
  - disassembler for many assembler programming languages
  - analyzers of binary and excutable file formats
  - a buit-in debugger and interfaces to external debuggers
- Radare2 is a command-line application
- Various graphical user interfaces exist as 3rd party tools
- Radare2 is open source (GPLv3)

Packer and Polymorphic Code

Definition (packer)

A *packer* is a program compressing (or optionally encrypting) executables in order to make the smaller and/or to potentially hide their internal structure.

Definition (polymorphic code)

Polymorphic code is code that changes itself every time it is executed without changing the functionality of the code.

- Packers just aiming to reduce the size of an executed are also called compressors.
- Packets aiming to obfuscate the original code are also called obfuscators.

Malware Detection

Malware Detection Example: YARA

- YARA is a tool to classify files using textual and binary pattern
- YARA was originally developed by Victor Alvarez (Virustotal)
- Initial release was 2013
- Written as C library (easily integrated into other tools)
- Open source license (BSD)
- Large collection of signature files

YARA Signature Rule Format

- mandatory rule name
- optional metadata
  - set of key/value pairs
- optional string definitions
  - hexadecimal strings
  - text strings
  - regular expressions
- mandatory condition
  - boolean, relational, bitwise operators
  - counting operator
  - offsets specifiers
  - iterators
  - rule references

```
rule example {

    meta:
        author = "Alice and Bob"

    strings:
        $hex = { 6d 6f 69 6e }
        $text = "moin"
        $regex = /moin/

    condition:
        $hex and $text and #regex > 0
}
```

Malware Detection Example: ClamAV

- ClamAV was originally developed by Tomasz Kojm
- ClamAV is meanwhile owned by Cisco Systems
- Initial release was 2001
- Written in C and C++
- Open source license (GPL)
- Huge collection of signatures

** Clamav Signatures
Clamav ¡https://www.clamav.net/¿ is an open source antivirus scanner which is widely used to scan email messages but it can be used to scan regular filesystems as well. Clamav uses signature formats that are specific for a given method of detection. Signatures can be collected into CVD files, which are digitally signed archives of signatures.
On Debian/Linux, clamav signatures are stored in /var/lib/clamav.
sigtool –info /var/lib/clamav/main.cvd sigtool –unpack /var/lib/clamav/main.cvd

Part: Security Incident Detection and Response

Intrusion Detections and Prevention

Intrusion Detection Systems

Definition (intrusion detection system)

An *intrusion detection system* (IDS) monitors a computing system for suspicious or malicious activity.

Definition (network-based intrusion detection system)

A *network-based intrusion detection system* (NIDS) monitors a network for suspicious or malicious traffic.

Definition (host-based intrusion detection system)

A *host-based intrusion detection system* (HIDS) monitors a single host for suspicious or malicious activity.

Detection Methods

Definition (signature-based intrusion detection)

A *signature-based intrusion detection* method compares observed data against signatures of known threats to identify possible intrusions.

Definition (anomaly-based intrusion detection)

An *anomaly-based intrusion detection* method is a statistical method for identifying possible intrusions by classifying observed behavior as normal or abnormal.

- Intrusion detection systems supporting both signature-based and anomaly-based methods are called hybrid intrusion detection systems.

Intrusion Prevention Systems

Definition (intrusion prevention system)

An *intrusion prevention system* is an intrusion detection system that can trigger an automated action to prevent or block intrusions that have been detected.

Network Intrusion Detection Example: Snort

- created in the late 1990s by Martin Roesch
- signature-based intrusion detection
- optionally intrusion prevention
- simple programmer-friendly rule format (in particular snort 3)
- large collections of detection rules
- written in C/C++
- distributed under the GNU GPLv2 license

Host-based Intrusion Detection Example: Wazuh

- fork of OSSEC
- integration with elastic search
- written in C
- distributed under the GNU GPLv2 license

Security Information and Event Management

Security Information and Event Management

Definition (security information and event management)

A *security information and event management* (SIEM) system (SIEM) provides real-time analysis of security information and security events.

Services provided by a SIEM are:

- data aggregation
- data correlation
- generation of alerts
- collection of compliance data
- focilities for data retention
- support for forensic analysis

Detection and Response Acronyms

EDR	Endpoint Detection and Response (threat detection and response on endpoints)
MDR	Managed Detection and Response (external company providing EDR service)
XDR	eXtended Detection and Response (EDR extended to cover non-host systems plus some SIEM)
MXDR	Managed eXtended Detection and Response (external company providing XDR service)

SIEM Example: Elastic Stack

The Elastic Stack consists of the following components:

- Elasticsearch: A distributed open source search and analytics engine
- Kibana: A dashboard for interacting with Elasticsearch
- Logstash: A data conversion pipeline feeding data into Elasticsearch
- Beats: A lightweight data shipper passing data to Logstash

All components are open source, mostly written in Java and Go.