

UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE
DIPARTIMENTO DI INFORMATICA

Reverse engineering: *debugging*

Andrea Lanzi <andrew@security.di.unimi.it>

A.A. 2015–2016

- esecuzione (monitorata) dell'applicazione
 - interazioni con l'ambiente (e.g., file system, rete, registro)
 - interazioni con il sistema operativo (system call)
 - **debugging**

Linux

- `lsdf` – list open files

```
$ lsdf -p 10220
```

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE	NODE	NAME
emacs	10220	sicurezza	cwd	DIR	8,8	4096	48193	/home/sicurezza
emacs	10220	sicurezza	rtd	DIR	8,3	4096	2	/
emacs	10220	sicurezza	txt	REG	8,3	5051492	131642	/usr/bin/emacs22-nox
emacs	10220	sicurezza	mem	REG	8,3	38444	785329	/lib/i686/cmov/libnss_nis-2.7.so
emacs	10220	sicurezza	mem	REG	8,3	87800	785313	/lib/i686/cmov/libnsl-2.7.so
emacs	10220	sicurezza	mem	REG	8,3	30436	785314	/lib/i686/cmov/libnss_compat-2.7.so
emacs	10220	sicurezza	mem	REG	8,3	42504	785326	/lib/i686/cmov/libnss_files-2.7.so
...								

- `netstat`, `/proc/<pid>/*`, `tcpdump`, `wireshark`, ...

Windows

- utility SysInternals (www.sysinternals.com)
- FileMon, RegMon, TcpView, ...

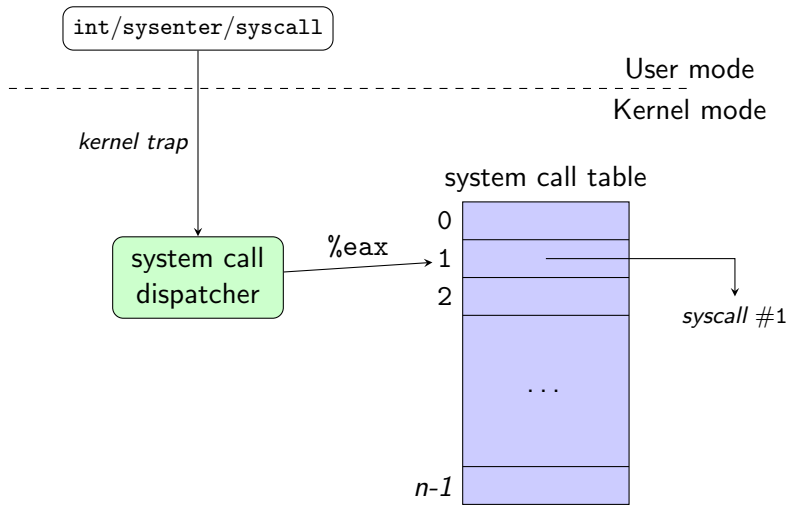
- interfaccia tra sistema operativo e applicazioni utente
- per compiere operazioni “al di fuori” dell’address space del processo
- rivelano informazioni utili sul comportamento di un processo

Metodi per invocazione *system call*

- `int` (Linux: `int 0x80`, Windows: `int 0x2e`)
- `sysenter/sysexit` (\geq Intel Pentium II)
`syscall/sysret` (\geq AMD K6)
- numero `syscall` in `%eax`
- parametri nei registri *general purpose*

System call

Come funzionano



strace

- visualizza chiamate a sistema, parametri, segnali, ...
- possibilità di seguire i processi figlio
- attach a processi in esecuzione
- basato su ptrace

Esempio

```
$ strace /bin/ls 2>&1 | head -n 10
execve(/bin/ls, [/bin/ls], [/* 34 vars */]) = 0
brk(0)                                = 0x8e93000
access(/etc/ld.so.nohwcap, F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7f74000
access(/etc/ld.so.preload, R_OK)      = -1 ENOENT (No such file or directory)
open(/etc/ld.so.cache, O_RDONLY)      = 3
fstat64(3, st_mode=S_IFREG|0644, st_size=58489, ...) = 0
mmap2(NULL, 58489, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f65000
close(3)                               = 0
access(/etc/ld.so.nohwcap, F_OK)      = -1 ENOENT (No such file or directory)
...
```

strace

- visualizza chiamate a sistema, parametri, segnali, ...
- possibilità di seguire i processi figlio
- attach a processi in esecuzione
- basato su ptrace

Opzioni *più* utili

```
-f          traccia anche i figli
-i          stampa l'indirizzo dell'istruzione che invoca la syscall
-e read=x,y,... stampa tutti i dati delle read su fd x,y,...
-e write=x,y,... stampa tutti i dati delle write su fd x,y,...
-ff -o filename salva su file
-s n       stampa n caratteri di ogni stringa (default 32)
```

- NTAPI (*native* Win32 API)
- BindView strace
 - simile a strace Linux
- WUSSTrace - by ex LaSER
(<https://code.google.com/p/wusstrace/>)
 - user-space
 - analisi precisa degli argomenti
 - interfaccia XML, binding Python

- NTAPI (*native* Win32 API)
- BindView strace
 - simile a strace Linux
- WUSSTrace - by ex LaSER
(<https://code.google.com/p/wusstrace/>)
 - user-space
 - analisi precisa degli argomenti
 - interfaccia XML, binding Python

Come funzionano?

- **kernel-mode**: hijack System Service Descriptor Table (SSDT)
- **user-mode**: hijack KiFastSystemCall & KiIntSystemCall (ntdll.dll)

Tracer per funzioni di libreria

Esempio

```
$ ltrace /bin/ls
__libc_start_main(0x804e5f0, 1, ... <unfinished...>
setlocale(6, '') = it_IT.ISO-8859-15@euro
bindtextdomain(coreutils, /usr/share/locale) = /usr/share/locale
textdomain(coreutils) = coreutils
__cxa_atexit(0x8051860, 0, 0, 0xb7f01ff4, 0xbf95cf98) = 0
isatty(1) = 0
getenv(QUOTING_STYLE) = NULL
...
getenv(BLOCK_SIZE) = NULL
getenv(COLUMNS) = NULL
ioctl(1, 21523, 0xbf95cf6c) = -1
...
```

ltrace: come funziona?

- basata su ptrace()
- breakpoint in corrispondenza dei simboli importati

```
$ cat test.c
int main()
{
    printf("*printf: \\x%.2x\\n", *(unsigned char*) printf);
    return 0;
}
$ ./test
*printf: \xff
$ ltrace ./test 2> /dev/null
*printf: \xcc
```

ltrace: come funziona?

- basata su ptrace()
- breakpoint in corrispondenza dei simboli importati

```
$ cat test.c
int main()
{
    printf("printf: \\x%.2x\\n", *(unsigned char*) printf);
    return 0;
}
$ ./test
printf: \xff
$ ltrace ./test 2> /dev/null
printf: \xcc
```

Domanda

- funziona con static linking?

- esecuzione “controllata” di un processo
- granularità a livello della *singola* istruzione
- possibilità di interrompere l'esecuzione (**breakpoint**/watchpoint)
- possibilità di esaminare lo stato di CPU/memoria

Debugger

- *Linux*: **gdb**
- *Windows*: OllyDbg, SoftIc3, WinDbg

Come funzionano?

- istruzione target sovrascritta con `int3`
- segnale SIGTRAP

Come funzionano?

- istruzione target sovrascritta con `int3`
- segnale `SIGTRAP`

```
EIP → mov  (%edx),%eax  
      mov  %eax,0xc(%ebp)  
      mov  (%ecx),%eax  
      mov  %eax,0x8(%ebp)  
      mov  (%esp),%ebx  
      mov  0x4(%esp),%esi  
      mov  0x8(%esp),%edi
```

Come funzionano?

- istruzione target sovrascritta con `int3`
- segnale SIGTRAP

```
EIP → mov  (%edx),%eax           ① set int3
      mov  %eax,0xc(%ebp)
      mov  (%ecx),%eax
      int3
      mov  (%esp),%ebx
      mov  0x4(%esp),%esi
      mov  0x8(%esp),%edi
```


Come funzionano?

- istruzione target sovrascritta con `int3`
- segnale SIGTRAP

```
EIP → mov  (%edx),%eax           ① set int3
      mov  %eax,0xc(%ebp)      ② continue
      mov  (%ecx),%eax
      int3
      mov  (%esp),%ebx
      mov  0x4(%esp),%esi
      mov  0x8(%esp),%edi
```

Come funzionano?

- istruzione target sovrascritta con `int3`
- segnale `SIGTRAP`

```
mov  %edx,%eax
mov  %eax,%ebx
mov  (%ecx),%eax
EIP → int3
mov  (%esp),%ebx
mov  0x4(%esp),%esi
mov  0x8(%esp),%edi
```

① set int3
② continue
③ SIGTRAP

Come funzionano?

- istruzione target sovrascritta con `int3`
- segnale `SIGTRAP`

<code>mov</code>	<code>(%edx),%eax</code>	①	<code>set int3</code>
<code>mov</code>	<code>%eax,0xc(%ebp)</code>	②	<code>continue</code>
<code>mov</code>	<code>(%ecx),%eax</code>	③	<code>SIGTRAP</code>
<i>EIP</i> →	<code>mov</code>	<code>%eax,0x8(%ebp)</code>	④ <code>restore original</code>
<code>mov</code>	<code>(%esp),%ebx</code>		
<code>mov</code>	<code>0x4(%esp),%esi</code>		
<code>mov</code>	<code>0x8(%esp),%edi</code>		

Come funzionano?

- istruzione target sovrascritta con `int3`
- segnale `SIGTRAP`

<code>mov</code>	<code>(%edx),%eax</code>	①	<code>set int3</code>
<code>mov</code>	<code>%eax,0xc(%ebp)</code>	②	<code>continue</code>
<code>mov</code>	<code>(%ecx),%eax</code>	③	<code>SIGTRAP</code>
<code>mov</code>	<code>%eax,0x8(%ebp)</code>	④	<code>restore original</code>
<i>EIP</i> →	<code>mov</code>	⑤	<code>single step</code>
	<code>(%esp),%ebx</code>		
	<code>mov</code>		
	<code>0x4(%esp),%esi</code>		
	<code>mov</code>		
	<code>0x8(%esp),%edi</code>		

Come funzionano?

- istruzione target sovrascritta con `int3`
- segnale SIGTRAP

<code>mov</code>	<code>(%edx),%eax</code>	①	<code>set int3</code>
<code>mov</code>	<code>%eax,0xc(%ebp)</code>	②	<code>continue</code>
<code>mov</code>	<code>(%ecx),%eax</code>	③	<code>SIGTRAP</code>
<code>int3</code>		④	<code>restore original</code>
<code>EIP</code> → <code>mov</code>	<code>(%esp),%ebx</code>	⑤	<code>single step</code>
<code>mov</code>	<code>0x4(%esp),%esi</code>	⑥	<code>set int3</code>
<code>mov</code>	<code>0x8(%esp),%edi</code>		

Problemi

- “invasivi”
- modificano l'address space del processo
- facili da rilevare
- software watchpoint sono molto lenti. . .

Problemi

- “invasivi”
- modificano l'address space del processo
- facili da rilevare
- software watchpoint sono molto lenti. . .

Soluzione

- **hardware** breakpoint

Breakpoint

Hardware breakpoint

- maggior trasparenza
- registri dedicati (DR0 → DR7 su x86)
- efficienti

- maggior trasparenza
- registri dedicati (DR0 → DR7 su x86)
- efficienti

Registri di debug x86

- DR0 → DR3: VA breakpoint
- DR4, DR5: *riservati*
- DR6: status register
- DR7: control register

Breakpoint

Hardware breakpoint

- maggior trasparenza
- registri dedicati (DR0 → DR7 su x86)
- efficienti

Registri di debug x86

- DR0 → DR3: VA breakpoint
- DR4, DR5: *riservati*
- DR6: status register
- DR7: control register

Problemi

- richiedono supporto hardware
- disponibili in numero limitato

- system call Linux
- un processo padre può osservare, controllare e manipolare l'esecuzione di un processo figlio

- system call Linux
- un processo padre può osservare, controllare e manipolare l'esecuzione di un processo figlio

Processo figlio

- `ptrace(PTRACE_TRACEME, 0, NULL, NULL); (+ exec())`
- parent notificato via `wait()` per segnali e chiamate a `execve()`

- system call Linux
- un processo padre può osservare, controllare e manipolare l'esecuzione di un processo figlio

Processo figlio

- `ptrace(PTRACE_TRACEME, 0, NULL, NULL); (+ exec())`
- parent notificato via `wait()` per segnali e chiamate a `execve()`

Processo padre

- lettura VA (`PTRACE_PEEKTEXT`) e registri (`PTRACE_PEEKUSER`)
- modifica VA (`PTRACE_POKETEXT`) e registri (`PTRACE_POKEUSER`)
- resume esecuzione (`PTRACE_CONTINUE`)
- single step a livello di istruzione (`PTRACE_SINGLESTEP`) o di entry/exit system call (`PTRACE_SYSCALL`)
- ...

```
$ wget http://security.di.unimi.it/sicurezza1314/antidbg.tar.gz
```

```
$ wget http://security.di.unimi.it/sicurezza1314/gdbinit73.txt  
$ mv gdbinit73.txt ~/.gdbinit
```

int3 & SIGTRAP {ex1}

```
void handler(int x)
{
    printf("Intercettato int 3!\n");
}

void main(int argc, char **argv)
{
    signal(SIGTRAP, handler);
    asm("int3\n");
}
```

```
(gdb) run
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0804847b in main ()
(gdb) signal SIGTRAP
Continuing with signal SIGTRAP.
Intercettato int 3!
```

Code checksumming {ex2}

```
void check(void *f, int checksum)
{
    // ... calc a generic checksum c of function f
    if(c != checksum)
        printf('Debugger present\n');
}
void main()
{
    check(main, 12345);
    // ....
}
```

Trappare su controllo checksum e patchare dinamicamente il risultato

Limiti di ptrace {ex3}

- un processo può essere tracciato da **un solo** altro processo
- in caso contrario ptrace() fallisce

```
#include <stdio.h>
#include <sys/ptrace.h>

int main(int argc, char **argv)
{
    if(ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
        printf("Debugger present!\n");
        return 1;
    }

    printf("Hello world!\n");
    return 0;
}
```

Soluzione

- `ptrace()` è anche una funzione di libreria
- \Rightarrow ridefinire `ptrace()` in modo che ritorni sempre 0

```
int ptrace(int a, int b, int c, int d)
{
    return 0;
}
```

```
$ gcc -shared ptrace.c -o ptrace.so
```

```
$ LD_PRELOAD=./ptrace.so ./antidebug
(gdb) set environment LD_PRELOAD=./ptrace.so
...
$ strace -fi -E LD_PRELOAD=./ptrace.so ./ex3
```

Checking for extra file descriptors {ex4} (gdb only)

```
void main()
{
    int fd = open('/dev/shm/test', O_CREAT|O_RDONLY, S_IRWXU);
    if(fd > 3) {
        printf('Debugger present %d!\n', fd);
        return 1;
    }
    printf('Hello world!\n');
    return 0;
}
```

```
$ gdb ./ex4
(gdb) run
Starting program: /home/sicurezza/antidbg/ex4
Debugger present 7!
```

- *molti* metodi
- spesso utilizzati dal malware

Alcuni esempi

- `kernel32!IsDebuggerPresent`
- `PEB!NtGlobalFlags` → flag particolari se creato in un debugger
- `ntdll!NtQueryInformationProcess`, con
`ProcessInformationClass = ProcessDebugPort`
- ...
- ...

- applicazione eseguita in ambiente emulato/virtualizzato
- informazioni registrate “dall'esterno” della VM
- utile per malware analysis
- *vantaggi*: trasparenza
- *svantaggi*: OS-dependent, anti-emulation/anti-VM

Strumenti

- Anubis
- CWSandbox
- Norman SandBox
- ...

Emulazione e virtualizzazione

- *emulazione*: **tutte** le istruzioni del programma sono emulate (e.g., QEMU, BOCHS)
- *virtualizzazione*: emulazione delle sole istruzioni (e.g., VMWare, VirtualBox) **privilegiate**

Emulazione e virtualizzazione

- *emulazione*: **tutte** le istruzioni del programma sono emulate (e.g., QEMU, BOCHS)
- *virtualizzazione*: emulazione delle sole istruzioni (e.g., VMWare, VirtualBox) **privilegiate**

Anti-virtualizzazione

```
Y:\>type test.c
typedef struct __attribute__((__packed__)) {
    uint16_t Limit;
    uint32_t Descriptors;
} IDT, *PIDT;

int main()
{
    IDT idtr;
    asm("sidt %0;" : "=m" (idtr));
    printf("addr: %.8x\n", idtr.Descriptors);
    return 0;
}

Y:\>test.exe           (macchina fisica)
addr: 8003f400

Y:\>test.exe           (macchina virtuale)
addr: f7406390
```

Emulazione e virtualizzazione

- *emulazione*: **tutte** le istruzioni del programma sono emulate (e.g., QEMU, BOCHS)
- *virtualizzazione*: emulazione delle sole istruzioni (e.g., VMWare, VirtualBox) **privilegiate**

Anti-emulazione

- sfruttare errori nell'implementazione dell'emulatore
- *A fistful of red-pills: How to automatically generate procedures to detect CPU emulators*

2 Dicembre @ 23.59.59

1) Reversare il programma che e' sul sito homework4. Scrivere una relazione (pdf/txt) che descriva il comportamento del programma, la chiave individuata al suo interno, i dettagli di ogni funzionalità di obfuscation e anti debugging riscontrata e come avete fatto a bypassarla.

2) Scrivere un programma C che, utilizzando la libreria PTRACE, effettui tracing delle system call di un programma (specificato da linea di comando). Per ogni syscall intercettata, il programma deve stampare a video: il numero della system call, il valore dei registri general purpose (%ebx,%ecx,%edx,%esi,%edi) e il valore di ritorno, secondo questo esempio:

```
syscall_#100(0xcafebabe, 0xdeadbabe, 0xbadbadoo, 0x0, 0x0) = 0x0
```

Inoltre, estrarre e stampare gli argomenti delle seguenti syscall:

```
exit, open, read, write, execve
```

esempio:

```
syscall_write(1, 'Hello world!', 0xc)
```

Anche in questo caso allegare una **breve** relazione (pdf/txt) che illustri eventuali problemi affrontati e soluzioni adottate durante lo sviluppo.