Defences

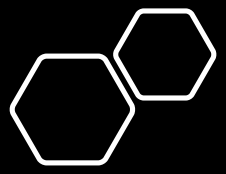
What have we learned

Certain features of programming languages can introduce security vulnerabilities, making systems susceptible to attacks.

Today, we will explore possible countermeasures to mitigate these risks

A first defence: awareness

State precisely what your program does



Defensive Programming

We look out for trouble before it happens, anticipate the unexpected, and never put ourselves into a position from which we can't extricate ourselves

Design by Contract: clients and suppliers must agree on rights and responsibilities.

The Pragmatic Programmer

From Journeyman to Master

Andrew Hunt

David Thomas

→ SW toolchains, Libraries, language, teams all contribute to security of the ecosystem

Security by construction

1. **Formal Methods & Verification** – Uses mathematical models to prove that a system adheres to security properties, reducing the likelihood of vulnerabilities.
2. **Secure Coding Practices** (→ C malloc) – Enforces strict adherence to secure programming techniques, such as memory safety and avoiding common pitfalls like buffer overflows.
3. **Least Privilege & Strong Typing** – Ensures that components and processes operate with minimal permissions, and strong type systems help prevent unintended behaviors.
4. **Design-Time Security Considerations** – Security is incorporated during system architecture and design phases rather than being bolted on later. (No patch on bad project)
5. **Automated Code Analysis & Tooling** – Uses static analysis, fuzz testing, and automated code reviews to detect potential security flaws early.
6. **Continuous Security Validation** – Implements secure development pipelines with ongoing security assessments.

↓ Provide feedback asap

DEFENCES (classical)

Apart from awareness, what else? Pulling patches: if a feature you have in a PL is bad, you need this.

Changing compilers and **run-time mechanisms** to prevent exploitation of security vulnerabilities.

↳ you will need to modify them

This has extra work on compilers

DEFENCES (Innovative)

The design of an evolutionary toolkits to support the design and development of secure programs enhances security within the software ecosystem by promoting design by construction.

This requires additional efforts in educating professionals to adopt secure design and development practices.

DEFENCES (Innovative)

STOP using wrong prog. languages unsecure, start with more secure ones.

Design and implement novel programming languages where security and memory safety IS a main concern from the very beginning

Classical defences

Mitigation 1: Security canaries

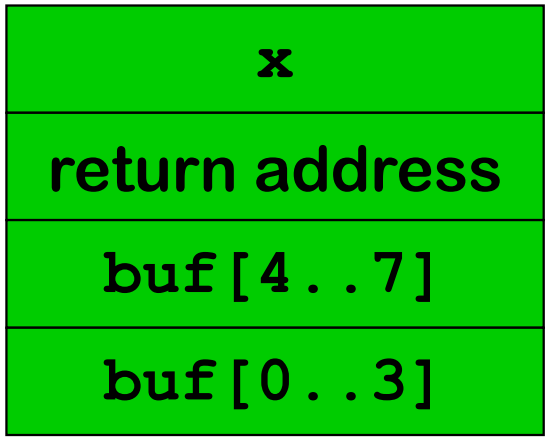
Security canaries are **runtime protection mechanisms** that detect **stack buffer overflows** before they can be exploited.

Security canaries act as **sentinels (canaries) placed in memory** to identify unauthorized modifications, similar to how **canary birds in coal mines** detected toxic gases.

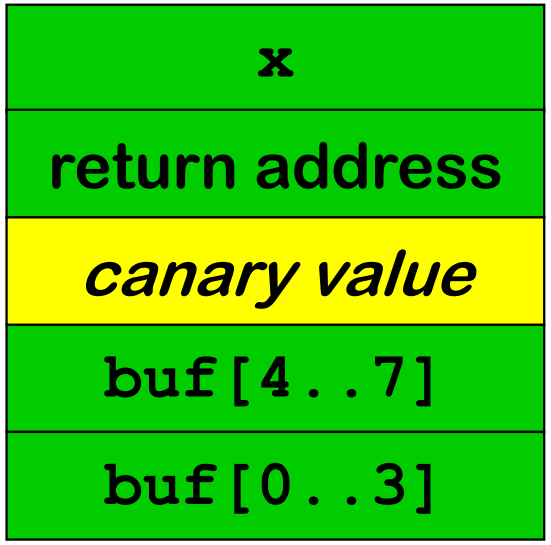
How canaries detect buffer overflow?

- **Canaries are values inserted in the activation record between local variables and the function return address on the stack**.
- If a buffer overflow occurs and an attacker **overwrites adjacent memory**, the canary value **is modified**.
- Before returning from a function, the program **checks if the canary has changed**.
  - If the value is **unchanged**, execution continues.
  - If the value is **modified**, the program **terminates immediately**, preventing exploitation.

Stack without canary



Stack with canary



What is a canary value?

Canaries are Random Values	Hard for attackers to predict.
• Generated randomly at program startup.	

Canaries in the compilation process

Canaries are run-time data structures to prevent **buffer overflow attacks**

The integration of **stack canaries** into the compilation process is a security enhancement.

The **compiler instruments the code to generate, insert, and verify canary values** at runtime.

Compilation steps

Canary Generation (Program Startup)

When the program starts executing, a **random canary value** is generated.

This value is **stored in a global memory location** (e.g., fs:0x28 on Linux for x86-64 systems).

Compilation steps

Canary Insertion (Prologue)

The compiler **modifies function entry code (prologue)** to insert the canary **before the return address**.

This ensures that **any overflow from local variables must first overwrite the canary** before reaching the return address.

Compilation steps

Canary Insertion (Example)

```
mov    %fs:0x28, %rax
# Load global canary value
```

At assembler level:

Prologue

```
mov    %rax, -0x8(%rbp)
# Store it onto the stack
```

Compilation steps

Canary Verification (Epilogue)

Before returning from a function, the compiler **inserts in the prologue code a check** to see if the canary is unchanged.

If the canary value **was modified**, the program triggers a **security alert and aborts execution**

Compilation steps

Canary Verification (Example)

```
mov    -0x8(%rbp), %rdx
# Load stored canary
xor    %rdx, %fs:0x28
# Compare with original value
jne    stack_smash
# If changed, abort execution
```

If jne stack_smash is triggered, the program calls:

__stack_chk_fail();

This function **terminates execution** to prevent further exploitation.

Canaries: Discussion

- **Prevents return address overwrites** (common in buffer overflow exploits).
- **Widely deployed in modern systems** (Linux, Windows, macOS).

Canaries: Weakness

- **Heap overflows & format string attacks** are **not protected**.
- **Bypassing techniques**:
  - **Memory disclosure** (leaks canary value).
  - **ROP (Return-Oriented Programming)** attacks avoid direct return address modifications:
  - Instead of injecting new code, the attacker **chains small snippets of existing code (gadgets)** to execute malicious operations.

What is ROP?

ROP is a method where an attacker takes control of a program's execution flow without injecting new code. Instead of writing their own shellcode (which would be blocked by modern protections), they reuse small sequences of instructions—called **gadgets**—that are already present in the program's memory (usually in libraries like libc or the program's own code). Each gadget typically ends in a `ret` (return) instruction, allowing the attacker to "chain" them together to perform arbitrary actions.

What are ROP Attacks?

ROP attacks exploit vulnerabilities like **buffer overflows** to overwrite the stack and control the return address. Instead of pointing to a single attacker-controlled address, they carefully redirect execution through a sequence of gadgets. This allows them to **bypass security mechanisms** (like non-executable memory) and execute malicious operations without injecting new code.

> You're absolutely right! If the canary is intact, the system won't detect an issue when the function returns. But there's still a small detail: **even if the attacker leaks the canary and overwrites the return address correctly, the program would still return only once.**
>
> ### *How Does ROP Avoid Canary Checks?*
>
> The trick is that **ROP doesn't rely on a single return**—it **chains multiple returns** without triggering a function epilogue that checks the canary. Here's how:

How Attackers Bypass Stack Canaries

If an attacker can leak the canary value, the attacker can overwrite the buffer **without modifying the canary**, ensuring that the check passes.

Leaking Canaries: with fomat strings

```
void vulnerable_function() {
    char buffer[64];
    printf("Enter input: ");
    gets(buffer); //  Buffer Overflow risk!
    printf(buffer); // String vulnerability leaks stack data
}

int main() {
    vulnerable_function();
    return 0;
}
```

Leaking Canaries: with fomat strings

The attacker **input:**
%p-%p-%p-%p-%p

```
void vulnerable_function() {
    char buffer[64];
    printf("Enter input: ");
    gets(buffer); //  Buffer Overflow risk!
    printf(buffer); // String vulnerability leaks stack data
}

int main() {
    vulnerable_function();
    return 0;
}
```

Leaking Canaries: with fomat strings

The attacker **input:**
%p-%p-%p-%p-%p

```
void vulnerable_function() {
    char buffer[64];
    printf("Enter input: ");
    gets(buffer); //  Buffer Overflow risk!
    printf(buffer); // String vulnerability leaks stack data
}

int main() {
    vulnerable_function();
    return 0;
}
```

Leaking Canaries: with fomat strings

The attacker **input:**
%p-%p-%p-%p-%p

```
void vulnerable_function() {
    char buffer[64];
    printf("Enter input: ");
    gets(buffer); //  Buffer Overflow risk!
    printf(buffer); // String vulnerability leaks stack data
}

int main() {
    vulnerable_function();
    return 0;
}
```

%p-%p-%p-%p-%p?
the format specifier **%p** prints **memory addresses**.
The program executed uses printf() with **%p-%p-%p-%p-%p** to **print out stack values**, including the **stack canary**

%p: print a memory address. I will display the structure of the stack and get access to it

Another example

```
void secret_function() {
    printf("You've bypassed the stack canary!\n");
    system("/bin/sh"); // Spawns a shell
}

void vulnerable_function() {
    char buffer[64];
    printf("Enter input: ");
    gets(buffer); // Buffer Overflow risk!
}

int main() {
    vulnerable_function();
    return 0;
}
```

Attacker knows that the canary is always put right after the return address. But what if I modify a function pointer without touching canary vs point? Attacker must know layout of AR.

Bypassing canaries: predicting memory addresses

- The attacker can **predict memory addresses**, making bypassing stack canaries easier.
- The issues: the stack canary is always at a predictable memory location.
  - If attackers can leak stack memory (e.g., format string attack), they can directly read the canary value.
  - With the known canary value, attackers can overwrite return addresses without triggering stack protection.

Example

```
void secret_function() {
    printf("You've bypassed the stack canary!\n");
    system("/bin/sh"); // Spawns a shell
}

void vulnerable_function() {
    char buffer[64];
    printf("Enter input: ");
    gets(buffer); // Buffer Overflow risk!
}

int main() {
    vulnerable_function();
    return 0;
}
```

The attck strategy:
overwrite a saved function pointer near the buffer.
Redirect execution to secret_function(), bypassing the canary check.

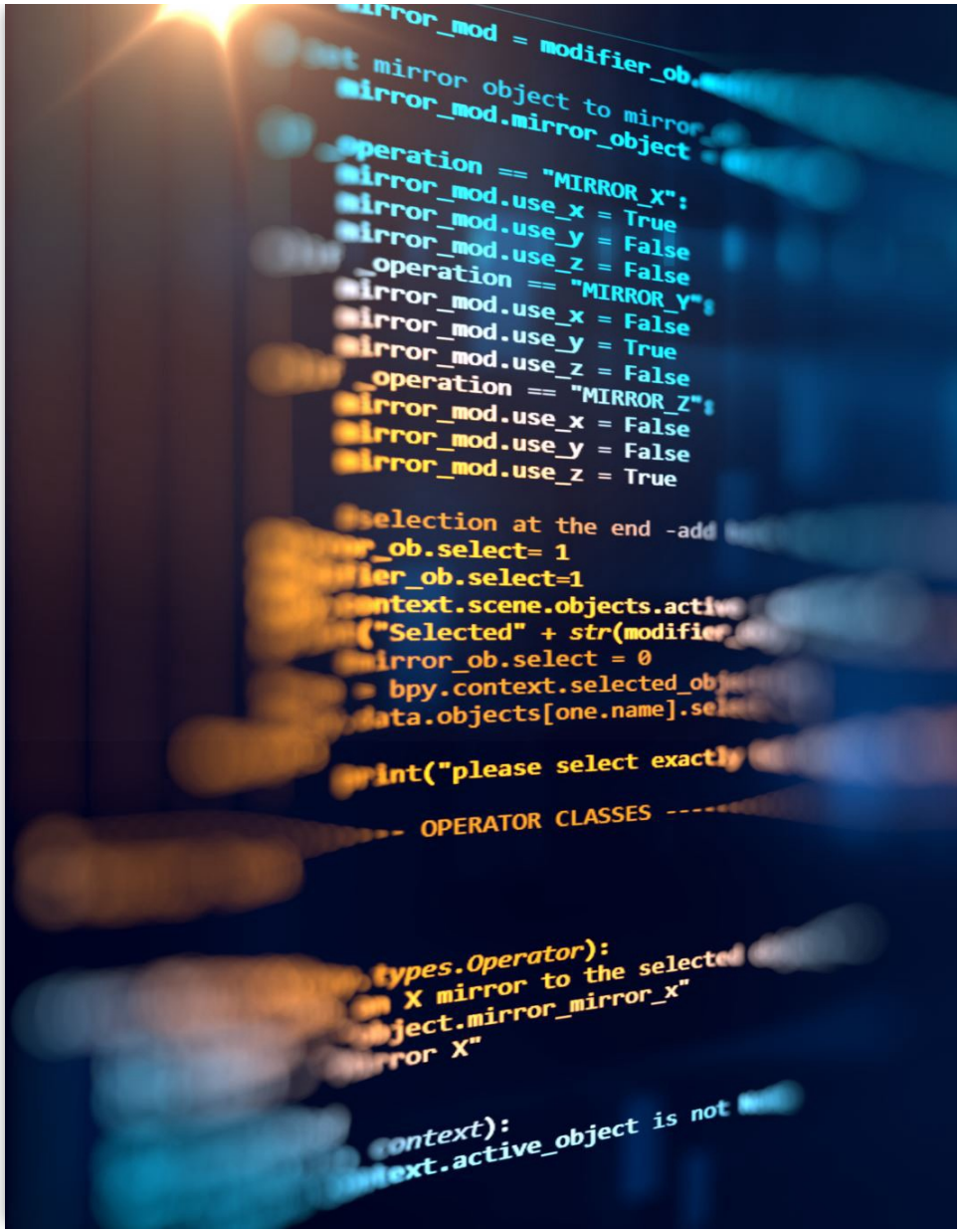
Address Space Layout Randomization (ASLR)

- **Address Space Layout Randomization (ASLR)** is a security technique used to **randomize memory addresses** of program components **at runtime**.
- This makes it **harder for attackers** to exploit memory-based vulnerabilities, such as **buffer overflows** and **return-oriented programming (ROP) attacks**.
- **Key Idea**: If an attacker **doesn't know where key memory structures are located**, they **can't reliably overwrite them** or execute malicious payloads.

How ASLR Work

Every time a program runs,
ASLR **randomizes** the locations of:

- **Stack**
- **Heap**
- **Shared libraries**
- **Executable binary**
- **Memory-mapped files**



0x400000 - Executable Code
0x600000 - Data Segment
0x7fff0000 - Stack
0x08048000 - Heap

Predictable Memory Layout

0x45d20000 - Executable Code
0x6af50000 - Data Segment
0x77bf9000 - Stack
0x08e34000 - Heap

Randomized Memory Layout

Today we work with them all randomized

Address Space Layout Randomization

- Attacker needs detailed info about memory layout
  - to jump to specific piece of code
  - to corrupt a pointer at known position on the stack
- Attacks become harder if compilers randomise the memory layout every time we start a program
  - change the offset of the heap, stack, etc, in memory by some random value
- Attackers can still analyse memory layout on their own machine, but will have to determine the offsets used on the victim's machine to carry out an attack

ASLR Benefits

- **Prevents Return-Oriented Programming (ROP) Attacks**
  - ROP attacks rely on **fixed addresses of instructions** to execute arbitrary code. With ASLR, attackers **cannot reliably predict gadget locations**.
- **Mitigates Buffer Overflow Exploits**
  - Overwriting a return address **without knowing its exact location** causes crashes instead of successful exploitation.
- **Prevents Memory Disclosure Attacks**
  - Even if an attacker leaks memory addresses, **they only work for the current execution**.
  - Next execution = **new random addresses**.

Shadow Stack

Extension of the run time data structure (stack) in the abstract machine

Only used to store return addresses

Mitigating Control Flow Attacks

Control flow attacks, such stack-based buffer overflows, exploit vulnerabilities in the program's control flow to execute malicious code.

A **shadow stack** is a security mechanism designed to counter such attacks by maintaining an independent, protected copy of return addresses.

How the Shadow Stack Works

A shadow stack ius a run time data structure that operates as follows:

Call Instruction: When a function is called, the return address is pushed onto both the normal stack and the shadow stack.

Return Instruction: Before executing a return instruction, the system verifies that the return address on the normal stack matches the corresponding entry in the shadow stack.

Mismatch Detection: If the return addresses do not match, it indicates a possible control flow attack, and the program is terminated or other security measures are triggered.

Protected Memory Region: The shadow stack is stored in a protected memory region that prevents unauthorized writes, ensuring its integrity.

Discussion

- **Performance Overhead**: Maintaining a separate stack and performing additional checks introduce some runtime overhead.
- **Compatibility Issues**: Legacy applications and certain optimization techniques may not be fully compatible with shadow stacks.
- **Bypass Techniques**: Sophisticated attackers may attempt to locate and tamper with the shadow stack, requiring further protections like hardware-based enforcement.

Discussion

- **Intel Control-Flow Enforcement Technology (CET)**: Includes shadow stack support to protect return addresses at the hardware level.
- **Compiler-Based Approaches**: Certain compilers (e.g., LLVM) implement software-based shadow stacks for enhanced security.

What we have learned

Run-time organization and code generation

Stack & Heap

- Stack Canaries
- ASLR
- Shadow stack

Code instrumentation

↳ generated by compiler should be written to use these solutions

Impact on compiler and runtime organization

Non Executable Memory

- **Non-executable (NX) memory** is a security feature that prevents certain memory regions from executing code, thereby mitigating exploits like **buffer overflows** and **code injection attacks**.
- Implemented using hardware-supports.

How NX works

- **Memory Segmentation**: The operating system marks specific memory regions (e.g., stack, heap) as non-executable.
- **Execution Prevention**: If an attacker injects and tries to execute malicious code in these protected regions, the CPU generates a **segmentation fault** or access violation.
- **Legitimate Execution Allowed**: Code execution is restricted to designated areas like text (code) segments, while writable regions remain non-executable.

Discussion

The interpreter refuses to execute non-executable code

Attackers can then no longer jump to their own attack code, as any input provide as attack code will be non-executable

Enhances Overall System Security: Reduces the attack surface by enforcing strict execution policies.

Regions now are strict.

But …

NX memory does not work for JIT (Just In Time) compilation, where e.g. JavaScript is compiled to machine code at run time.

Just-In-Time (JIT) Compilation

- **JIT compilation** is a technique where code is **compiled at runtime** instead of ahead of time. This allows languages like JavaScript or Java to convert high-level code into machine code **just before execution** rather than being pre-compiled.

The Conflict

The issue arises because **JIT compilation requires the ability to execute code in places that might be marked as non-executable**. For example:

- When a JavaScript engine (like V8 in Chrome or SpiderMonkey in Firefox) executes JavaScript code, it often converts the script into **machine code** dynamically using JIT compilation for performance.
- This **dynamically generated machine code** needs to be **executed** in memory. If the memory where this machine code is generated is **non-executable**, the system will prevent it from running, causing an issue for JIT-based environments.

2. Performance Considerations

- JIT compilers are designed to run code **on-the-fly**, optimizing for performance and responsiveness. This means they typically generate machine code in memory locations that are **closer** to the execution context (e.g., within the runtime memory of a web page, application, or script).
- Relocating this code to predefined, protected executable sections would **introduce overhead** and might reduce the performance benefit of JIT, because JIT compilation is meant to be fast and flexible. It would require extra steps to manage where and how the code is generated and executed.

Enclaves

Security enclaves

HW trusted execution environment

A **security enclave** is a **protected execution environment** designed to isolate sensitive computations and data from the rest of the system.

Enclaves isolates part of the code together with its data

- Code outside the enclave cannot access the enclave's data
- Code outside the enclave can only jump to valid entry points for code inside the enclave

Security enclaves ensure **confidentiality, integrity, and code authenticity**, making them crucial for building **trusted execution environments**.

Code outside the island cannot assess the enclave.
You enclose code and data in an island. Code of attacker is outside the enclave, so attacker cannot run what's in the enclave

Example: security-sensitive code in larger program

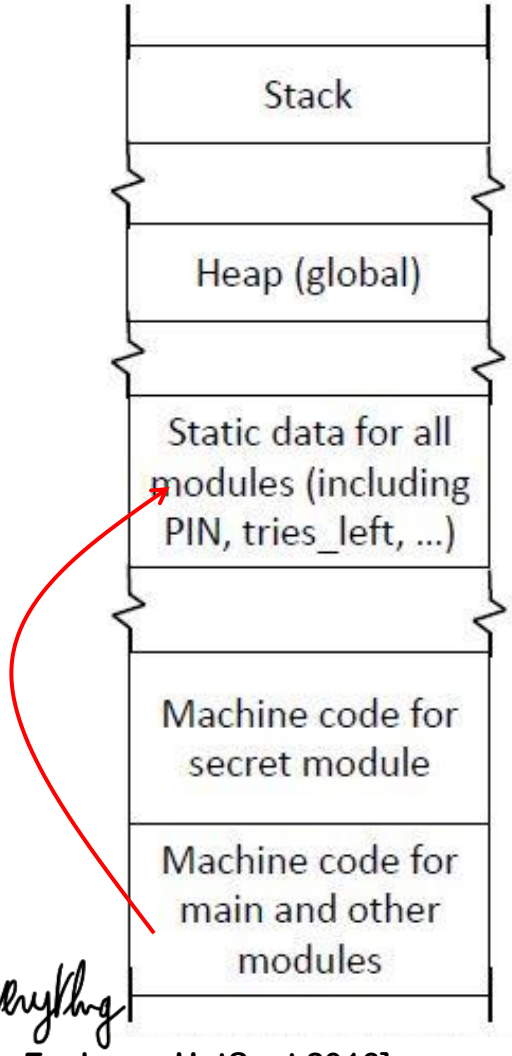
secret.c

```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 int get_secret (int pin_guess) {
   if (tries_left > 0) {
    if ( PIN == pin_guess) {
      tries_left = 3; return secret; }
    else {
      tries_left--; return 0 ;}
 } }
```

main.c

```
# include "secret.h"
... // other modules
void main () {
...
}
```

Bugs or malicious code *anywhere* in the program could access the high-security data



Example from [N. van Ginkel et al, Towards Safe Enclaves, HotSpot 2016]

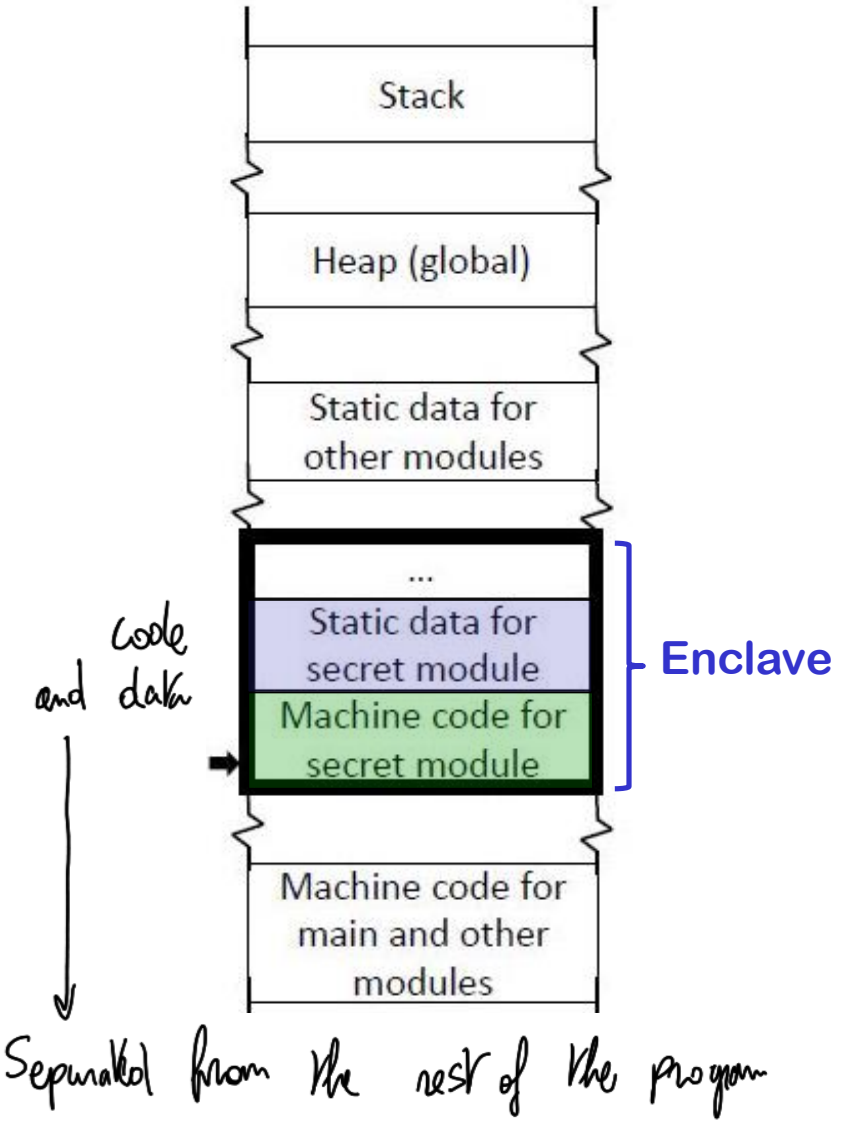
Isolating security-sensitive code with secure enclaves

secret.c

```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 int get_secret (int pin_guess) {
   if (tries_left > 0) {
    if ( PIN == pin_guess) {
      tries_left = 3; return secret; }
    else {
      tries_left--; return 0 ;}
 } }
```

main.c

```
# include "secret.h"
... // other modules
void main () {
...
}
```



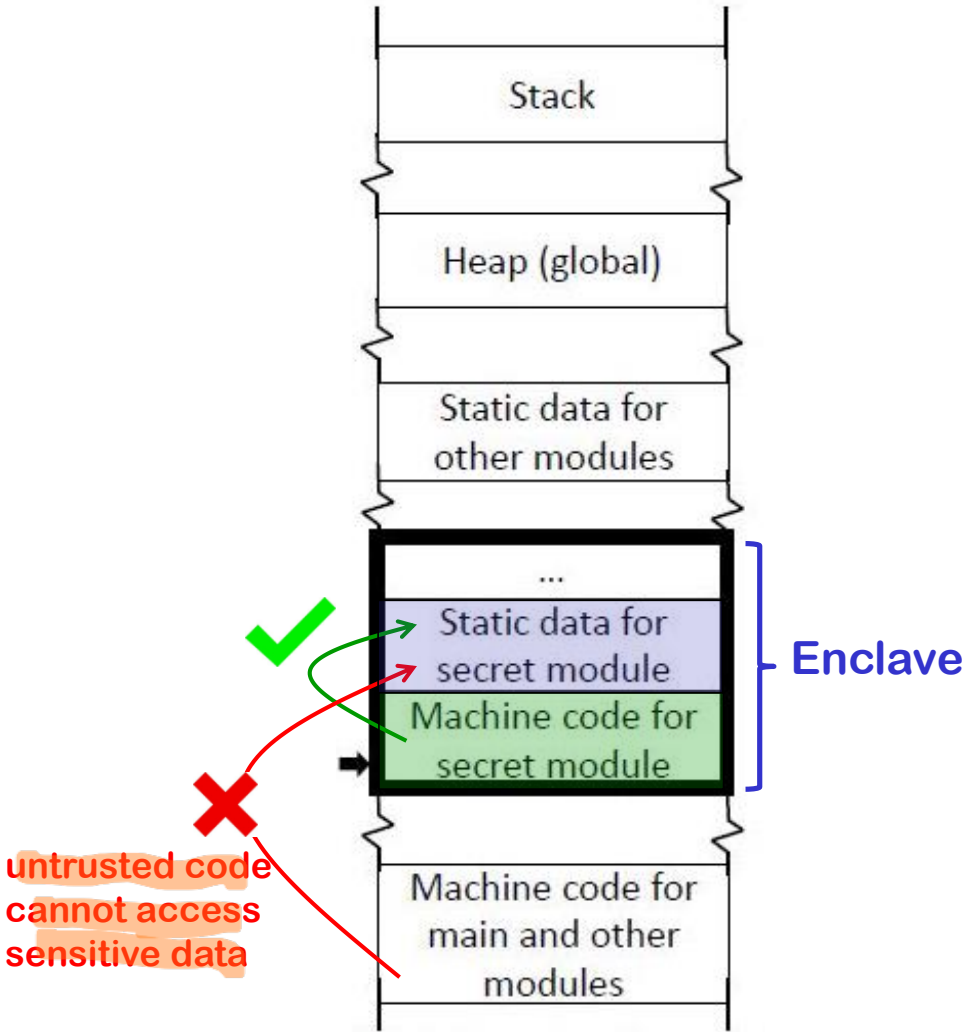
Isolating security-sensitive code with secure enclaves

secret.c

```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 int get_secret (int pin_guess) {
   if (tries_left > 0) {
    if ( PIN == pin_guess) {
      tries_left = 3; return secret; }
    else {
      tries_left--; return 0 ;}
 } }
```

main.c

```
# include "secret.h"
... // other modules
void main () {
...
}
```



Isolating security-sensitive code with secure enclaves

secret.c

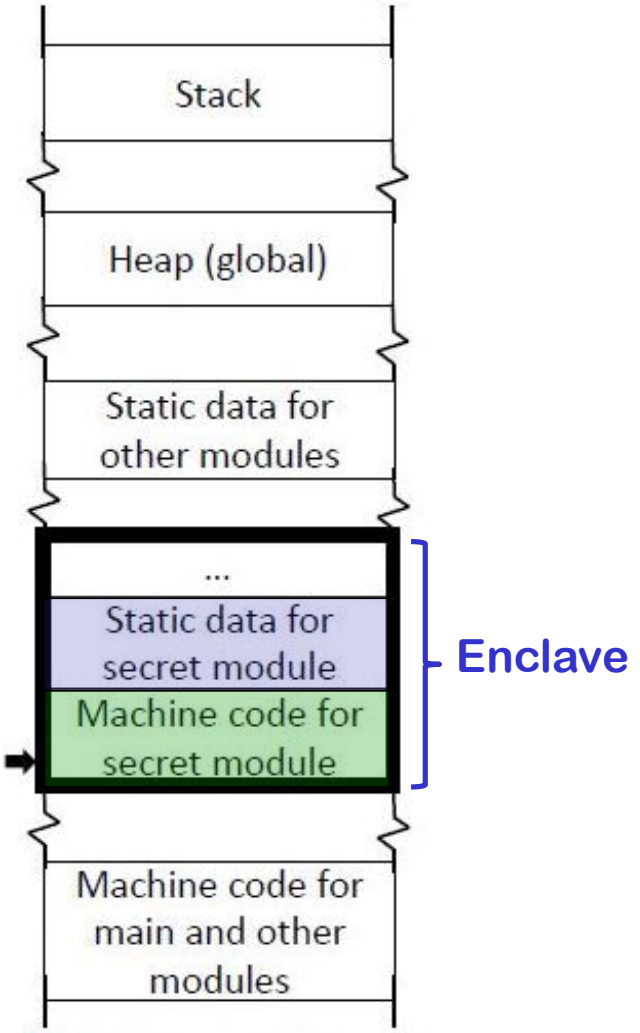
```
static int tries_left = 3;
 static int PIN = 1234;
 static int secret = 666;

 int get_secret (int pin_guess) {
   if (tries_left > 0) {
    if ( PIN == pin_guess) {
      tries_left = 3; return secret; }
    else {
      tries_left--; return 0 ;}
 } }
```

main.c

```
# include "secret.h"
… // other modules
void main () {
…
}
```

Only allowed entry point (for get_secret)

Untrusted code should not be able to jump to the middle of get_secret code (recall return-to-libc & ROP attacks)



How enclave work

- **Isolated Execution**: Code and data inside the enclave are protected from external access, including from privileged software like the OS and hypervisor.
- **Hardware-Based Protection**: Modern CPUs (e.g., Intel SGX) provide dedicated instructions to create and manage enclaves.
- **Safe Memory**: Data in an enclave is stored in encrypted memory, preventing unauthorized access.
- **Attestation**: Enclaves can cryptographically prove their integrity to external parties, ensuring they haven't been tampered with.

↳ a sort of certificate proving their integrity

Secure enclave: hardware support

- **Intel SGX (Software Guard Extensions)**: Hardware-supported enclaves for isolated execution.
- **AMD SEV (Secure Encrypted Virtualization)**: Protects VMs from host-level threats.
- **ARM TrustZone**: Provides a separate secure execution environment for mobile and IoT devices.
- **Apple Secure Enclave**: A dedicated co-processor for secure biometric authentication and cryptographic operations.

AMD TrustZone: key features

Hardware-Enforced Isolation: The processor is divided into two execution environments:

- **Secure World**: Runs trusted code, handling sensitive operations like cryptographic functions and secure boot.
- **Normal World**: Runs regular applications and the operating system but has restricted access to secure resources.

Secure Boot & Firmware Protection: FW prot. to avoid modification

- Ensures that only verified and signed firmware can execute, protecting against unauthorized modifications or malware injections.

Trusted Applications & Key Management:

- Enables secure execution of applications handling sensitive data, such as **payment processing, biometric authentication, and digital rights management (DRM)**.
- Secure storage and management of cryptographic keys to prevent tampering or extraction.

Memory and I/O Security:

- Secure memory regions prevent unauthorized access to critical data, even if the normal OS is compromised.
- Protects communication between system components (e.g., CPU, GPU, peripherals).

This is a trusted execution environment

Run-time & memory organization

code generation

Stack, Heap and code area

- Stack Canaries
- Reorder layout of AR and Heap elements (randomization)
- Shadow stack
- NX memory
- Enclave

Code instrumentation

↳ Gen. of the code has to be modified (for ex. to exploit enclaves)

Code Instrumentation as a Security Mechanism

use it as SW mech. to protect code with respect to certain attacks

- **Code instrumentation** is a security technique that involves modifying a program's code—either at compile-time, runtime, or dynamically—to **monitor, analyze, or enforce security policies.** (dynamically)
- It is widely used for **detecting vulnerabilities, preventing exploits, and enforcing security policies**.

Code instrumentation examples

- The management of canaries and shadow stack are examples of security instrumentation

How to instrument pointer?

QUITE TRICKY!!!

Which is the weakness of pointers? With pointer anywhere you can access everything

Cyclone

level of C prog. language, but in a more secure approach

Cyclone is a **safe dialect of C** designed to prevent common vulnerabilities like buffer overflows, dangling pointers, and format string attacks.

One of its notable features is the use of **fat pointers**, which enhance memory safety while maintaining efficiency.

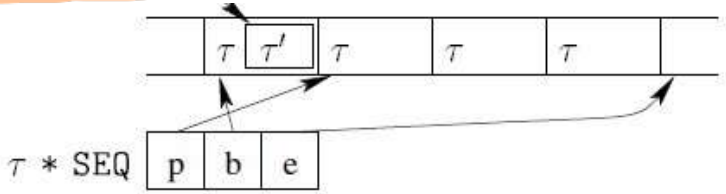
This is SW instrumentation
Rust exploits the fat pointers idea

FAT POINTERS

- Idea: change the representation of pointers to carry bounds information
  - "int *p" becomes

    struct st {int *ptr, int *b, int *e};

    struct st p;
  - the field b points to the beginning of the buffer, and e the end of buffer



MEMORY OPS

- "x=*p" becomes

```
if (p.ptr < p.b || p.ptr > p.e) abort();
x= * (p.ptr);
```

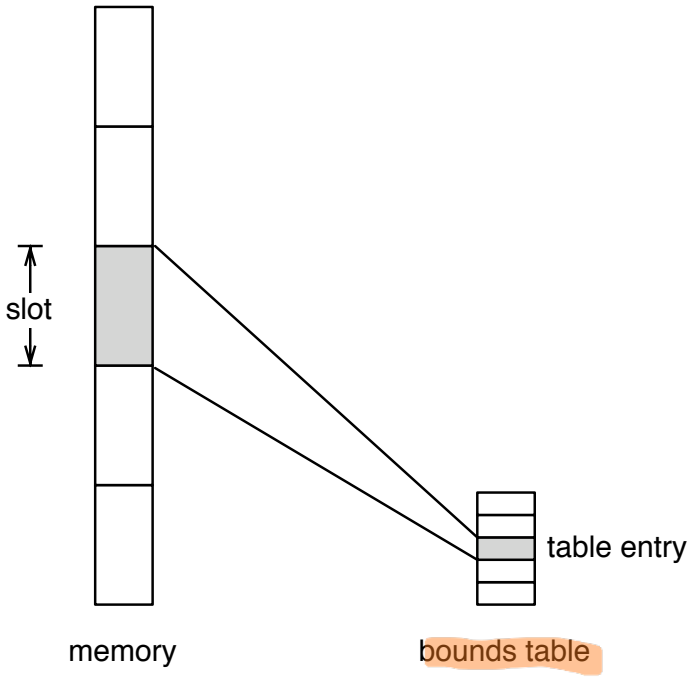
Ptrs characterise the region of a memory

Bounds Tracking : any op. has to check the boundaries of a region

- Fat pointers in Cyclone store **both a base address and a length**. This ensures that any access using the pointer stays within allocated bounds.
- The compiler and the runtime enforce boundary checks, preventing buffer overflows.

FAT POINTERS: bounds checking

- Compiler-based instrumentation to enforce spatial memory safety for C/C++.
- The general idea is to keep information about all pointers in *disjoint* metadata, indexed by pointer location



Is This Related to Fat Pointers?

Yes, but it's **not exactly the same thing**. **Fat pointers** are a specific solution where the **pointer itself** contains the metadata, but this slide seems to be describing an **alternative** or **complementary** approach where the metadata is **separated** from the pointer. Instead of modifying the pointer itself, the compiler tracks metadata about each pointer separately.

2. Indexed by Pointer Location

- **Metadata** related to a pointer (like its size, bounds, or type information) is stored separately, not with the pointer itself.
- This metadata is "**indexed**" by the **location** where the pointer is stored in memory.
  - For example, if the pointer `p` is stored at memory location `0x1000`, the metadata (like the bounds of the memory region it points to) could be stored in a table where the entry at `index 0x1000` corresponds to `p`.
- In simpler terms, when the program uses a pointer (e.g., dereferences it), the system can **look up** the metadata for that pointer by **finding the pointer's location in memory** and retrieving the associated metadata from the metadata table.

Discussion

- **Preventing Dangling Pointers:** Cyclone's fat pointers integrate with its **region-based memory management**, ensuring that pointers remain valid only within their allocated lifetime.
- **Safe Array and String Operations:** Cyclone replaces **raw C-style arrays and strings** with fat pointers that carry length metadata.
  - Functions operating on strings and arrays use this metadata to prevent unsafe memory access.
- **NULL and Uninitialized Pointer Safety:** Cyclone's fat pointers **cannot be used unless they are properly initialized**.
- **Performance Overhead:** Since fat pointers include additional metadata, they consume more memory than traditional C pointers. However, Cyclone optimizes their usage to reduce unnecessary runtime checks when safety can be statically guaranteed.

We gain in memory safety, lose in expressivity (limitation of arithmetics). Ops for buffers, strings are safer;

Run-time & memory organization

code generation

Stack, Heap and code area

- Stack Canaries
- Reorder layout of AR and Heap elements (randomization)
- Shadow stack
- NX memory
- Enclaves

Code Instrumentation

- FAT Pointers

Security Instrumentation: examples

- **Detecting Buffer Overflows** – Instruments code to track memory accesses and prevent overflows (e.g Java and JVM)
- **Enforcing Control Flow Integrity (CFI)** – Ensures execution follows a valid control flow path (later).
- **Runtime Behavior Analysis** – Detects suspicious activity such as unauthorized system calls or memory modifications (later)

(MY) SUMMARY

Ecosystem

Principles and implementation guidance for software safety and security are well established and broadly accepted.

- Despite these efforts, common types of software defects prevail, and many occupy top ranks of “worst vulnerabilities” lists such as the OWASP Top 10 or the CWE Top 25 Most Dangerous Software Weaknesses for years if not decades.

Common types of defects are introduced during design, development, and deployment they arises from the structure of the *developer ecosystem*

- the end-to-end collection of tools and processes in which developers design, implement, and deploy software. This includes programming languages, software libraries, application frameworks, source repositories, build and deployment tooling and so forth.

SW is part of the echosystem
SW=code, PL, Toolkits associated etc.

Memory Safety and Security Here

Promote the use of memory-safe programming to reduce attack surface.

A gentle introduction to RUST

Promote the adoption of formal methods and related toolkits to detect much earlier rather than tracing back from externally visible systems failures.

A gentle introduction to formal methods