

```
root@hacklab:~/parte-02/lezione-16
```

ETHICAL HACKING POTENZIATO DALL'IA // PARTE II

LEZIONE 16

Binary Exploitation e sviluppo di exploit

Sfruttamento della memoria e sviluppo di exploit affidabili.

docente **Vincenzo Calabro** fase Specializzazione avanzata: Binary Exploitation

```
root@hacklab:~/lezione-16/agenda $
```

LEZIONE 16 / AGENDA

In questa lezione

// PERCORSO

- > Memoria del processo e registri
- > Stack overflow e format string
- > Heap exploitation
- > Shellcoding, ret2libc e ROP
- > Mitigazioni e bypass

metodo obiettivi -> teoria -> strumenti -> laboratorio -> caso di studio -> etica -> verifica

```
root@hacklab:~/lezione-16/obiettivi $
```

LEZIONE 16 / OBIETTIVI

Obiettivi di apprendimento

// AL TERMINE SAPRAI (TASSONOMIA DI BLOOM)

- > Comprendere – l'organizzazione della memoria di un processo
- > Applicare – overflow e corruzioni della memoria
- > Analizzare – il funzionamento delle mitigazioni moderne
- > Valutare – affidabilità e portabilità di un exploit
- > Creare – un exploit funzionante documentato

```
root@hacklab:~/lezione-16/inquadramento $
```

LEZIONE 16 / INQUADRAMENTO

Inquadramento

// DI COSA PARLIAMO

Scendiamo al livello della memoria per capire come nascono e si sfruttano i bug binari, e perché le mitigazioni moderne contano. Un terreno tecnico in cui rigore e ambiente isolato sono irrinunciabili.

// PERCHÉ CONTA

Sfruttamento della memoria e sviluppo di exploit affidabili.

```
root@hacklab:~/lezione-16/mappa-insieme $
```

LEZIONE 16 / MAPPA // INSIEME

Mappa d'insieme del modulo

■ Attacco/tecnica ■ Difesa ■ Norme & metodo ■ IA

FONDAMENTI

01 Memoria, registri e calling convention

02 Classi di vulnerabilità binarie

03 Shellcoding e ROP/JOP

04 Mitigazioni e loro bypass

05 Fuzzing e triage dei crash

APPROFONDIMENTI

06 Calling convention e stack frame

07 Use-after-free e heap

08 JOP e catene di gadget

09 Panoramica delle mitigazioni

10 Dal crash all'exploit

```
root@hacklab:~/lezione-16/mappa-1 $
```

LEZIONE 16 / MAPPA // FONDAMENTI

Mappa dei concetti – Fondamenti

■ Attacco/tecnica ■ Difesa ■ Norme & metodo ■ IA



```
root@hacklab:~/lezione-16/concetto-1 $
```

LEZIONE 16 / CONCETTO 1/10

[Attacco/tecnica]

Memoria, registri e calling convention

// SPIEGAZIONE

Sfruttare un binario richiede di capire come il processo usa la memoria: lo stack per le chiamate, lo heap per l'allocazione dinamica, i registri per il controllo del flusso. Le calling convention definiscono come passano argomenti e valori di ritorno.

// ESEMPIO

> Il return address salvato sullo stack è il bersaglio classico di un overflow.

```
root@hacklab:~/lezione-16/concetto-2 $
```

LEZIONE 16 / CONCETTO 2/10

[Attacco/tecnica]

Classi di vulnerabilità binarie

// SPIEGAZIONE

Lo stack overflow sovrascrive dati di controllo oltre un buffer; le format string lette male espongono e alterano memoria; i bug di heap come use-after-free sfruttano puntatori a memoria già liberata. Ognuno apre una via al dirottamento del flusso.

// ESEMPIO

> Un use-after-free riusa un oggetto liberato per controllare un puntatore a funzione.

```
root@hacklab:~/lezione-16/concetto-3 $
```

LEZIONE 16 / CONCETTO 3/10

[Difesa]

Shellcoding e ROP/JOP

// SPIEGAZIONE

Quando non si può iniettare codice eseguibile si riusa quello esistente: `ret2libc` salta a funzioni di libreria, ROP e JOP concatenano frammenti (gadget) già presenti per costruire la logica dell'attacco aggirando le difese.

// ESEMPIO

> Una catena ROP mette in sequenza gadget per invocare `execve` senza iniettare shellcode.

```
root@hacklab:~/lezione-16/concetto-4 $
```

LEZIONE 16 / CONCETTO 4/10

[Difesa]

Mitigazioni e loro bypass

// SPIEGAZIONE

Le difese moderne rendono l'exploitation ardua: ASLR randomizza gli indirizzi, DEP/NX impedisce l'esecuzione dei dati, i canary proteggono lo stack, PIE e RELRO irrobustiscono il binario. Bypassarle richiede leak di memoria e tecniche combinate.

// ESEMPIO

> Un info leak rivela un indirizzo e vanifica l'ASLR, abilitando la catena ROP.

```
root@hacklab:~/lezione-16/concetto-5 $
```

LEZIONE 16 / CONCETTO 5/10

[Difesa]

Fuzzing e triage dei crash

// SPIEGAZIONE

Il fuzzing bombarda il programma con input anomali per provocare crash che rivelano bug; il triage distingue i crash sfruttabili da quelli innocui. Le tecniche valgono anche su Windows, con mitigazioni e strumenti propri.

// ESEMPIO

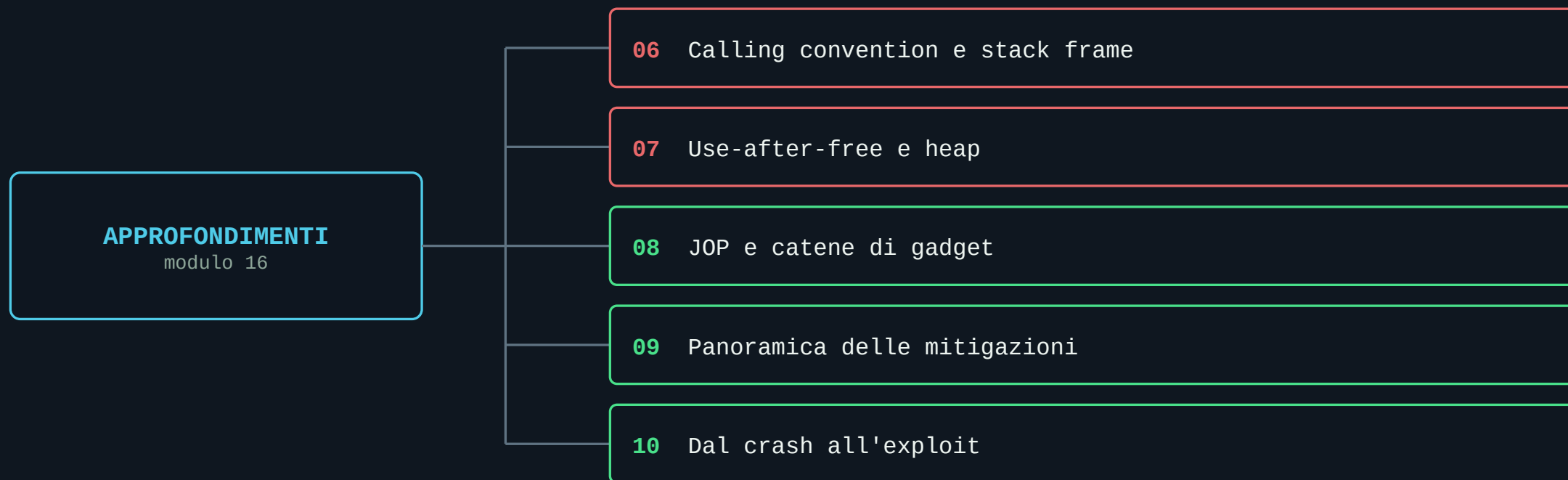
```
$ afl-fuzz -i input -o out ./target (genera crash poi classificati per sfruttabilita)
```

```
root@hacklab:~/lezione-16/mappa-2 $
```

LEZIONE 16 / MAPPA // APPROFONDIMENTI

Mappa dei concetti – Approfondimenti

■ Attacco/tecnica ■ Difesa ■ Norme & metodo ■ IA



```
root@hacklab:~/lezione-16/concetto-6 $
```

LEZIONE 16 / CONCETTO 6/10

[Attacco/tecnica]

Calling convention e stack frame

// SPIEGAZIONE

Senza sapere come sono disposti argomenti, return address e variabili locali nello stack frame non si capisce cosa si sovrascrive e in quale ordine. Le convenzioni cambiano tra architetture e sistemi operativi.

// ESEMPIO

> In x64 i primi argomenti passano nei registri, non sullo stack: cambia la strategia.

```
root@hacklab:~/lezione-16/concetto-7 $
```

LEZIONE 16 / CONCETTO 7/10

[Attacco/tecnica]

Use-after-free e heap

// SPIEGAZIONE

L'allocatore riusa i blocchi liberati secondo liste chiamate bin: capirne il funzionamento permette di far riallocare un oggetto controllato al posto di uno liberato, trasformando un use-after-free in controllo del flusso.

// ESEMPIO

> Si libera un oggetto e si rialloca al suo posto una struttura con un puntatore controllato.

```
root@hacklab:~/lezione-16/concetto-8 $
```

LEZIONE 16 / CONCETTO 8/10

[Difesa]

JOP e catene di gadget

// SPIEGAZIONE

Jump-oriented programming usa gadget che terminano con salti invece di return, eludendo difese basate sul return address. Costruire catene affidabili richiede di trovare i gadget giusti nel binario o nelle librerie caricate.

// ESEMPIO

> Una catena JOP evita i controlli sul ritorno saltando tra gadget selezionati.

```
root@hacklab:~/lezione-16/concetto-9 $
```

LEZIONE 16 / CONCETTO 9/10

[Difesa]

Panoramica delle mitigazioni

// SPIEGAZIONE

Ogni mitigazione chiude una porta: NX i dati eseguibili, ASLR e PIE gli indirizzi prevedibili, i canary la sovrascrittura dello stack, RELRO la GOT, CFG i salti indiretti su Windows. La robustezza nasce dal loro insieme.

// ESEMPIO

> Con DEP/NX attivo non si esegue shellcode sullo stack: serve il riuso di codice (ROP).

```
root@hacklab:~/lezione-16/concetto-10 $
```

LEZIONE 16 / CONCETTO 10/10

[Difesa]

Dal crash all'exploit

// SPIEGAZIONE

Non ogni crash è sfruttabile: il triage analizza registri e memoria al momento del crash per capire se si controlla il flusso. Su Windows cambiano strumenti come WinDbg e le mitigazioni, ma la logica resta la stessa.

// ESEMPIO

> Un crash con il program counter sotto controllo è un candidato forte a diventare exploit.

```
root@hacklab:~/lezione-16/strumenti $
```

LEZIONE 16 / STRUMENTI

Strumenti della lezione

// TOOLKIT

GDB (pwndbg/GEF)

pwntools

Ghidra

Radare2

checksec

R0Pgadget

AFL++

// NOTA OPERATIVA

> Ogni strumento si usa in ambiente virtualizzato e isolato, solo su bersagli di proprietà o autorizzati. Output e artefatti vanno protetti e, a fine ingaggio, gestiti o distrutti.

```
root@hacklab:~/lezione-16/lab-1 $
```

LEZIONE 16 / LABORATORIO 1

Laboratorio 1

\$ Stack overflow con shellcode

// PROCEDURA

[1] Sfrutta uno stack overflow con shellcode su un binario senza mitigazioni

[!] Verifica: A cosa serve il canary e come si rileva la sua assenza?

```
root@hacklab:~/lezione-16/lab-2 $
```

LEZIONE 16 / LABORATORIO 2

Laboratorio 2

\$ ROP e bypass di DEP/ASLR

// PROCEDURA

[1] Costruisci una catena ROP usando un info leak per sconfiggere ASLR

[!] Verifica: Perché DEP rende insufficiente lo shellcode classico?

```
root@hacklab:~/lezione-16/lab-3 $
```

LEZIONE 16 / LABORATORIO 3

Laboratorio 3

\$ Fuzzing e triage (AFL++)

// PROCEDURA

[1] Usa AFL++ per scoprire un crash e farne il triage fino alla causa

[!] Verifica: Quali crash sono realmente sfruttabili?

```
root@hacklab:~/lezione-16/caso-di-studio $
```

LEZIONE 16 / CASO DI STUDIO

Caso di studio

// SCENARIO

> Vulnerabilità di memory corruption storica e la sua weaponizzazione.

// ANALISI RICHIESTA

- > Ricostruisci la dinamica dei fatti con metodo e tracciabilità
- > Individua errori, cause radice e punti di decisione critici
- > Proponi contromisure, controlli e lezioni apprese

```
root@hacklab:~/lezione-16/etica $
```

LEZIONE 16 / ETICA & DIRITTO

Nota etico-legale

// PRINCIPI INDEROGABILI

> autorizzazione scritta scope definito ambiente isolato dati protetti responsible disclosure

// SPECIFICO DELLA LEZIONE

[!] Solo binari di laboratorio/CTF; exploit per software reale -> autorizzazione e responsible disclosure.

```
root@hacklab:~/lezione-16/ricapitolazione $
```

LEZIONE 16 / RICAPITOLAZIONE

Ricapitolazione

// PUNTI CHIAVE

- > L'organizzazione della memoria di un processo
- > Overflow e corruzioni della memoria
- > Il funzionamento delle mitigazioni moderne
- > Affidabilità e portabilità di un exploit
- > Un exploit funzionante documentato

```
root@hacklab:~/lezione-16/verifica $
```

LEZIONE 16 / AUTOVALUTAZIONE

Autovalutazione

// DOMANDE DI VERIFICA

- ? A cosa serve il canary e come si rileva la sua assenza?
- ? Perché DEP rende insufficiente lo shellcode classico?
- ? Quali crash sono realmente sfruttabili?

// VALUTAZIONE

Sviluppo di un exploit completo e documentato.

// LETTURE

Erickson, The Art of Exploitation; doc. pwntools.

lezione 16 completata

\$ next

Binary Exploitation e sviluppo di exploit

[!] Le competenze di questa lezione si impiegano esclusivamente in contesti legali, autorizzati e deontologicamente corretti, entro lo scope e con autorizzazione scritta.

root@hacklab:~# _

root@hacklab:~/contatti \$

CONTATTI / RISORSE

Restiamo in contatto

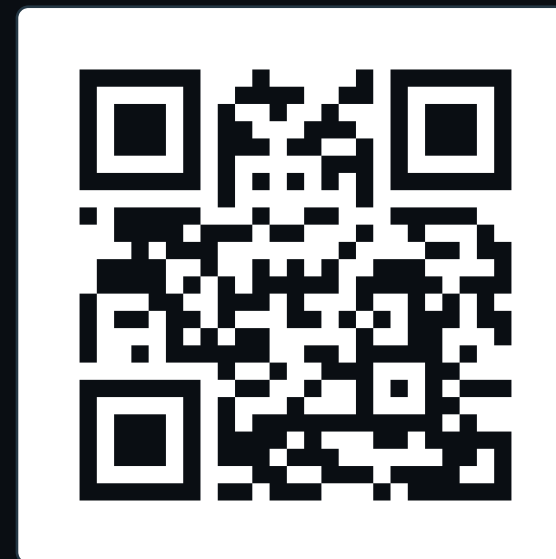


Vincenzo Calabro

cyber security / digital forensics /
docenza e divulgazione

// SCANSIONA O VISITA

> vincenzocalabro.it



Inquadra il QR con la fotocamera
per aprire il sito.