Ricerca 11102024

Cos'è Assembly

⚜️ **Tag:** #assembly #linguaggiMacchina #sicurezza

Assembly è un linguaggio di programmazione a basso livello strettamente legato all'architettura del processore. Ogni istruzione assembly corrisponde direttamente a un'operazione che il processore può eseguire, rendendolo estremamente efficiente ma difficile da scrivere e leggere rispetto ai linguaggi di alto livello come Python o C. È il linguaggio più vicino al **linguaggio macchina**, che è composto da istruzioni binarie.

Principali caratteristiche:

1. **Lingua di traduzione diretta:** Ogni istruzione assembly viene tradotta da un assembler in un'istruzione eseguibile dal processore (codice macchina).
2. **Registri:** L'assembly manipola direttamente i registri, che sono piccole aree di memoria all'interno della CPU usate per velocizzare il calcolo.
3. **Efficienza:** Permette il controllo preciso delle risorse di sistema, rendendolo ideale per ottimizzare le prestazioni in situazioni critiche.
4. **Architettura-dipendente:** Ogni architettura hardware (ad esempio, x86, ARM) ha il proprio set di istruzioni assembly, quindi il codice assembly scritto per una CPU non può essere usato direttamente su un'altra.

Rilevanza nella sicurezza informatica:

- **Reverse Engineering:** Il codice assembly viene utilizzato per analizzare programmi chiusi, come il malware o software senza codice sorgente, cercando vulnerabilità sfruttabili.
- **Exploit Development:** Scrivere exploit spesso richiede la comprensione del comportamento a livello di registro e memoria, un aspetto cruciale dell'assembly. Ad esempio, sfruttare un buffer overflow richiede di capire come i dati sovrascrivono lo stack.

WinDBG

⚜️ **Tag:** #WinDBG #debugger #sicurezza

WinDBG è uno dei principali debugger utilizzati su piattaforma Windows, particolarmente utile per il **reverse engineering**, l'**analisi post-mortem** dei crash (tramite i dump), e lo sviluppo di exploit.

Funzionalità principali:

1. **Debugging Kernel e User Mode:** Può essere utilizzato per eseguire il debug sia del kernel di Windows che delle applicazioni utente.
2. **Analisi Post-Crash:** Con i file di crash dump, è possibile analizzare lo stato del sistema o del programma nel momento in cui si è verificato il crash. Questo è molto utile per individuare vulnerabilità come buffer overflow.
3. **Breakpoints:** Consente di fermare l'esecuzione del codice in punti specifici per analizzare lo stato dei registri e della memoria.

Comandi comuni in WinDBG:

- **bp [indirizzo]:** Imposta un breakpoint su un indirizzo specifico, permettendo di interrompere l'esecuzione per analizzare lo stato del programma.
- **dd [indirizzo]:** Visualizza i dati in memoria a partire da un certo indirizzo.
- **k (stack trace):** Fornisce uno stack trace per vedere l'elenco delle funzioni chiamate fino a quel momento.

Rilevanza nella sicurezza informatica:

- **Analisi dei malware:** WinDBG può essere utilizzato per eseguire il debug di malware in esecuzione, osservando il loro comportamento senza compromettere il sistema.
- **Sviluppo exploit:** Aiuta a capire come manipolare la memoria e il flusso di esecuzione per creare exploit di vulnerabilità software.

Mona.py

⚜️ **Tag:** #monapy #exploit #pentesting

Mona.py è uno script Python per Immunity Debugger, progettato per automatizzare e semplificare molti aspetti del processo di **exploit development**. Viene utilizzato principalmente per trovare e sfruttare vulnerabilità legate al controllo del flusso di esecuzione, come buffer overflow e **Return-Oriented Programming (ROP)**.

Funzionalità principali di Mona.py:

1. **!mona findmsp:** Trova l'offset esatto dove avviene la sovrascrittura del pointer di ritorno (EIP) durante un buffer overflow. Questo è il punto in cui un exploit potrebbe deviare il flusso di esecuzione.
2. **!mona modules:** Mostra i moduli caricati nel processo, indicandone lo stato in relazione a protezioni come **ASLR (Address Space Layout Randomization)** o **DEP (Data Execution Prevention)**.
3. **!mona jmp:** Cerca indirizzi di istruzioni utili come `JMP ESP`, che possono essere utilizzate per reindirizzare l'esecuzione in un exploit di buffer overflow.
4. **!mona rop:** Cerca gadget ROP all'interno dei moduli caricati, necessari per bypassare le protezioni DEP o ASLR sfruttando ROP chain.

Rilevanza nella sicurezza informatica:

- **Buffer Overflow:** Mona.py semplifica l'individuazione di posizioni in cui l'overflow sovrascrive registri o puntatori cruciali come EIP, necessari per deviare il flusso del programma e inserire codice malevolo.
- **Protezione bypass:** Permette di identificare vulnerabilità in moduli senza protezioni come ASLR, e fornisce gadget ROP per bypassare DEP.
- **Sviluppo exploit:** Con Mona.py è possibile automatizzare gran parte delle operazioni ripetitive nel processo di sviluppo exploit, come il calcolo degli offset e la ricerca di indirizzi utili.

🔑 **Chiavi:**
assembly, WinDBG, mona

Concetti base di Assembly x86

⚜️ **Tag:** #assembly #x86 #sicurezza

- **Registri:** I registri principali su x86 sono:
  1. **EAX, EBX, ECX, EDX:** Registri generali per operazioni aritmetiche/logiche.
  2. **ESP (Stack Pointer):** Indica la posizione corrente dello stack.
  3. **EBP (Base Pointer):** Usato per mantenere il riferimento del frame di chiamata.
  4. **EIP (Instruction Pointer):** Indica la prossima istruzione da eseguire.
- **Stack:** È un'area di memoria per salvare dati temporanei, indirizzi di ritorno e variabili locali. La gestione dello stack è cruciale per capire vulnerabilità come **buffer overflow** e **stack smashing**.
- **Istruzioni principali:**
  1. **MOV:** Sposta dati tra registri o memoria.
  2. **PUSH/POP:** Inserisce o rimuove valori dallo stack.
  3. **CALL/RET:** Gestisce le chiamate di funzione e il ritorno da esse.
  4. **INT:** Esegue interrupt, utilizzato in exploit come **interrupt 0x80** per chiamate di sistema.
- EIP Hijacking: In un buffer overflow, sovrascrivendo EIP, è possibile far eseguire codice arbitrario, solitamente inserito tramite uno shellcode.
- NOP sled: Una tecnica comune consiste nell'usare un'area di memoria piena di istruzioni NOP (No Operation, 0x90), seguita da uno shellcode, per aumentare le probabilità di esecuzione dello shellcode.
- Reverse Engineering e Debugging, attraverso l'uso di tool come IDA Pro, Ghidra o radare2 permette di convertire il codice macchina in assembly per comprendere il funzionamento interno del programma.

WinDBG (Windows Debugger)

⚜️ **Tag:** #WinDBG #debugging #sicurezza

WinDBG è uno strumento avanzato di debugging, sviluppato da Microsoft, utilizzato per diagnosticare e analizzare problemi legati a **sistemi operativi Windows**, **driver**, e **applicazioni**. Fa parte del pacchetto **"Debugging Tools for Windows"** e viene utilizzato per il **debugging del kernel**, l'analisi dei **crash dump**, e per identificare errori nei driver o nelle applicazioni.

Caratteristiche principali di WinDBG

1. **Supporto per il Kernel-Mode e User-Mode Debugging:**
   - **Kernel-Mode Debugging:** Permette il debug a livello di sistema operativo, utile per analizzare problemi come i crash del sistema (es. **Blue Screen of Death** - BSOD) e la risoluzione di errori nei driver.
   - **User-Mode Debugging:** Usato per fare il debug di applicazioni a livello di processo, utile per trovare bug e crash in singole applicazioni.
2. **Analisi dei Crash Dump:**
   - WinDBG è particolarmente efficace nell'analisi dei **crash dump** (minidump o memory dump), file che Windows genera durante un crash. Caricando questi dump, è possibile identificare il modulo o il driver che ha causato il crash.
3. **Simboli (Symbols):**
   - I **simboli** sono una mappa che collega gli indirizzi di memoria alle funzioni e variabili definite nel codice sorgente. Per un debugging efficace, è necessario configurare il percorso ai simboli. Microsoft fornisce un **server di simboli pubblico**, che WinDBG può usare automaticamente.
4. **Script e Automazione:**
   - Supporta l'automazione tramite script in **JavaScript** o **DML (Debugging Markup Language)**, che permette di automatizzare operazioni ripetitive o complesse nel debug.
5. **Estensioni:**
   - WinDBG dispone di estensioni come **!analyze**, che esegue un'analisi preliminare automatica dei dump di memoria, fornendo indicazioni utili sulle cause del crash.
6. **Interfaccia e Comandi:**
   - WinDBG utilizza un'interfaccia a riga di comando con comandi specifici per navigare attraverso la memoria, registri e stack di chiamate. Alcuni comandi fondamentali includono:
     - **k o kb:** Visualizza lo stack di chiamate.
     - **!analyze -v:** Analizza automaticamente un crash dump con un output dettagliato.
     - **lm:** Elenca i moduli caricati.
     - **dd:** Mostra il contenuto della memoria a partire da un indirizzo.

Utilizzi principali di WinDBG

1. **Debugging del Kernel:**
   - Essenziale per i programmatori di driver, permette di analizzare e correggere bug a livello di kernel, identificando crash causati da driver difettosi.
2. **Debugging di applicazioni:**
   - Viene utilizzato per identificare bug che causano crash o comportamenti anomali nelle applicazioni in esecuzione in user-mode.
3. **Analisi dei Crash Dump:**

- È uno degli strumenti primari per analizzare i crash del sistema operativo, come i famosi **Blue Screen of Death (BSOD)**, fornendo indicazioni su quali driver o moduli abbiano causato il crash.

4. **Reverse Engineering e Analisi di Malware:**
   - Spesso utilizzato nel contesto del **reverse engineering** o per l'**analisi di malware**, WinDBG permette di analizzare il comportamento del codice malevolo in esecuzione.

Approfondimento sui Simboli in WinDBG

⚜️ **Tag:** #simboli #WinDBG #debugging

Simboli sono essenziali nel debugging poiché collegano gli indirizzi di memoria a nomi di funzioni, variabili e file sorgente. I simboli facilitano l'interpretazione del codice macchina e semplificano l'analisi del flusso di esecuzione di un programma o crash dump.

Cos'è un file di simboli?

Un **file di simboli** contiene informazioni di debug generate durante la compilazione di un programma. Include dettagli come:

- **Nomi di funzioni e variabili:** Collega gli indirizzi di memoria a nomi significativi.
- **Indirizzi delle funzioni:** Mappa gli indirizzi delle funzioni nel codice eseguibile.
- **Punti di origine nel codice sorgente:** Collega gli indirizzi eseguibili alle righe del codice sorgente.
- **Tipi di dati delle variabili:** Include il tipo di variabili (es. int, float, struct).

Tipi di Simboli

1. **Simboli Privati:**
   - Contengono informazioni dettagliate, incluse le variabili locali e i nomi delle funzioni private. Utilizzati internamente dagli sviluppatori, non vengono distribuiti con il software per motivi di sicurezza.
2. **Simboli Pubblici:**
   - Includono solo informazioni limitate (nomi di funzioni pubbliche e variabili globali) e sono distribuiti più frequentemente per garantire privacy e sicurezza nel debugging.

Funzionamento dei simboli nel debugging:

- **Mappatura degli indirizzi:** Ogni volta che un modulo viene caricato, gli indirizzi di memoria vengono mappati ai simboli per fornire contesto.
- **Caricamento dei simboli:** WinDBG carica automaticamente i simboli dai server configurati se non sono disponibili localmente.
- **Ricerca e visualizzazione:** I simboli permettono di associare indirizzi di memoria a nomi di variabili e funzioni, rendendo comprensibile il codice.

🔑 **Chiavi:**
WinDBG, simboli, debugging, reverse engineering

Mona.py

⚜️ **Tag:** #monapy #exploit #pentesting

- **Cos'è Mona.py?** È uno script Python per Immunity Debugger, ampiamente utilizzato per lo sviluppo di exploit. Aiuta nell'identificazione di buffer overflow, ricerca di indirizzi per ROP gadgets, e gestione delle shellcode.
- **Funzionalità principali:**
  1. **!mona modules:** Visualizza i moduli caricati e le loro proprietà (ad esempio, se sono compilati con protezioni come ASLR o DEP).
  2. **!mona findmsp:** Trova l'offset di sovrascrittura del pointer di ritorno (EIP) in attacchi buffer overflow.
  3. **!mona jmp:** Cerca indirizzi utili per eseguire salti a shellcode.
  4. **!mona rop:** Ricerca gadget ROP (Return-Oriented Programming) per bypassare protezioni come DEP.
- **Rilevanza per la sicurezza:**
  1. **Exploit buffer overflow:** Mona.py facilita la scrittura di exploit sfruttando buffer overflow identificando indirizzi per sovrascrivere EIP.
  2. **Protezione bypass:** Aiuta nel bypass delle tecnologie di protezione moderne (ASLR, DEP) attraverso ROP chain e gadget.
  3. **Sviluppo di exploit custom:** Mona.py permette di automatizzare e semplificare molti passaggi nel processo di creazione di exploit, rendendo il lavoro di un pentester o sviluppatore di exploit più efficiente.

🔑 **Chiavi:**
assembly, x86, WinDBG, mona

---

mona.py codice

In questo esempio, mostro una delle funzioni comuni di `mona.py`, ovvero la generazione di un pattern ciclico, utile per trovare offset nei buffer overflow.

Ecco un codice semplificato con commenti riga per riga:

```python
# Importa le librerie necessarie
import string  # Fornisce strumenti per lavorare con stringhe
import itertools  # Fornisce strumenti per creare iteratori complessi

# Funzione per generare un pattern ciclico
def create_pattern(length):
    """
    Questa funzione genera un pattern ciclico utilizzato per identificare
    l'offset di un buffer overflow. Ad esempio, è usata per trovare l'offset
    dove un EIP (Instruction Pointer) viene sovrascritto.

    :param length: La lunghezza totale del pattern
    :return: Una stringa contenente il pattern ciclico
    """

    # Definizione dei set di caratteri per il pattern
    chars1 = string.ascii_uppercase  # Lettere maiuscole
    chars2 = string.ascii_lowercase  # Lettere minuscole
    chars3 = string.digits           # Numeri

    # Crea un iteratore che combina i tre set di caratteri
    pattern = itertools.product(chars1, chars2, chars3)

    # Genera il pattern ciclico combinando i gruppi creati dall'iteratore
    result = ''.join([''.join(x) for x in pattern])

    # Taglia il pattern alla lunghezza specificata
    return result[:length]

# Funzione per trovare l'offset in base a un dato valore
def find_offset(value, length):
    """
    Questa funzione cerca un determinato valore all'interno del pattern ciclico,
    per trovare l'offset corrispondente.

    :param value: Il valore che si vuole cercare (come EIP sovrascritto)
    :param length: La lunghezza del pattern da cercare
    :return: L'offset dove il valore appare nel pattern
    """
```

```
    # Genera un pattern ciclico con la lunghezza specificata
    pattern = create_pattern(length)

    # Cerca il valore nel pattern ciclico
    offset = pattern.find(value)

    # Restituisce l'offset se trovato, altrimenti -1
    return offset if offset != -1 else "Valore non trovato nel pattern."

# Funzione principale che esegue il codice di test
def main():
    # Lunghezza del pattern ciclico che vogliamo creare
    pattern_length = 500

    # Genera un pattern di esempio
    pattern = create_pattern(pattern_length)
    print(f"Pattern generato: {pattern[:50]}...")  # Mostra i primi 50 caratteri del pattern

    # Esempio di valore da cercare nel pattern (ad esempio un EIP sovrascritto)
    search_value = "Aa3A"

    # Trova l'offset del valore nel pattern
    offset = find_offset(search_value, pattern_length)

    # Mostra il risultato
    print(f"Offset trovato: {offset}")

# Punto di ingresso del programma
if __name__ == '__main__':
    main()
```

Spiegazione del codice:

1. **Importazione delle librerie**:
   - `string`: Usata per accedere rapidamente a caratteri alfabetici e numerici.
   - `itertools`: Fornisce uno strumento per creare una combinazione di set di caratteri (maiuscole, minuscole, numeri).
2. **Funzione** `create_pattern(length)`:

- Crea un pattern ciclico che combina tre gruppi di caratteri: maiuscole, minuscole e numeri.
- Questo pattern viene usato durante gli attacchi di buffer overflow per individuare dove avviene la sovrascrittura dell'EIP.
- `itertools.product(chars1, chars2, chars3)` crea tutte le possibili combinazioni di caratteri maiuscoli, minuscoli e numeri.
- La funzione restituisce il pattern tagliato alla lunghezza specificata.

3. **Funzione** `find_offset(value, length)`:
   - Cerca un valore specifico (ad esempio, una porzione del pattern che sovrascrive l'EIP) all'interno del pattern ciclico.
   - Utilizza la funzione `find` per identificare la posizione (offset) di quel valore nel pattern ciclico.
4. **Funzione** `main()`:
   - Genera un pattern di 500 caratteri.
   - Cerca un esempio di valore (`"Aa3A"`) all'interno del pattern generato.
   - Stampa i primi 50 caratteri del pattern generato e l'offset trovato del valore cercato.
5. **Blocco** `if __name__ == '__main__'`:
   - Esegue il codice solo se lo script viene eseguito direttamente (non importato come modulo).

Utilizzo nel contesto di `mona.py`:

In `mona.py`, funzioni come queste vengono utilizzate per generare automaticamente i pattern ciclici e determinare dove l'EIP (o altri registri) vengono sovrascritti durante l'esecuzione di un exploit di buffer overflow. Queste operazioni aiutano l'exploit developer a determinare con precisione dove avviene il crash e come manipolare l'exploit per ottenere il controllo del flusso del programma.

MEET
OUR TEAM

MICHELE GUIDO

SUSHANTO ROMA

FRANCESCO LETO

ANGELO LOMBARDI

MATTIA DELEU

NICOLO' BIASIO

GET
IN TOUCH

WWW.WOLFEH.COM