# Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

# Software Security (Part 4)

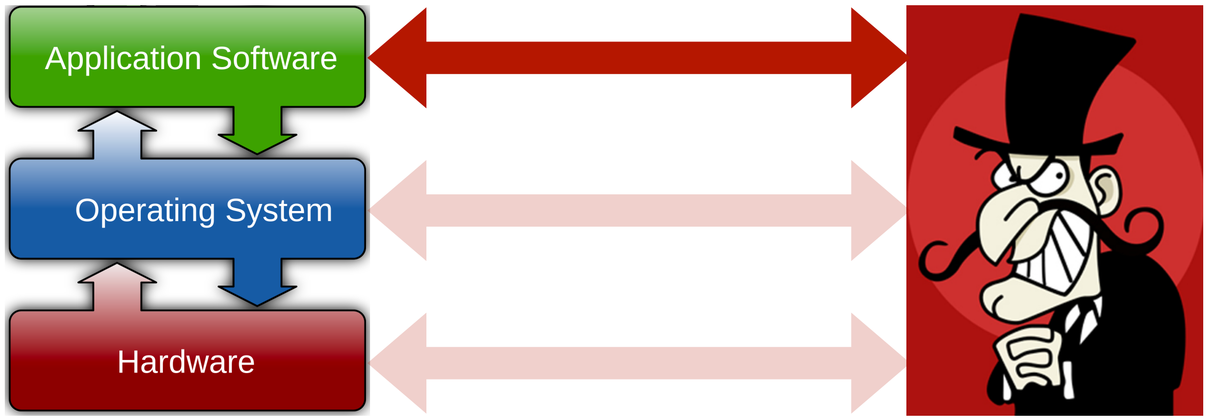
**Hugo Pacheco**
**hpacheco@fc.up.pt**

# Attacks we have seen

- **Stack/Heap Buffer Overflows:** overwriting the return address in the stack/heap to execute malicious code
- **Use After Free:** make use of program that is using memory outside its control to read malicious code from such memory
- **Format Strings:** control string formatters and/or explore incorrect number of variadic arguments to read/write data from stack/heap memory
- **Integer Overflows:** abuse arithmetic precision errors to trigger, e.g., wrong memory allocation
- ...

# What is there in common?

- **Data from outside the security perimeter** may interfere with the **control flow of the program**

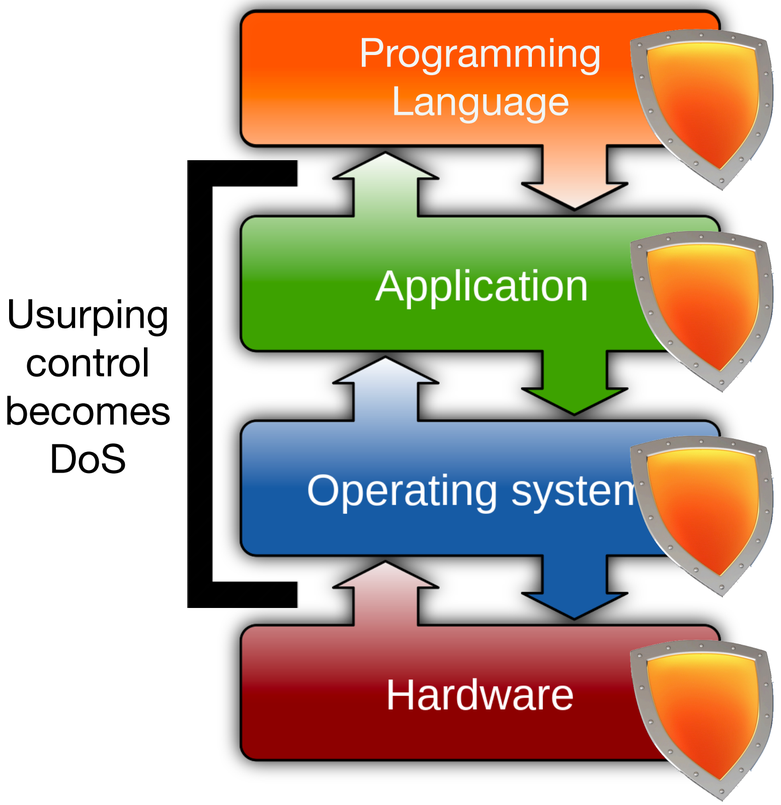



- E.g., buffer with data stored in the stack next to a return address

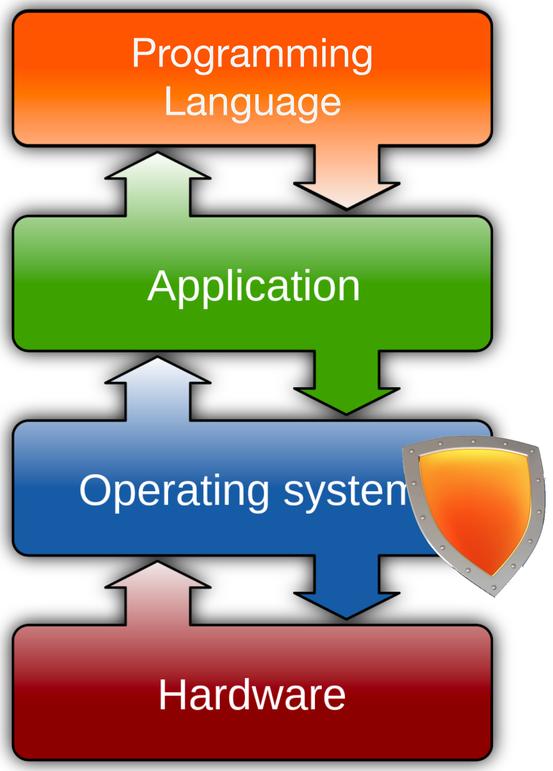
- A recurring problem… (many more examples later in Web Security)
- Even if adopting good principles, implementation errors may break all assumptions/guarantees

# Defense in Depth

- Countermeasures at the level of the **Programming Language**:
  - Use *memory safe* languages (Java, Haskell, Go, Rust, etc.)
  - Program verification (more effort, only choice for critical legacy code)
  - Secure compilation
- Countermeasures at the level of the **Application**:
  - Detect hijacking attempts; monitor the stack, tag memory, etc.
- Countermeasures at the level of the **Operating System**:
  - Prevent the execution of malicious code
- Countermeasures at the level of the **Hardware**
  - Instruction set, trusted execution environments, secure boot, etc.



# Operating System Countermeasures



# Data Execution Prevention

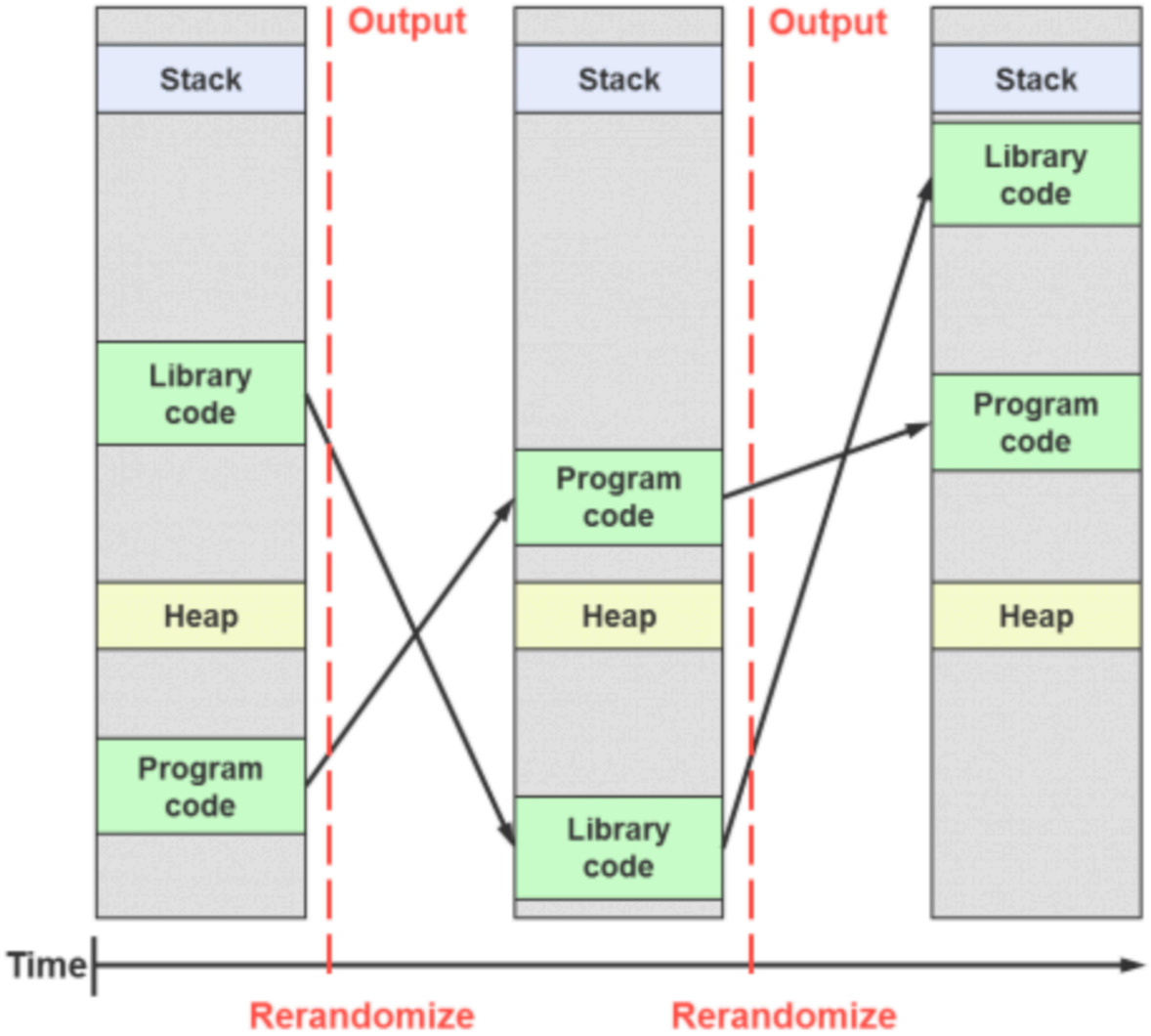
- Also called Executable Space Protection or W ^ X (mentioned before):
  - motivation for Return-Oriented Programming 
- Memory can never **simultaneously**:
  - Be **writable** by a program
  - Contain **executable** code
- Can be implemented in HW (NX bit) or emulated in SW

# Data Execution Prevention

- The most popular architectures offer HW support:
  - AMD, Intel, ARM: disallowing execution at the level of memory pages
- Implemented by the most popular OSes since the 2000's
- Limitations  :
  - By itself does not forbid exploits from reusing code, e.g., ROP
  - Creates problems for JIT (e.g., in modern browsers where trusted code may run alongside untrusted client code)

# Address Space Layout Randomisation

- The location of code and data in memory is determined at random **in each execution**:
  - stack, heap, shared libraries, base code
- **Advantages:** attacker cannot easily predict the addresses of useful code (stack, ROP)
- Usage:
  - In more popular OSes since the 2000's
  - More entropy to make prediction harder (8 bits ⇒ 24 bits)




https://www.daniloaz.com/en/differences-between-aslr-kaslr-and-karl/

# Address Space Layout Randomisation

- **Disadvantages:** “shuffled” regions have the same structure

- Two executions

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
082ac000-082cd000 rw-p 082ac000 00:00 0          [heap]
b7dfe000-b7f53000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f53000-b7f54000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7f54000-b7f56000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf966000-bf97b000 rw-p bffeb000 00:00 0          [stack]
```

- Each region has a random offset, but a fixed layout

```
exploit:~# cat /proc/self/maps | egrep '(libc|heap|stack)'
086e8000-08709000 rw-p 086e8000 00:00 0          [heap]
b7d9a000-b7eef000 r-xp 00000000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7eef000-b7ef0000 r--p 00155000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
b7ef0000-b7ef2000 rw-p 00156000 08:01 1750463    /lib/i686/cmov/libc-2.7.so
bf902000-bf917000 rw-p bffeb000 00:00 0          [stack]
```

# Address Space Layout Randomisation

- **Attacker** (without crashing the program)**:**

  - Extract addresses from other vulnerabilities
  - Use spraying techniques
  - Guess many times under the same process
  - Use OS kernel code?

https://www.ndss-symposium.org/wp-content/uploads/2017/09/how-make-aslr-win-clone-wars-runtime-re-randomization.pdf

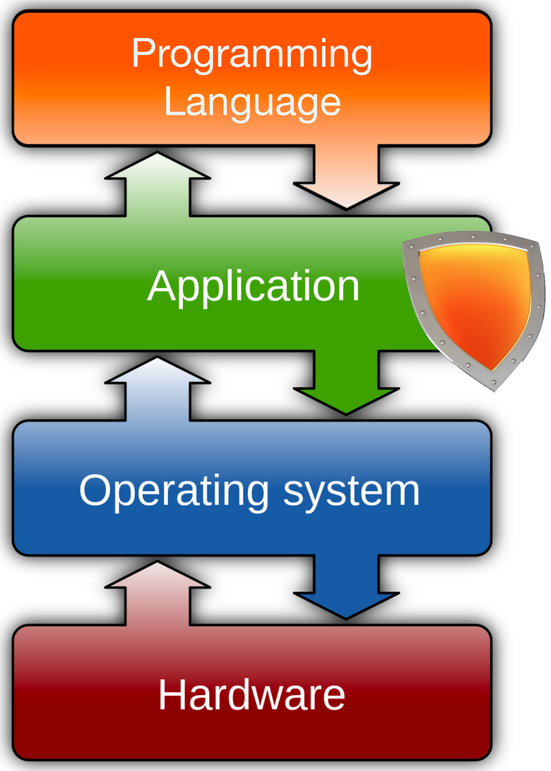
- Other (cutting-edge) solutions:
  - **Runtime ASLR**: “shuffle” the address space on forks
  - **Kernel ASLR (KASLR)**: only changes at boot time; attacker still just needs to predict the kernel base
  - **Kernel Address Randomised Link (KARL)**: the kernel code itself is “shuffled”

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.

https://www.bleepingcomputer.com/news/security/openbsd-will-get-unique-kernels-on-each-reboot-do-you-hear-that-linux-windows/

# Application Countermeasures



# Compiler Countermeasures

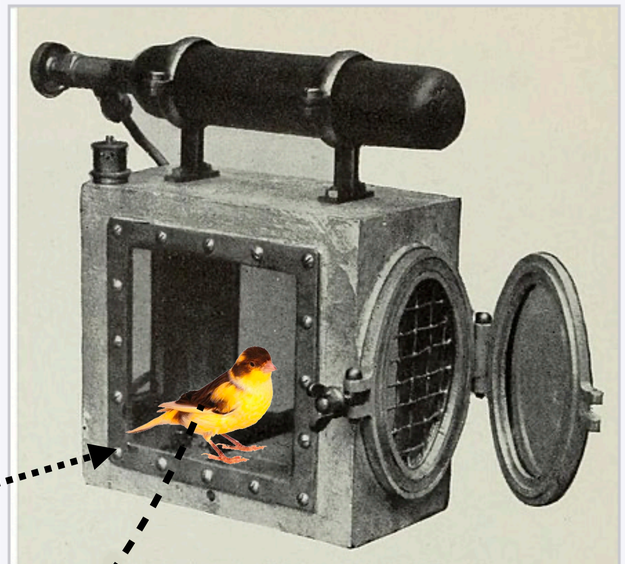
- **Position Independent Executables/Code (PIE/PIC)**
  - The compiler generates (blocks of) code whose execution is **independent from absolute addresses** ⇒ each time it is executed, can be loaded into a different address
  - Attacker still just needs to find the base address (of the code block)
- **Code instrumentation** to perform “run-time monitoring”:
  - The compiler attaches simple code to “monitor” more complex code and **verify that it is performing the expected functionality**
  - Transforms **arbitrary code execution exploits** into **DoS attacks**
    ⇒ the program aborts

# 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,[15] after the idea had been suggested in 1895 by John Scott Haldane.[16] Toxic gases such as carbon monoxide or asphyxiant gases such as methane[17] 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.[18] Later, canaries were found to be more sensitive a a more effective indicator as they showed more visible signs of distress. Their use in mining is documented from around 1900.[19] The birds were sometimes kept in carriers which had small oxygen bottles attached to revive the birds.[20][21] The use of miners' canaries in British mines was phase out in 1986.[22][23]

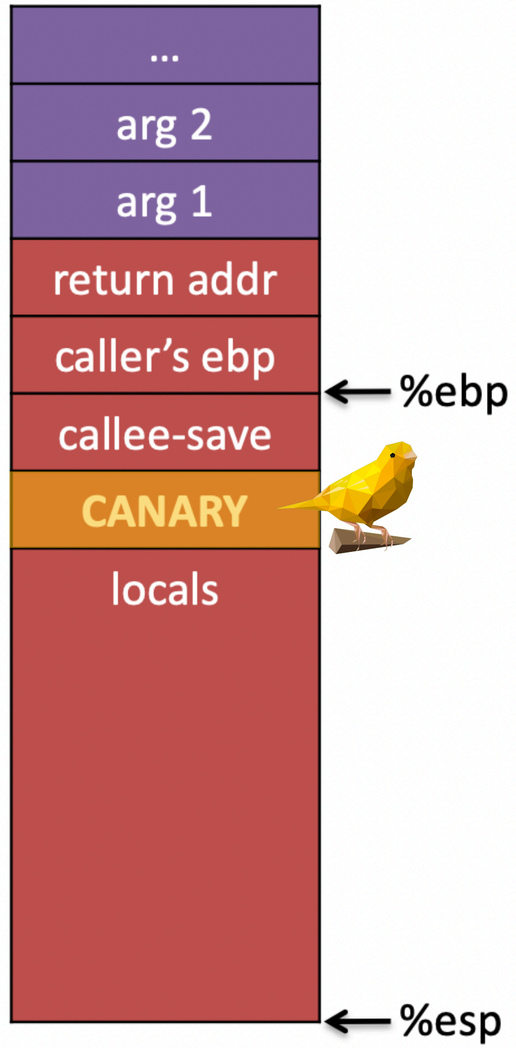
The phrase "canary in a coal mine" is frequently used to refer to a person or thing which serves as an early warning of a coming crisis. By analogy, t term "climate canary" is used to refer to a species (called an indicator species) that is affected by an environmental danger prior to other species,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


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

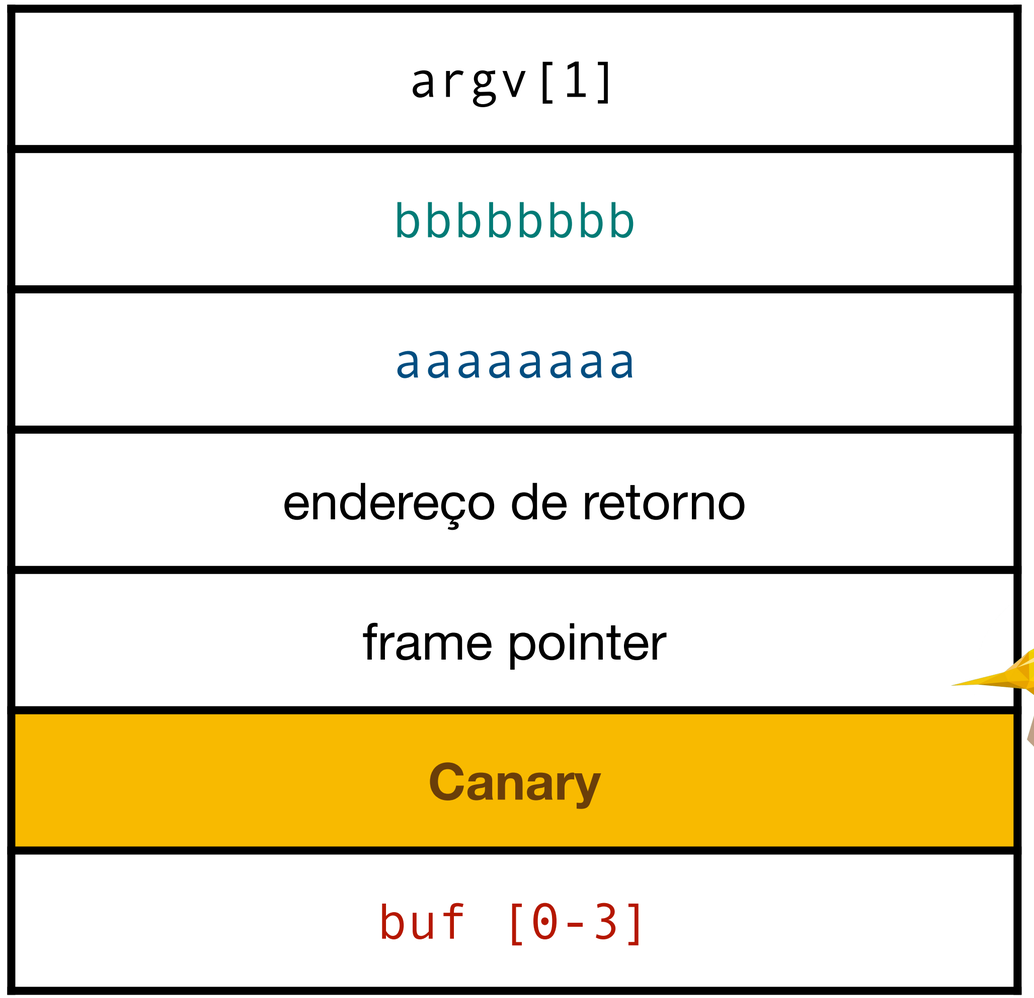
- **Objective:** preventing code injection by detecting stack modifications
- **Idea:**
  - Introduce “**canaries**” generated dynamically **between local variables and the frame pointer and return address values** stored in the stack
  - Verify canary before using the return address
- **Implementation:** compiler instruments entry/exit of functions



# Stack Canaries (Example)

```
void func(int a, int b, char *str) {
  char buf[4];
  strcpy(buf,str);
}


int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```



# Stack Canaries (Variants)

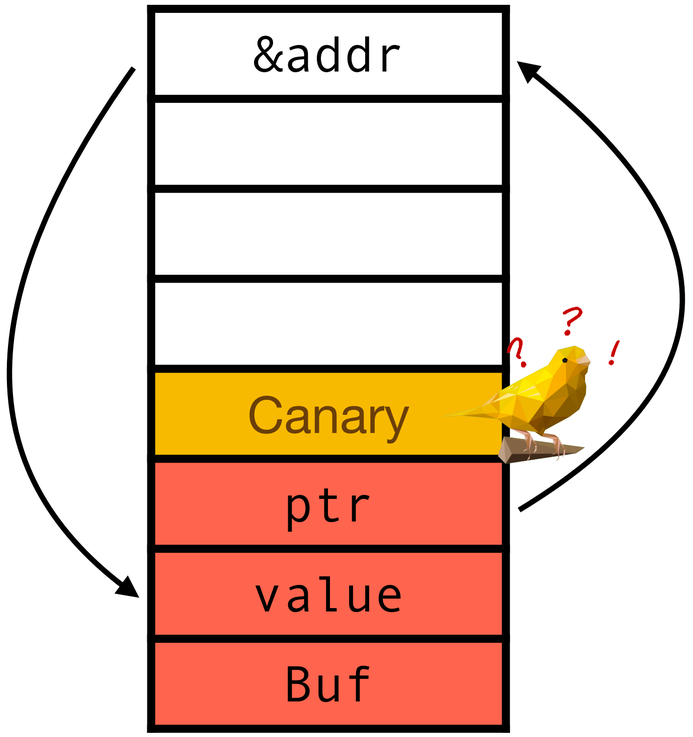
- Random canary (requires “good” randomness)
  - At the beginning of execution the program chooses an **array of random bytes**
  - Those bytes are used to generate canaries that are placed in all stack frames
  - Before returning from a function, the program verifies the **canary’s integrity**
  - If there is some change, the program **terminates**
- Termination canary (using some `‘\0','\n',EOF` instead of fully random bytes)
  - String manipulation functions will always stop in such values

# Stack Canaries (Compilers)

- Implemented by most C compilers:
    - GCC StackGuard (-fstack-protector-strong), MS BufferSecurityCheck, many others…
- **Advantages:**
    - simple and easy to put into practice
- **Disadvantages:**
    - space overhead (1 address per function call)
    - performance (~10%)

# Defeating Canaries

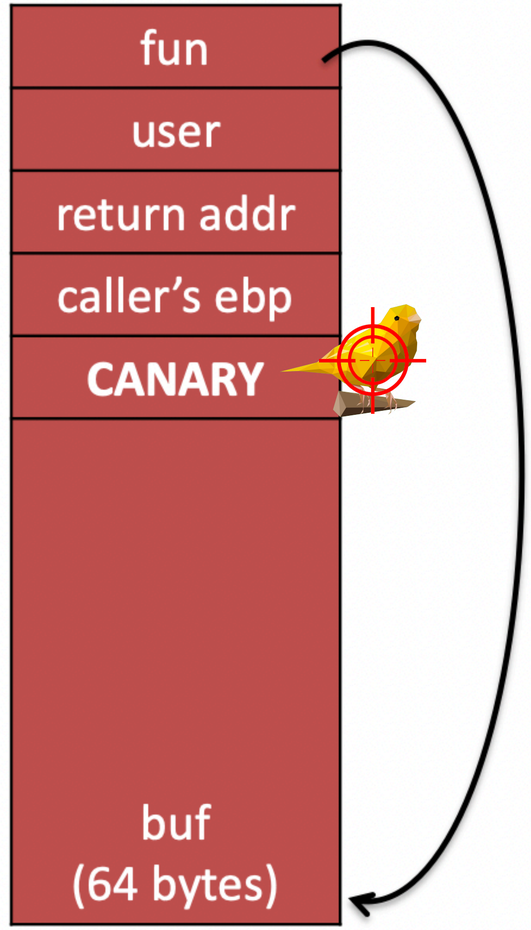
- Heap & Integer overflows are **not affected**
- **Read the canary** using other vulnerability + stack overflow and put back the canary's value
- **Circumvent the canary using existing pointers** 😈:
  - If the code has a **pointer in the stack** (`ptr`) and writes to that address the `value` of another local variable …
  - … a stack overflow may change both and write to an arbitrary memory position



# Defeating Canaries

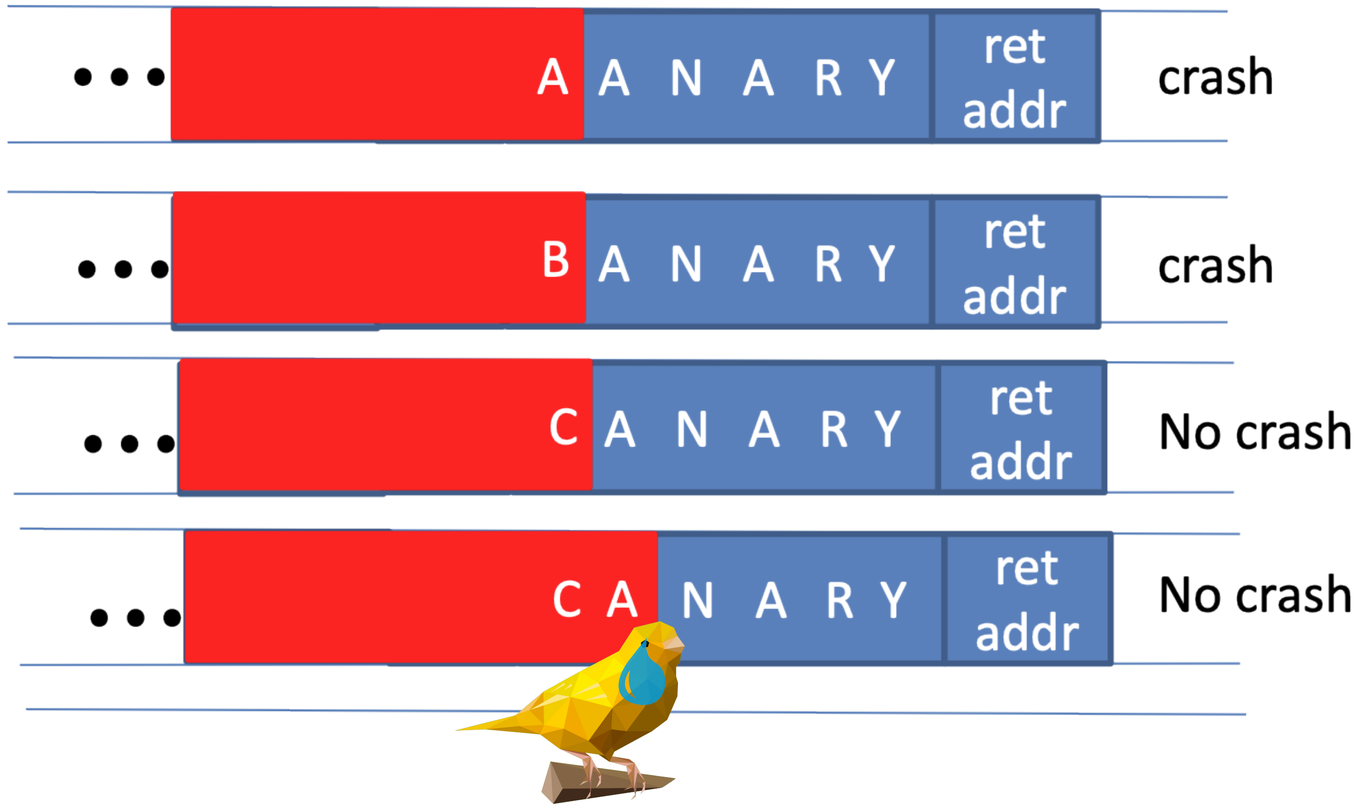
```
void contrived(const char *user, void (*fun)(char *)) {
    char buf[64];
    strcpy(buf, user);
    fun(buf);
}
```

- **Overwrite and ignore the canary using function pointers 😈:**
  - If a program receives a function pointer as an argument …
  - … changing such pointer may allow jumping to an arbitrary address before the function returns



# Defeating Canaries

- **Guess the canary 😈:**
  - If the canary is reused many times, e.g., by being preserved on forks …
  - … it is susceptible to brute-force attacks
  - It is “easy” to  “retrieve the canary byte-by-byte

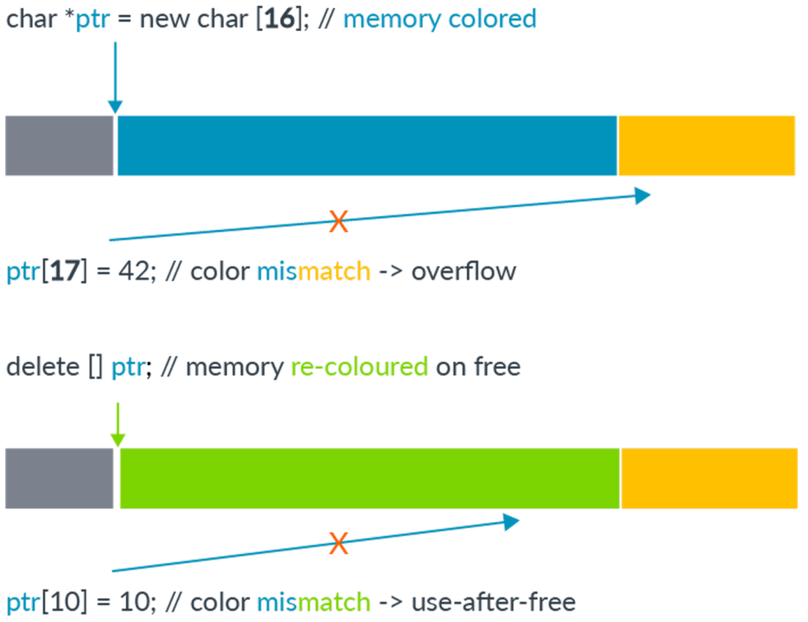


# Additional Stack Protections

- **Changing the memory convention**, e.g. GCC:
  - Making sure that local buffers are always close to canaries ⇒ when overflow does not reach canary
  - Copying function arguments to the top of the stack, to addresses below local buffers ⇒ immune to overflow
- **Adding stack redundancy**, e.g., Shadow Stack:
  - Redundant stack only for control (frame pointer and return address)
  - Check consistency of original and shadow stacks before returning from the function
  - HW support: Intel Control Flow Enforcement Technology, shadow stack placed in memory only writable by `call`/`ret` operations

# Memory Tagging

- ARM Memory Tagging Extension:
  - HW support for creating tags (new instructions)
  - “Binds” pointers to the pointed memory regions
    - Both must have the same 4-bit tag
  - Tags are compared on memory access:
    overflow ⇒ exception
  - `free` updates tags: use after free ⇒ exception




- Designed for **debugging** and extra protection against overflows

https://www.usenix.org/system/files/login/articles/login_summer19_03_serebryany.pdf

# Control-Flow Integrity

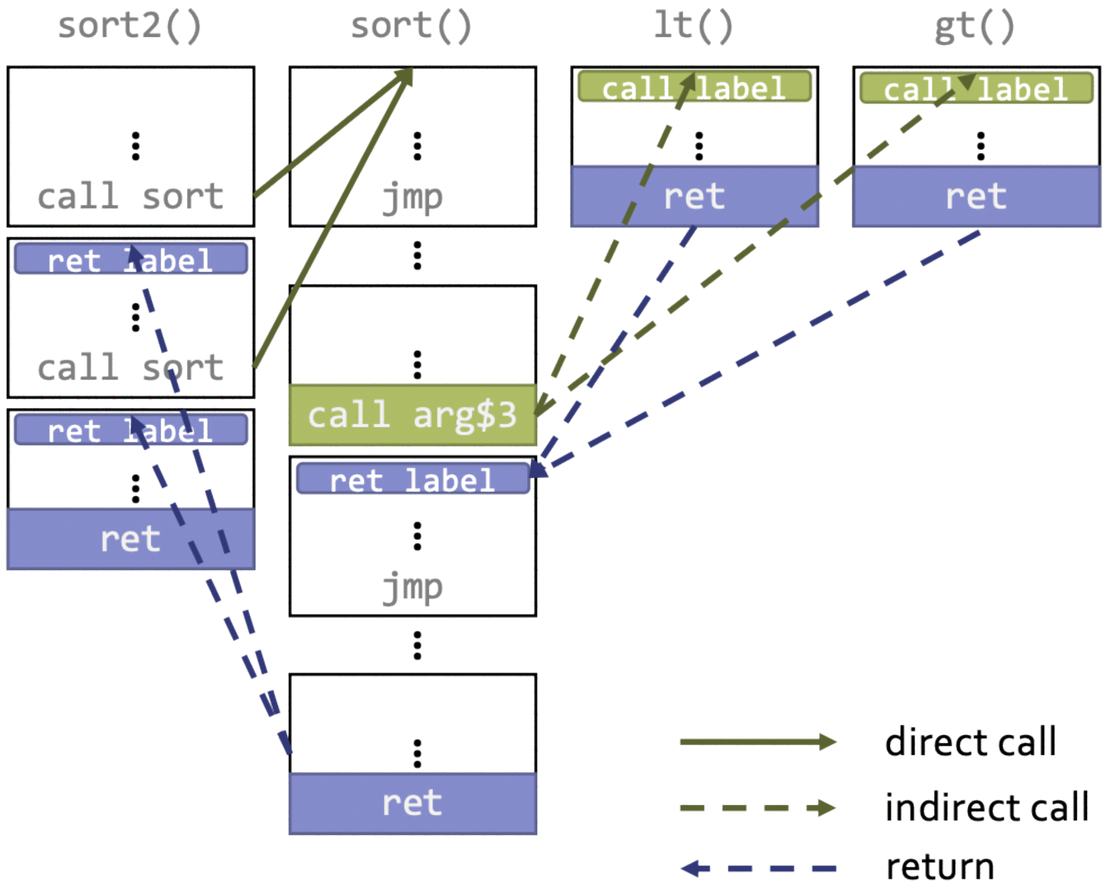
- **So far:** countermeasures **focused on memory** (preventing/detecting modifications)
- **CFI focused on control flow**:
    1. At compile time: **determining a control flow graph (set of possible sources for each target)**
    2. At execution time: **verifying consistency w.r.t. such information**
- **Not necessary to protect direct calls:** jumps and static calls whose target is hardcoded into the code ⇒ the attacker cannot change the compiled code
- **Necessary to protect indirect calls:** jumps/returns whose addresses are dynamically manipulated by the program, including function pointers

# Control-Flow Integrity (Example)

```
void sort2( int a[],
            int b[],
            int len )
{
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y)
{ return x < y; }

bool gt(int x, int y)
{ return x > y; }
```



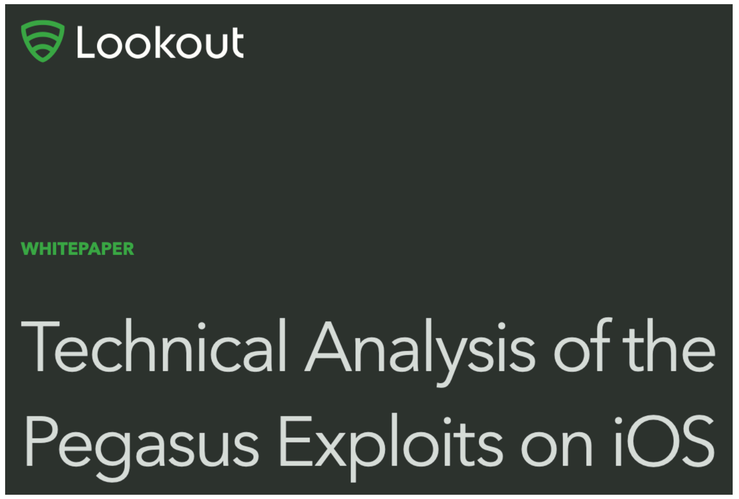
# Control-Flow Integrity

- **Most basic version:** consider only function entry points as valid targets + confirm that every function entry arose from a call (OS (e.g., kBouncer) & HW (e.g., Intel's Last Branch Recording) support)
  - Problem: does not prevent function calls from following a different sequence, e.g., jumping over an authentication function
- **Classical version:** over-approximate a Control Flow Graph (CFG) + check that each indirect call is valid (implemented in Clang, MS Control Flow Guard, etc)
  - Challenge: statically compute a precise CFG
- **Advanced version:** using cryptography (CCFI):
  - Every time an address is written, store also a cryptographic MAC + check the MAC every time the address is accessed + cryptographic key not stored in memory (e.g., in a register)
- **Status:** many competing proposals; significant performance overhead ⇒ **HW support is vital!**

# Are attacks still possible?

- With all countermeasures, attacks are getting more elaborate but still possible…
- **Trident exploit** ⇒ **Pegasus rootkit**:
  - Allowed to remotely unlock an iOS phone and install spyware
  - Detected in the real world and attributed to a company called NSO Group (60 minutes documentary )
  - Chained together a **series of exploits** to vulnerabilities in browser + kernel
- **Takeaway:** essential to ensure that there are **no "holes"** in the processing of potentially **malicious inputs**!

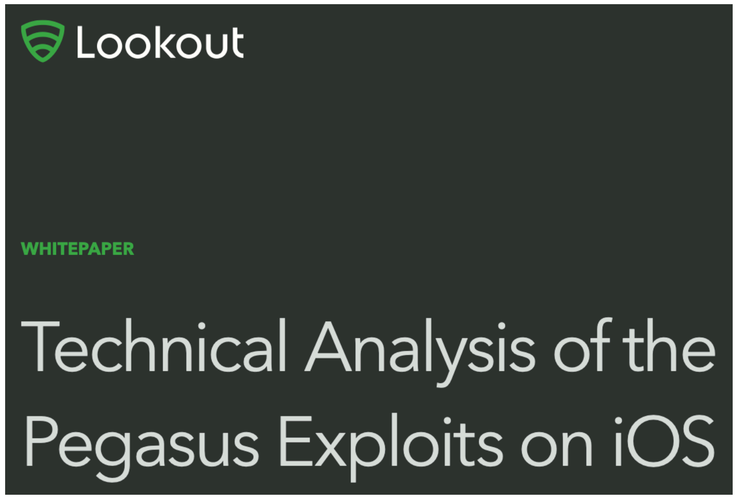
https://www.cbsnews.com/news/interview-with-ceo-of-nso-group-spyware-maker-fighting-terror-khashoggi-murder-and-saudi-arabia-60-minutes-2019-08-18/




# 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.

https://blog.lookout.com/trident-pegasus-technical-details




# Trident ⇒ Pegasus

- Some details:

1. Use-after-free vulnerability in Safari's GC ⇒ change the size of an allocated array ⇒ access to a large memory space with RW permissions

2. Kernel vulnerability ⇒ finding magic numbers + kernel location

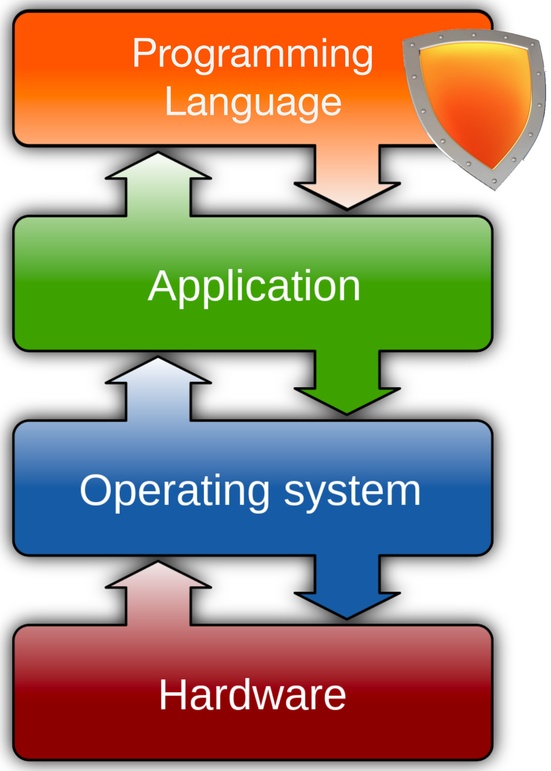
3. Safari JIT vulnerability ⇒ inject shellcode

4. Previous vulnerabilities ⇒ find the address of compiled object

5. Kernel vulnerability ⇒ Jailbreak + evading detection

https://blog.lookout.com/trident-pegasus-technical-details

# Programming Language Countermeasures



# Program Safety

- Important concept in the **Programming Languages** area:
  - Informally: the execution of the **program remains within the behaviour specified in the language's semantics** (e.g., C Standard)
  - Example: the semantics describes what should happen when we access a valid memory region, but is omissive in the remaining cases
    - A program that accesses an invalid memory region is **unsafe**
  - Example: the semantics indicates that integer arithmetic with limited precision are only consistent with $\mathbb{Z}$ within a specific range
    - A program that assumes consistency with $\mathbb{Z}$ outside of such range is **unsafe**

# Memory Safety ⊂ Program Safety

- There are various kinds of unsafe memory operations:
  - Possibly **“undefined”** behaviour in the semantics of the language, (e.g. C Standard)
    - **Reading** from invalid memory region
    - **Writing** to invalid memory region
    - **Executing** invalid memory regions / invalid instructions
- Spatial Safety: validity depends on memory location
- Temporal Safety: validity depends on the lifetime of an object/variable

# Ensuring Safety

- **Strongly typed** languages (Java, Haskell, Rust) verify a large set of safety conditions (types, array bounds, etc) at compile time
- Some **weakly typed** languages (Python) verify safety at execution time
- **Interpreted** languages (Java, Python) detect memory safety problems during execution
    - Invalid memory accesses raise exceptions ⇒ control attacks become DoS
- Rust includes various static and dynamic checks to ensure safety

CYBERSECURITY & INFRASTRUCTURE SECURITY AGENCY

# The Urgent Need for Memory Safety in Software Products

**Released:** September 20, 2023

Bob Lord, Senior Technical Advisor

For over half a century, software engineers have known malicious actors could exploit a class of software defect called "memory safety vulnerabilities" to compromise applications and systems. During that time, experts have repeatedly warned of the problems associated with memory safety vulnerabilities. Memory unsafe code even led to a major internet outage in 1988. Just how big a problem is memory unsafety? In a blog post, Microsoft reported that "~70% of the vulnerabilities Microsoft assigns a CVE [Common Vulnerability and Exposure] each year continue to be memory safety issues." Google likewise reported that "the Chromium project finds that around 70% of our serious security bugs are memory safety problems." Mozilla reports that in an analysis of security vulnerabilities, that "of the 34 critical/high bugs, 32 were memory-related."

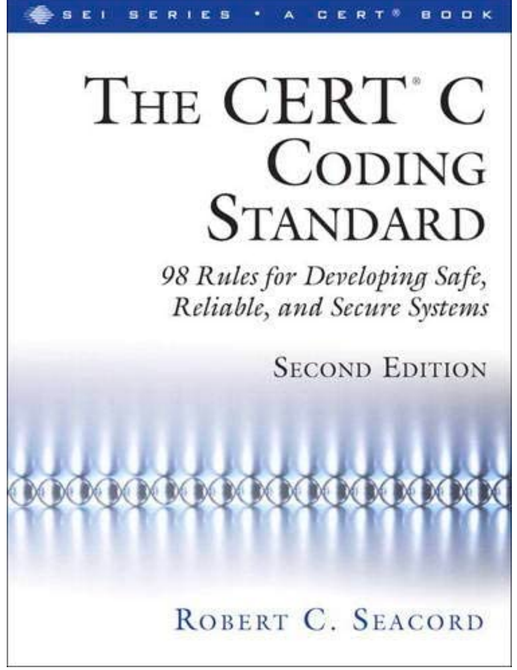
Different products will require different investment strategies to mitigate memory unsafe code. The balance between C/C++ mitigations, hardware mitigations, and memory safe programming languages may even differ between products from the same company. No one approach will solve all problems for all products. The one thing software manufacturers cannot do, however, is ignore the problem. The software industry must not kick the can down the road another decade through inaction.

https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

- **How difficult is it to guarantee safety?**
- **How important is it to guarantee safety?**
- **Even low-level unsafe languages (C, Assembly) can have safety guarantees**
  - Program verification tools
    - some are dynamic & incomplete, but automatic (valgrind, fuzzing, symbolic execution, …)
    - some are static & complete, but largely manual (FramaC, Dafny, …)

# Just safety?




- **Defensive programming**, e.g., CERT C Coding Standard https://www.securecoding.cert.org/confluence/display/
- **Taint analysis,** a form of information flow analysis:
  - User input is tainted
  - Critical functions accept only untainted arguments
  - Examples: gets(), printf(), strcpy()
- **Constant-time analysis:** protection against timing side-channels
- **Secure compilation:** prove that the compilation process does not introduce security errors
- …

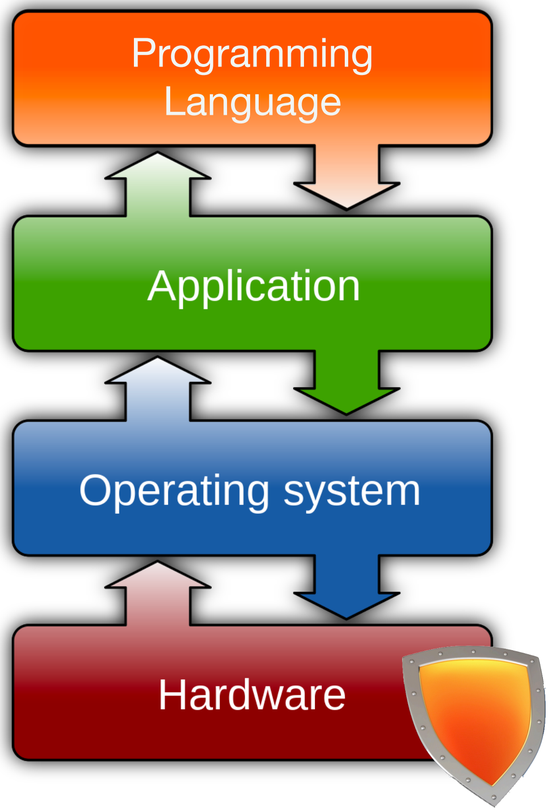
```
int printf(untainted char *fmt, ...);
tainted char *fgets(...);

tainted char *name = fgets(...,network_fd);
printf(name); // FAIL: tainted ≠ untainted
```

```
int check(char *arg, char *pass)
{
  // secret password of size 5 bytes
  int i;
  for (i=0; arg[i]==pass[i] && i < 5; i++);
  return (i==n);
}
```

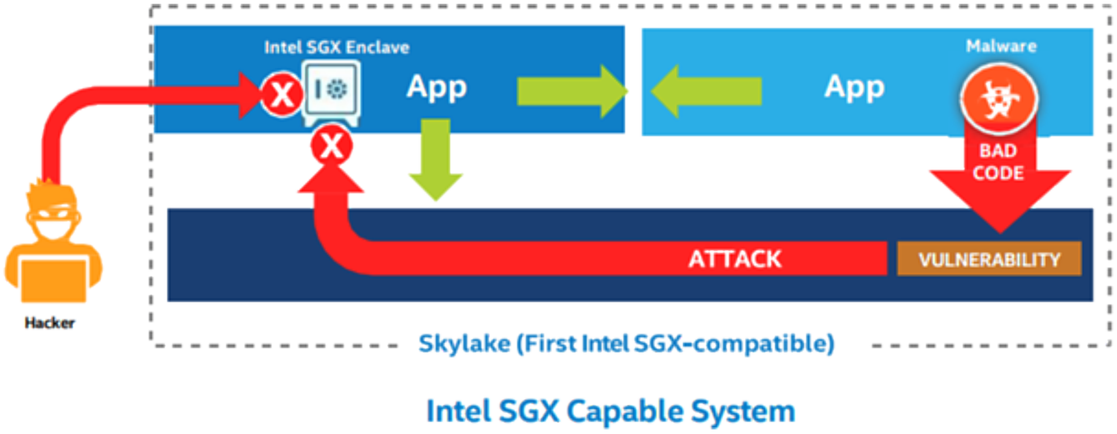
# In the end…

# Hardware Countermeasures



# Hardware Protections


- Many of the techniques that we have seen have some HW support
- Trusted Platform Module (a secure crypto-processor)
- Trusted Execution Environments (Intel SGX, ARM TrustZone)
- UEFI Secure Boot (firmware trusted by the manufacturer)




Intel SGX Capable System

## Windows 11 and Secure Boot

*Published August 2021*

This article is intended for users who are not able to upgrade to Windows 11 because their PC is not currently Secure Boot capable. If you are unfamiliar with this level of technical detail, we recommend that you consult your PC manufacturer's support information for more instructions specific to your device.

Secure Boot is an important security feature designed to prevent malicious software from loading when your PC starts up (boots). Most modern PCs are capable of Secure Boot, but in some instances, there may be settings that cause the PC to appear to not be capable of Secure Boot. These settings can be changed in the PC firmware. Firmware, often called BIOS (Basic Input/Output System), is the software that starts up before Windows when you first turn on your PC.

# Why do vulnerabilities exist?

- Programmers make errors ⇒ crucial to use tools which assist in avoiding bugs
- Programmers often have no notion of the implications of a bug

  ⇒ **training in cybersecurity! this course!** 
- (General-purpose) languages are **not** designed to guarantee **security**…
  - Still, some languages are **better** at ensuring **safety** (Java, Haskell, Python, Rust, etc.)
  - Program verification of critical code
  - Secure engineering development process 

# It is actually getting worse?

RESEARCH-ARTICLE

## Do Users Write More Insecure Code with AI Assistants?

**Authors**: Neil Perry, Megha Srivastava, Deepak Kumar, Dan Boneh | Authors Info & Claims

CCS '23: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security • Pages 2785 - 2799
https://doi.org/10.1145/3576915.3623157

**Published**: 21 November 2023 Publication History

Check for updates

28 1,600

PDF eReader

***Abstract***

AI code assistants have emerged as powerful tools that can aid in the software development life-cycle and can improve developer productivity. Unfortunately, such assistants have also been found to produce insecure code in lab environments, raising significant concerns about their usage in practice. In this paper, we conduct a user study to examine how users interact with AI code assistants to solve a variety of security related tasks. Overall, we find that participants who had access to an AI assistant wrote significantly less secure code than those without access to an assistant. Participants with access to an AI assistant were also more likely to believe they wrote secure code, suggesting that such tools may lead users to be overconfident about security flaws in their code. To better inform the design of future AI-based code assistants, we release our user-study apparatus to researchers seeking to build on our work.

https://doi.org/10.1145/3576915.3623157

# AI-Generated Code is Causing Outages and Security Issues in Businesses

Published September 13, 2024 | Written by Fiona Jackson

**Tariq Shaukat, CEO of Sonar, is "hearing more and more" about companies that have used AI to write their code experiencing downtime.**

Businesses using artificial intelligence to generate code are experiencing downtime and security issues. The team at Sonar, a provider of code quality and security products, has heard first-hand stories of consistent outages at even major financial institutions where the developers responsible for the code blame the AI.

## Could 'insufficient reviews' be a factor?

In late 2023, more than half of organisations said they encountered security issues with poor AI-generated code "sometimes" or "frequently," as per a survey by Snyk. But the issue could worsen, as 90% of enterprise software engineers will use AI code assistants by 2028, according to Gartner.

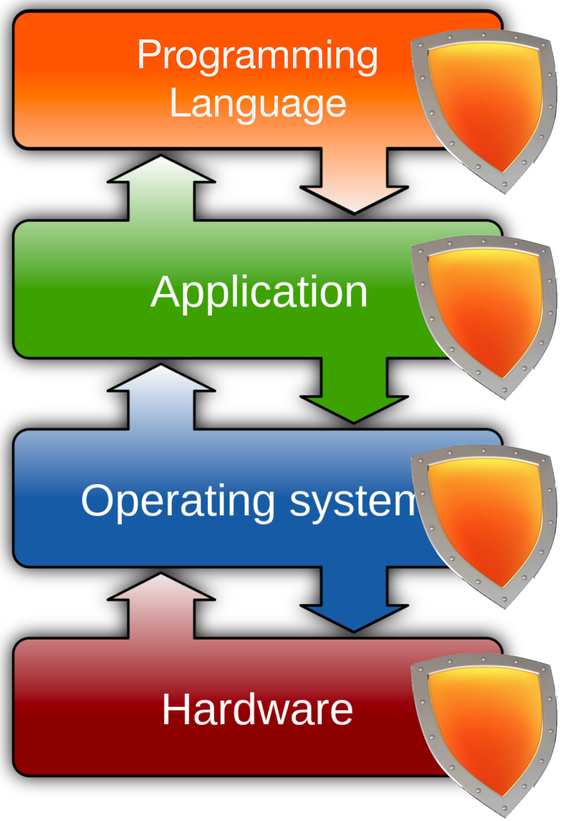
Tariq Shaukat, CEO of Sonar and a former president at Bumble and Google Cloud, is "hearing more and more about it" already. He told TechRepublic in an interview, "Companies are deploying AI-code generation tools more frequently, and the generated code is being put into production, causing outages and/or security issues.

"In general, this is due to insufficient reviews, either because the company has not implemented robust code quality and code-review practices, or because developers are scrutinising AI-written code less than they would scrutinise their own code.

https://www.techrepublic.com/article/ai-generated-code-outages/

# Defense in Depth

- There are many countermeasures at various levels
  - processor, compiler, programming language, ...
  - Automatic bound verification (e.g., Rust, Java, Python)
  - Randomisation of addresses, W^X, etc
  - Control-Flow Integrity
  - Trusted Execution Environments, etc
- But all countermeasures can fail, no definite fallback
  - We are still exposed to DoS attacks…
  - How to recover from crashes? Reposing the state? ⇒ Can facilitate brute-force 



# Acknowledgements


- This lecture's slides have been inspired by the following lectures:
  - CSE127: Low-level Software Security III: Integer overflow, ROP and CFI
  - CS155: Control Hijacking: Defenses
  - CS343: Control-flow hijacking defenses