Una breve rassegna (cocktail) di vulnerabilità, attacchi e contromisure

Control Hijacking

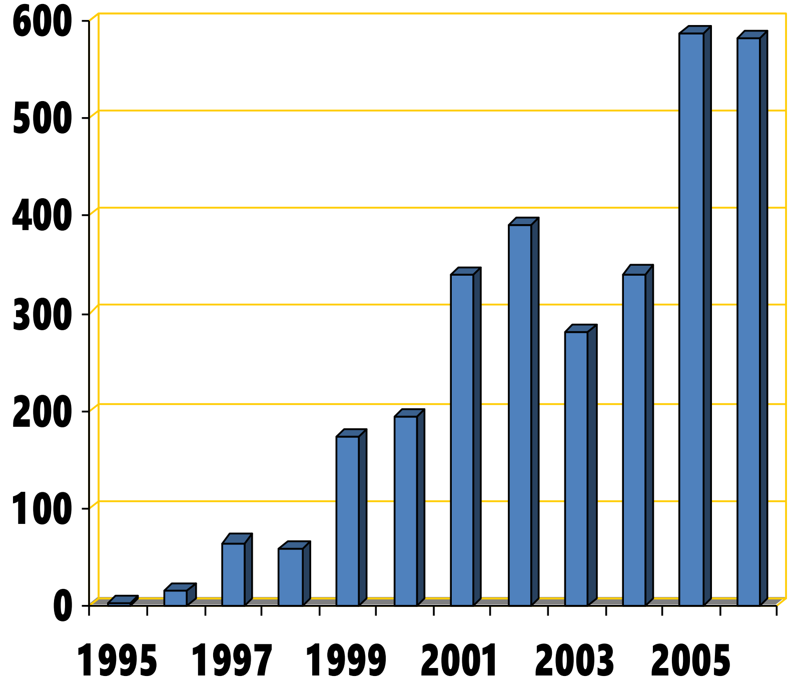
Basic Control Hijacking Attacks

Control hijacking attacks

- Attacker's goal:
  - Take over target machine (e.g. web server)
    - Execute arbitrary code on target by hijacking application control flow
- Examples.
  - Buffer /Integer overflow attacks
  - Format string vulnerabilities

Example 1: buffer overflows

- Extremely common bug in C/C++ programs.
  - First major exploit: 1988 Internet Worm. fingerd.



≈ 20% of all vuln.

2005-2007: ≈ 10%

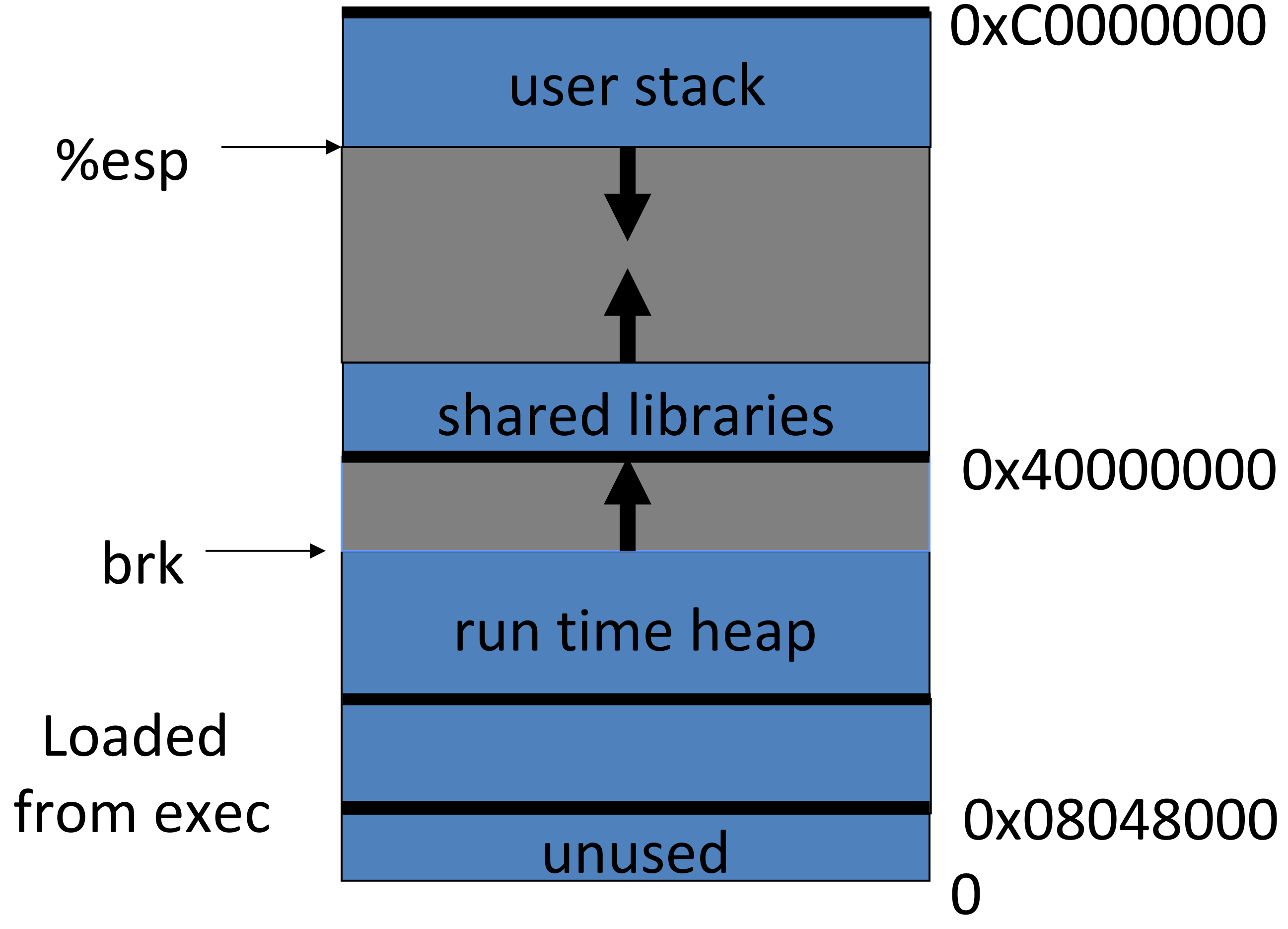
Source: NVD/CVE

What is needed

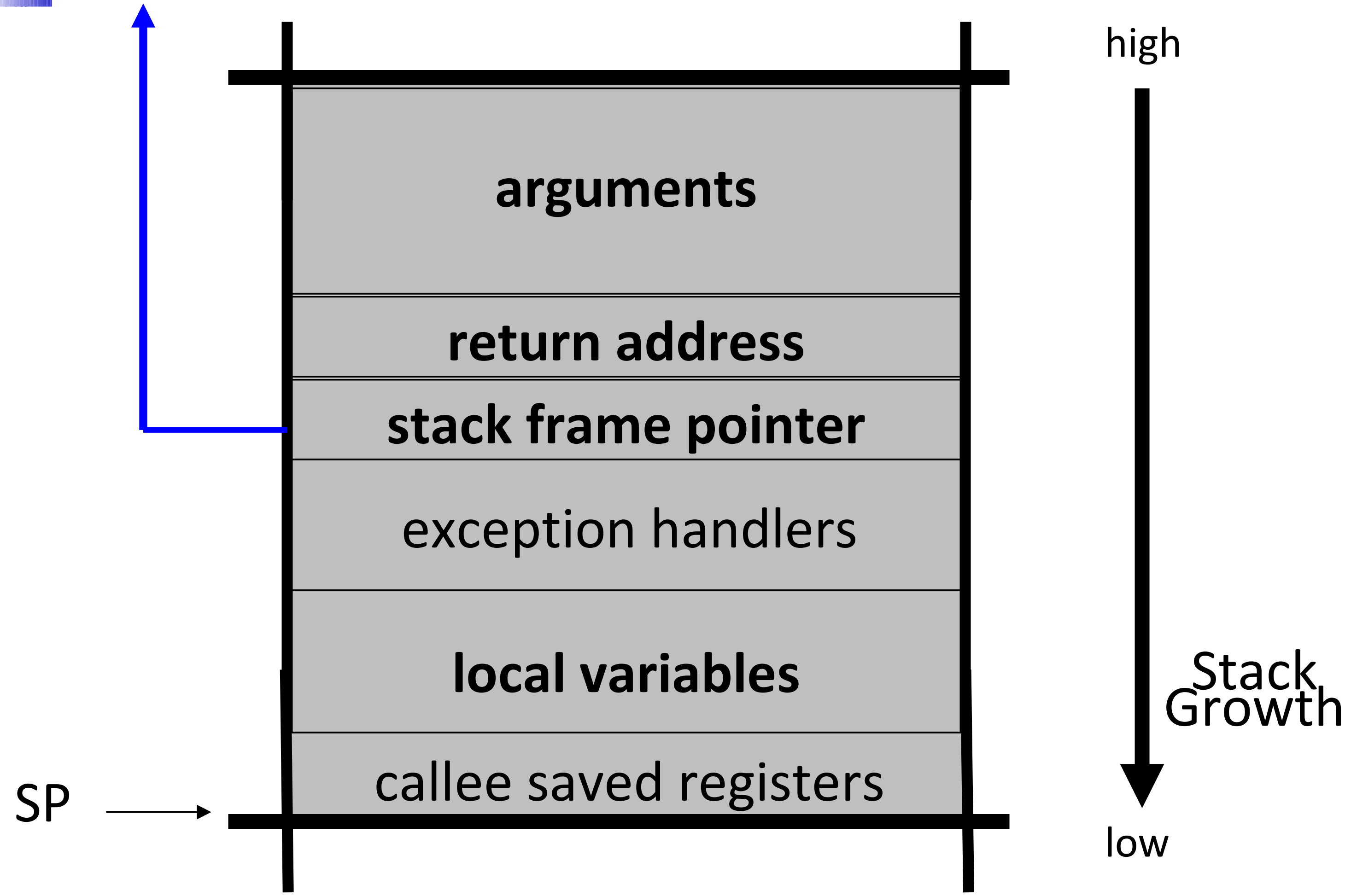
- Understanding C functions, the stack, and the heap.
- Know how system calls are made

- The exec() system call
- Attacker needs to know CPU and OS used on the target machine:
  - Our examples are for x86 running Linux or Windows
  - Details vary slightly between CPUs and OSs:
    - Little endian vs. big endian   (x86 vs. Motorola)
    - Stack Frame structure   (Unix vs. Windows)

Linux process memory layout



Stack Frame

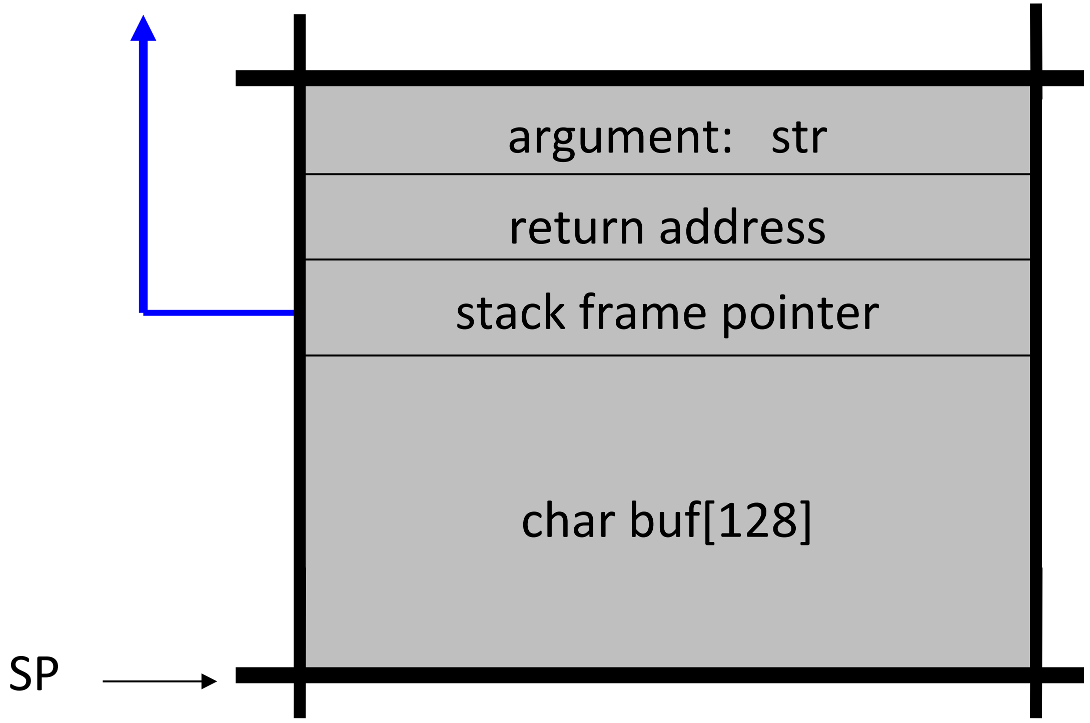

high
arguments
return address
stack frame pointer
exception handlers
local variables
callee saved registers
SP
Stack Growth
low

What are buffer overflows?

Suppose a web server contains a function:

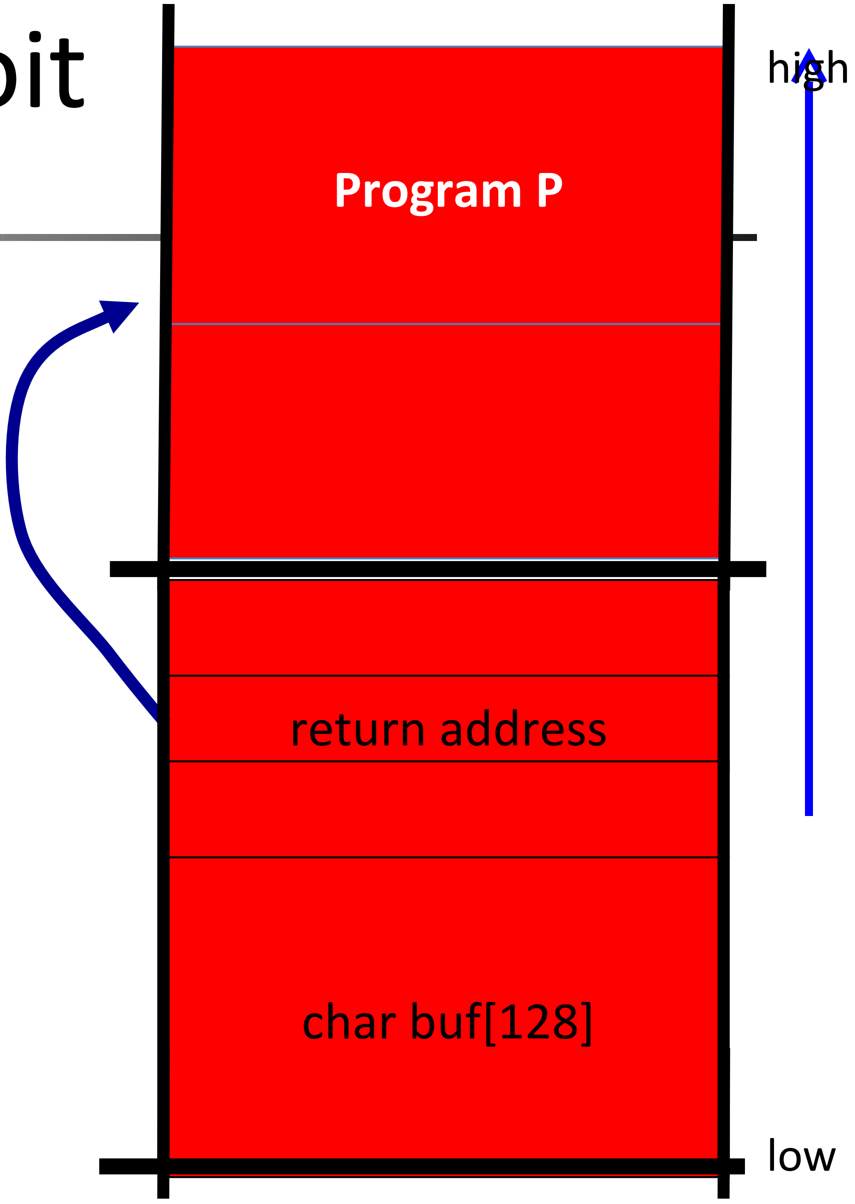
```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```

When func() is called stack looks like:



Basic stack exploit

- Suppose *str is such that after strcpy stack looks like:
- Program P: exec("/bin/sh")
- When func() exits, the user gets shell !
- Note: attack code P runs *in stack*.



The NOP slide

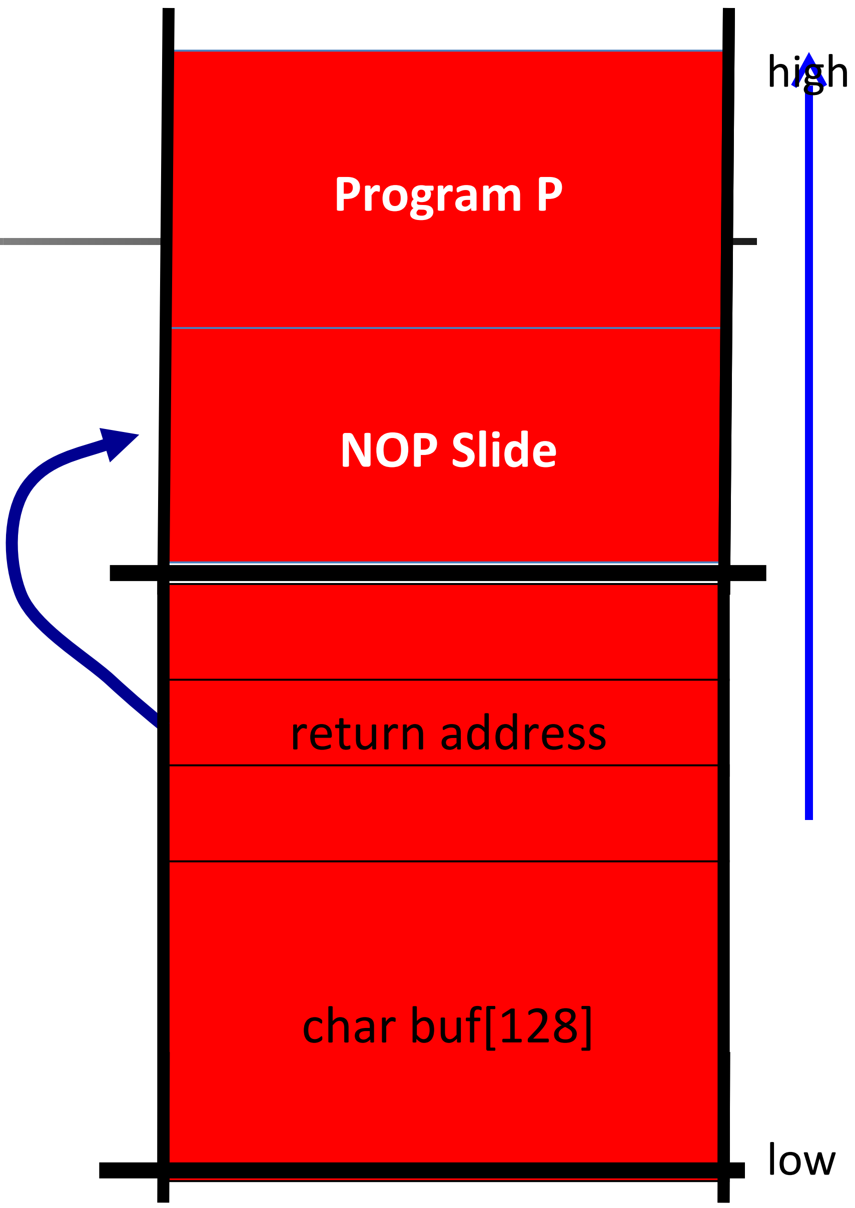
Problem: how does attacker determine ret-address?

Solution: NOP slide

- Guess approximate stack state when func() is called

- Insert many NOPs before program P:

nop , xor eax,eax , inc ax



Details and examples

- Some complications:
  - Program P should not contain the ‘\0’ character.
  - Overflow should not crash program before func() exists.
- Sample <u>remote</u> stack smashing overflows:
  - (2007) Windows animated cursors (ANI), LoadAniIcon()
  - (2005) Symantec Virus Detection

Many unsafe libc functions

strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

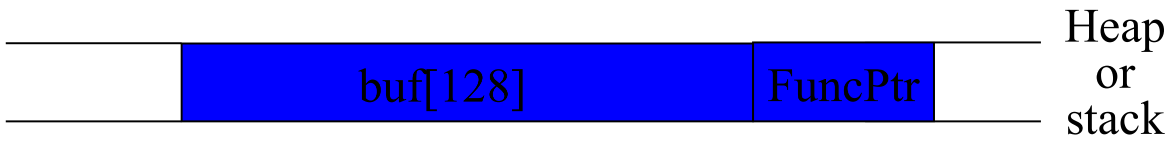
scanf (const char *format, …)   and many more.

- "Safe" libc versions strncpy(), strncat() are misleading
  - e.g. strncpy()   may leave string unterminated.

- Windows C run time (CRT):
  - strcpy_s (*dest, DestSize, *src):   ensures proper termination

Buffer overflow opportunities

- Exception handlers: (Windows SEH attacks)
  - Overwrite an exception handler address in stack frame.



- Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)
  - Overflowing buf will override function pointer.

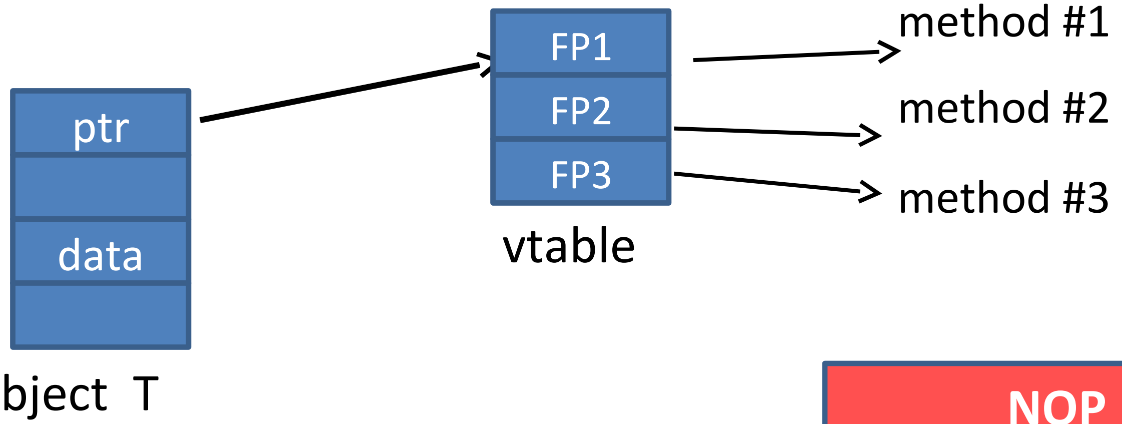
- Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

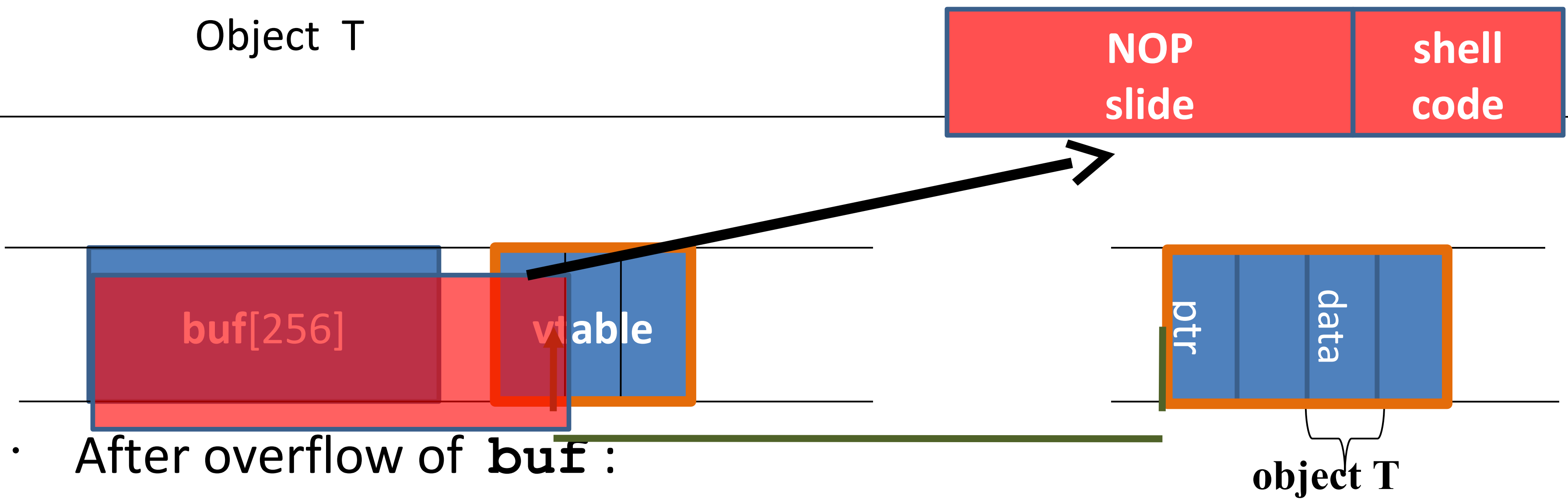
SEH attack

- It executes arbitrary code by abusing the 32-bit Windows exception dispatching facilities
- A stack-overflow overwrites an exception registration record (ERR) on a thread's stack.
- An ERR includes a next pointer and an exception handler function pointer. The next pointer links to the next record in the list of registered exception handlers. The exception handler function pointer is used when an exception occurs.
- After an exception registration record has been overwritten, an exception must be raised so that the exception dispatcher will attempt to handle it.

Corrupting method pointers

- Compiler generated function pointers (e.g. C++ code)





- After overflow of **buf** :

Finding buffer overflows

- To find overflow in a web server:
  - Run server on local machine
  - Issue malformed requests (ending with "$$$$$")
    - Many automated tools exist (called fuzzers – next module)
  - If web server crashes,
    search core dump for "$$$$$" to find overflow location
- Construct exploit (not easy given latest defenses)

More Hijacking Opportunities

- **Integer overflows**: (e.g. MS DirectX MIDI Lib)

- **Double free**: double free space on heap.
  - Can cause memory mgr to write data to specific location
  - Examples: CVS server

- **Format string vulnerabilities**

Integer Overflows (see Phrack 60)

Problem: what happens when int exceeds max value?

int m; (32 bits) **short s; (16 bits)** **char c; (8 bits)**

c = 0x80 + 0x80 = 128 + 128 $\Rightarrow$ c = 0

s = 0xff80 + 0x80 $\Rightarrow$ s = 0

m = 0xffffff80 + 0x80 $\Rightarrow$ m = 0

Can this be exploited?

Integer overflow

Since an integer is a fixed size (e.g. 32 bits), there is a fixed maximum value it can store. When an attempt is made to store a value greater than this maximum value it is known as an integer overflow.

The ISO C99 standard says that an integer overflow causes "undefined behaviour", meaning that compilers conforming to the standard may do anything they like from completely ignoring the overflow to aborting the program.

Most compilers seem to ignore the overflow, resulting in an unexpected or erroneous result being stored.

Integer overflow

- Integer overflows cannot be detected after they have happened, so there is not way for an application to tell if a result is in fact correct.

- This can get dangerous if the calculation has to do with the size of a buffer or how far into an array to index.

- Most integer overflows are not exploitable because memory is not being directly overwritten, but sometimes they can lead to other classes of bugs,frequently buffer overflows.

- Integer overflows can be difficult to spot, so even well audited code can spring surprises.

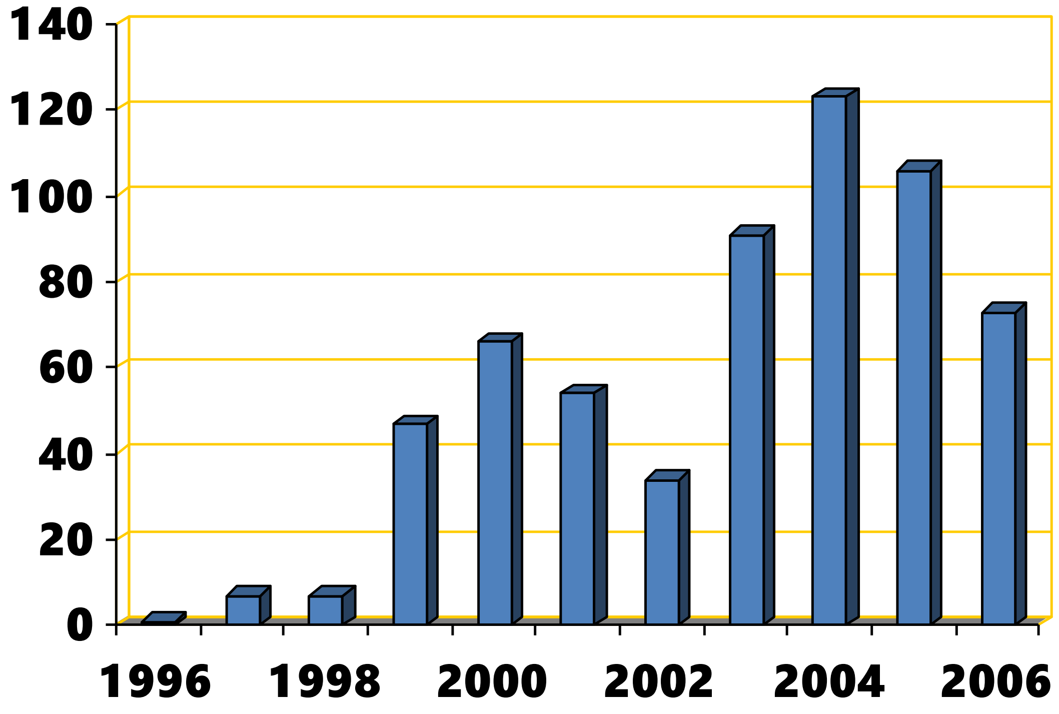
An example

```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {

      char temp[256];

      if  (len1 + len2 > 256)  {return -1}          // length check

      memcpy(temp, buf1, len1);                     // cat buffers

      memcpy(temp+len1, buf2, len2);

      do-something(temp);                           // do stuff

}
```

If **len1 = 0x80,**   **len2 = 0xffffff80** ⇒ len1+len2 = 0

Second memcpy() will overflow heap !!

Integer overflow exploit stats



Source: NVD/CVE

Format string bugs

Format string problem

```
int func(char *user)   {

            fprintf( stderr, user); }
```

```
int fprintf(FILE *stream, char *formato,
argomenti ...);
```

Problem: what if *user = “%s%s%s%s%s%s%s” ??

- Most likely program will crash: DoS.
- If not, program will print memory contents. Privacy?
- Full exploit using user = “%n”

Correct form: `fprintf( stdout, “%s”, user);`

Format string problem

Se si passa a una funzione che stampa una stringa a schermo (printf del C) una stringa che in realtà contiene una serie di parametri di specifica dell'input (tipicamente %s e %x per esaminare il contenuto della memoria e %n per sovrascriverne parti , in particolare dello stack) si permette l'avvio di un attacco di tipo stack overflow e return to libc.

Per proteggersi da questo attacco, quando si vuole stampare una stringa s usando la printf() o una qualsiasi funzione C che accetti un numero illimitato di identificatori di formato, bisogna scrivere la funzione printf("%s", s) e non printf(s)

History

- First exploit discovered in June 2000.
- Examples:

– wu-ftpd 2.* :	remote root
– Linux rpc.statd:	remote root
– IRIX telnetd:	remote root
– BSD chpass:	local root

Vulnerable functions

Any function using a format string.

Printing:

printf, fprintf, sprintf, …

vprintf, vfprintf, vsprintf, …

Logging:

syslog, err, warn

Exploit

- Dumping arbitrary memory:
  - Walk up stack until desired pointer is found.
  - printf(“%08x.%08x.%08x.%08x|%s|”)

- Writing to arbitrary memory:
  - printf(“hello %n”, &temp) - ‘6’ into temp.
  - printf(“%08x.%08x.%08x.%08x.%n”)

Control Hijacking

Platform Defenses

=

Contromisure

Preventing hijacking attacks

a) <u>Fix bugs</u>:

- Audit software
  - Automated tools: Coverity, Prefast/Prefix.
- Rewrite software in a type safe languange (Java, ML)
  - Difficult for existing (legacy) code …

b) Concede overflow, but <u>prevent code execution</u>

c) Add <u>runtime code</u> to detect overflows exploits

- Halt process when overflow exploit detected
- StackGuard, LibSafe, …

Marking stack and heap as **non-executable**

NX-bit on AMD Athlon 64,

- XD-bit on Intel P4 Prescott
- NX bit in every Page Table Entry (PTE)

Deployment: Linux (via PaX project);

- OpenBSDWindows: since XP SP2 (DEP)
- Visual Studio: **/NXCompat[:NO]**

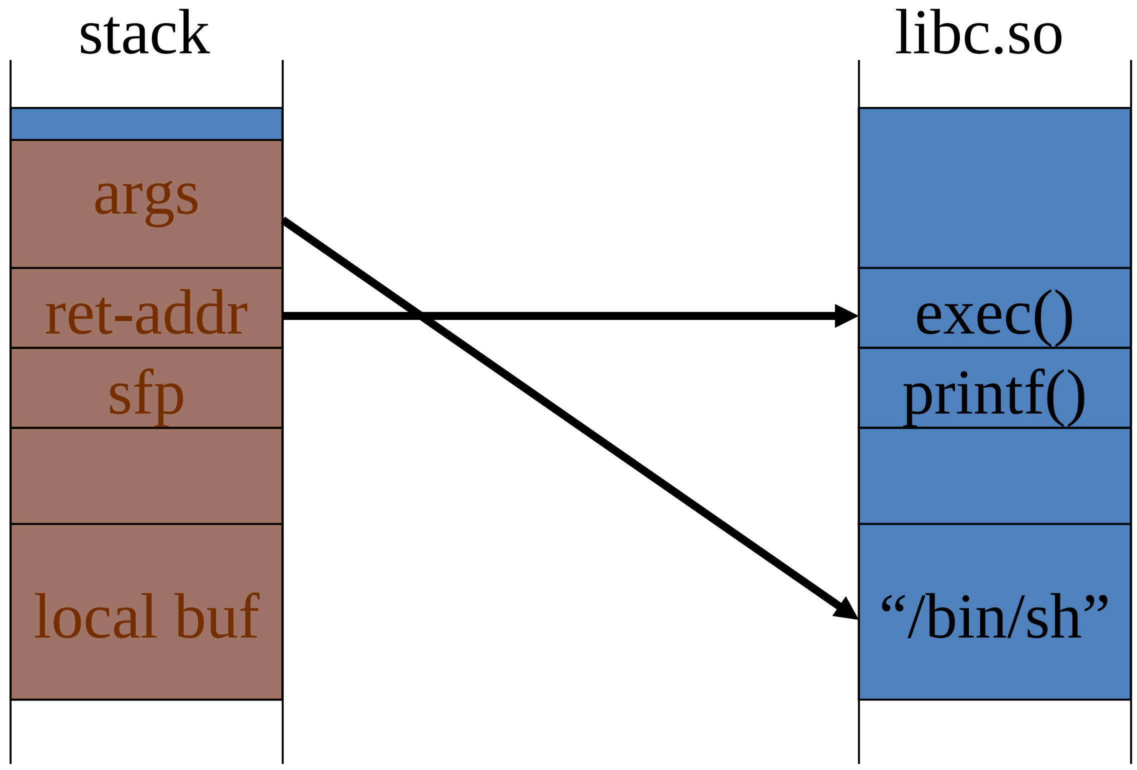
Limitations:

– Some apps need executable heap (e.g. JITs).

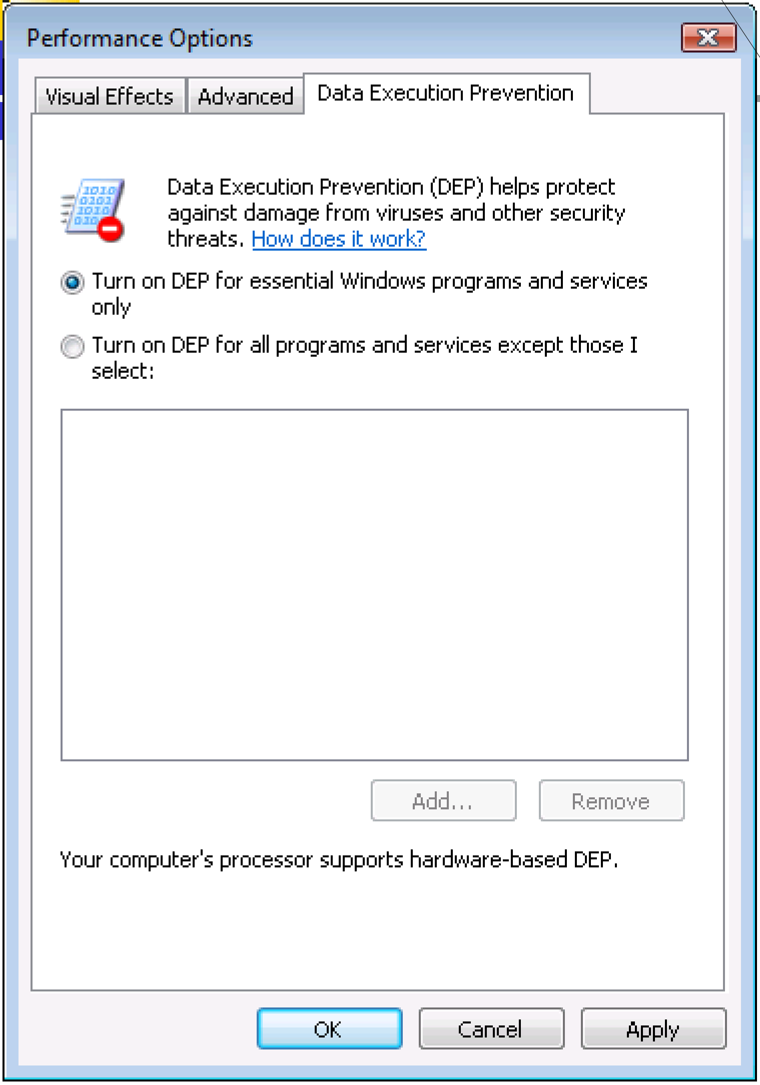
– Does not defend against `**Return Oriented Programming**’

Attack: Return Oriented Programming (ROP)

- Control hijacking without executing code



Examples: DEP controls in Windows

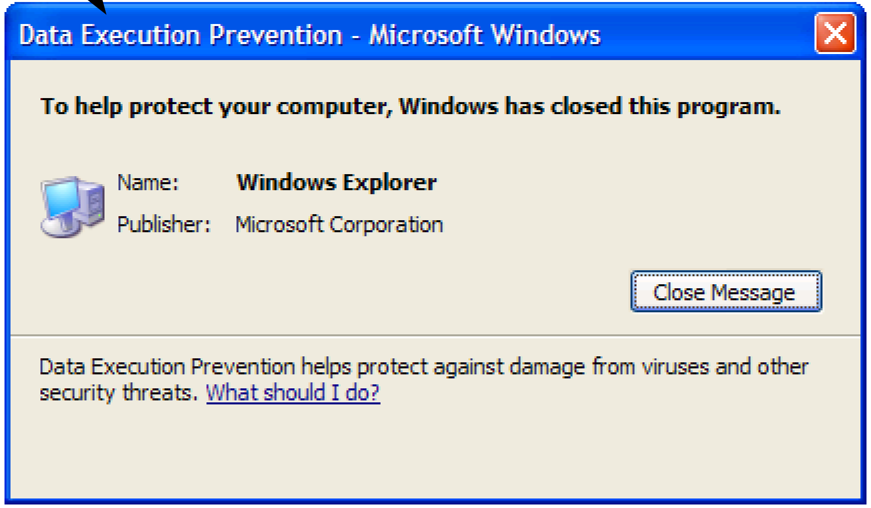

Performance Options

Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

[✓ Turn on DEP for essential Windows programs and services only]

Turn on DEP for all programs and services except those I select:

Your computer's processor supports hardware-based DEP.


Data Execution Prevention - Microsoft Windows

To help protect your computer, Windows has closed this program.

Name: **Windows Explorer**
Publisher: Microsoft Corporation

Data Execution Prevention helps protect against damage from viruses and other security threats. What should I do?

DEP terminating a program

Response: ASLR =Address space layout randomization

- – Shared libraries to random location in process memory

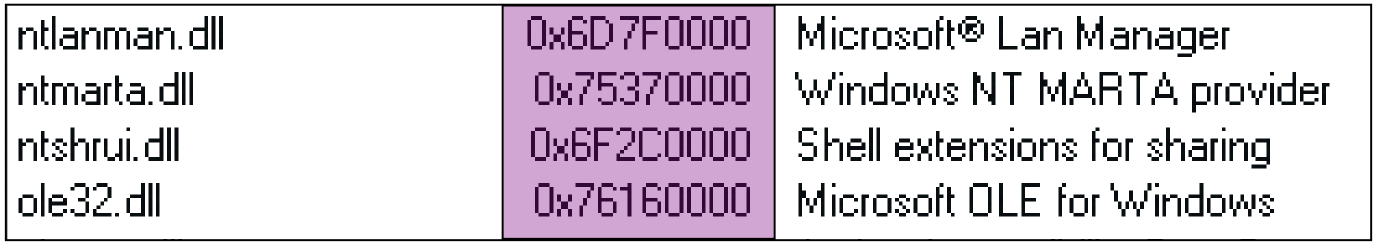
$\Rightarrow$ Attacker cannot jump directly to exec function

- – <u>Deployment</u>: (/DynamicBase)
  - **Windows Vista:** 8 bits of randomness for DLLs
    - – aligned to 64K page in a 16MB region $\Rightarrow$ 256 choices
  - **Windows 8:** 24 bits of randomness on 64-bit processors

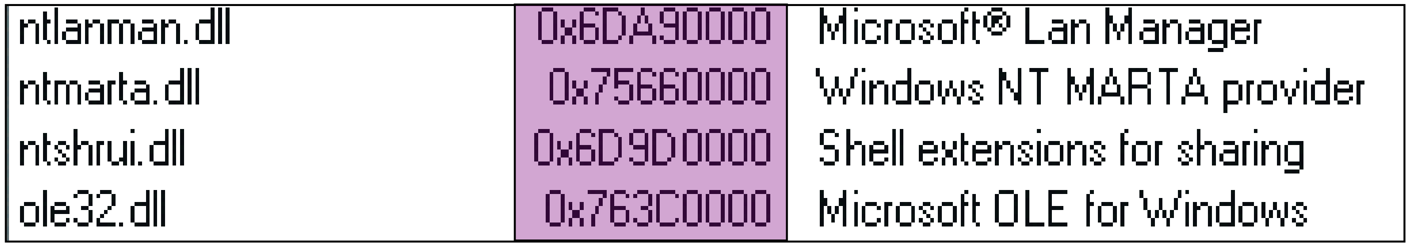
- <u>Other randomization methods</u>:
  - – Sys-call randomization: randomize sys-call id’s
  - – Instruction Set Randomization (ISR)

ASLR Example

Booting twice loads libraries into different locations:



ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows



ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

Note: everything in process memory must be randomized
stack, heap, shared libs, image

- Win 8 **Force ASLR**: ensures all loaded modules use ASLR

More attacks :   spraying

A heap spray cannot be used to break any security : a separate vulnerability is needed.

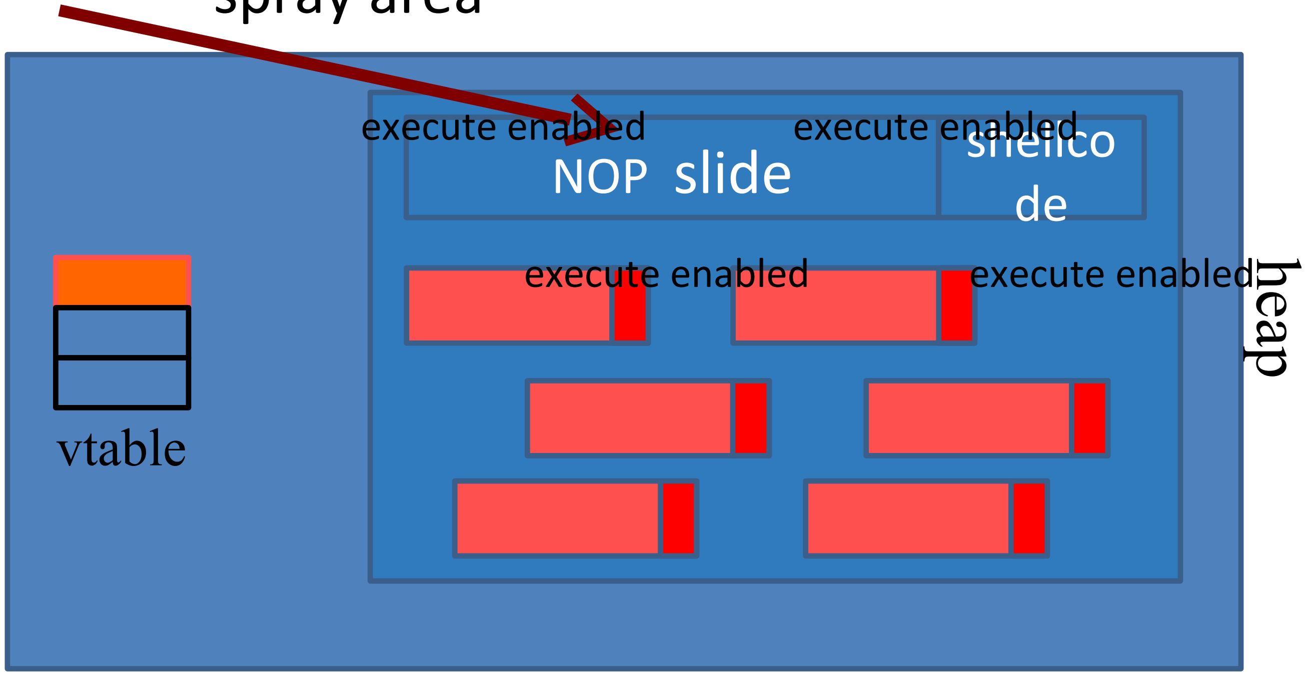
Exploiting security issues is often hard because various factors can influence this process. Chance alignments of memory and timing introduce a lot of randomness (from the attacker's point of view). A heap spray can be used to introduce a large amount of order to compensate for this and increase the chances of successful exploitation. Heap sprays take advantage of the fact that the start location of large heap allocations is predictable and consecutive allocations are roughly sequential. This means that the heap will roughly be in the same location each and every time the heap spray is run.

This strategy aims to use the heap spray as a very large NOP sled (for example, 0x0c0c0c0c is often used as non-canonical NOP[1])

More attacks : JiT spraying

Idea:

1. Force Javascript JiT to fill heap with executable shellcode
2. then point Saved Frame Pointer anywhere in spray area



More attacks :   JiT spraying

Most modern interpreters implement a Just-In-Time (JIT) compiler to transform the parsed input or bytecode into machine code for faster execution.

JIT spraying is the process of coercing the JIT engine to write many executable pages with embedded shellcode.
This shellcode will entered through the middle of a normal JIT instruction.

For example, a Javascript statement such as "var x = 0x41414141 + 0x42424242;" might be compiled to contain two 4 byte constants in the executable image
("mov eax, 0x41414141; mov ecx, 0x42424242; add eax, ecx").
By starting execution in the middle of these constants, a completely different instruction stream is revealed.

Control Hijacking

Run-time Defenses

StackGuard

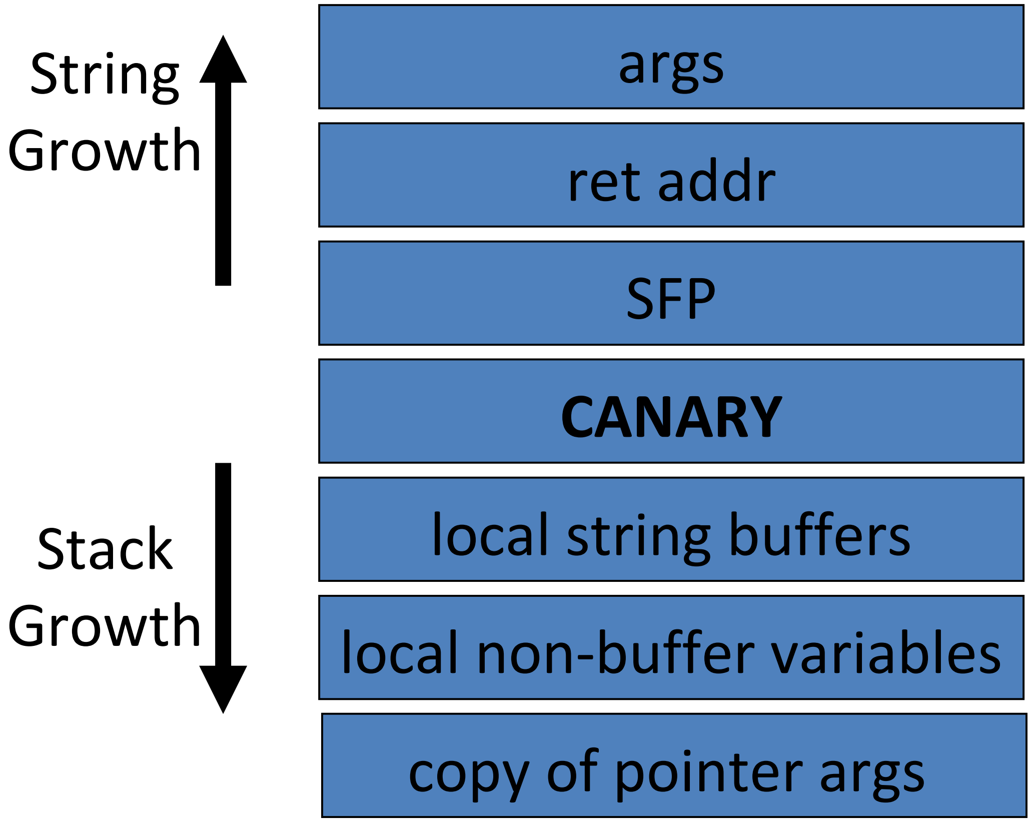
- Minimal performance effects: 8% for Apache.
- StackGuard implemented as a GCC patch.
  - Program must be recompiled.
- Note: Canaries don't provide full proof protection.
  - Some attacks leave canaries unchanged
- Heap protection: PointGuard.
  - Protects pointers and buffers by encryption
  - Less effective, more noticeable performance effects

Heap protection: PointGuard.

- Protects pointers and buffers by encryption
- Key generated when the program starts
- Never shared so it is secure
- Less effective, more noticeable performance effects

StackGuard enhancements: ProPolice IBM

Rearrange stack layout to prevent ptr overflow.



Protects pointer args and local pointers from a buffer overflow

pointers, but no arrays

ProPolice IBM

- reorder local variables to place buffers after pointers to avoid the corruption of pointers
- copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers
- omission of instrumentation code from some functions to decrease the performance overhead.

MS Visual Studio /GS [since 2003]

Compiler /GS option:

- Combination of ProPolice and Random canary.
- If cookie mismatch, default behavior is to call **_exit(3)**

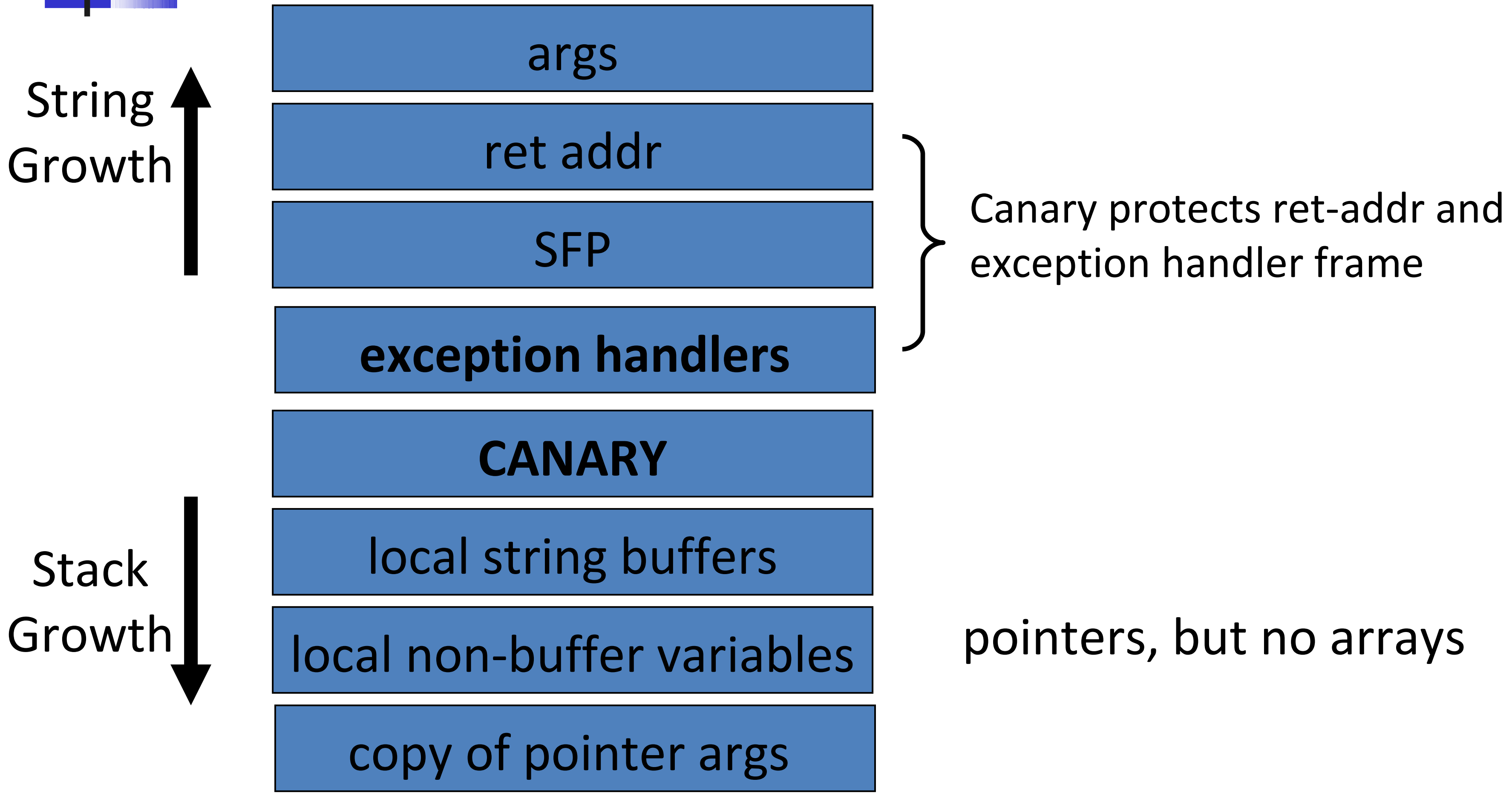
Function prolog:
```
    sub   esp, 8      // allocate 8 bytes for cookie
    mov   eax, DWORD PTR ___security_cookie
    xor   eax, esp     // xor cookie with current esp
    mov   DWORD PTR [esp+8], eax  // save in stack
```

Function epilog:
```
    mov   ecx, DWORD PTR  [esp+8]
    xor   ecx, esp
    call  @__security_check_cookie@4
    add   esp, 8
```

Enhanced /GS in Visual Studio 2010:

- /GS protection added to all functions, unless can be proven unnecessary

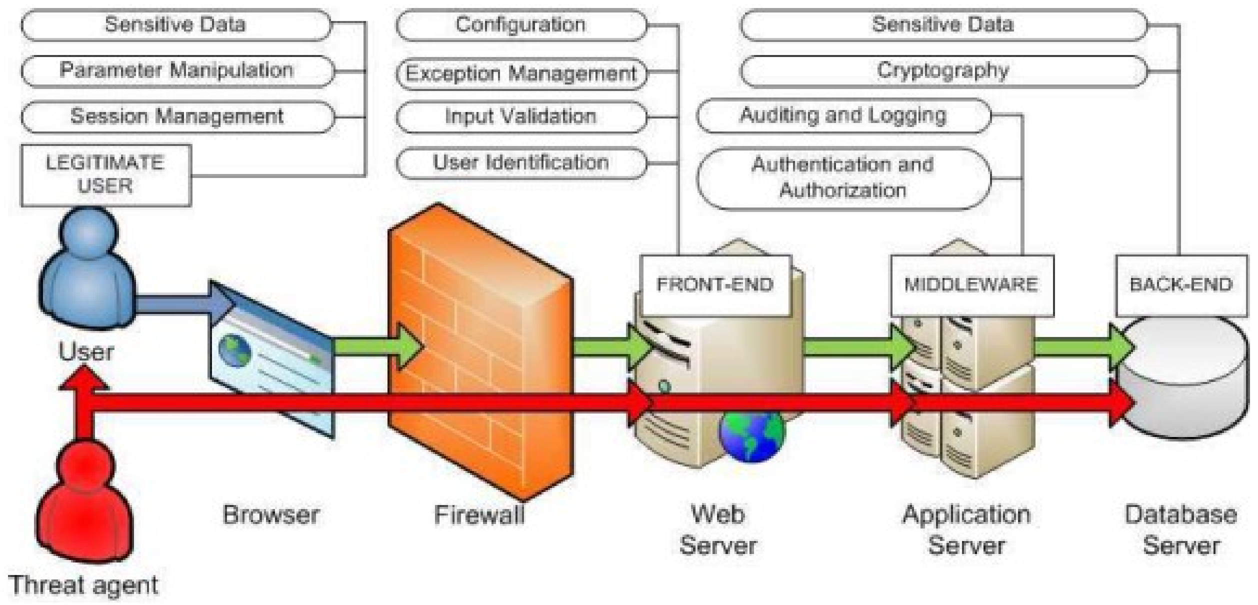
/GS stack frame



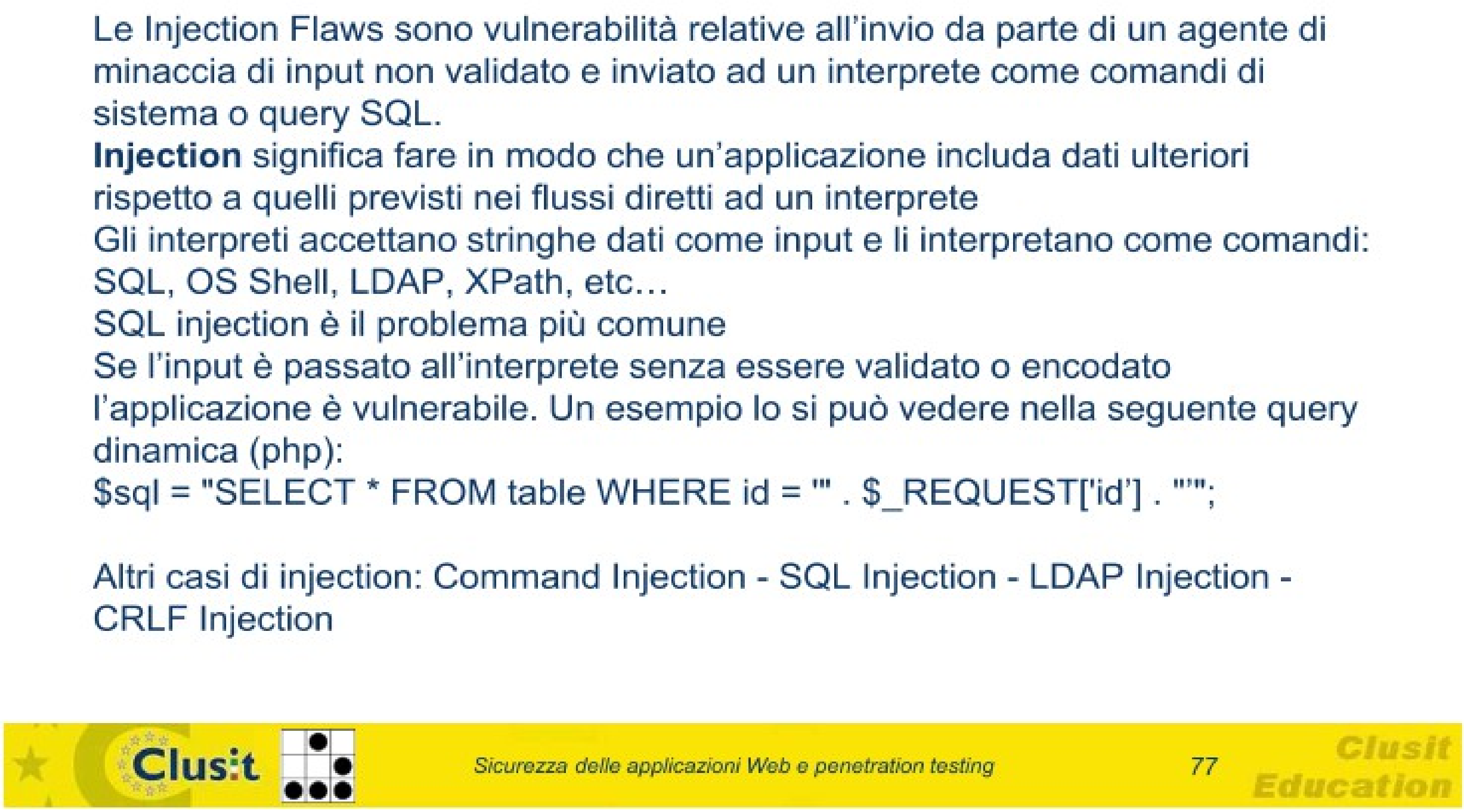
Summary: Canaries are not full proof

- Canaries are an important defense tool, but do not prevent all control hijacking attacks:
  - Heap-based attacks still possible
  - Integer overflow attacks still possible
  - /GS by itself does not prevent Exception Handling attack

Attacchi web based



Injection Flaw e Attack



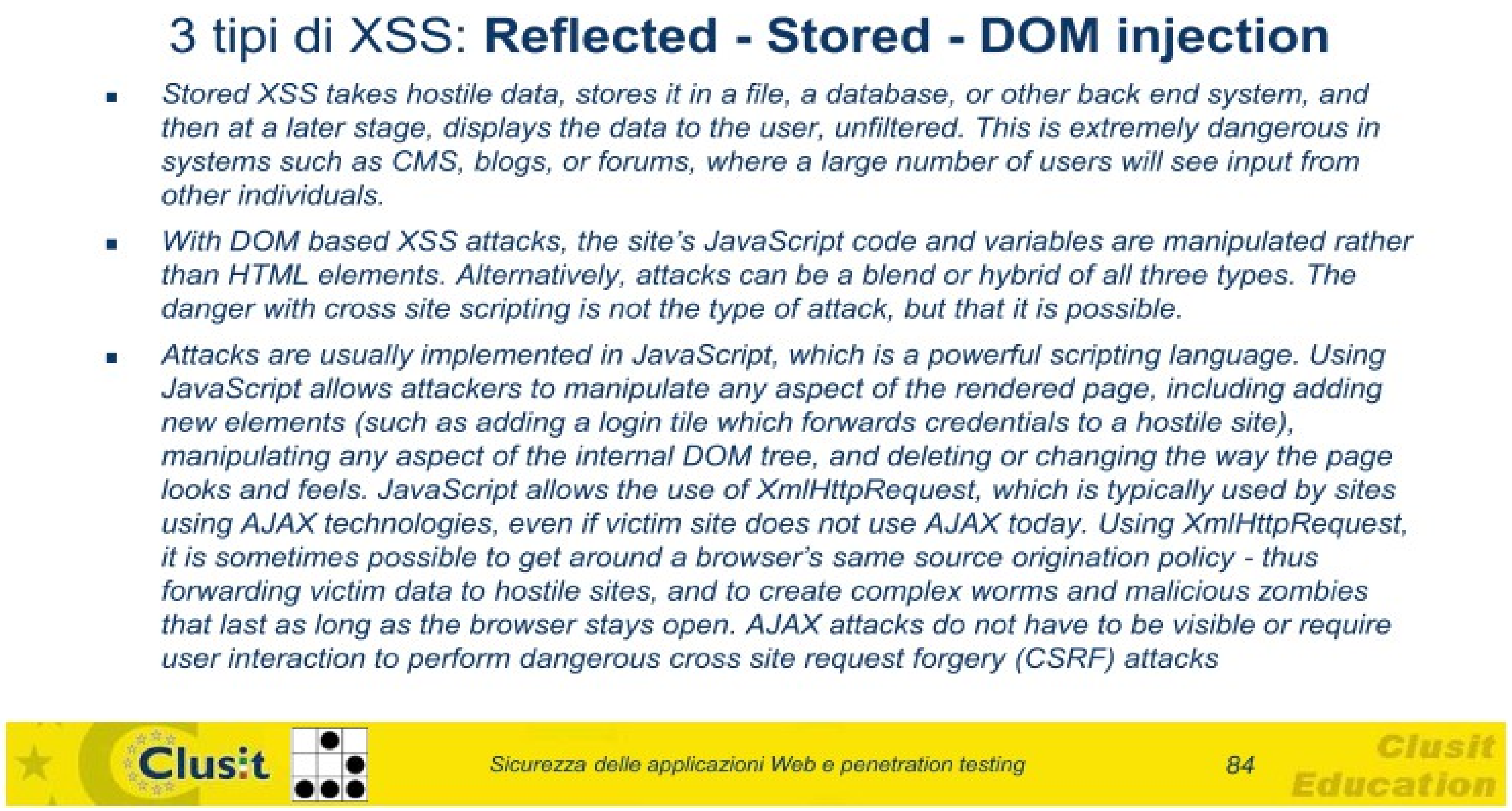
Le Injection Flaws sono vulnerabilità relative all’invio da parte di un agente di minaccia di input non validato e inviato ad un interprete come comandi di sistema o query SQL.
Injection significa fare in modo che un’applicazione includa dati ulteriori rispetto a quelli previsti nei flussi diretti ad un interprete
Gli interpreti accettano stringhe dati come input e li interpretano come comandi: SQL, OS Shell, LDAP, XPath, etc…
SQL injection è il problema più comune
Se l’input è passato all’interprete senza essere validato o encodato l’applicazione è vulnerabile. Un esempio lo si può vedere nella seguente query dinamica (php):
$sql = "SELECT * FROM table WHERE id = '" . $_REQUEST['id’] . "'";

Altri casi di injection: Command Injection - SQL Injection - LDAP Injection - CRLF Injection

XSS – Cross Site Scripting

- Le falle di tipo XSS si verificano quando un'applicazione web riceve dei dati provenienti da fonti non affidabili e li invia ad un browser senza una opportuna validazione e/o "escaping". Il XSS permette agli attaccanti di eseguire degli script malevoli sui browser delle vittime; tali script possono dirottare la sessione dell'utente, fare un deface del sito web o redirezionare l'utente su un sito malevolo.

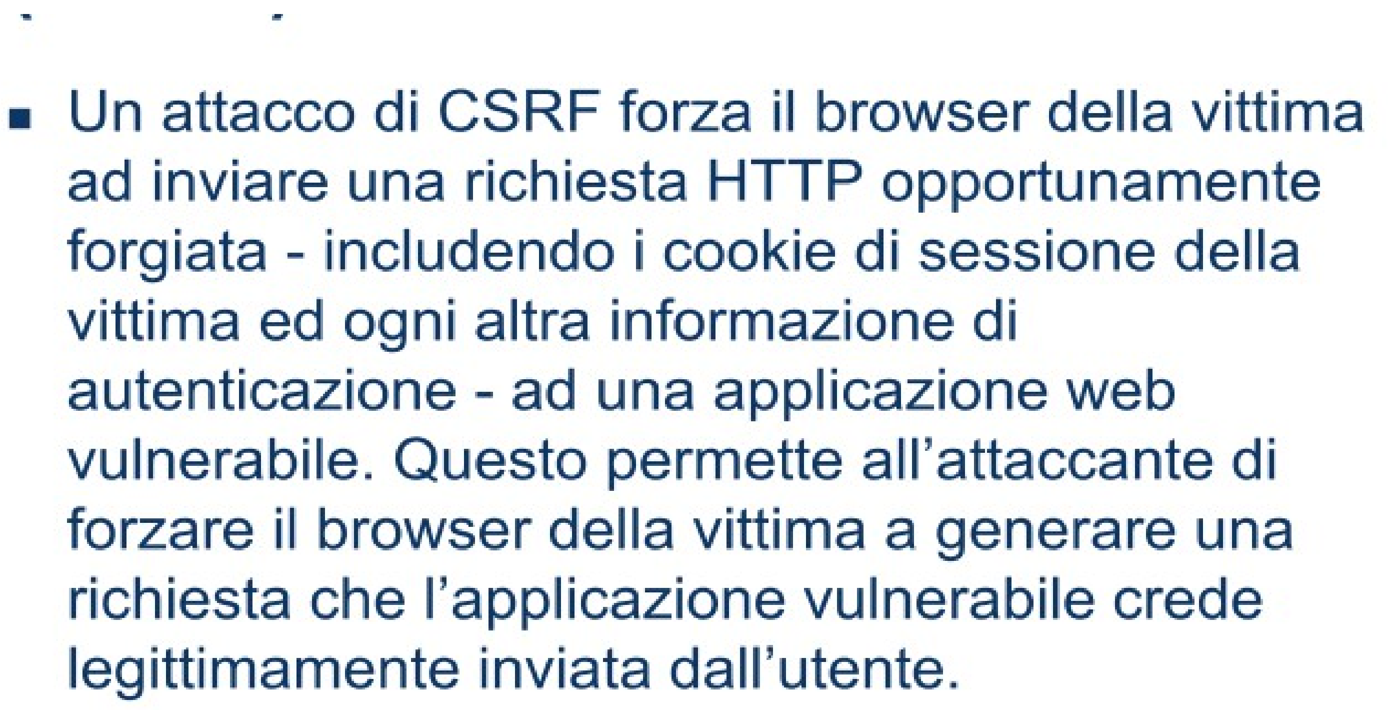
XSS



3 tipi di XSS: **Reflected - Stored - DOM injection**

- *Stored XSS takes hostile data, stores it in a file, a database, or other back end system, and then at a later stage, displays the data to the user, unfiltered. This is extremely dangerous in systems such as CMS, blogs, or forums, where a large number of users will see input from other individuals.*
- *With DOM based XSS attacks, the site's JavaScript code and variables are manipulated rather than HTML elements. Alternatively, attacks can be a blend or hybrid of all three types. The danger with cross site scripting is not the type of attack, but that it is possible.*
- *Attacks are usually implemented in JavaScript, which is a powerful scripting language. Using JavaScript allows attackers to manipulate any aspect of the rendered page, including adding new elements (such as adding a login tile which forwards credentials to a hostile site), manipulating any aspect of the internal DOM tree, and deleting or changing the way the page looks and feels. JavaScript allows the use of XmlHttpRequest, which is typically used by sites using AJAX technologies, even if victim site does not use AJAX today. Using XmlHttpRequest, it is sometimes possible to get around a browser's same source origination policy - thus forwarding victim data to hostile sites, and to create complex worms and malicious zombies that last as long as the browser stays open. AJAX attacks do not have to be visible or require user interaction to perform dangerous cross site request forgery (CSRF) attacks*

CSRF – Cross Site Request Forgery



- Un attacco di CSRF forza il browser della vittima ad inviare una richiesta HTTP opportunamente forgiata - includendo i cookie di sessione della vittima ed ogni altra informazione di autenticazione - ad una applicazione web vulnerabile. Questo permette all'attaccante di forzare il browser della vittima a generare una richiesta che l'applicazione vulnerabile crede legittimamente inviata dall'utente.

CSRF – Cross Site Request Forgery

Un attacco di tipo Cross Site Request Forgery è mirato **a forzare il Browser di una vittima ad eseguire operazioni da lui non espressamente richieste** sfruttando vulnerabilità applicative come una non corretta implementazione delle sessioni ed un passaggio dei parametri e dei rispettivi valori di una richiesta attraverso l'uso del metodo GET.

Nota la differenza con XSS: nel csrf è la vittima che viene forzata a compiere un'azione, con xss è l'attaccante che si impadronisce dell'identità della vittima e richiede javascript.
Inoltre: xss richiede che un sito accetti malicious code, mentre con CSRF il codice malevole è in un sito di terze parte.

CSRF – Cross Site Request Forgery

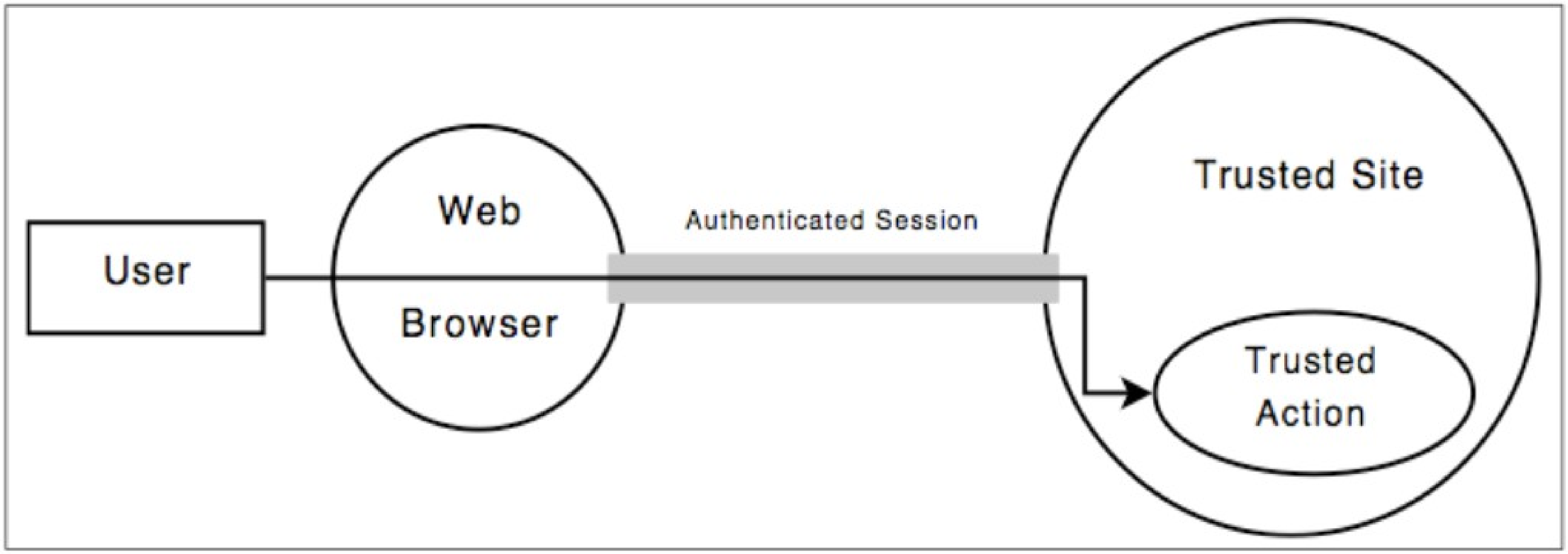


Figure 2: *A valid request. The Web Browser attempts to perform a Trusted Action. The Trusted Site confirms that the Web Browser is authenticated and allows the action to be performed.*

CSRF – Cross Site Request Forgery

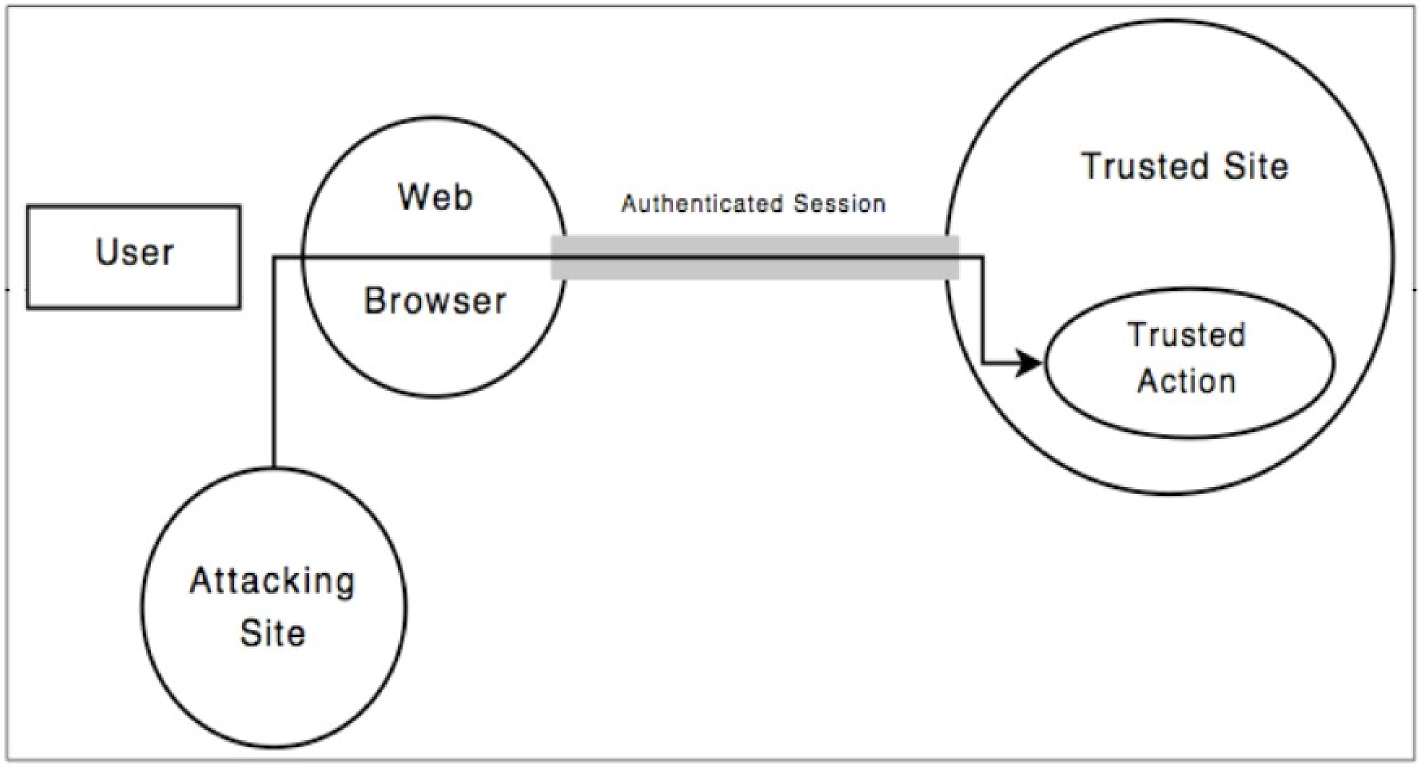
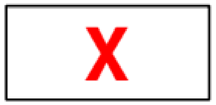


Figure 3: *A CSRF attack. The Attacking Site causes the browser to send a request to the Trusted Site. The Trusted Site sees a valid, authenticated request from the Web Browser and performs the Trusted Action. CSRF attacks are possible because web sites authenticate the web browser, not the user.*

CSRF – Cross Site Request Forgery

Supponiamo che l'utente Alice stia navigando su un forum, o un blog, o stia visualizzando una mail in formato HTML dove l'utente Bob ha postato un'immagine malevola così forgiata:



```
<img src="http://bancadialice.com/prelievo.php?da=conto_alice&a=conto_bob"
alt="Errore nella visualizzazione dell'immagine”>
```

Supponiamo ora che Alice sia autenticata in quel momento sul sito della sua banca, quindi sul suo computer sia presente un cookie che testimonia l'avvenuta autenticazione. In tal caso, la richiesta effettuata sarà perfettamente valida, e l'URL in questione eseguito da Alice con ovvie conseguenze. In genere questo tipo di attacchi vengono perpetrati attraverso immagini o iframe "abusivi" nel codice.