Девятая независимая научно-практическая конференция «Разработка ПО 2013»
23 - 25 октября, Москва

Легковесное профилирование разделяемых библиотек в Linux для встраиваемых систем

Кирилл Кринкин, Марк Заславский, Эдуард Рябиков
Motivation

Popular Linux Profilers (gprof, gcov, GPT, Valgrind) have following problems:

- Need to **recompile** with special options (gprof, gcov)
- Need to **relink** program with 3rd-party libraries (GPT)
- Need to use **special environment** for profiling (Valgrind)
- Need to use **superuser rights**
Project goals

We need a tool for system-wide ELF executables performance analysis.

This tool should allow user next things do easily:

- Profile function calls without recompilation and relinking with 3rd-party libraries
- Profile only given set of C/C++ functions from shared libraries in Linux
- Profile both dynamically linked and dynamically loaded functions
- Profile without creating of special environment
- Get information about number and total duration of function calls
- Perform profiling on x86/x64 platforms
“Non-invasive” profiling

Main ideas:

- Profiler **can not be implemented** in the program code
- Profiling should be performed at **well-defined points of function calls**
- Profiling process **should not corrupt the algorithm** of profiled application
- Profiling process should use **minimum amount of system resources**
- Results of profiling should be **as accurate as possible**
Ways to implement

- Infiltration into the **symbol relocation process**
- Modification of Linux **dynamic linker** (ld-linux.so)
- Modification of **dynamic loading library** (libdl.so)
Dynamic Linking

IDL LINUX SO

_dl_fixup

_dl_profile_fixup

_dl_call_pltexit

LD_LIBRARY_PATH

LD_LINKER_PATH

anylib.so

anyfunc1()

anyfunc2()

...

anyfuncn()

exe

int main()
{
    ...
    anyfunc1();
    ...
    return 0;
}
ELF Parsing by Dynamic Linker

Executable object file

<table>
<thead>
<tr>
<th>ELF header</th>
<th>Process image</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table (required for executables)</td>
<td>init and shared lib segments</td>
<td>0x080483e0</td>
</tr>
<tr>
<td>.text section</td>
<td>.text segment (r/o)</td>
<td>0x08048494</td>
</tr>
<tr>
<td>.data section</td>
<td>.data segment (initialized r/w)</td>
<td>0x0804a010</td>
</tr>
<tr>
<td>.bss section</td>
<td>.bss segment (uninitialized r/w)</td>
<td>0x0804a3b0</td>
</tr>
<tr>
<td>.symtab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.rel.text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.dynamic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Section header table (required for relocatables)

Virtual Address

28 October 2013
1. Resolving the Dependencies

- When linking a dynamic executable, one or more shared objects are explicitly referenced. These objects are recorded as dependencies within the dynamic executable.

- The runtime linker uses this dependency information to locate, and load, the associated objects.

- Once all the dynamic executable’s dependencies are loaded, each dependency is inspected, in the order the dependency is loaded, to locate any additional dependencies.
1. Resolving the Dependencies

The Linux runtime linker looks in two default locations for dependencies /lib and /usr/lib.

The dependencies of a dynamic executable or shared object can be displayed using **ldd**. For example, the file /usr/bin/cat has the following dependencies:

```
$ ldd /usr/bin/cat
    libc.so.1 => /lib/libc.so.1
    libm.so.2 => /lib/libm.so.2
```

The dependencies recorded in an object can be inspected using **dump**. Use this command to display the file’s .dynamic section, and look for entries that have a NEEDED tag.

```
$ dump -Lvp prog
prog:
    [INDEX] Tag Value
    [1] NEEDED libfoo.so.1
    [2] NEEDED libc.so.1
    [3] RUNPATH /home/me/lib:/home/you/lib
```

........
typedef struct {
    Elf32_Word st_name;
    Elf32.Addr st_value;
    Elf32_Word st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half st_shndx;
} Elf32_Sym;
Parsing other sections of ELF

- For dynamic linking, the Dynamic linker primarily uses two processor-specific tables:
  - **Global Offset Table** (GOT)
  - **Procedure Linkage Table** (PLT)
- Dynamic linkers support PIC Code through the GOT in each shared library.
- The GOT contains absolute addresses to all of the static data referenced in the program.
Dynamic Loading

libdl.so

| dlopen | dlsym | dlclose | dlerror |

anylib.so

| anyfunc1() | anyfunc2() | anyfuncn() |

... exe

LD_LIBRARY_PATH

ld-linux.so

? int main()
{
    void* p = dlopen("anylib.so", RTLD_LAZY);
    ...
    void(*f)() = dlsym(p, "anyfunc1");
    ...
    f();
    ...
    dlclose(p);
    ...
}
Profiler components

- **Shared library** `libelfperf.so`
  - Call redirection and function wrapping mechanisms
  - Collecting of calls statistics
  - Memory management

- **Modified dynamic linker** (`ld-linux.so`)
  - Uses `libelfperf.so` for profiling of dynamically linked functions
  - Displays the results of profiling

- **Modified dynamic loading library** (`libdl.so`)
  - Uses `libelfperf.so` for profiling of dynamically loaded functions
Call redirection mechanism

Calls redirection mechanism (**Redirector**) is a set of machine codes for the next assembly instructions:

```
push $fcnPtr
jmp $wrapper_addr
```

All they do is:

- Save address of profiled function *in program stack*
- Jump to *wrapper-function*
Redirector workflow

Code

```c
... void(*f)() = dlsym(p, "any_function");
... f();
...
```

Redirector

1. `push $fcnPtr`
2. `jmp $wrapper_addr`

Wrapper

3. `call preProfile`
4. `call postProfile`
5. `ret`

```c
... void(*f)() = dlsym(p, "any_function");
... f();
...
```

any_function

```c
push %ebp
mov %esp, %ebp
...
leave
ret
```
Redirector details

- Each redirector is created individually for each profiled function
- Redirectors are placed into data segment of process virtual memory
- The operating system allows to mark these memory areas as executable
Wrapping mechanism

Function Wrapping mechanism (or Wrapper) is a function that does next things:

- **Takes control** from redirector
- Performs **pre-profile operations**
- Performs **replacement of return address**
- Performs **jump into** profiled **function**
- Again **takes control after** the work of profiled **function**
- Performs **post-profile operations**
- Returns **to caller**
Working scheme of Wrapper

Wrapper

Context
- Return address
- Function address
- Start time
- End time

Params

Function
- jmp
- ret

Return value

28 October 2013
Implementation details (x86)

Wrapper

```c
void wrapper()
{
    // push %ebp
    // movl %esp, %ebp
    asm volatile ( "popl %ebp\n"
                   "pushal\n"
                   "pushl 32(%esp)\n"
                   "pushl 40(%esp)\n"
                   "call preProfile\n"
                   "addl $8, %esp\n"
                   "movl $wrapper_rp, 36(%esp)\n"
                   "popal\n"
                   "retl\n"
    );

    asm volatile ( "wrapper_rp:\n"
                   "pushl $0\n"
                   "pushal\n"
                   "call postProfile\n"
                   "movl %eax, 32(%esp)\n"
                   "popal\n"
                   "retl\n"
    );
}
```

Function

```assembly
push %ebp
mov %esp,%ebp
...
leave
retl
```

Context

- Ret address
- Start time
- Function address
- End time

Stack Base

- ...:
- fcn param #n:
- ret address:
  - %EAX, %EBX, %ECX, %EDX, %ESI, %EDI:
  - %ESI, %EDI

Registers

- %EAX: raddr
- %EBX: new
- %ECX: new
- %EDX: new
- %EDI: new
- %ESI: new
Wrapper details

- Wrapper doesn’t corrupt stack content
- Wrapper exists in a single copy for all functions in each profiler implementation (x86 or x64)
- Saving/Restoring of registers’ state allows to escape of uncontrollable changes in the program state
- Allows to profile wide set of C/C++ functions
Interaction of ElfPerf’s components

- elfperf-ld-linux.so
  - _dl_fixup
  - _dl_profile_fixup
  - _dl_call_pltexit

- elfperf-libdl.so
  - dlopen
  - dlsym
  - dlclose
  - dlerror

- elfperf so
  - Wrapper
  - Redirectors
  - Statistics
  - Memory management

- ElfPerf Storage
  - Function Infos
  - Function Statistics

Shared memory

LD_PRELOAD
Conclusion

Now we have:

- «Light» profiler based on «patched» ld-linux.so and libdl.so
- Support of profiling for C/C++ functions from shared libraries (including libs compiled with `-fomit-frame-pointer` flag)
- Collecting of information about number and total duration of function calls
- Support of both x86 and x64 platforms
Links

- **Project resources:**
  - [https://github.com/OSLL/elfperf](https://github.com/OSLL/elfperf)

- **Contacts:**
  - [http://osll.ru/](http://osll.ru/)
  - kirill.krinkin@gmail.com
  - edward.ryabikov@gmail.com
  - mark.zaslavskiy@gmail.com