

Tecniche per l'analisi di codice binario

InfoSecurity 2003

Andrea Lanzi

andrew@security.dsi.unimi.it

Lorenzo Martignoni

lorenzo@security.dsi.unimi.it

Computer Forensics

Uno dei settori del **Computer Forensics** è l'analisi di programmi in formato binario, solitamente chiamati **malware**, trovati all'interno di sistemi compromessi.

Obiettivi

- Identificare il tipo di binario (architettura / s.o.)
- Individuare il target del binario (agente DDoS, worm, sniffer, ...)
- Determinare che tipo di input sono accettati (rete, riga di comando, segnali, variabili d'ambiente, ...)
- Determinare quali azioni sono eseguite in base agli input ricevuti

Metodi di analisi

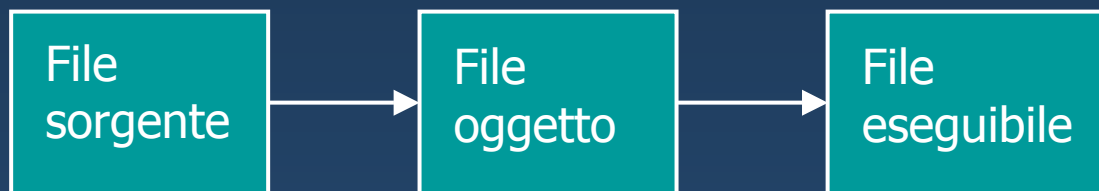
- Analisi attiva: il binario viene studiato a run-time
- **Analisi passiva**: il binario viene studiato senza essere eseguito
- Analisi black-box: il binario viene studiato monitorando i suoi input ed output
- Analisi post-mortem: il binario viene analizzato studiando le conseguenze della sua esecuzione

Analisi passiva

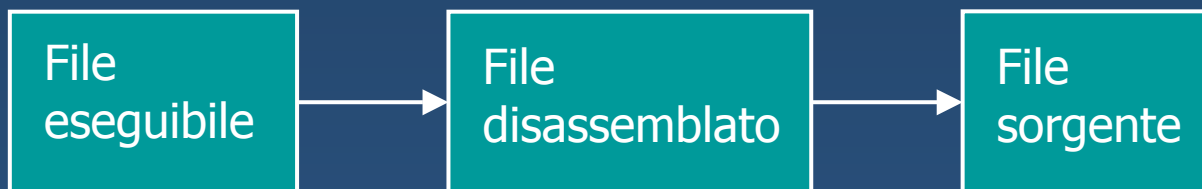
- Vantaggi:
 - Permette di rilevare comportamenti raramente producibili
 - Non necessita di un ambiente (hw/sw) nel quale eseguire il binario
- Svantaggi:
 - Dispendiosa in termini di tempo

Analisi passiva

Compilazione e linking



Decompilazione



Identificazione del tipo di binario

- Architettura e sistema operativo
- Statico / dinamico
- Con / senza tabella dei simboli

```
$ file binary1  
binary1: ELF 32-bit LSB executable, Intel  
80386, version 1 (SYSV), for GNU/Linux  
2.2.0, statically linked, stripped
```

Ricerca di stringhe ASCII

- Si cercano stringhe “interessanti” all’interno del binario, in grado di:
 - Rivelare informazioni relative al compilatore usato
 - Rivelare informazioni relative alle librerie usate dal binario

Ricerca di stringhe ASCII

```
$ strings -a binary1
```

```
@Ph@9Mu.@t1=Ph@
```

```
@P777897@K1sw9823h1v@__##@!#$$@
```

```
...
```

```
MD5 part of OpenSSL 0.9.7 31 Dec 2002
```


```
...
```

```
GCC: (GNU) 3.2.2 20030109 (Debian release)
```

```
...
```

Traduzione del codice macchina

Il codice macchina deve essere tradotto in linguaggio assembler:

50						push	%eax
54						push	%esp
52						push	%edx
68	38	c3	08	08		push	\$0x808c338
68	b4	80	04	08		push	\$0x80480b4
51						push	%ecx
56						push	%esi
68	d0	81	04	08		push	\$0x80481d0
e8	1f	0c	00	00		call	0x8048d20

Librerie di sistema

- Ogni programma compilato staticamente contiene al suo interno tutte le librerie di sistema usate
- Per esempio il programma "hello world" compilato staticamente contiene 80.000 istruzioni assembler
- L'obiettivo è quello di isolare le funzioni di sistema da quelle create dal programmatore

Funzioni di libreria

Librerie:

- Individuazione delle librerie candidate
- **Fingerprint** del **codice invariante** di tutte le funzioni di ogni libreria
- Creazione di un database contenente i fingerprint di tutte le funzioni trovate

Funzioni di libreria

Binario:

- Individuazione di tutte le sue funzioni (es. ricerca delle istruzioni `call`)
- Selezione del codice di ogni funzione (es. ricerca delle istruzioni `leave` e `ret`)
- Fingerprint del codice invariante di ogni funzione
- Ricerca dei fingerprint nel database delle librerie candidate

Funzioni di libreria

```
push    %ebp
mov     %esp, %ebp
sub     $0x4, %esp
nop
mov     0x8(%ebp), %eax
cmp     0xc(%ebp), %eax
jl      11 <foo+0x11>
jmp     33 <foo+0x33>
mov     0xc(%ebp), %edx
mov     %edx, %eax
sar     $0x1f, %eax
shr     $0x1f, %eax
lea    (%eax, %edx, 1), %eax
sar     %eax

cmp     %eax, 0x8(%ebp)
jne     2e <foo+0x2e>
mov     0x8(%ebp), %eax
mov     %eax, -0x4(%ebp)
jmp     3a <foo+0x3a>
inc    0x8(%ebp)
jmp     7 <foo+0x7>
movl   -0x1, -0x4(%ebp)
mov    -0x4(%ebp), %eax
leave
ret
```

HASH

Individuazione del `main()`

- Tracciare il grafo delle chiamate a funzione
- Studiare come viene chiamata dal compilatore la funzione `main()`

```
__start()  
{  
    ...  
    __libc_start_main(..., &main);  
}
```

```
__start()  
{  
    ...  
    main();  
}
```

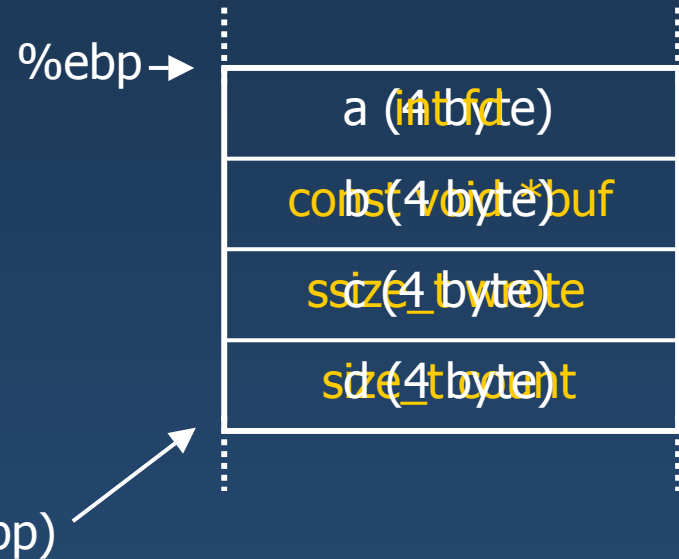
Individuazione delle strutture dati

Obiettivo: individuare le strutture dati utilizzate dal programma.

- Utilizzare i prototipi delle funzioni di libreria:
 - individuare i tipi dei buffer passati come parametri alle funzioni
 - individuare il tipo del buffer ritornato dalle funzioni

Individuazione delle strutture dati

```
push    -0xc(%ebp)
push    -0x8(%ebp)
push    -0x4(%ebp)
call    1d # write
add     $0x10,%esp
mov     %eax,-0x10(%ebp)
```



Prototipo `write()` :

```
ssize_t write(int fd, const void *buf, size_t count);
```



```
-0x10(%ebp) = write(-0x4(%ebp), -0x8(%ebp),  
                  -0xc(%ebp));
```

Individuazione delle strutture dati

- Individuare i tipi di dati attraverso la **propagazione dei tipi**, utilizzando:
 - istruzioni di assegnamento
 - istruzioni di confronto
 - istruzioni di copia
- Signed / unsigned
- Ricostruzione delle strutture dati complesse (struct, union e array)

Individuazione delle strutture dati

```
struct sockaddr *a;
```

```
??? b;
```

```
memcpy(&b, a, sizeof(*a));
```



```
struct sockaddr b;
```

```
int a;
```

```
??? b;
```

```
if(a == b)
```



```
int b;
```

Strutture dati

È possibile distinguere operandi con segno da operandi senza segno analizzando gli operatori di confronto.

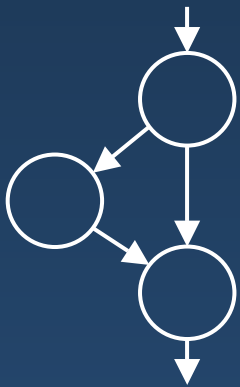
- Operatori di confronto per operandi con segno:

`jg/jnl` `jge/jnl` `jle/jng` `jns` `js`

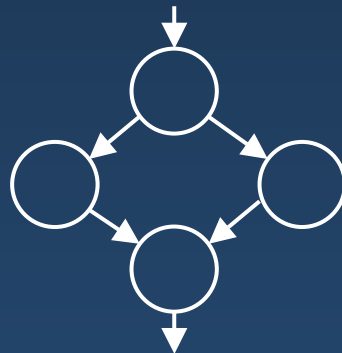
- Operatori di confronto per operandi senza segno:

`ja/jnbe` `jae/jnb` `jb/jnae` `jbe/jna`

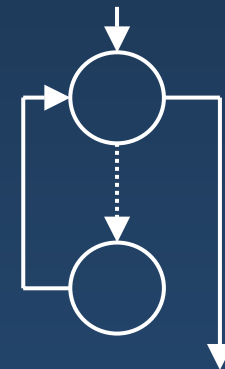
Strutture di controllo di flusso



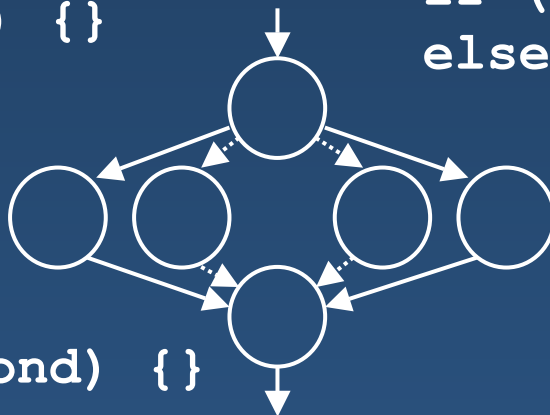
`if (cond) {}`



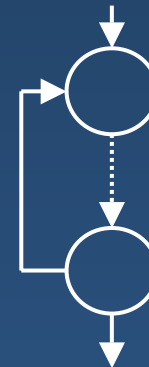
`if (cond) {}
else {}`



`while (cond) {}`



`switch (cond) {}`



`do {} while (cond);`

Salto condizionato ad una via

Salto condizionato ad una via:

```
push    %ebp
mov     %esp,%ebp
sub     $0x8,%esp
cmpl   $0xa,-0x4(%ebp)
jne     1c <foo+0x1c>
```

```
sub     $0xc,%esp
push    $0x0
call   15 <foo+0x15>
add     $0x10,%esp
```

```
mov     -0x8(%ebp),%edx
lea     -0x4(%ebp),%eax
add     %edx,(%eax)
```



```
if (a == 10) {
    /* if body */
}
```

Salto condizionato a 2 vie

Salto condizionato a due vie:

```
cmp1    $0xa, -0x4(%ebp)
```

```
jne     1e <foo+0x1e>
```

```
sub     $0xc, %esp
```

```
push   $0x0
```

```
call   15 <foo+0x15>
```

```
add    $0x10, %esp
```

```
jmp    2e <foo+0x2e>
```

```
sub     $0xc, %esp
```

```
push   $0x9
```

```
call   27 <foo+0x27>
```

```
add    $0x10, %esp
```

```
mov    -0x8(%ebp), %edx
```

```
lea   -0x4(%ebp), %eax
```



```
if (a == 10) {  
    /* if body */  
}  
else {  
    /* else body */  
}
```

Salto condizionato a 2 vie

Salto condizionato a due vie con doppia condizione:

```
cmp1    $0xa, -0x4(%ebp)
jne     24 <foo+0x24>
cmp1    $0x4, -0x8(%ebp)
jg      24 <foo+0x24>
call   1b <foo+0x1b>
jmp     34 <foo+0x34>
call   2d <foo+0x2d>
mov     -0x8(%ebp), %edx
lea    -0x4(%ebp), %eax
add     %edx, (%eax)
```



```
if(a == 10 && b < 5) {
    func1();
} else {
    func2();
}

a += b;
```

Salto condizionato a n vie

```
cmp    $0x1a,%eax
ja     80484dd
jmp    *0x80485c4(,%eax,4)
mov    %esi,%esi
...
jmp    80484dd
add   $-0xc,%esp
...
jmp    80484dd
add    $-0xc,%esp
push   $0x804858e
...
...
call   8048300 <printf>
...
```

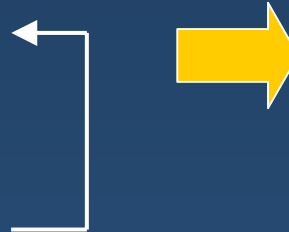
```
switch (a) {
  case 1:
    ...;
    break;
  ...
  case n:
    ...;
    break;
  default:
    ...;
}
```



Ciclo con test in coda

Ciclo con test in coda:

```
push    %ebp
mov     %esp, %ebp
sub     $0x8, %esp
movl    $0x0, -0x4(%ebp)
lea    -0x4(%ebp), %eax
incl   (%eax)
cmpl   $0x9, -0x4(%ebp)
jle    d <foo+0xd>
mov     -0x8(%ebp), %edx
lea    -0x4(%ebp), %eax
add    %edx, (%eax)
```



```
do {
    /* loop body */
} while (a < 10);
```

Ciclo con test in testa

Ciclo con test in testa:

```
movl    $0x0, -0x4(%ebp)
cmpl    $0x9, -0x4(%ebp)
jle     15 <foo+0x15>
jmp     2a <foo+0x2a>
sub     $0x8, %esp
pushl   -0x4(%ebp)
push    $0x0
call    21 <foo+0x21>
add     $0x10, %esp
jmp     d <foo+0xd>
movl    $0x0, -0x4(%ebp)
```

```
while (a < 10) {
    /* loop body */
}
```

Analisi del codice sorgente

Una volta terminata la fase di decompilazione, inizia la fase finale che consiste nell'analizzare il sorgente ad alto livello che si è prodotto nella fase precedente.



Metodi anti-analisi

- Divisione tra istruzioni e dati
- Codice assembler offuscato
- Codice polimorfico
- Codice automodificante
- Assenza di codice