# **- Introdução**

**Porquê comprometer a máquina de um utilizador?**

* Razão #1:
* credenciais: roubar passwords bancárias, empresariais, para jogos
* Razão #2: ransomware
* Dia 1: vulnerabilidade utilizada por agências governamentais divulgada
* 3 semanas depois: Wannacry
* Razão #3:
* para utilizar o processador => minerar bitcoin
* Razão #4:
* para usurpar o endereço de rede e parecer um utilizador normal
* Spam: o spam funciona (spamalytics)
* Denial of Service
* Clicks (e.g., Clickbot.a)
* Todos estes serviços são vendidos na internet:

Para isso é necessário controlar um conjunto de máquinas

**Porquê comprometer servidores?**

* Data breaches
* Números de cartão de crédito e credenciais de utilizadores <https://en.wikipedia.org/wiki/List_of_data_breaches>
* Motivações políticas e geo-estratégicas
* DNC, Infraestrutura elétrica na Ucrânia, Stuxnet
* Para depois infetar as máquinas dos utilizadores:
* supply-chain attacks (infetar servidor que distribui software)
* web-server attacks (infetar servidor web, que depois compromete browsers)

**Comprometer utilizadores**

* Servidores que distribuem software permitem disseminar malware:
* Exemplo: SolarWinds, ferramentas de monitorização
* Servidores Web permitem comprometer browsers vulneráveis:
* Exemplo: MPack conjunto de ferramentas server side para “website defacing”

**Comprometer developers**

* Um dos ataques mais simples é o chamado “typo squatting”
* Exemplo: PyPI gestor de pacotes >300,000 projetos
* Supply-chain attack:
* Pacote malicioso com nome similar => developer instala o pacote errado

**“Segurança”?**

* Uma definição comum
* “A propriedade de um sistema que se comporta como esperado”
* Esta definição não diz o que o sistema deve ou não deve fazer:
* Não existe uma definição universal ou teste

**Atores**

* Atores ou participantes são entidades que intervêm no sistema:
* Pessoas, organizações, empresas, máquinas, …
* A segurança define-se do ponto de vista destes atores
* Muitas vezes deposita-se confiança em alguns atores/componentes
* e.g., Trusted Third Party (TTP), Trusted Agent (TA)
* Sistema “seguro" se pressuposto de confiança se verificar

senão? por ex, defesa em profundidade

* Noutras os atores são potenciais atacantes (externos/internos)

**Adversário/Atacante**

* Na segurança informática analisamos o comportamento de sistemas quando interagem com adversários/atacantes:
* Atores com intenção explícita de utilizar o sistema/recursos de forma indevida ou de impossibilitar a sua utilização
* É crítico conhecer o nosso adversário (motivação, capacidades, acesso):
* “Script-kiddies” (curiosos mas incapazes)
* Atacantes ocasionais que visam compreender os sistemas
* Pessoas com intenção de causar danos/destruição
* Grupos organizados e tecnicamente sofisticados (e.g., ciber crime)
* Competidores (espionagem industrial)
* Países/Estados/Governos
* É preciso pensar como um adversário/ atacante:
* Procurar sempre o elo mais fraco
* Identificar pressupostos de confiança subjacentes à segurança

São válidos?

* Olhar para o sistema "out of the box"

Quem desenha um sistema está sempre "preso" ao que ele é suposto fazer

# **- “Segurança”**

**Correção ⊂ Segurança**

* Correção: bom input => bom output
* e.g. “utilizador e data válidos” => “histórico do utilizador nessa data”
* Segurança: mau input =/> mau output
* e.g. “dados fidedignos” + “outros dados nunca são revelados”

**Dois modelos:**

* **Binário (seguro vs inseguro)**
* Típico em criptografia e sistemas confiáveis
* Definir formalmente capacidades dos atacantes X
* Definir formalmente objetivos de segurança Y
* Nenhum atacante limitado a X consegue quebrar Y
* Terminologia típica: prova de segurança, secure by design

O problema de segurança é suficientemente pequeno/simples que se consegue escrever um modelo matemático para o descrever. Esse modelo matemático captura as capacidades do atacante definidas formalmente e também se consegue definir formalmente o que é suposto não acontecer, ou seja, o que é que significa o adversário ter sucesso. É possível provar-se matematicamente que se considerarmos uma classe de atacantes bem definida e um sistema concreto para um determinado objetivo de segurança, não há nenhum atacante naquela classe que consiga quebrar o esquema/protocolo.

* **Gestão de Risco (mais seguro vs menos seguro)**
* Típico em engenharia de software e segurança no mundo real
* Minimizar risco em função das ameaças mais prováveis
* Otimizar o custo das medidas de segurança vs potenciais perdas
* Terminologia típica: resiliência, mitigação, risco

**Ambos têm limitações**

* Modelo binário:
* Não escala para sistemas complexos
* Modelo abstrato ≠ sistema concreto
* Os modelos formais podem ignorar problemas reais, e.g., side-channels
* Gestão de risco:
* A análise de risco pode estar errada
* Nunca estamos verdadeiramente seguros (se é que isso existe!)
* Uma ameaça mal classificada pode deitar tudo por terra

**Segurança = CIA**

* 3 eixos essenciais de segurança:
* Confidencialidade (segredo, privacidade)
* Integridade (não alteração, dados fidedignos, autenticidade de origem)
* Disponibilidade (existência, constância)

**Em geral**

* A segurança é um processo de gestão de custos
* Por vezes sai mais barato simplesmente aceitar o risco de que o ataque ocorra (e.g., fraude no cartão de crédito vs ataque nuclear)

Quanto custaria um possível ataque

↑ ↓

Quanto custa implementar defesa

**Gestão de Risco: Ativos**

* **Gerimos o risco** de que ativos (recursos) relevantes sejam usados indevidamente ou estejam indisponíveis.
* Um **ativo** é um recurso que detém valor para um ator do sistema:
* e.g. informação, reputação/imagem, dinheiro/recurso com valor monetário intrínseco, infraestrutura, etc.
* O risco que se pretende acautelar é o de que um ativo perca o seu valor durante a utilização do sistema
* CIA => Risco de perda de valor por quebra de:
* Confidencialidade (acessos não autorizados)
* Integridade (modificações não autorizadas)
* Disponibilidade (quebras de serviço)

Table

Description automatically generated**Matriz de análise de risco**

* eixos: ativos + ameaças
* A ação a tomar depende de dois fatores críticos:
* o potencial impacto no ativo
* a probabilidade de materialização da ameaça
* cada par (ameaça,ativo) tem de ser avaliado nestes dois eixos
* a decisão de mitigação depende do risco avaliado

**Vulnerabilidades**

* Uma **vulnerabilidade** é uma falha que está acessível a um adversário que poderá ter a capacidade de a explorar
* As vulnerabilidades têm geralmente origem em erros de conceção:
* Software de má qualidade
* Análise de requisitos desadequada
* Configurações erradas
* Utilização errada

**Ataques**

* Um **ataque** ocorre quando alguém tenta explorar uma vulnerabilidade
* Tipos de ataques:
* Passivos (e.g., eavesdropping)
* Ativos (e.g., adivinhar passwords)
* Denial-of-Service
* Quando um ataque é bem-sucedido diz-se que um sistema foi comprometido

**Estrutura de um ataque**

* Um ataque precisa de MOM:
* **Motivo/Ameaça:** perturbação, roubo de informação, …
* **Oportunidade/Vulnerabilidade:** algo que pode ser explorado
* **Método/Exploit:** forma de explorar a vulnerabilidade

**Reporting**

* Novas vulnerabilidades aparecem todos os dias
* A gestão desses processos chama-se vulnerability reporting
* Toda a comunidade trabalha em conjunto para:
* Identificar, classificar, divulgar, detetar, eliminar
* Um utilizador pode:
* ser notificado, obter informação, obter uma ferramenta que verifica se a vulnerabilidade existe

**Ameaças**

* Uma ameaça é uma causa possível para um incidente que possa trazer consequências negativas para um sistema, pessoa ou organização
* Dependem do contexto:
* Podem ser pessoas, eventos naturais, falhas acidentais, ser causadas intencionalmente
* Geralmente definem-se pelo seu tipo e origem:
  + Tipo de ataque: dano físico, perda de serviços, quebra de proteção de informação, falhas técnicas, abuso de funcionalidades
  + Tipo de atacante: ação deliberada (implica definir a fonte), ação negligente (implica definir a fonte), acidente (implica definir o componente afetado), evento ambiental.
* Dependendo da comunidade (e.g., finança vs distribuição de água)
  + As ameaças estão identificadas e classificadas quanto à relevância

**Modelo de Ameaças**

* + Objetivos de segurança: ativos, o que queremos proteger e de quem?
  + Quem são os nossos adversários?
  + Motivação
  + Capacidades
  + Tipo de Acesso
  + Que tipos de ataque temos de precaver (pensar como um atacante)
  + Âmbito: que tipos de ataque podemos descartar/ignorar?
* Perímetro de segurança
  + fronteira que delimita um contexto com o mesmo nível de segurança
  + quaisquer inputs provenientes do exterior são suspeitos
* Superfície de ataque
  + pontos/formas de contacto com o exterior no perímetro de segurança

**Mecanismo de segurança**

* + Um mecanismo de segurança é um método, ferramenta ou procedimento que permite implantar uma (parte de uma) política de segurança
  + Os mecanismos podem ser não técnicos, e.g., exigir identificação pessoal na entrada de um edifício
  + Parte do nosso modelo de confiança consiste em acreditar que um (conjunto de) mecanismo(s) de segurança cumpre a sua função, que está corretamente instalado e administrado, etc.

**Política de segurança**

* + Uma política de segurança determina:
  + um conjunto de processos/mecanismos que devem ser seguidos
  + para garantir segurança num determinado modelo de ameaças
  + Uma política de segurança pode ter como objetivos:
* prevenção, deteção e/ou recuperação

**Gerir a Complexidade**

* + Em sistemas complexos o problema da segurança tem de ser tratado de forma estruturada
  + Segurança de redes, de sistemas/computadores, de programas/software
  + Segurança física
  + Privacidade
  + …

**Confiabilidade**

* + É essencial que a "segurança" não seja um “salto de fé”
  + Engenharia de segurança:
  + construir confiança através de argumentos rigorosos

**1)**

* Definir o problema:
  + Análise de requisitos de segurança
  + Definir o modelo de ameaças

Ativos e objetivos de segurança

Classes relevantes de ameaças (o adversário)

**2)**

* Definir o modelo de confiança:
  + em que componentes/atores confiamos
  + para fazer o quê (o que nos é dado como ponto de partida/âncora)
* Definir a solução:
  + Conceber as políticas de segurança que se aplicam ao sistema
  + Identificar os mecanismos de segurança necessários e suficientes

**3)**

* Validar e justificar a solução:
  + Validar a adequação dos modelos à aplicação concreta em análise

(o adversário não é obrigado a seguir o nosso modelo de ameaças)

* + Demonstrar (formal ou informalmente) que

os mecanismos de segurança subjacentes são suficientes,

em conjunto com os pressupostos de confiança assumidos,

para implantar as políticas de segurança (justificação)

**Testes de penetração: ethical hacking**

* + Método de avaliação do nível de segurança de um sistema
  + Simulação de um ataque de procura de vulnerabilidades que poderiam ser exploradas
  + Black box: semelhante a um ataque real
  + White box: com conhecimento privilegiado

# **- Segurança de Software: Controlo (Parte 1)**

**Vamos começar “bottom-up”**

* + O software executa numa plataforma computacional
* Armazenamento dinâmico (memória) e persistente (disco)
* Sistema Operativo
* Hardware de processamento + Interfaces com o exterior
  + É sempre uma potencial arma contra essa plataforma:
* Pode permitir ganhar o controlo sobre essa plataforma
* Pode depois ser utilizado para diversos fins: DoS, SpyWare, Jailbreak, etc.

Diagram

Description automatically generated**Modelo de Memória**

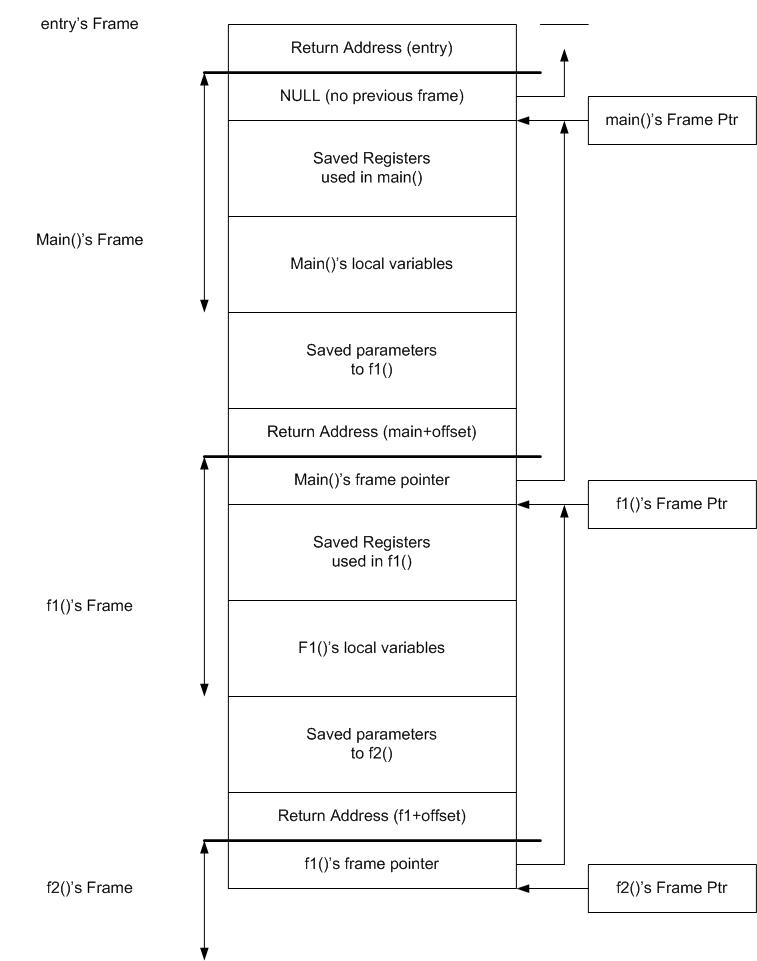
* Memória com código (text)
* Memória com dados globais/estáticos (data)
* Memória alocada "on demand” (heap)
* Memória alocada automaticamente (stack):
  + endereço de retorno
  + variáveis locais
  + argumentos para funções

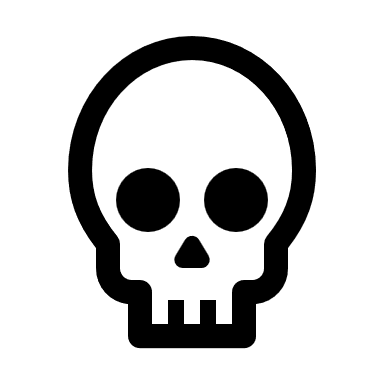
**Usurpação de Controlo**

* + Ocorre quando um input externo leva o programa a quebrar a convenção:
  + alterando a sequência de instruções que é executada
  + substituindo a sequência de instruções esperada por uma sequência de instruções controlada pelo atacante
  + pode ocorrer na stack, na heap, nas chamadas ao sistema, etc.
  + muito mais difícil hoje do que há 30 anos
  + mas ainda muito comum

**Funcionamento típico da Stack**

* + Cada função funciona com base num contexto local chamado stack frame:
  + frame pointer guardado num registo (e.g., ebp) aponta para a base dessa região (variáveis locais)
  + stack pointer guardado num registo (e.g., esp) aponta para o topo (endereço mais baixo
  + contém: parâmetros, dados guardados para o retorno, variáveis locais, outros valores guardados temporariamente.
  + Main(…) → f1(…) → f2(…)



**Diagram

Description automatically generatedStack Smashing**

* + Se str vier de fora do perímetro de segurança:
  + atacante pode preencher stack com código
  + substituir endereço de retorno
  + executar código arbitrário
  + Não há separação entre dados e programa!

**Text

Description automatically generated**

**Que código injetar?**

* + "Shell code" é uma sequência de instruções que executa comandos shell:
  + codificado em código máquina
  + precedido de algumas operações NOP (porque o seu endereço em memória pode variar)
  + injetado em memória para ser executada no retorno da função onde se provoca o buffer overflow

**O diabo está nos detalhes**

* + A construção de um exploit na prática exige muito "fine tuning"
    - e.g., a codificação do código como array de bytes não pode conter '\0' (porquê?)
  + Um exploit robusto funcionará de forma consistente em execuções arbitrárias do código vulnerável em plataformas do mesmo tipo
* Muitas variantes/truques permitem obter essa robustez
* Exemplo para evitar ter de adivinhar endereço específico onde reside shellcode:
  + - colocar shell code no topo da stack depois do retorno
    - substituir endereço de retorno por endereço na memória de código (mais estáveis) onde esteja o opcode *jmp sp*

**Porquê tão comum?**

* + - Manipulação de strings/buffers usando bibliotecas tipo libc
    - Muitas funções são simplesmente unsafe:
    - não garantem escrita limitada região de memória pré-definida
    - strcpy, strcat, gets, scanf
    - Noutros casos isso é garantido, mas pode ser criado outro tipo de problemas: strncpy escreve em espaço pré-definido, mas não garante que string está corretamente terminada
    - Ou simplesmente código implementado de raiz com os mesmos problemas:
    - assume-se que o input vem de fonte confiável =>
    - No geral, as vulnerabilidades de gestão de memória são mesmo muito subtis

# **- Segurança de Software: Controlo (Parte 2)**

**Overflows na Heap**

* + - Quando ocorre um buffer overflow na heap a obtenção de controlo também é possível.
    - Existem muitos candidatos para alteração que permitem obter controlo:
    - apontadores para funções (e.g., virtual function tables em C++ e global offset tables para bibliotecas com dynamic linking)
    - exception handlers (e.g. lista ligada de apontadores para funções em C++)
    - *longjmp* buffers para tratamento de exceções em C

**Tratamento de Exceções**

* + - Em linguagens com tratamento de exceções (e.g., C++):
    - A stack frame de uma função inclui uma lista ligada/tabela de apontadores para as diferentes rotinas de tratamento de exceções
    - Se existe um buffer overflow na heap, pode escrever-se por cima de um desses endereços
    - Se a exceção ocorrer, a execução prosseguirá para o endereço da nossa escolha.

**malloc/free**

* + - Alocar e dealocar blocos de memória na heap (chunks)
    - Além da memória alocada, mantêm uma certa estrutura:
    - prev\_size: size of the previous chunk in memory
    - size: size of this chunk use
    - fd: forward pointer in free list
    - bk: backward pointer in free list
* Uma generalização chamada "smashing the heap" funciona da seguinte forma:
  + - para além de colocar a "payload" em memória, reescreve-se informação de gestão utilizada por malloc/free
    - as operações de malloc/free seguintes são utilizadas para alterar outras partes da memória:

o free apaga um elemento de uma lista duplamente ligada reescrevendo apontadores

manipulado para escrever numa parte da memória completamente diferente

de facto reescreve um apontador para uma função com o endereço de código malicioso

**Criação do Exploit**

* + - Domínio de programação baixo nível (assembly) e debugging de código binário
    - Compreender as causas do overflow e como o desencadear de forma controlada
    - Replicar das condições de execução do código alvo:
    - prever endereços a alterar e
    - prever localização de shellcode
    - evitar crash antes de tomada de controlo

**Use After Free**

* + - Em linguagens orientadas a objetos (C++)
    - uma instância guarda endereços de métodos
    - E se o código chama um método de uma instância destruída?
    - Talvez não haja crash se a memória permanecer estável
    - Mas se entretanto o adversário conseguir preencher essa zona de memória com endereços à sua escolha?

Alocação de um input do adversário => execução de shellcode

**Overflow de Inteiros**

* Quando a representação de inteiros do processador causa perda de informação
* Caso mais simples: truncatura por passagem para tipo mais pequeno
* A aritmética pode dar resultados inesperados por overflow

**O que faz o adversário?**

* + Se controlar um inteiro no input pode provocar um overflow de inteiros num cálculo intermédio:
  + provoca a criação de um buffer demasiado pequeno => buffer overflow
  + utiliza as técnicas descritas anteriormente
  + ou, causar um crash é um ataque valido! (DoS)
  + Caso típico:
  + tamanho a alocar: *expressao*(y)
  + y é controlado pelo atacante
  + *expressao*(y) <= 0 por overflow

**Strings de formatação**

**…**

# **- Segurança de Software: Controlo (Parte 3)**

**Controlo está mais difícil**

* + O que é necessário:
* Injetar código malicioso na memória (shellcode)
* Alterar controlo de execução para saltar para essa zona: done!
  + Sistemas modernos: Data Execution Prevention (DEP) ou W ^ X (Write xor Execute)
* Uma página de memória executável não pode ser escrita
* Uma página de memória que pode ser escrita não pode ser executável
* Algumas exceções em casos de “Just-In-Time” Compilation
  + Sistemas modernos: Address Space Layout Randomization (ASLR)
* A localização em memória de partes críticas da memória de um programa é "baralhada" em cada execução

**Reutilização de Código**

* + Pensar como um atacante:
  + se não podemos injetar o nosso código
  + temos de reutilizar código que já está em memória executável
  + bibliotecas!

**Bibliotecas!**

* + e.g. Google Chrome:
  + utiliza perto de 100 bibliotecas
  + A libc é utilizada por (quase) todos os programas
  + Contém funções úteis:
  + system: correr comando shell
  + mprotect: alterar permissões de memória
  + Cujos endereços são mais fáceis de prever

**Usar libc** **como shellcode**

* + Utilizam-se as técnicas apresentadas nas aulas anteriores
  + Mas o endereço de retorno será o de uma função da biblioteca
  + e.g., se quisermos usar a função system ou mprotect temos de configurar a stack de acordo com o que essas funções necessitam:
* int system(const char \*command);
* int mprotect(void \*addr, size\_t len, int prot);
  + É preciso perceber em detalhe como funciona a utilização da stack
  + Os exemplos que vamos ver são versões simplificadas

**FlashBack: Stack Smashing**

* A função que faz a chamada armazena parâmetros e endereço de retorno
* Quando se entra na função é armazenado o frame pointer anterior e criado espaço para variáveis locais
  + Cada função limpa o espaço que cria na stack: o processador depois utiliza o endereço de retorno automaticamente, e a função que fez a chamada limpa os parâmetros

**Como chamar system?**

* + Os dados injetados têm de:
  + substituir o endereço de retorno da função atual por &system
  + configurar a stack para uma entrada correta na função system:

o endereço de uma string com o comando shell (parâmetro)

um endereço de retorno para quando system terminar

* Para evitar um crash (porquê?): evitar deteção
  + Colocar string de shell para cima na stack longe da frame atual
  + Retornar para função que encerra programa sem erro e.g., exit(0)

**Como chamar mprotect?**

* Os dados injetados têm de:
  + substituir o endereço de retorno da função atual por &mprotect
  + configurar a stack para uma entrada correta na função mprotect:

o endereço de memória onde está o shellcode injetado, comprimento e permissões (parâmetros)

um endereço de retorno para quando mprotect terminar shellcode

* Não é fácil porque os parâmetros a passar a mprotect (endereços) geralmente contêm '\0', o que reduz as possibilidades de exploit (não funciona com strcopy, mas talvez com memcpy)

**Será que conseguimos chamar N funções em sequência?**

**Mais complexo, mas possível**

* + É necessário construir toda uma estrutura na stack que esteja preparada para a sequência de funções:
  + retornamos para a primeira função a chamar
  + parâmetros + endereço de retorno = segunda função a chamar
  + colocámos na stack os parâmetros para a primeira função, mas ...
  + A primeira função não limpa!
  + A segunda função não é responsável pela limpeza da stack
  + quando entramos nessa função a stack contém ainda "lixo" da chamada anterior

**Casos mais simples**

* + Se uma função não recebe parâmetros, a função que a chama apenas armazena na stack o endereço de retorno!
  + Se quisermos que N funções sem parâmetros sejam executadas em sequência, basta colocar os seus endereços por ordem na stack
  + A função f1 retorna para f2, que retorna para f3, etc
  + Depois de uma sequência de qualquer tamanho de funções que não recebem parâmetros podemos chamar uma função que recebe parâmetros e terminar chamando uma função que não recebe parâmetros

Chart

Description automatically generated**Casos mais complexos**

* Existem outras combinações que funcionam neste estilo
* Mas e se quisermos chamar uma sequência arbitrária?
* f1(t); f2(u,v); f3(w,x,y); …
* É necessário "criar" versões alternativas destas funções que limpam os seus próprios parâmetros da stack quando retornam. Como?
* colocamos um outro endereço na stack entre as duas funções f1 e f2
* esse endereço é de um pedaço de código arbitrário que:

retira da stack o número de bytes necessário (por pop ou alteração direta do stack pointer)

retorna para a função seguinte => f2!

* Esta técnica pode utilizar-se para encadear um número arbitrário de chamadas

**Return Oriented Programming**

* + Considerar o assembly de código em memória (programa ou biblioteca) como uma "sopa de letras"
  + Identificar sequências de código úteis (geralmente terminando em ret, mais recentemente também sequências terminando em jmp ou call)
  + Colecionar essas sequências como "gadgets" reutilizáveis
  + "Colar" os gadgets como uma sopa de letras

**Que operações se podem fazer?**

* + Depende do código a que temos acesso
  + Operações aritméticas/lógicas e tudo o que seja "straight-line" é mais simples (nomeadamente shellcode):
  + composição sequencial de pequenos blocos de código
  + pesquisar de forma automática no código alvo um conjunto de gadjets
  + quando compostos devem ser equivalentes ao código que pretendemos
  + Conceptualmente Turing-complete!

**Wrap-up de Buffer Overflows**

**O que vimos**

* + Se for possível a um atacante influenciar um input:
  + é essencial garantir que todos os acessos a memória são corretos
  + Se um atacante conseguir escrever fora da zona de memória reservada para essa finalidade:
  + pode alterar o controlo de fluxo do programa
  + utilizar o programa contra a própria plataforma onde está a correr
  + E se o atacante conseguir só ler fora da zona correta?

**Como são descobertas as vulnerabiblidades?**

* Geralmente um erro na gestão de memória
* O primeiro sinal é um crash do programa
* Para procurar por vulnerabilidades:
* utilizar um "fuzzer" para fornecer inputs cegamente a um programa
* se o programa crashar, investigar porquê
  + Se o overflow se verificar, é geralmente reconhecido imediatamente como uma potencial vulnerabilidade
  + A construção de um exploit geralmente exige muito mais trabalho (como vimos)

**Ethical hacking**

* É crucial haver quem procure estas vulnerabilidades para termos sistemas mais seguros
* É importante ter muito cuidado com a forma como se divulgam essas vulnerabilidades
* Os sistemas de bug bounty e vulnerability reporting são desenhados para:
* evitar que "zero-day vulnerabilities" detectadas por hackers bem intencionados acabem por causar danos

# **- Segurança de Software: Controlo (Defesas)**

**Relembrar os ataques**

* **Stack smashing:** reescrever o endereço de retorno na stack para código malicioso
* **Heap spraying:** evitar ter de prever o endereço exato do código malicioso, criando muitas cópias em memória
* **Use after free:** aproveitar que programa está a utilizar memória que está já fora do seu controlo para modificar essa memória antes da utilização incorreta
* **Format string vulnerabilities:** uso de strings de formatação criadas pelo adversário
* **Integer overflows:** erros em cálculos que levam, por exemplo, a alocação de memória insuficiente para a utilização
* ...

**O que há de comum**

* Os dados, potencialmente provenientes de fora do círculo de confiança, podem interferir com o controlo do programa
* Por exemplo: buffer de dados armazenado na stack ao lado do endereço de retorno
* Problema recorrente (mais exemplos em Segurança Web)
* Mesmo com princípios corretos, os erros de implementação podem deitar tudo a perder

**Defesa em profundidade**

* Diagram

  Description automatically generatedMedidas de proteção na **Linguagem de Programação**:
* Usar linguagens memory safe (Java, Haskell, Go, Rust, etc.)
* Verificação de programas (mais esforço, mas a única hipótese para código legacy)
* Medidas de proteção na **Aplicação**:
* detetar tentativa de highjacking; monitorização do estado da stack, tagging de memória, etc.
* Medidas de proteção no **Sistema Operativo**:
* impedir a execução de código malicioso
* Medidas de proteção no **Hardware**
* instruction set, trusted execution environments, secure boot, etc.

**Mitigações no Sistema Operativo**

**Data Execution Prevention**

* + Também chamada Executable Space Protection ou W ^ X, referida anteriormente:
* motivação para Return Oriented Programming
  + A memória nunca pode ser simultaneamente:
  + passível de ser escrita por um programa
  + passível de conter código executável
  + Pode ser implementada em hardware (NX bit) ou emulada em SW
* As arquiteturas mais populares oferecem suporte em hardware:
* AMD, Intel, ARM: permitem desativar execução ao nível da página
* Implementado pelos sistemas operativos mais utilizados desde meados dos anos 200X
* Limitações:
* só por si não evita exploits que reutilizem código, tal como ROP
* cria problemas em cenários JIT (e.g., nos browsers para correr código do lado do cliente)

**Address Space Layout Randomization**

* + A localização de código e dados em memória é gerada de forma aleatória em cada execução:
* stack, heap, bibliotecas partilhadas, código de base
  + Vantagens: atacante não consegue prever os endereços de código útil (stack, ROP)
  + Utilizações:
* sistemas operativos mais populares desde meados dos anos 200X
* cada vez maior entropia para tornar previsão mais difícil (primeiro 8 bits, depois 24 bits)
  + O que pode fazer um atacante:
* outras vulnerabilidades fornecem informação sobre endereços (e.g. format strings)
* tentar muitas localizações (heap spraying) sem crashar o programa
* utilizar código do sistema operativo (kernel)?
  + Proteção também aplicada ao endereço do Kernel (KASLR):
* problema: apenas muda de boot para boot
* problema: sabendo a base do kernel, podem prever-se outros endereços
* solução: Kernel Address Randomized Link (KARL) o próprio código do kernel é “baralhado"

**Mitigações na Aplicação**

**Deteção em Tempo de Execução**

* + Instrumentação de código, um segundo nível de proteção:
* o compilador adiciona código que permite detetar um potencial ataque
* transforma exploits que permitem execução arbitrária de código em ataques DoS porque a execução do programa termina
  + De uma forma mais geral pode falar-se de "run-time monitoring":
* uma técnica de verificação que executa código adicional para "vigiar" código mais complexo que está a desempenhar a funcionalidade pretendida

**Stack Canaries**

* + Objetivo:
* prevenir injeção de código detetando modificações à stack
* Ideia:
* introduzir "**canários**" gerados dinamicamente entre variáveis locais e os valores do frame pointer e do endereço de retorno guardados na stack
* verificar canário antes de utilizar o endereço de retorno
* Implementação: compilador modifica entrada e saída das funções

**Variantes**

* + Canário aleatório (exige “boa” aleatoriedade)
* No início da execução o programa escolhe um array de bytes aleatório
* Esses bytes são colocados em todas as stack frames
* Antes de retornar de uma função verifica-se a sua integridade
* Se houver alguma alteração o programa termina
  + Canário de terminação (usam '\0','\n',EOF em vez de bytes aleatórios)
* As funções que manipulam strings irão sempre parar nestes valores

**StackGuard**

* + Implementado pelo GCC (-fstack-protector-strong), muitos outros nomes, e.g., MS BufferSecurityCheck
  + Vantagens:
* simples e fácil de colocar em prática
  + Desvantagens:
* overhead de espaço (1 endereço por function call) e performance (~10%)
  + Como fazer bypass aos canários?

**Derrotando os canários**

* + Heap e Integer overflows não são afetados
  + Ler o canário com outra vulnerabilidade + overflow da stack e repor o valor do canário
  + Aproveitar apontadores:
* se o código tem um apontador na stack e escreve para esse endereço outra variável local
* um buffer overflow pode alterar ambas e escrever num ponto arbitrário em memória
* Apontadores para funções:
  + se um programa recebe um apontador para função como argumento
  + alterar esse apontador pode permitir saltar para endereço arbitrário antes de a função retornar
* Se o canário for reutilizado muitas vezes, e.g., em cada fork, pode fazer-se um brute-force

É “fácil” otimizar para um byte de cada vez: como?

**Mitigações adicionais na stack**

* + Garantir que os buffers estão sempre junto ao canário
  + Copiar sempre os argumentos da função para o topo da stack (abaixo dos buffers locais)
  + O GCC também implementa estas proteções adicionais
  + Shadow stack (suportado, e.g., Windows 10 2020):
  + stack redundante apenas para controlo (endereços frame e retorno)
  + antes de retornar da função verifica-se consistência
  + e.g., Intel Control Flow Enforcement Technology permite colocar shadow stack em páginas marcadas para escrita apenas por call/ret
* Outra possibilidade é ter os dados "perigosos" numa stack separada do control flow

**Memory Tagging**

* ARM Memory Tagging Extension:
  + suporte de hardware para criar tags (novas instruções)
  + permitem "ligar" apontadores às regiões para onde apontam
  + uma zona de memória alocada pode ter associada uma tag de 4 bits
  + o apontador que aponta para essa memória terá a mesma tag
  + compara-se tags no acesso: overflow => exceção
  + free altera tags: use after free => exceção
* Apresentado como suporte ao debugging e proteção extra contra overflows (adivinhar a tag)

**Control Flow Integrity**

* Até agora olhámos para mitigações que se focam na memória e tentam impedir/detetar a sua alteração
* CFI foca-se na sequência de controlo e chamadas a funções:
* em tempo de compilação: determinar um conjunto válido de pontos de origem para cada destino válido
* em tempo de execução: verificar consistência com essa informação
* Não é necessário proteger saltos diretos/chamadas estáticas que estão codificadas nas instruções => o atacante não consegue alterar o código
* É necessário proteger todos os saltos/retornos cujos endereços são criados dinamicamente pelo programa, nomeadamente chamadas indiretas (apontadores para funções)
* Versão mais básica:
* considerar apenas pontos de destino válidos as entradas de funções + confirmar que sempre que chegamos à entrada de uma função é resultado de um call (*à la* kBouncer)
* problema: não impede a chamada de funções numa sequência diferente, por exemplo, saltando por cima de um sistema de autenticação.
  + Versões elaboradas implicam computar um Control Flow Graph (CFG) que define todos os saltos válidos => difícil calcular um grafo preciso estaticamente
  + Usando criptografia (CCFI):
* sempre que um endereço é escrito algures na memória, guardar também um MAC criptográfico + sempre que se usa o endereço verifica-se o MAC + chave que não está em memória (um registo por exemplo)
* soluções e ataques têm sido muito discutidas no meio académico; implicam um overhead significativo; suporte de hardware é vita

**É possível mesmo assim?**

* Apesar de todas estas mitigações os ataques continuam a aparecer
* São cada vez mais elaborados:
  + Trident exploit: permite remotamente desbloquear um iOS e instalar spyware
  + História interessante: detetado no mundo real e associado a empresa denominada NSO Group (ver 60 minutes)
  + Encadeia uma série de exploits de vulnerabilidades no browser e no kernel
* Conclusão: é essencial garantir que não existem "buracos" no tratamento de inputs potencialmente hostis

**Mitigações na Linguagem de Programação**

**Program Safety**

* Conceito importante na área das linguagens de programação:
* informalmente significa que a execução do programa permanece dentro do âmbito especificado pela semântica da linguagem (e.g., C Standard)
* Exemplo: a linguagem descreve o que deve acontecer quando escrevemos e lemos de uma região válida de memória, mas é omissa nos outros casos

um programa que utiliza uma zona inválida de memória éunsafe

* Exemplo: a semântica da linguagem indica que os cálculos sobre inteiros de precisão limitada apenas são consistentes com dentro de uma determinada gama

um programa que assume consistência com fora dessa gama é unsafe

**Memory Safety ⊂ Program Safety**

* Nas operações de memória existem diversos tipos de operações unsafe:
* em geral todas elas aparecem como tendo resultados "undefined" na semântica de linguagens como o C

Read: ler de zona inválida

Write: escrever em zona inválida

Execute: interpretar como instruções executáveis dados inválidos

* + Safety espacial: validade depende apenas da localização
  + Safety temporal: validade depende do tempo de vida do objeto/variável

**Garantir Program Safety**

* Algumas linguagens de programação oferecem mais automação do que outras para garantir safety:
* Linguagens strongly typed como Java ou Haskell verificam um conjunto muito alargado de condições em tempo de compilação:

os tipos de dados são compatíveis com as operações que sobre eles são realizadas (arrays)

* + Outras linguagens como Python são weakly typed, mas verificam type safety na execução
  + Linguagens interpretadas como Java ou Python detectam problemas de safety na execução

acessos errados a memória/containers originam exceções => controlo passa a DoS

* O Rust inclui um conjunto de checks estáticos e dinâmicos para garantir safety, etc.

**Verificação de Programas**

* Mesmo em linguagens como o C ou assembly é possível ter garantias de safety:
  + existem ferramentas que nos permitem verificar essas propriedades
  + algumas são dinâmicas/incompletas, mas automáticas: valgrind, fuzzing, symbolic execution
  + outras são completas e baseadas em matemática/lógica:

os programas são anotados (comentados) com meta-informação que descreve as propriedades funcionais (e.g., o que é tocado em memória)

é necessário construir uma prova matemática de que se verificam (Frama-C, etc.)

aqui pode haver diferentes níveis de automação no auxílio da construção de uma prova

**Mitigações no Hardware**

**Proteções em Hardware**

* + Muitas das técnicas que vimos têm algum suporte de hardware
  + Trusted Platform Module (um cryptoprocessador seguro)
  + Trusted Execution Environments (Intel SGX, ARM TrustZone)
  + UEFI Secure Boot

**Porque existem vulnerabilidades?**

* Os programadores fazem erros
* é preciso utilizar ferramentas para os ajudar a evitar erros
  + Os programadores muitas vezes não têm noção das implicações de um erro => formação! (esta UC!)
  + As linguagens de programação não são desenhadas com o objetivo de garantir segurança:
  + ainda assim, umas linguagens são melhores que outras, e oferecem validação estática de algumas propriedades de safety e run-time checks de outras
  + Java, Haskell, Python, Rust, etc.

**Defesa em profundidade**

* Existem mecanismos de mitigação a diversos níveis
  + processador, compilador, linguagem de programação, ...
  + verificação automática de bounds (e.g., Rust, Java, Python)
  + randomização de endereços, W^X, etc.
  + Control-Flow Integrity
  + Trusted Execution Environments, etc.
* Mas podem falhar todos e não temos fallback
  + Pior: continuamos sujeitos a ataques por DoS (exceções, crash, etc.)
  + Como se recupera de falhas? Repomos o estado? => Pode permitir brute-force

# **- Segurança de Sistemas (Parte 1)**

**Economia nos mecanismos**

* Keep it simple:
  + um sistema deve ter apenas as funcionalidades (e.g., serviços a correr) necessários; evitar funcionalidades “nice to have”
  + de todos os mecanismos de segurança equivalentes, devem ser adotados os mais simples
* Facilita a implementação, usabilidade, validação, etc.

**Proteção por omissão**

* A configuração de qualquer sistema deve, por omissão (default) impor um nível de proteção conservador
* E.g., um novo utilizador deve, por omissão, ter o mínimo de permissões
* "Fail closed": por oposição a reverter para uma posição permissiva

**Desenho aberto**

* A arquitetura de segurança e os detalhes de funcionamento de um sistema devem ser públicos:
* não deve basear-se segurança em "security through obscurity"
* Os segredos são parâmetros do sistema, que podem ser alterados:
* chaves criptográficas, passwords, etc.
* Racional:
* permite o escrutínio => mais provável sabermos que estamos vulneráveis
* podemos mudar os segredos sem mudar o sistema

**Defesa em Profundidade**

* Já vimos antes: todos os mecanismos de segurança podem falhar
* Não podemos depositar toda a nossa confiança num só mecanismo:
  + e.g., blindar perímetro e assumir que não existem ameaças internas
    - Inerente a qualquer sistema que faça qualquer coisa de útil para o exterior!

**Privilégio Mínimo**

* Cada utilizador, compartimento, programa, etc.
* deve ter apenas os privilégios/permissões essenciais para desempenhar a sua função
* racional: need-to-know

contrariar este princípio amplifica sem necessidade o impacto de uma brecha local na segurança

* + exemplo de violação: correr todos os serviços como root

**Separação de Privilégios**

* As funcionalidades/utilização de recursos devem ser compartimentadas:
  + cada compartimento deve estar isolado dos outros
  + todos os compartimentos devem encarar os restantes como estando num outro domínio de confiança
* Essencial combinar com defesa em profundidade e privilégio mínimo:
  + comprometer um compartimento tem consequências localizadas
  + um compartimento tem apenas acesso aos recursos de que necessita

**Mediação Completa**

* Um sistema gere, invariavelmente, recursos: ficheiros, dispositivos de hardware, memória, etc.
* Mediação completa implica:
  + Para todos os recursos definir uma política de proteção
  + Validar todos os acessos a recursos relativamente a essa política
  + Exemplo que já vimos => memória virtual
  + Todos os processos acedem a um espaço de memória virtual
  + Todos os acessos a memória são mediados pelo sistema operativo
  + Veremos diversas formas de controlo de acessos = mediação

**Controlo de Acessos**

* + Controlo de acessos refere-se a uma família de mecanismos de segurança que visam aplicar alguns dos princípios anteriores:
  + **Privilégio mínimo:** atribuir os privilégios estritamente necessários às entidades que interagem com/no âmbito do sistema
  + **Mediação completa:** garantir que todos os acessos a um recurso são efetuados por entidades com privilégios adequados
  + **Separação de privilégios:** relacionado com os outros dois
  + Quando tratamos de controlo de acessos falamos de
  + **Ator** (entidade que realiza uma ação)
  + **Recurso** (sobre o qual se realiza a ação)
  + **Operação** (o tipo de ação que é realizada)
  + A combinação de recurso/operação chama-se uma permissão
  + UNIX: Um **Processo** não deve poder **ler** **memória** de outros processos
  + Mobile: Uma **App** só deve poder **editar** os seus próprios **dados**
  + Web: Um **Domínio** não deve poder **ler** os **cookies de outros domínios**

**Matriz de acessos**

* + Descreve todos os possíveis acessos:
  + **Ator**, **Recurso**, **Operação**
  + E.g., ler, escrever, executar
  + Vantagem: clareza, eficiência
  + Problema: tamanho
  + As soluções seguintes são alternativas a esta solução, que permitem escalar para sistemas de grande dimensão

**Listas de controlo de acessos (ACL)**

* + Calendar

    Description automatically generatedPara cada **recurso**, as **permissões** dos **atores** sobre esses recursos (omissão = ausência de permissão, intuição = lista de convidados)
  + Vantagens:
  + faz-nos pensar sobre como proteger cada recurso individualmente => armazenamento associado ao próprio recurso
  + permite garantir facilmente que um recurso apenas pode ser acedido por um número limitado de atores (isolamento)
* Desvantagens:
  + Não é possível determinar as permissões que um ator tem sem ver todos os recursos (e.g., para remover um ator)
  + Não há agregação de perfis de permissões

**Listas de permissões (Capabilities)**

* Calendar

  Description automatically generatedPara cada **ator**, as **operações** que pode realizar sobre cada **recurso** (omissão = ausência de permissão, intuição = passe de transportes).
* Vantagens:
  + faz-nos pensar sobre como lidar com cada ator individualmente => armazenamento associado ao ator
  + permite garantir facilmente que um ator apenas acede a um número limitado de recursos
  + Desvantagens:
  + Não é possível determinar todos os atores que podem aceder a um recurso sem percorrer todos os atores
  + Não há agregação de perfis de permissões

**Role Based Access Control**

* + Modelo que permite separar a gestão de **recursos** da gestão de **atores**:
  + Ligação de **perfis** (roles) a **conjuntos de permissões** (agregação, menos espaço)
  + Ligação de atores a perfis
  + As relações entre **perfis e permissões** são geralmente muito estáveis:
  + são administrados por quem gere **recursos** => semelhante a ACL
  + Os **perfis** dos **atores** podem ser mais dinâmicos:
  + são administrados por quem gere atores (e.g., utilizadores) => semelhante a listas de permissões

**Attribute-based Access Control**

* + O RBAC é um caso particular de ABAC:
  + **atores** e **recursos** têm **atributos** associados
  + matriz de acessos descreve **permissões com base nos atributos**:

para aceder a recurso com atributo A o ator deve possuir atributo B

* + permite políticas mais expressivas, como por exemplo ter em conta o contexto geográfico e temporal, ou sistemas hierárquicos

**Sistemas Operativos - Aplicação dos Princípios Anteriores**

**Sistema Operativo**

* + Interface entre os utilizadores de um computador e o hardware:
    - gestão do acesso a **recursos** por parte de **aplicações**

armazenamento, processador, memória, I/O, rede, etc.

partilha destes recursos por vários **utilizadores** e aplicações.

software que trata operações de baixo nível e oferecem **abstrações** convenientes para o desenvolvimento de aplicações

* + - muito complexos e gerem aspetos críticos para a segurança

**Aspetos de segurança**

* + Múltiplos utilizadores com diferentes níveis de acesso:
    - administradores, utilizadores frequentes, utilizadores esporádicos, etc.
    - diferentes necessidades e privilégios relativamente a recursos a utilizar
    - o SO tem de garantir que estes requisitos são cumpridos e, ao mesmo tempo, garantir que não existem comportamentos abusivos
    - **os utilizadores são sempre potenciais ameaças**, que modelo de ameaças?
    - **os recursos são sempre ativos a proteger**, que propriedades?
  + Múltiplas aplicações/serviços a executar em simultâneo:
    - as **aplicações são também potenciais ameaças** (aulas sobre controlo)
    - devem estar protegidas de interferência de outras => **isolamento**
    - esta proteção aplica-se mesmo quando não estão a executar:

a informação que armazenam como estado está geralmente em recursos partilhados (e.g., disco)

* + Os sistemas operativos modernos visam garantir:
    - isolamento virtual entre utilizadores, aplicações e processos
    - apesar de estarem a partilhar os recursos disponíveis no sistema
    - através de mecanismos de acesso mediado
    - Administrar um sistema operativo => configurar estes mecanismos
* devem aplicar-se os princípios fundamentais: privilégio mínimo, separação de privilégios, etc. => boas práticas

# **- Segurança de Sistemas (Parte 2)**

**Kernel**

* + - **Diagram

      Description automatically generated**O **Kernel** é a parte do sistema operativo que desempenha as operações mais críticas (mediação completa):
* o processador está em **kernel mode** e todas as operações são permitidas a esse código
* os processadores permitem geralmente definir vários níveis de privilégio (ring em Intel)
* em muitos casos usam-se apenas dois: **kernel mode** e **user mode**
* o código em **user mode** não tem acesso direto aos recursos do sistema
* qualquer troca de informação entre os dois níveis faz parte de uma superfície de ataque
  + - O **Kernel** está protegido dos processos em modo utilizador:
    - Diagram

      Description automatically generated with medium confidencetem um espaço de memória gerido de forma independente (pelo próprio kernel)
    - o processador garante que apenas código que corre em **kernel mode** pode executar um conjunto de instruções privilegiadas (separação de privilégios)
    - qualquer processo em **user mode** (incluindo device drivers) tem de aceder aos recursos do sistema usando **system calls** (privilégio mínimo)
    - parte do código das **system calls** executa em **kernel mode**!

**Confinamento (veremos mais à frente)**

* + - Os **pontos de entrada em system calls são críticos**:
* para causar danos críticos, um processo em modo utilizador tem de o fazer através de uma system call! (relembrar kBouncer)
* Isto implica implementar sistemas de monitorização e controlo de chamadas ao sistema
* **Reference monitor** (mediação completa + registo de compromissos):

sempre presente (se terminar, têm de ser terminados os processos monitorizados)

tem de ser simples para poder ser analisado e validado mais facilmente que o sistema todo

**System Calls**

* + - * Controlo de processos (e.g., fork, load, execute, wait, alloc, free)
      * Acesso a ficheiros (criar, ler, escrever, etc.)
      * Gestão de dispositivos (obter acesso, escrever/ler, etc.)
      * Configuração do sistema (hora, data, características, estado, etc.)
      * Comunicações (estabelecer ligação, enviar/receber mensagens, etc.)
      * Proteção (alterar/obter permissões de acesso a recursos)
      * Para entrar num modo de funcionamento com mais privilégios, o código **user mode** deve:
      * preparar argumentos, e identificar um ponto permitido para acesso a **kernel mode**
      * executar uma **instrução especial** que passa o controlo para o kernel
      * **Existe um número limitado deste tipo de pontos de entrada** (economia de mecanismos):
      * registos específicos para parâmetros, que são tipicamente apontadores para memória de processos em user mode
      * o processamento dessa informação é da total responsabilidade do kernel
      * Um **número limitado de pontos de entrada** => **superfície de ataque bem definida**

**Processos**

* O **Kernel define a noção de processo**: uma instância de um programa que está a executar
* Os programas começam por estar guardados em armazenamento não volátil (e.g., código no disco)
* Para serem executados têm de ser carregados para memória e receber um identificador como processo
* Cada processo deve executar num contexto em que tem acesso a um conjunto de **recursos**, que devem estar disponíveis independentemente de outros processos
* A fronteira entre **processos** é uma fronteira de confiança: os processos têm de estar confinados/isolados entre si (separação de privilégio)
* O **Kernel** garante acesso a **recursos**:
  + atribui uma fração razoável de tempo de processador a cada **processo** (time slicing)
  + atribui um espaço da memória sobre o qual o processo pode trabalhar
  + concede acesso a outros recursos através de system calls
  + Nos SO multi-utilizador, existe um conjunto de processos base que

interagem com o utilizador humano:

* + quando o utilizador lança uma aplicação, o SO vê um dos processos que interagem com o utilizador (e.g., shell, GUI, etc.) a criar um novo processo => forking
  + Os SO gerem uma **hierarquia de processos**, em que tipicamente os descendentes herdam os privilégios dos seus criadores
  + Em Linux pode ver-se a árvore com *pstree*
  + O processo raiz é o *init* (PID 1)
  + PID é um identificador único de processo
  + As permissões atribuídas a cada processo dependem do utilizador que o cria
  + cada utilizador tem um UID (único para o utilizador) e um GID (único para o grupo)
  + tipicamente o 0 é reservado para o super-user (root)
  + o processo é associado aos mesmos UID, GID: sintaxe *processo* (*GID*)
  + Muitas vezes é necessário **comunicar entre processos** (IPC) **system calls**

• através do sistema de ficheiros

• memória partilhada

• mensagens síncronas: pipes, sockets

• mensagens assíncronas: signals

* + **Daemons, services:**
  + processos que não são visíveis pelo utilizador (diretamente)
  + E.g., indexação, login remoto, impressoras, sincronização de ficheiros, etc.
  + geralmente arrancados antes dos próprios processos que interagem com o utilizador
  + executam tipicamente com privilégios superiores aos dos utilizadores e sobrevivem às suas sessões

**Modelo de Confiança**

* A confiança depositada nos processos lançados é **indutiva**:
  + o código armazenado no computador (nomeadamente a BIOS e o kernel) após uma instalação é **“confiável"** (proteção por omissão)
  + o processo de boot utiliza este código para colocar o kernel em memória e passar-lhe o controlo, **preservando o estado “confiável”** em memória
  + o kernel lança processos com permissões que garantem que **nenhum novo processo pode alterar o estado de confiança**
  + os processos de hibernação preservam o estado de confiança
  + os administradores podem alterar o software instalado no sistema e o sistema de permissões, mas garantem que qualquer atualização **preserva o estado de confiança**
* O que significa "confiável":
  + que o sistema faz exatamente (e apenas) aquilo que foi especificado
  + exemplo: não transmite a nossa informação sensível para o exterior sem autorização
  + exemplo: garante que as nossas comunicações são estabelecidas com as entidades com quem queremos comunicar (e.g., servidores Google)
  + exemplo: cifra toda a informação em disco e limpa a memória quando fazemos shutdown

**Modelo de Ameaças**

* Ataques em todos os níveis do boot:
  + BIOS corrompida
  + ficheiros de hibernação corrompidos
  + bootloader corrompido
  + cold boot attacks (side-channel)
* Vulnerabilidades que afetam a implementação dos mecanismos de arranque **fazem um bypass** completo ao modelo de confiança (o arranque é a âncora)

**Medidas de Mitigação**

* + A maioria dos problemas de segurança surgem através de erros de administração
  + Veremos exemplos de processos maliciosos (malware) e das formas que utilizam para corromper o modelo de confiança, bem como medidas de mitigação
  + A **monitorização** é uma forma de mitigação comum para detetar quebras neste modelo (registo de compromissos):
  + logs de eventos permitem detetar comportamentos suspeitos, como o crash repetido de um processo que está a tentar explorar uma vulnerabilidade
  + aplicações de monitorização de processos permitem visualizar os processos que estão a executar, os recursos que utilizam, e os ficheiros de código que os originam
  + a **mediação** de instalação/execução de código com base em assinaturas digitais fornece também um entrave à introdução de código malicioso num sistema (mediação completa + proteção por omissão)
  + Para já vamos estudar os mecanismos de segurança que são utilizados nos sistemas operativos para garantir isolamento entre processos
  + impedem que um utilizador com poucos privilégios (e portanto fora do círculo de confiança) possa utilizar o sistema para além do que lhe é permitido
  + impedem que um processo que contenha uma vulnerabilidade não abra uma porta que corrompa todo o sistema (defesa em profundidade)
  + vamos focar-nos na **gestão da memória, processos e sistema de ficheiros**, que são aspetos fundamentais comuns a todos os SO

**Memória**

* + A regra fundamental da gestão de memória diz que:
  + **um processo não pode aceder ao espaço de memória de outro processo**
  + a confidencialidade, integridade e **controlo de fluxo do kernel tem de ser protegida** de todos os processos que executam em modo utilizador
  + Como se garante?
  + em run-time os acessos são mediados por um conjunto de mecanismos de hardware e software geridos pelo kernel (e.g., partição swap, memory protection keys)
  + as partes da memória virtual que estão em disco podem ser alvo de ataque off-line se um adversário puder aceder a essa informação => disco cifrado
  + O espaço de memória gerido por um SO é muito maior do que o espaço físico:
  + o processador dá suporte a mecanismos de memória virtual
  + o espaço de endereçamento de um processo está dividido em páginas
  + algumas estão em memória outras em armazenamento não volátil
  + quando é necessário trocar, diz-se que ocorreu um "page fault"

**Tradução de Endereços**

* + A tradução de endereços necessária à implementação de mecanismos de memória virtual cumpre dois propósitos:
  + **isolamento:** cada processo acede a uma zona de memória que não existe na realidade, e que dá visão/acesso limitados aos recursos (mediação completa)
  + **eficiência:** esconde mecanismos de otimização (caching, speculative access, paging, etc.)
  + Nos últimos anos ficou claro que algumas otimizações criam, de facto, novos pontos de ataque (side-channel como Spectre e Meltdown)
  + A memória virtual está dividida em páginas, e.g., 4KB
* O sistema tem de armazenar, para cada página (se utilizada) a sua localização física
* Page-table: árvore esparsa com informação nas folhas
* Processador oferece suporte para gerir estas estruturas
  + Aceder a uma page table (que está em memória física) é penalizador
  + Translation Lookaside Buffer (TLB):
  + cache de páginas traduzidas recentemente
  + informação para controlo de acessos em cada página (ACL)
  + Read/Write/eXecute (NX bit)
* Como lidar com system calls?
  + uma gestão totalmente independente dos espaços de endereçamento tornaria as mudanças de contexto muito ineficientes
* Kernel mapping:
  + parte da memória virtual do kernel está mapeada diretamente na memória virtual de cada processo, mas com permissões diferentes

**User mode (UR,UW,UX)**

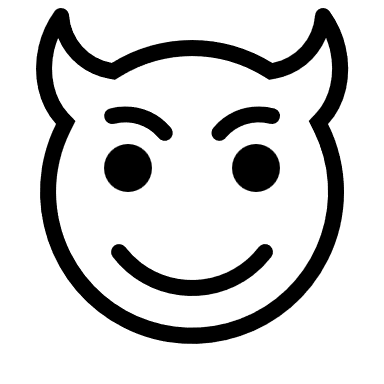
**Kernel/privileged mode (PR,PW,PX)**

**Memória Virtual**

**Kernel Mapping**

* Quando um processo faz uma system call não é necessário alterar o sistema de mapeamento de páginas:
  + a memória relevante para o kernel já está mapeada
  + mais importante: a parte da memória do processo relevante à system call coexiste no mesmo espaço de endereçamento
* Quando se muda de processo de utilizador, as tabelas de páginas são alteradas, mas as que dizem respeito à memória do kernel são as mesmas (relembrar Pegasus)

**Defesa em profundidade**

* Que permissões deve ter o kernel sobre a memória dos outros processos:
* todas as permissões => !
* Defesa em profundidade:
  + nem o kernel deve ser capaz de violar a regra W^X
  + impedir o kernel de escrever em partes da memória do utilizador é uma forma de impedir fugas de informação/código malicioso no caso de kernel corrompido

**Chart, timeline

Description automatically generated with medium confidence**

# **- Segurança de Sistemas (Parte 3)**

**Sistema de Ficheiros**

* Veremos como exemplo os sistemas \*n?x:
  + **atores:** utilizadores, processos
  + **recursos**: ficheiros e pastas
  + **ações**/acessos:

r/w/x para ficheiros: evidente qual o significado

r/w/x para pastas: listar conteúdo, criar conteúdo adicional, "entrar" na pasta

alterar as permissões?

* Diagram, schematic

  Description automatically generatedCada utilizador pertence a um grupo: permite uma forma de RBAC
* Cada recurso tem um owner e um grupo
  + as permissões são atribuídas de forma independente a

owner (ACL com uma única entrada)

membros do grupo associado ao recurso (RBAC rígido)

todos os outros utilizadores (RBAC rígido)

* + Superuser:
  + antigamente um utilizador especial (*root*)
  + hoje em dia um papel/role: *sudo*
  + *uid* = 0 utilizado para identificar esse utilizador/papel
  + boas práticas: utilização mínima (privilégio mínimo)
  + Alteração de permissões:
  + sempre permitido ao **superuser** / administrador
  + permissões podem ser alteradas pelo **owner** (*chmod*)
  + **owner** pode ser alterado pelo **superuser** (*chown*)
  + **grupo** poder ser alterado por **owner** e **superuser** (*chgrp*)
  + **Discretionary Access Control**: o **owner** pode alterar permissões
  + **Mandatory Access Control**: apenas o **superuser** pode
  + e.g., SELinux (Security-Enhanced Linux)
* Permissões de **processos**:
  + os **utilizadores** interagem com o sistema através de **processos**
  + cada **processo** tem associado um **effective user id**

determina as permissões do processo

em geral: **uid do utilizador** que lançou o processo

**existem exceções**: e.g., *sudo* ou mudar a password usando *passwd*

* + Como funciona o login?
  + o sistema executa um processo login como root/super user
  + esse processo **autentica o utilizador** (tem acesso às credenciais no sistema)
  + **altera o seu próprio uid e gid para os associados ao utilizador**
  + lança o processo de *Shell*
  + **Crítico: o login executa *drop privileges***
  + Se incorretamente implementado, continua a ser possível ***capability leaking***
  + O reverso (***elevate privileges***) deve ser impossível (e o *passwd*?)
  + O **bit *setuid*** associado a um ficheiro:
  + Permite fixar o utilizador associado um processo ao owner do executável (e não ao utilizador que executa)
  + Pode ser ativado pelo superuser e pelo owner do ficheiro
  + Implicações:

se o owner tiver muitos privilégios **=> permite elevação de privilégios!**

* + No caso do passwd o owner é o utilizador *root*.
  + **Tudo é um ficheiro** (economia de mecanismos):
  + como minimizar o número de system calls/superfície de ataque?
  + utilizar a mesma interface construída para o sistema de ficheiros para outros recursos
  + Em \*n?x: sockets, pipes, dispositivos de I/O, objetos do kernel, etc.
  + **O sistema de controlo de acessos é sempre o mesmo!**

**Privilégios de Processos**

* **Quando executamos um processo, tipicamente executa com o UID do utilizador que o lançou**
  + pode aceder aos mesmos recursos
  + Alguns processos são executados com o UID do owner do ficheiro executável (bit setuid = 1)
  + **Os processos do kernel arrancam com UID = 0 (root/super user)**
  + acesso a todos os recursos => privilégio máximo!
* A transição de privilégios é mais complexa do que parece à partida
* Um processo tem, de facto, três UIDs:
* **Effective User ID (EUID):** determina as permissões
* **Real User ID (RUID):** utilizador que lançou o processo
* **Saved User ID (SUID):** utilizado em transições, lembra o anterior
* O que é possível fazer em tempo de execução?
* O utilizador root pode usar a system call setuid(x) para alterar estes valores para UIDs arbitrários:
* **EUID** => x, **RUID** => x, **SUID** => x
* Utilizadores comuns apenas podem mudar **EUID** para **RUID** (eles próprios) ou **SUID** (andar para trás)
* Isto permite a um processo reduzir os próprios privilégios:
* quando o Apache (corre como root para usar porta 80) cria um processo para atender um utilizador reduz os privilégios do processo descendente
* É possível fazer uma redução temporária de privilégios:
* A system call *seteuid* (x) altera apenas o **EUID** e preserva o **RUID** e o **SUID** (e.g., daemon precisa de usar **RUID** para criar um recurso)
* Utilização típica: baixar privilégios => executar código => restaurar privilégios
* Perigo: usar seteuid quando root pretende baixar privilégios permanentemente (porquê?) **=> é possível voltar para RUID usando setuid!**
* **Complexidade:**
* mesmo com um sistema tão simples
* existe um sistema de transições entre estados de confiança
* onde é muito fácil cometer erros
* **Importante:** registo de compromissos

**Conclusão**

* O sistema de controlo de acessos em \*n?x é essencialmente uma implementação de Access Control Lists, com algum batching
* Vantagem => simples e funciona na prática
* Desvantagem => pouco robusto e pouco flexível
* uma falha num processo tipo *passwd* ou ssh (*euid* = 0) tem consequências catastróficas
* *root* utilizado para muita coisa => erros de administração
* **Um atacante ganhando root, não é possível tirar-lhe privilégios**