

# Fundamentos de Segurança Informática (FSI)

2021/2022 - LEIC

**Manuel Barbosa**  
**mbb@fc.up.pt**

# **Aula 6**

## **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
- A falta de separação entre dados e controlo é um problema recorrente:
  - mais à frente vamos ver isto novamente em XSS na segurança Web
- Mesmo quando os princípios de gestão de memória são corretos, os erros de implementação podem deitar tudo a perder

# Defesa em profundidade

- Minimizar a probabilidade de erros na gestão de memória
  - Usar linguagens de programação que garantem memory safety (Java, Go, Rust, etc.), ou
  - Verificar que os programas estão corretos (mais esforço, mas a única hipótese para código legacy)

- Medidas de proteção na própria plataforma: impedir a execução de código malicioso
- Medidas de proteção no executável: detectar tentativa de highjacking
  - monitorização do estado da stack, tagging de memória, etc.

controle  
passa a  
DoS

# Mitigações na Plataforma

# Data Execution Prevention

- Também chamada Executable Space Protection, 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

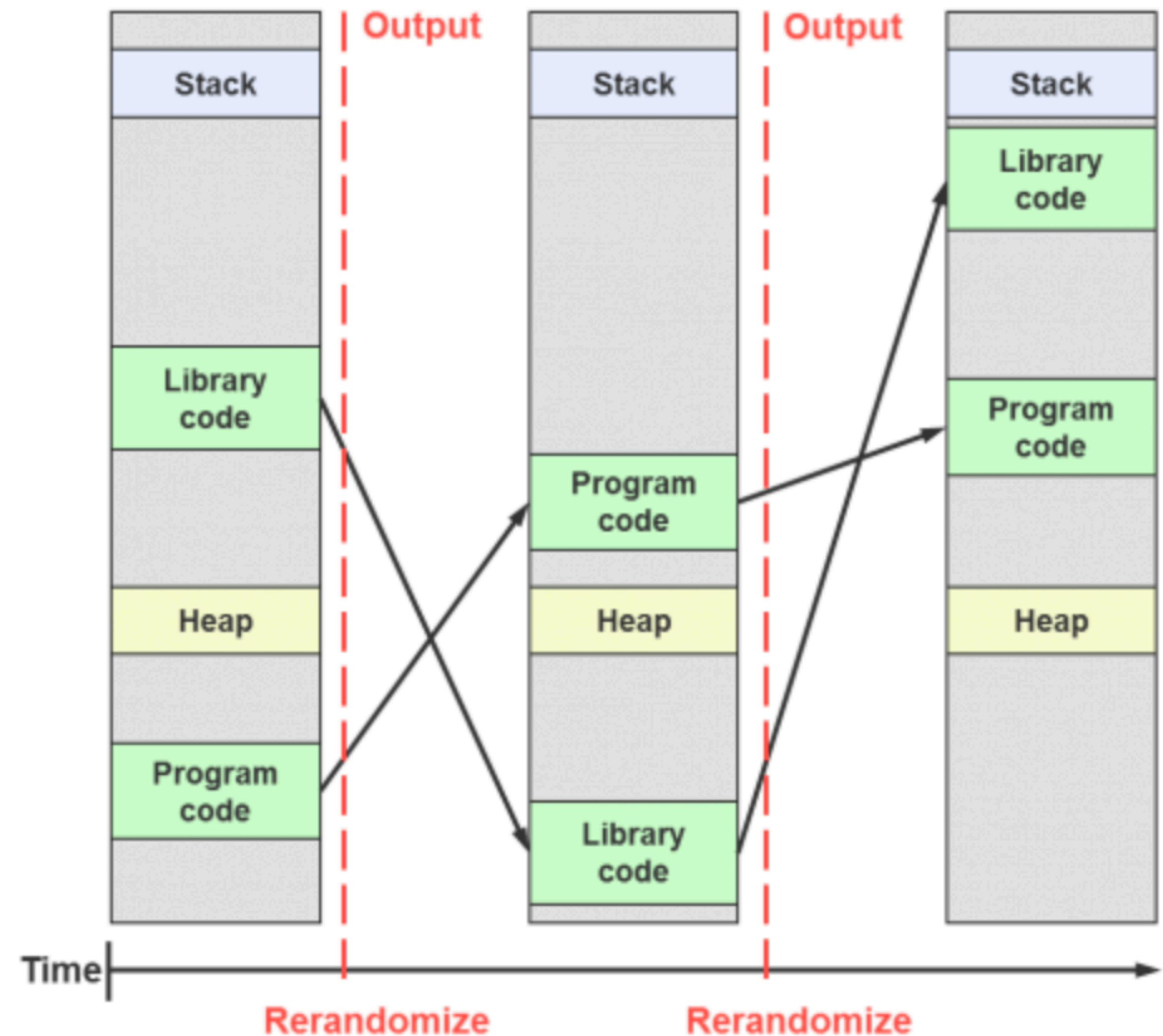
# Data Execution Prevention

- 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 de Just-in-Time compilation (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 (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)





# Address Space Randomization

- 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”



A new feature added in test snapshots for OpenBSD releases will create a unique kernel every time an OpenBSD user reboots or upgrades his computer.

This feature is named KARL — Kernel Address Randomized Link — and works by relinking internal kernel files in a random order so that it generates a unique kernel binary blob every time.

# Mitigações no Executável

# Detecção em Tempo de Execução

- Um segundo nível de proteção:
  - o compilador adiciona código que permite detectar 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

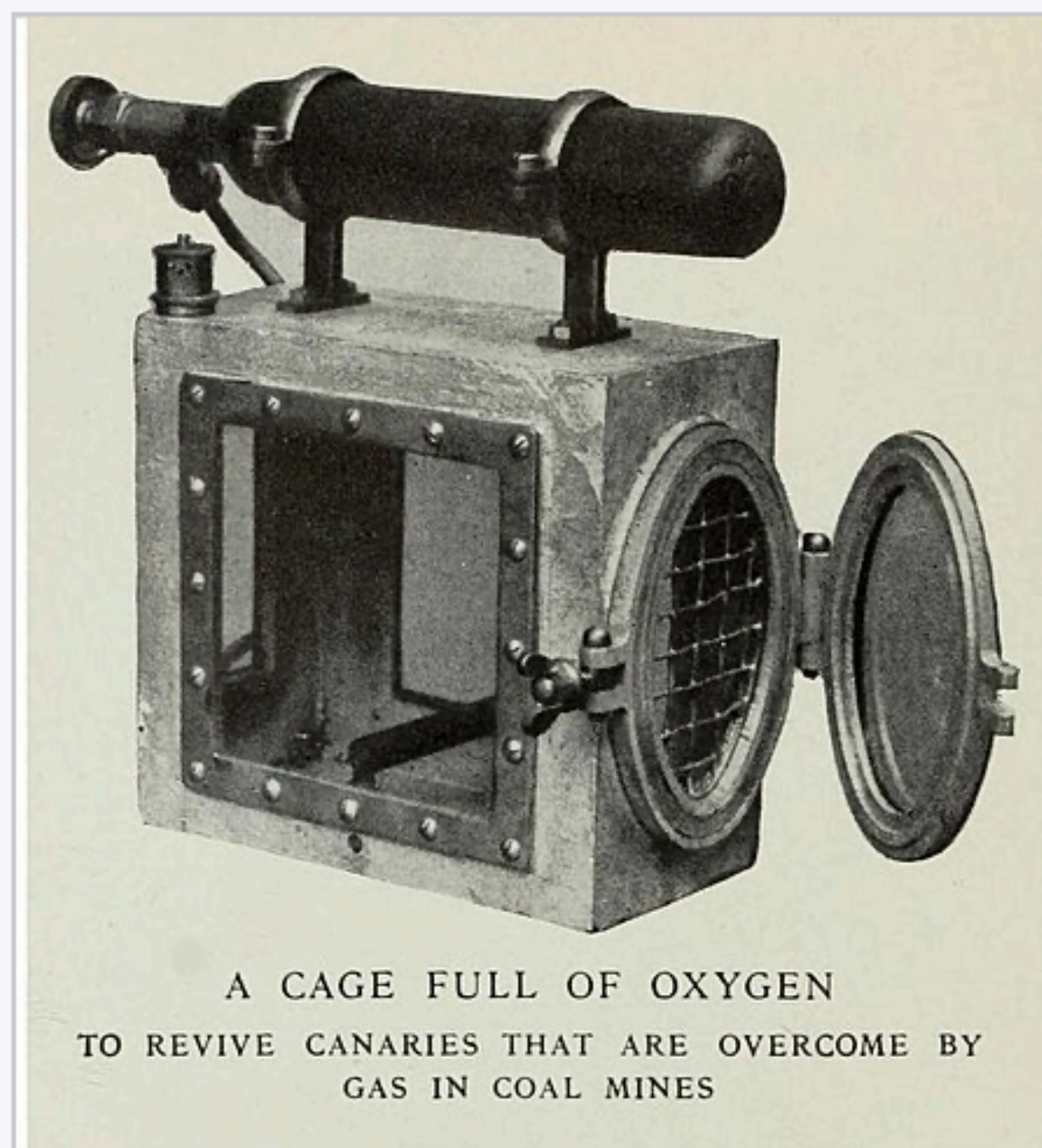


# Stack Canaries

## Miner's canary [\[ edit \]](#)

Mice were used as [sentinel species](#) for use in detecting carbon monoxide in British [coal mining](#) from around 1896,<sup>[15]</sup> after the idea had been suggested in 1895 by [John Scott Haldane](#).<sup>[16]</sup> Toxic [gases](#) such as [carbon monoxide](#) or [asphyxiant gases](#) such as [methane](#)<sup>[17]</sup> in the mine would affect small warm-blooded animals before affecting the miners, since their respiratory exchange is more rapid than in humans. A mouse will be affected by carbon monoxide within a few minutes, while a human will have an interval of 20 times as long.<sup>[18]</sup> Later, canaries were found to be more sensitive and a more effective indicator as they showed more visible signs of distress. Their use in mining is documented from around 1900.<sup>[19]</sup> The birds were sometimes kept in carriers which had small oxygen bottles attached to revive the birds.<sup>[20][21]</sup> The use of miners' canaries in [British](#) mines was phased out in 1986.<sup>[22][23]</sup>

The phrase "[canary in a coal mine](#)" is frequently used to refer to a person or thing which serves as an early warning of a coming crisis. By analogy, the term "climate canary" is used to refer to a species (called an [indicator species](#)) that is affected by an environmental danger prior to other species, thus serving as an early warning system for the other species with regard to the danger.<sup>[24]</sup>



Resuscitation cage with an oxygen cylinder serving as a handle used to revive a canary for multiple uses in detecting carbon monoxide pockets within mines

# Stack Canaries

- Objectivo:
  - prevenir injeção de código detectando 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

# Exemplo

```
void func(int a, int b, char *str) {  
    int c = 0xdeadbeef;  
    char buf[4];  
    strcpy(buf, str);  
}  
  
int main(int argc, char**argv) {  
    func(0xaaaaaaaa, 0xbbbbbbbb, argv[1]);  
    return 0;  
}
```

argv[1]
bbbbbbbb
aaaaaaaa
endereço de retorno
frame pointer
canário
deadbeef
buf[0-3]

# Variantes

- Canário aleatório
  - 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 ', E0F em vez de bytes aleatorios)
  - As funções que manipulam strings irão sempre parar nestes valores



# StackGuard

- Implementado pelo GCC (-fstack-protector-strong)
- Vantagens:
  - simples e fácil de colocar em prática
- Desvantagens:
  - overhead de espaço e performance
- Como fazer bypass aos canários?

# Derrotando os canários

- 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
- Outra vulnerabilidade pode permitir "aprender" o canário, ou
- 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 tipos de dados "perigosos" numa stack separada

# 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)
- [https://www.usenix.org/system/files/login/articles/login\\_summer19\\_03\\_serebryany.pdf](https://www.usenix.org/system/files/login/articles/login_summer19_03_serebryany.pdf)

# Control Flow Integrity

- Até agora olhámos para mitigações que se focam na memória.
- Uma alternativa é validar que a sequência de controlo está dentro de um conjunto de sequências válidas:
  - 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, mas
- É necessário proteger todos os saltos/retornos cujos endereços são criados dinamicamente pelo programa

# Control Flow Integrity

- 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
  - 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 grafo que define todos os saltos válidos
- Usando criptografia (veremos mais à frente):
  - sempre que um endereço é escrito algures na memória, guardar também um autenticador criptográfico
  - sempre que se usa o endereço verifica-se o autenticador
  - usa-se uma chave que não está em memória (um registo por exemplo)
- soluções e ataques têm sido muito discutidas no meio académico; implicam um overhead significativo

# Control-Flow Integrity

- Implementações existem no Clang, MS Control Flow Guard, MS Return Flow Guard, Google Indirect Function-Call Checks, ...
- A shadow control stack, que referimos anteriormente, pode ser vista como parte do control-flow integrity
- As plataformas Intel permitem utilizar uma flag de HW para assinalar os entry points válidos para saltos, que têm de explicitamente desligar a flag
- O MS CFG funciona de forma parecida, mas utiliza uma tabela protegido por um hypervisor para gerir os endereços válidos

# 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



# Technical Analysis of the Pegasus Exploits on iOS

# Trident/Pegasus

1. CVE-2016-4657: Memory Corruption in WebKit - A vulnerability in Safari WebKit allows the attacker to compromise the device when the user clicks on a link.
2. CVE-2016-4655: Kernel Information Leak - A kernel base mapping vulnerability that leaks information to the attacker that allows him to calculate the kernel's location in memory.
3. CVE-2016-4656: Kernel Memory corruption leads to Jailbreak - 32 and 64 bit iOS kernel-level vulnerabilities that allow the attacker to silently jailbreak the device and install surveillance software.
4. The Pegasus Persistence Mechanism used for remaining on the device after compromise.

# Technical Analysis of the Pegasus Exploits on iOS

# Trident/Pegasus

- Alguns detalhes:
  - vulnerabilidade *use-after-free* no GC do safari
  - é possível alterar o tamanho de array alocado para aceder a um espaço enorme de memória (read/write)
  - leituras => descobrir números mágicos + localizações
  - Descobrir objeto que vai ser compilado (JIT) => injetar código malicioso

# Program Safety

# 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
  - 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 é unsafe
  - Exemplo: a semântica da linguagem indica que os cálculos sobre inteiros de precisão limitada apenas são consistentes com  $\mathbb{Z}$  dentro de uma determinada gama
    - um programa que assume consistência com  $\mathbb{Z}$  fora dessa gama é unsafe

# Memory 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
  - 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



# Conclusão





# 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, 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<sup>X</sup>, etc.
  - Control-Flow Integrity
- 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