

## DISCUSSION STACK CANARIES

A simple function

```
void a_function(const char* input) {  
    char buffer[12];  
    strcpy(buffer, input);  
}
```

A simple function

```
void a_function(const char* input) {  
    char buffer[12];  
    strcpy(buffer, input);  
}
```

Assume that stack canaries are enabled,  
the compiler (eg gcc and Clang) will automatically  
transform the program to manage canaries into the prologue and epilogue.

A simple function

```
void a_function(const char* input) {  
    char buffer[12];  
    strcpy(buffer, input);  
}
```

The canary-enabled prologue must push the canary on the stack, immediately after the control data (return address) and before the allocation of the local variables. The canary-enabled epilogue must check that the canary still contains the original value, before restoring the saved control information and returning.

queste sono informazioni globali il che vuol dire che sono informazioni di strutture dati presenti nel runtime, vuol dire che per operare queste trasformazioni il compilatore ha bisogno di estendere il tradizionale run time support per poter fare generazione del codice ed eseguire il codice oggetto generato tenendo conto delle nuove potenzialità di esecuzione

# The compiler transformation (more or less)

```
extern uintptr_t __stack_chk_guard;  
noreturn void __stack_chk_fail(void);
```

Variabile globale che tiene le informazioni relative al valore usato da mettere nella canaries (random, xor tra il punto di ritorno e un valore random etc..)

funzione che non restituisce il controllo al chiamante e viene invocata quando si scopre che la canaries è stata modificata, vuol dire che siamo potenzialmente in una situazione in cui c'è stato un attacco

```
void a_function(const char* input) {
```

```
    uintptr_t canary = __stack_chk_guard;
```

```
    char buffer[12];
```

```
    strcpy(buffer, input);
```

```
    if ((canary = canary ^ __stack_chk_guard) != 0 ) {  
        __stack_chk_fail(); } }
```

la canary viene introdotta prima della esecuzione della funzione e viene data alla canary il valore di questa variabile globale. Si fanno le operazioni e poi si va a fare il controllo che il valore della canary e il valore del canary guard siano identici, viene fatto con un operazione di xor bitwise, se da un risultato diverso da 0 vuol dire che sono diversi dunque c'è stato un attacco

# The compiler transformation (more or less)

```
extern uintptr_t __stack_chk_guard;  
noreturn void __stack_chk_fail(void);
```

```
void a_function(const char* input) {
```

```
    uintptr_t canary = __stack_chk_guard;
```

```
    char buffer[12];  
    strcpy(buffer, input);
```

```
    if ((canary = canary ^ __stack_chk_guard) != 0) {  
        __stack_chk_fail();  
    } }
```

il prologo deve mettere il valore della canary ovvero deve prendere il valore della canary usando le informazioni disponibili nel run time support per generare valori in modo trusted della canary.

→ PROLOGUE

→ EPILOGUE

L'epilogo fa le operazioni di controllo, ovvero se il valore della canary non è stato modificato da un potenziale overflow dunque da un attaccante, se scopre che c'è stato un potenziale attacco deve abortire il programma

# The compiler transformation (more or less)

```
extern uintptr_t __stack_chk_guard;
noreturn void __stack_chk_fail(void);

void a_function(const char* input) {

    uintptr_t canary = __stack_chk_guard;

    char buffer[12];
    strcpy(buffer, input);

    if ((canary = canary ^ __stack_chk_guard) != 0 ) {
        __stack_chk_fail(); } }
```

→ contains the value of the stack protection canary word

→ callback function invoked when a stack buffer overflow is detected

# The compiler transformation (more or less)

```
extern uintptr_t __stack_chk_guard;
noreturn void __stack_chk_fail(void);

void a_function(const char* input) {

    uintptr_t canary = __stack_chk_guard;

    char buffer[12];
    strcpy(buffer, input);

    if ((canary = canary ^ __stack_chk_guard) != 0 ) {
        __stack_chk_fail();
    }
}
```

contains the value of the stack protection canary word

callback function invoked when a stack buffer overflow is detected

EXTENSION OF THE  
RUN-TIME SUPPORT

Mentre lo stack del runtime può essere attaccato, ovvero lo stack del runtime non è una componente della trusted computer base dell'implementazione del linguaggio della programmazione, allora se vogliamo che il meccanismo delle canaries funzioni adeguatamente, dobbiamo assumere in modo particolare che non si possa corrompere il valore della canaries ovvero che rimanga trusted, ovvero stiamo assumendo che questi 2 componenti in figura fanno parte della trusted computer base dell'esecuzione del linguaggio.



# The gcc output prologue

1. **pushq %rbp**
2. **movq %rsp, %rbp**
3. *;push other registers, if needed*
- 4 **subq \$x, %rsp** *;space for variables + canary*
5. **movq %fs:0x28, %rax**
6. **movq %rax, -8(%rbp)**

**Line 5 reads the global canary from offset 0x28**

**Line 6 copies the canary just above the frame pointer.**

# The gcc output: prologue

1. **pushq %rbp**
2. **movq %rsp, %rbp**
3. *;push other registers, if needed*
- 4 **subq \$x, %rsp** *;space for variables + canary*
5. **movq %fs:0x28, %rax**
6. **movq %rax, -8(%rbp)**

THE RUN-TIME SUPPORT BECOMES  
THE TCB

**Line 5 reads the global canary from offset 0x28**

**Line 6 copies the canary just above the frame pointer.**

# The gcc output: epilogue

1. **mov** -8(%rbp), %rax ;*load the canary*

2. **xor** %fs:0x28, %rax ;*compare with the global one*

3. **je** good

4. **call** \_\_stack\_chk\_fail@plt ;*modified: abort*

5. **good:**

6. **leave**

7. **ret**



indica che il loader deve caricare anche le metainformazioni dove troverà le funzioni per il support della defention dall'attacco

# Discussion

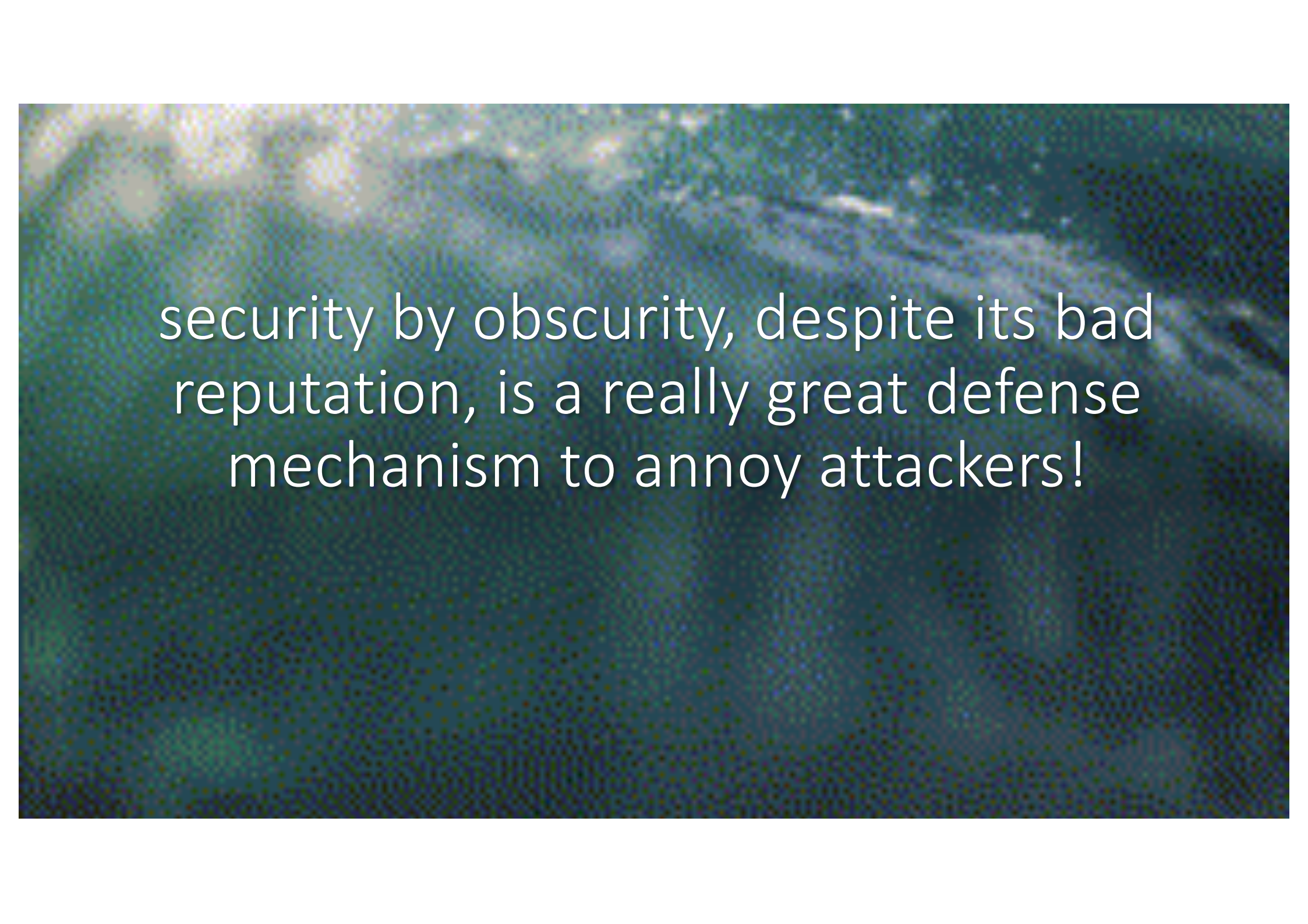
- An information disclosure flaw in a different part of the program could disclose the global **\_\_stack\_chk\_guard** value. This would allow an attacker to write the correct canary value and overwrite the function return address.
- Canaries cannot prevent heap-based buffer overflows.
- The attacker can try to write to the stack frame after the canary, without overwriting the canary value itself.

## Discussion (2)

- If a function has multiple local data structures and pointers to functions, these are allocated on the stack as well, before the canary value.
  - A buffer overflow in any one of these structures can be used by the attacker to get arbitrary code execution.
  - A main compiler issues: the arrangement of data on the stack.

## Discussion (3)

- Multi-threaded programs: the canary `__stack_chk_guard` is typically stored in Thread Local Storage, usually located after the end of the thread's stack.
  - A sufficiently large overflow can overwrite both canary and `__stack_chk_guard` to the same value, causing the detection to incorrectly fail.



security by obscurity, despite its bad reputation, is a really great defense mechanism to annoy attackers!

# Non-eXecutable memory (NX)

**X: executable memory**

- for storing code)

**W: writeable, non-executable memory**

- for storing data

**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**

Intel calls it eXecute-Disable (XD)

AMD calls it Enhanced Virus Protection

un interprete tradizionale sostanzialmente è un astrazione del ciclo fetch-execute visto a livello di linguaggio di programmazione che si sta eseguendo dunque carica l'istruzione corrente, la decodifica e la esegue. Se l'istruzione corrente proviene da un'area di memoria eseguibile allora l'interprete funziona come prima altrimenti l'interprete genera un errore. Questa semplice modifica della struttura della memoria ha un impatto sull'implementazione a tempo di esecuzione della macchina virtuale di esecuzione del linguaggio perchè ha un impatto sull'interprete. E' chiaro che se si compila in codice oggetto che è direttamente eseguibile questa cosa qui è mitigata dal fatto che si usa subito il supporto hardware della macchina che arriva subito a far parte della macchina virtuale di esecuzione del linguaggio perchè essa coincide con la macchina hardware su cui viene compilato il programma. In caso di macchine virtuali, dove si compila su .NET o bytecode con Java ad esempio, questo ha un effetto sulla struttura della macchina virtuale, in questo caso l'attaccante non può iniettare codice che va a chiedere all'interprete di eseguire del codice che sta in una parte che non è certificata eseguibile quando viene caricato il programma in esecuzione. Dunque quella parte lì sarà codice non eseguibile.



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

## Code optimization: safer strategies

- More later!!

A blue banner with a folded ribbon effect on the left side, containing the text "But ...".

But ...

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

ovviamente la memoria strutturata in memoria solo leggibile e memoria eseguibile non va bene in quanto il codice che stiamo compilando a runtime è memorizzato in una parte dati che a quel punto è un dato non eseguibile e dunque rendendolo eseguibile fallisce questa contromisura

## Libc attacks

- With NX, code injection attacks no longer possible, but code reuse attacks still are...
- Attackers can no longer corrupt code or insert their own code, but can still corrupt code pointers
  - instead of jumping to own attack code corrupt return address to jump to existing code esp. library code in libc
- libc is a rich library that offers lots of functionality, eg. `system()`, `exec()`, which provides attackers with all they need...



## return oriented program **Ming** (ROP)



Next stage in evolution of attacks, as people removed or protected dangerous libc calls such as `system()`

Instead of using entire library call, attackers can

- look for **gadgets**, small snippets of code which end with a return, in the existing code base


```
...; ins1 ; ins2 ; ins3 ; ret
```

- chain these gadgets together as subroutines to form a program that does what they want

This turns out to be doable

- Most libraries contain enough gadgets to provide a **Turing complete programming language**
- **ROP compilers** can then translate arbitrary code to a string of these gadgets

A newer variant is Jump-Oriented Programming (JOP) which uses a different kind of code fragment as gadgets



A black and white photograph of a large splash of water or sand against a dark background. The splash is captured in mid-air, with many droplets and particles visible. The text 'Other compilation issues' and 'Control Flow' is overlaid in white, centered on the image.

# Other compilation issues

## Control Flow

## Control flow: function call

- A compiler translates function calls in source code to **call <address>** in machine code where <address> is the location of the code for the function.
  - **For a function call f(...) in C a static address (or offset) of the code for f may be known at compile (static) time.**
- If compiler can hard-code this in the binary, it is hard for the attacker to mess with, esp. with NX

se fosse possibile, quando il compilatore genera il codice oggetto, staticamente l'offset della porzione di memoria dove verrà memorizzato il codice oggetto della funzione che viene chiamata, e lo riesce a determinare a static time, allora per un attaccante è difficile rompere il flusso di controllo della chiamata della funzione perchè questa parte viene messa in codice eseguibile in una porzione di memoria che è solo eseguibile, se il compilatore riesce a fare questa cosa staticamente abbiamo una buona sicurezza sul flusso di controllo. MA QUESTO VALE SOLO NEI LINGUAGGI IN CUI QUESTO SI PUO' FARE STATICAMENTE..

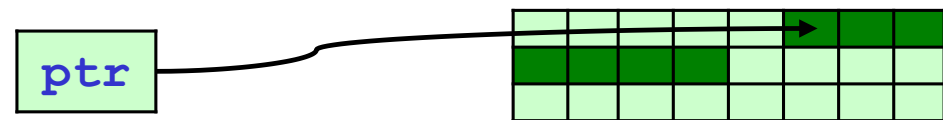
## Control flow: method dispatch

For a virtual function call **`o->m(...)`** in C++ the address of the code for the method **`m`** typically has to be determined at runtime, by inspecting the virtual method table (vtable). Even with NX attackers may be able to mess with (code pointers in) these tables

## More memory safety

Additional book keeping of meta-data  
& extra runtime checks to prevent illegal memory access

Different possibilities



- add information to **pointer** about size of **memory chunks** it points to (**fat pointers**)
- add information to **memory chunks** about their size (**Spatial safety with object bounds**)
- ...

In alcuni linguaggi di programmazione, la struttura dei puntatori è differente a runtime, per esempio alcuni linguaggi di programmazione, i descrittori di dato che sono presenti a tempo di esecuzione e servono per memorizzare i dati conservano maggiori informazioni rispetto alla normale.



## Fat pointers

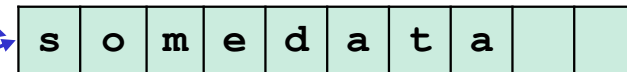
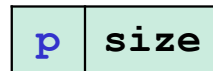
The compiler

- records size information for all pointers
- adds runtime checks for pointer arithmetic & array indexing

A pointer



A **fat pointer**



Downsides

- Considerable execution time overhead
- Not binary compatible – ie all code needs to be compiled to add

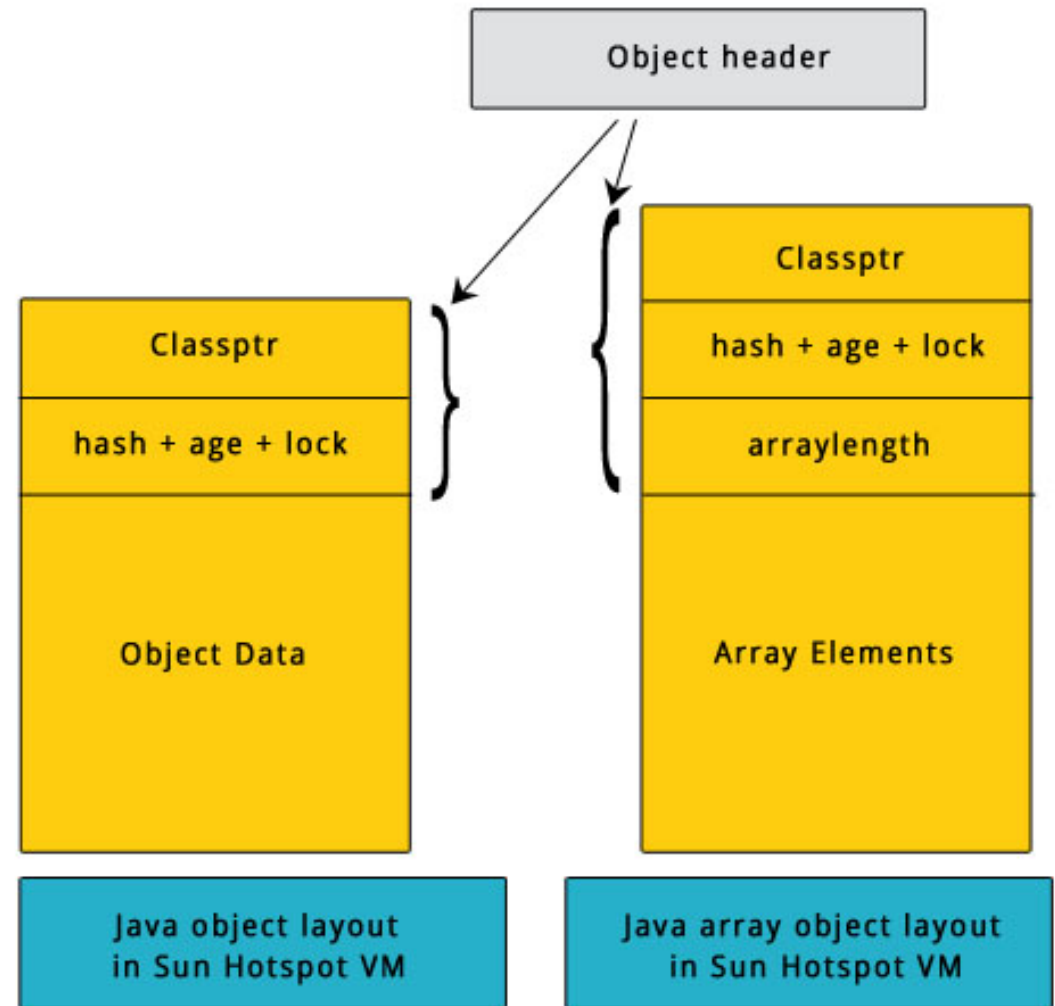
this book keeping for all pointers

se prendo una libreria binaria ed opero con questa, devo ricompilare tutto dall'inizio non posso semplicemente prendere il binario

# Typed data descriptor

- JVM: Typed data descriptors contain information about data

usando il puntatore si vuole accedere a un oggetto dell'array e vado ad accedere ad una posizione specifica, prima di accedere faccio un controllo se questa posizione è all'interno della dimensione, se non è all'interno della dimensione genero un'eccezione a runtime, e quindi anche in questo caso qui i descrittori permettono di evitare le operazioni che vanno al di fuori della dimensione dell'array ma stiamo pagando con un overhead dovendo fare un numero di controllo di tipi maggiore.



Se diverse funzioni si chiamano vicendevolmente, quello che potrebbe accadere è che l'attaccante si mette in mezzo e mette una funzione che è una funzione non corretta nel flusso di esecuzione che stiamo portando avanti. Esamina il flusso di esecuzione e se per caso scopre che ci sono dei flussi di controllo, ovvero le chiamate a ritorno non lecite, chiude il programma.

## Control Flow Integrity (CFI)

Extra bookkeeping & checks to spot unexpected control flow

- **Dynamic return integrity**

**Stack canaries**, or **shadow stack** that keeps copies of all return addresses, providing extra check against corruption of return addresses

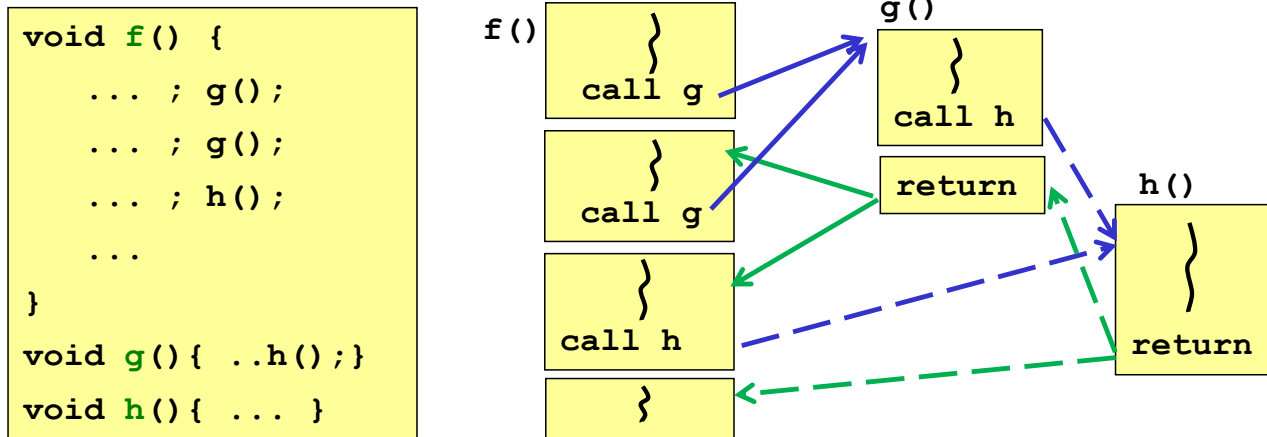
- **Static control flow integrity**

Idea: **determine the control flow graph (cfg) and monitor jumps in the control flow to spot deviant behavior**

If  $f()$  never calls  $g()$ , because  $g()$  does not even occur in the code of  $f()$ , then call from  $f()$  to  $g()$  is suspect, as is a return from  $g()$  to  $f()$



## Static control flow integrity: example code & CFG



Before and/or after every control transfer (**function call** or **return**)  
we could check if it is legal – ie. allowed by the cfg

Some weird returns would still be allowed

- eg if we call `h()` from `g()`, and the return is to `f()`, this would be allowed by the static cfg
- Additional *dynamic* return integrity check can narrow this down to actual call site – using recorded call site on shadow stack

# 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
- FAT Pointer & Data Descriptor

### Code optimization: safer strategies

- Control Flow Integrity
- More later!!

---



# SAFE PROGRAMMING LANGUAGES

## Unsafe vs safe

- **Unsafe:** accept that a program statement may be executed when it does not really make sense. The semantics of the statement is undefined in these cases: essentially, anything may happen.
- **obligation of the programmer** to ensure that statements are only ever executed in situations where they make sense

se in un linguaggio senza garbage collector eseguo una free, la free in presenza di alias toglie sotto il tappeto l'oggetto all'altro puntatore che è alias, è compito del programmatore gestire tutti i cammini di accesso alle strutture dati presenti sullo heap ed essere sicuro che tutti i cammini di accesso siano safe quando si fa una free. E' evidente che si lascia troppo carico al programmatore di gestire aspetti critici del linguaggio e si ottengono programmi error prone.

## Unsafe vs safe

**Safe: the language ensures that a statement is only ever executed when it makes sense, or, when it does not, signals some error in a precisely defined manner, for example by throwing an exception.**

- **obligation of the language** to somehow prevent or detect the execution of statements when this does not make sense.

è obbligo del linguaggio e non del programmatore prevedere che i comportamenti unsafe non previsti siano bloccati, è evidente che questo lascia un maggiore overhead a tempo di esecuzione che vuol dire privilegiare la correttezza dell'esecuzione rispetto all'efficienza.



# Unsafe vs safe

- C and C++ are examples of unsafe languages.
- Java and C# are meant to be safe,
- Java and C# still have some unsafe features.
  - the Java Native Interface (JNI) allows Java programs to call native machine code,
  - pointer arithmetic is allowed in C# in code blocks that are marked as unsafe.



# Lessons learned

- A safe programming language is one that provides **memory safety** and **type safety**, but there are other forms of safety
- IDEA: Safe programming languages provide some guarantees about the possible behaviour of programs, which can protect against security vulnerabilities due to unwanted behaviour.
- In a safe programming language, programmer can trust the abstraction mechanisms provided by the language

In un linguaggio di programmazione safe si può fare affidamento oltre che sulle primitive di un linguaggio di programmazione anche sulla macchina virtuale dell'implementazione del linguaggio. Dunque nei linguaggi safe, la TCB associata all'esecuzione del linguaggio deve essere ben chiara e definita e deve gestire i comportamenti del linguaggio.



# COMPOSITIONALITY

- In a safe programming language, it is possible to understand the behaviour of a program in a modular way, and to make guarantees about part of a program by inspecting only that part and not the entire program.

# Memory unsafety breaks compositionality

- Consider a program consisting of two modules, P and Q, written in a language that is not memory safe.
- Code belonging to module Q can make changes anywhere in memory, including data belonging to module P.
  - code belonging to module Q could corrupt data that module P relies on, for instance breaking data invariants.
- This means we cannot make guarantees about P just by looking at P, we also have to make sure Q does not interfere with P.
- Note that this means we cannot make guarantees about extensible programs, as any extension could corrupt existing behaviour.



In principle ...

If a language is memory-safe then programs can never crash with a segmentation fault. One might even consider it safe to switch off the checks the operating system performs on memory access for these programs.

# But...

---

The execution of the memory-safe program might rely on some interpreter or execution engine that is written in an unsafe language, which could still cause out-of-bounds memory access.

---

Also, memory safety might rely on the correctness of compilers and type checkers, which may contain bugs.



## Defence in Depth

- The principle of Defence in Depth: keeping the memory access control performed by the operating system switched on is always a wise thing to do.