

Managed Program Execution

Operating Systems I

Viktor Iakovlev (Victor Yacovlev)

Why to Manage

- Debugging Purposes
 - Step-by-step execution and watching
 - Time measurements
- Untrusted Program Execution
 - Restrict access to resources
- Change Third-Party Program Behaviour
 - The way to adopt software with no sources

Interpreting v.s. Native

- Interpreting Languages: Python, Perl, PHP
Manageable by design
- Native Execution: C/C++, Pascal, Fortran
Hard to control execution
- Hybrid: Java, C#
Might be controled by design

Techniques to Manage

- Low-Level: use CPU capabilities for debugging
- Kernel-Level: tracing at syscall-level
- Userspace-Level: replace library functions at load time
- Userspace-Level: replace library functions at link time

Link-Time Wrappers

- gcc's option `-Wl, SOMETHING` passes an option to ld linked at last stage
- ld's option `--wrap=SOME_NAME`:
 - replaces original `SOME_NAME` to `__real_SOME_NAME`
 - forces `SOME_NAME` to be linked within `__wrap_SOME_NAME`

Link-Time Wrappers

- The way to replace standard functions to non-standard
- Requires source code or object files
- Use cases:
 - Update legacy code by minimum changes
 - Increase security
 - Stress testing

Demo: gcc wrap example for sigaction()

Load-Time Wrappers

- The ELF executable depends on libraries
- `/lib[64]/ld.so` loads an executable and dependent libraries
- ... and libraries from `LD_PRELOAD` environment variable first

Demo: `LD_PRELOAD` example for `fakelib0.c`

Runtime Libraries Loading

- Before `_start` - loaded by `ld.so`
- After `_start` might be loaded using `dlopen`
- Widely used for “Plug-Ins” implementation

dlopen / dlsym

```
void *dlopen(const char *lib_name, int flags)  
HMODULE LoadLibraryA(LPCSTR lib_name, DWORD flags)  
--- load a library
```

```
void *dlsym(void *lib, const char *func_name)  
FARPROC GetProcAddress(HMODULE lib, LPCSTR  
func_name)  
--- find a function within library
```

dlopen /dlsym

```
#include <dlfcn.h>
void some_func() {
    void* lib =
        dlopen(
            "libSDL.so",
            RTLD_LAZY
        ); // Check for NULL!!!

    void* func_ptr =
        dlsym(lib, "SDL_Init");
        // Check for NULL!!!

    (*func_ptr)(); // Call
}
```

```
#include <Windows.h>
void some_func() {
    HMODULE lib =
        LoadLibraryA(
            "winhttp.dll"
        ); // Check for NULL!!!

    FARPROC func_ptr =
        GetProcAddress(
            lib, "WinHttpConnect"
        ); // Check for NULL!!!

    (*func_ptr)(....); // Call
}
```

Call Function by Name

```
def func():  
    print("Hello, World!");  
  
func_name = input("Please enter func name: ")  
  
func_ref = globals()[func_name]  
  
func_ref()    # Hello, World!
```

Call Function by Name

```
#include <dlfcn.h>

void callable() {}

void some_func() {
    void* lib =
        dlopen(NULL, 0);

    void* func_ptr =
        dlsym(lib, "callable");
        // Check for NULL!!!

    (*func_ptr)(); // Call
}
```

```
#include <Windows.h>

__declspec(dllexport)
void callable() {}

void some_func() {
    HMODULE lib =
        LoadLibrary(NULL);

    FARPROC func_ptr =
        GetProcAddress(
            lib, "callable"
        ); // Check for NULL!!!

    (*func_ptr)(); // Call
}
```

Symbol Resolving

- dlopen flag:
 - RTLD_GLOBAL - makes symbols available within process
 - used by ld.so while loading libraries
- dlsym special library values (GNU-specific):
 - NULL == (void*) 0 == RTLD_DEFAULT
 - find first occurrence within RTLD_GLOBAL
 - (void*) -1L == RTLD_NEXT
 - find first occurrence from *the next library to caller*
- LD_PRELOAD libraries are loaded before all

Demo: LD_PRELOAD example for fakelib.c

Constructors and Destructors

- «Constructor»: the function to be executed after the library loaded

```
__attribute__((constructor))  
void some_function() { ..... }
```

- «Destructor»: the function to be executed before the library to be unloaded

```
__attribute__((destructor))  
void some_another_function() { ..... }
```

Managing System Calls

- The strace command shows all system calls for a process
- PTrace API for Linux and *BSD
- Implemented by ptrace system call

Ptrace System Call

```
long ptrace(enum __ptrace_request request,  
            pid_t pid,  
            void *addr, void *data);
```

- Some request types:
 - ATTACH and TRACEME - start tracing
 - PEEK or POKE something
 - SINGLESTEP

Demo: ptrace catching the write() system call

Single Stepping

- Most processors support single-stepping by hardware
- Widely used in debuggers
- Works too slow in generic
- Have limited count of hardware-watched values