Übungen zur Vorlesung Systemsicherheit

Library Preloading

Tilo Müller, Reinhard Tartler, Michael Gernoth

Lehrstuhl Informatik 1 + 4

15. Dezember 2010

Aufgabeblatt 3

Shared Libraries

Aufgabenblatt 3 | Shared Libraries

Beispiele:

- Wichtigste: **GNU C Library** (`libc`)
  - Standard C Bibliothek
  - C hat nur wenig fest eingebaute Funktionalität
    (*Keywords*: `int for return sizeof + &&` ...)
  - ca. 200 häufige Funktionen
  - Beispiele: `printf strcmp memcpy malloc read write open` ...
  - Alternativen: BSD's Lib-C, diet libc, uClibc, ...
- Weitere Bibliotheken:
  - OpenSSL (`libcrypto`, `libssl`)
  - GUI (`libcairo`, `libglib`, `libgtk`, ...)
  - Math (`libm`), Crypto (`libcrypt`), ...
  - etc.

Aufgabenblatt 3 | Dynamically linked

Statisch vs. dynamisch gelinkt:

- **Statisch**: Bibliotheks-Fkt. werden schon beim Linken fest in das Binary kopiert
- **Dynamisch**: Bibliotheks-Fkt. werden erst beim Ausführen dynamisch geladen
  - Windows: Dynamic Linked Library (`.dll`); Linux: Shared Library (`.so`)
  - Vorteile: einfache Bibliothek-Updates, sparen RAM, kleine Binaries, Library Preloading (bspw. `libtrash`), ...
  - Nachteile: mögliche Bibliotheks-Inkompatibilitäten, schlechtere Portierbarkeit, (minimale) Geschwindigkeitseinbußen, Library Preloading (bspw. A3.1), ...

- GCC: Linkt standardmäßig dynamisch; statisch per `-static`
  $\longrightarrow$ die meisten Linux Programme sind dynamisch gelinkt

ltrace-Werkzeug

```
>> ltrace ls
__libc_start_main(0x804dfe0, 1, 0xffc73604, 0x80567e0, 0x8056790 <unfinished ...>
isatty(1)                                                    = 1
getenv("TABSIZE")                                            = NULL
getopt_long(1, 0xffc73604, "abcdfghiklmnopqrstuvw:xABCDFGHI:"..., 0x8059580, NULL)
                                                             = -1
__errno_location()                                           = 0xf7ce568c
malloc(36)                                                   = 0x805cca0
__errno_location()                                           = 0xf7ce568c
malloc(36)                                                   = 0x805ccc8
malloc(12800)                                                = 0x805ccf0
malloc(16)                                                   = 0x805fef8
strlen(".")                                                  = 1
malloc(2)                                                    = 0x805ff10
memcpy(0x805ff10, ".", 2)                                    = 0x805ff10
__errno_location()                                           = 0xf7ce568c
opendir(".")                                                 = 0xf78c2008
readdir64(0xf78c2008)                                        = 0xf78c2020
readdir64(0xf78c2008)                                        = 0xf78c2038
readdir64(0xf78c2008)                                        = NULL
closedir(0xf78c2008)                                         = 0
_setjmp(0x805bba0, 0xffc730f8, 0xf7dd86f8, 0xf78c2008, 0)    = 0
qsort(0x805ccf0, 0, 128, 0x8049f20)                          = <void>
free(0x805ff10)                                              = <void>
free(NULL)                                                   = <void>
free(0x805fef8)                                              = <void>
exit(0) <unfinished ...>
__fpending(0xf7e904e0, 1, 0, 0x8048f9f, 80)                  = 0
fclose(0xf7e904e0)                                           = 0
```

Aufgabe 3.1
Library Preloading

Aufgabe 3.1 | LD_PRELOAD

Unter Linux kann der dynamische Linker beeinflusst werden durch die Umgebungsvariablen:

- **LD_LIBRARY_PATH**
- **LD_PRELOAD**

→ Sucht an anderen Orten als den Standard-Pfaden nach Libraries
→ Das *Überladen* von Bibliotheken wird so möglich
→ Dieses Überladen hat positive wie negative Auswirkungen:

- Behebung von Kompatibilitäts-Problemen (explizit ältere Libs laden)
- Sichere Bibliotheks-Wrapper (bspw. für `strcpy`, `gets` etc.)
- Monitoring, Debugging, Reverse Engineering
- Rootkits, Backdoors (A3.1)
- Isolation, (fake)chroot, Overlay Filesystems (A3.2)

Aufgabe 3.1 | LD_PRELOAD Example

Beispiel: Schutz des Dateisystems vor nicht vertrauenswürdigen Zugriffen.

open() wrapper

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
int open(const char *path, int mode, ...) {
      printf(‘‘access denied. good bye.’’);
      exit(-1);
}
```

Kompilieren und ausführen

```
gcc -fPIC -c -o open.o open.c
gcc -shared -o open.so open.o -ldl
export LD_PRELOAD=‘‘./open.so’’
vi open.c
access denied. good bye.
```

Aufgabe 3.1 | Evil LD_PRELOAD

Aufgabe:

- Erstellen Sie eine Library `read.so`, die den Befehl `read()` überschreibt. Das modifizierte `read()` soll sich normal verhalten, beim Magic String `\xde\xadXmas` aber die nachfolgenden Zeichen bis zum Nullbyte als `system()` Befehl ausführen.
- Zeigen Sie, wie Sie unter Verwendung von `LD_PRELOAD=./read.so` ein Standard-Programm wie `cat` durch Lesen von `evil.txt` dazu bewegen über Netcat eine Bind-Shell auf Port 7235 zu öffnen.

Beispiel

```
$ gcc -fPIC -c -o read.o read.c
$ gcc -shared -o read.so read.o -ldl
$ export LD_PRELOAD=``./read.so''
$ cat evil.txt     #opens bind shell

$ nc localhost 7235     #connects to bind shell
```

Aufgabe 3.2

Overlay FS

Aufgabe 3.2 | Overlay FS

Aufgabe:

- Erstellen Sie eine Library, die Zugriffe auf zugangsbeschränkte Verzeichnisse umleitet
- Ein temporäres Verzeichnis wird für den Prozess überlagert: `/etc` → `/tmp/syssec-10477/etc`

Beispiel

```
$ ls /protected
in_real_dir
$ ls /tmp/syssec-10477/protected
in_shadow_dir
$ LD_PRELOAD=./overlayfs.so ls /protected
in_real_dir in_shadow_dir
$ touch /protected/new_file
touch: cannot touch ‘/protected/new_file’: Permission denied
$ LD_PRELOAD=./overlayfs.so touch /protected/new_file
$ LD_PRELOAD=./overlayfs.so ls /protected
in_real_dir in_shadow_dir new_file
```

Aufgabe 3.2 | Zugriff auf den dynamischen Linker

Explizites Laden von Funktionen aus Bibliotheken zur Laufzeit:

Anfragen an den dynamischen Linker während des Programmablaufs

```
void *dlopen(const char *filename, int flag);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
```

Benötigt zusätzliche Bibliothek:

Linken im Makefile

```
SOURCES = overlayfs.c utils.c
overlayfs.so: $(SOURCES)
        $(CC) $(CFLAGS) $^ -o $@ -shared -ldl
```

Aufgabe 3.2 | Zugriff auf überladene Funktionen

Ursprüngliche Funktionen können mit `dlsym()` aufgerufen werden:

Nutzung von dlsym()

```
#define _GNU_SOURCE 1
#include <dlfcn.h>

FILE *fopen(const char *path, const char *mode) {
        FILE* (*func) (const char*, const char*) = NULL;

        func = (FILE* (*) (const char*, const char*))

        dlsym(RTLD_NEXT, "fopen");

        return (*func) (path, mode);
}
```

Aufgabe 3.2 | Overlay FS Funktionen

- Für die Aufgabe müssen nur die wichtigsten Dateisystemfunktionen überladen werden
- Folgende Funktionen der libc müssen mindestens überladen werden:
  - `open`, `open64`
  - `fopen`, `fopen64`
  - `__xstat`, `__xstat64`, `__lxstat`, `__lxstat64`
  - `opendir`, `readdir`, `readdir64`
  - `creat`, `creat64`
  - `access`
  - `chmod`, `fchmod`
  - `acl_extended_file`
- Andere Funktionen mit Dateinamen können ignoriert werden

Aufgabe 3.2 | Overlay FS Hinweise (1/2)

- Bei Dateioperationen muss in zwei Verzeichnissen gesucht werden:
  - dem “eigentlichen” Verzeichnis
  - dem “Schattenverzeichnis”
- Bei `readdir()` wird zusätzlich der DIR-Pointer des Schattenverzeichnisses benötigt
  - Dieser kann z.B. bei `opendir()` zusätzlich erzeugt und gespeichert werden
- Für die einfache Benutzung der Bibliothek bietet sich ein Shell-Wrapper an:

Shell-Wrappper

```
#!/bin/sh

LD_PRELOAD=/path/to/overlayfs.so "$@"
```

Aufgabe 3.2 | Overlay FS Hinweise (2/2)

- `open{,64}()`
  - Hat eine variable Anzahl an Argumenten
  - Ein evtl. vorhandenes drittes Argument muss extrahiert werden:

open()-Argumente

```
int open (__const char *__file, int flags, ...) {
        mode_t mode = 0;
        va_list args;
        ...
        if (flags & O_CREAT) {
                va_start(args, flags);
                mode = va_arg(args, mode_t);
                va_end(args);
        }
```

Ende

Nächstes Jahr:

Aufgabenblatt 4: Binary Exploitation

Aufgabenblatt 5: Real World Exploits