REPORT 11/01/2023 LORENZO SCORBAIOLI

TASK:

Dato il seguente codice:

0x00001141 <+8>: mov EAX, 0x20
0x00001148 <+15>: mov EDX, 0x38
0x00001141 <+28>: add EAX, EDX
0x00001141 <+30>: mov EBP, EAX
0x00001141 <+33>: cmp EBP, 0xa extended base pointer
0x00001141 <+37>: jge 0x1176 <main+61>
0x00001141 <+49>: mov EAX, 0x0
0x00001141 <+54>: call 0x1030 <printf@plt>

identificare lo scopo di ogni istruzione, inserendo una descrizione per ogni riga di codice

- **0x00001141 <+8>: mov EAX, 0x20**

Come abbiamo visto nella lezione teorica di questa mattina sappiamo che “EAX” è un registro definito General Purpose, cioè un registro che viene utilizzato dalla CPU durante l’esecuzione e che viene usato per salvare variabili o indirizzi di memoria. In questa specifica riga viene assegnato al registro EAX, tramite l’operazione “mov”, il valore esadecimale 0x20, che in valore decimale equivale a 32.

- **0x00001148 <+15>: mov EDX, 0x38**

Come già spiegato sopra, tramite l’istruzione “mov”, viene definito il valore del registro “EDX”, a cui viene assegnato il valore esadecimale 0x38, che in valori decimali equivale a 56. In linguaggio C si potrebbe tradurre con

- **0x00001141 <+28>: add EAX, EDX**

In questa riga viene introdotto l’operatore aritmetico “add” per sommare il valore in questo caso dei 2 registri EAX e EDX, andando ad aggiornare il valore di EAX con il nuovo valore ottenuto. Sappiamo quindi che EAX equivale a 32 mentre EDX a 56. La somma ha valore 88 e quindi il registro EAX assume il nuovo valore di 88.

- **0x00001141 <+30>: mov EBP, EAX**

In questa riga di comando abbiamo nuovamente l’operatore “mov”. In questa riga abbiamo quindi un’operazione che assegna al registro EBP il valore contenuto nel registro EAX, che come abbiamo visto sopra contiene il valore 88.

- **0x00001141 <+33>: cmp EBP, 0xa**

In questa riga viene introdotta l’istruzione “cmp”. Essa si comporta in maniera similare all’istruzione “sub”, usata per sottrarre i valori di 2 registri, senza andare a modificare i 2 operandi. Tuttavia questa operazione va a modificare il zero flag (ZF) e il carry flag (CF). Questi sono degli status flag, valori che possono avere solo valori 1 o 0, che vengono posti in un apposito registro e in base al valore che assumono questi flag. In questa specifica riga di comando abbiamo quindi 88 (EBP) – 10 (0xa) =78. Quindi ZF e CF assumo rispettivamente i valori di 0 e 0 in quanto 88 >10.

- **0x00001141 <+37>: jge 0x1176 <main+61>**

In questa riga viene introdotta l'istruzione jump. In questo caso l'istruzione è per saltare all'indirizzo di memoria 0x1176 nel caso in cui la destinazione abbia un valore maggiore o uguale al valore dell'istruzione "cmp" contenuta nella riga vista precedentemente. Il <main+61> dovrebbe essere un **position-independent executable (PIE)**, un corpo di codice macchina che, essendo collocato da qualche parte nella memoria primaria, viene eseguito correttamente indipendentemente dal suo indirizzo assoluto.

https://stackoverflow.com/questions/52820264/what-are-those-numbers-in-gdb
https://en.wikipedia.org/wiki/Position-independent_code#PIE

- **0x00001141 <+49>: mov EAX, 0x0**

In questa riga viene assegnato il valore esadecimale 0x0, che in sistema decimale equivale a 0, al registro EAX.

- **0x00001141 <+54>: call 0x1030 <printf@plt>**

In questa riga viene utilizzata l'istruzione "call". Questa istruzione viene utilizzata per chiamare una funzione. Essa esegue 2 operazioni: la prima applica un push dell'indirizzo immediatamente successivo (0x1030) sullo stack. Dopodichè cambia l'extended instruction point nella destinazione di chiamata. In questo modo viene trasferito il controllo al target dell'istruzione"call" e comincia l'esecuzione da lì. Il <printf@plt> è in realtà un piccolo stub che (alla fine) chiama la vera funzione printf, modificando le cose lungo il percorso per rendere più veloci le chiamate successive. La vera funzione printf è mappata in una posizione arbitraria in un dato processo (spazio di indirizzi virtuale), così come il codice che sta tentando di chiamarla. Uno stub è una porzione di codice utilizzata per simulare il comportamento di funzionalità software (come una routine su un sistema remoto) o l'interfaccia COM e può fungere anche da temporaneo sostituto di codice ancora da sviluppare. Sono pertanto utili durante il porting di software, l'elaborazione distribuita e in generale durante lo sviluppo di software e il software testing.

https://stackoverflow.com/questions/5469274/what-does-plt-mean-here
https://it.wikipedia.org/wiki/Stub_(informatica)
https://www.aldeid.com/wiki/X86-assembly/Instructions/call