myDBG — Subject

version #

Epita systems/security laboratory <recrutement@lse.epita.fr>

Copyright

Contents

1 Introduction **4**

2 Basic Rules **4**
- 2.1 Build System . . . 4
- 2.2 Authorized Libraries . . . 4
- 2.3 Coding Style . . . 5

3 First Steps **5**
- 3.1 Basic structure . . . 5
- 3.2 Automated command registry . . . 5
- 3.3 Types of binaries . . . 6
- 3.4 info_regs . . . 7
- 3.5 info_memory . . . 7
- 3.6 Breakpoints . . . 8
- 3.7 Continue execution . . . 8
- 3.8 Single step . . . 8
- 3.9 Read memory . . . 8
- 3.10 Backtrace . . . 9
- 3.11 Temporary Breakpoints . . . 9
- 3.12 Next instruction . . . 9

4 Features **9**
- 4.1 Attach a process . . . 10
- 4.2 Breakpoints on functions . . . 10
- 4.3 Breakpoints on syscalls . . . 10
- 4.4 List of the breakpoints . . . 10
- 4.5 Delete a breakpoint . . . 10
- 4.6 Disassemble . . . 10
- 4.7 Finish . . . 10
- 4.8 List . . . 10
- 4.9 Step line . . . 11
- 4.10 Next source line . . . 11
- 4.11 Breakpoint on a line . . . 11
- 4.12 Backtrace with function names . . . 11
- 4.13 Backtrace with function parameters . . . 11
- 4.14 Handling libraries . . . 11

*https://acu.epita.fr

4.15 Handling 32bit binaries . . . 11
4.16 follow fork mode . . . 11
4.17 Hardware Breakpoints . . . 11
4.18 Memory breakpoints . . . 11
4.19 Checkpoint/Restore . . . 12
4.20 Drop libunwind . . . 12
4.21 Other OSes support . . . 12

5 Conclusion **12**

1 Introduction

This project is the opportunity to show us that you are able to work on LSE's projects. You will approach some notion around binary formats, debugging, and x86 architecture. You will have to write a debugger.

We are evaluating here lots of different things, but mostly that you are able to write good, clean C code.

A concise `README` file is welcomed to explain what you have done or what you tried to do.

Remember that all your code will be evaluated (and read) manually, but we can't guess how to use it. On the other side, your text outputs don't need to match some kind of spec or anything, common sense should dictate what you write.

You can do anything you want besides what is written in this subject, just remember to add it in your `README`.

If you have any question, do not hesitate to contact us either on the EPITA newsgroup `labos.lse`, or by email at `recrutement@lse.epita.fr`. Use the tags `[RECRUT]` `[DBG]` for both of them. (Remember that we don't care about your login in the subject line, this is for tickets only...)

Have fun and impress us!

The repository to push your code is:

```
git@git.cri.epita.net:p/2020-lse-recrutement/subject-$login
```

2 Basic Rules

2.1 Build System

- Your build system must be clean and make sense.
- **You can choose to use either:**
  - autotools
  - cmake
  - make
  - meson
- the binary `my-dbg` should be produced at the root directory.

2.2 Authorized Libraries

You can use any library shipped via the following packages (on Archlinux):

- glibc
- readline
- capstone
- libunwind

If you need anything else, you need to ask for permission on the news.

2.3 Coding Style

We don't check your coding style for this project, but since your code will be read by humans, it *must* be understandable and clean.

Some rules though, we *hate* 2 spaces indent. Please use 4 spaces, or 1 tab (with a size of 8). Also, we have small terminals, and like lines that have at most 80 columns.

At the LSE, we love the Linux kernel coding style[1].

You are of course authorized to use:

- explicit cast: you will need them
- `goto`: because error checking is a good thing
- assembly code
- `__attribute__`: more fun!

3 First Steps

3.1 Basic structure

A debugger looks like a command interpreter, so let's build it like this.

```
$ ./my-dbg /path/to/binary arguments...
dbg> help
info_regs: display registers
continue: continue program being debugged
quit: exit
help: display this help message
```

You'll need to be able to register multiple commands in your shell.

Completion with readline is not hard to implement[2] and will be a plus.

Like any interpreter, your debugger should exit on `^D`.

You debugger should `fork()`, launch the debugged binary and attach to it. This may look scary, but it is quite simple actually, thanks to `ptrace()`.

```c
int main(int argc, char **argv)
{
    pid_t pid = fork();
    if (pid == 0) {
            ptrace(PTRACE_TRACEME, 0, 0, 0);
            execvp(argv[1], argv + 1);
    }
    waitpid(pid, NULL, 0);
    /* we now have our debugged program in a stopped state */
}
```

3.2 Automated command registry

Here is a simple example of a generic and extensible command system.

[1] https://www.kernel.org/doc/html/latest/process/coding-style.html
[2] http://www.delorie.com/gnu/docs/readline/rlman_48.html

When linking multiple object files in a program, all sections are merged into one with the same name.

Also, if a section name is a valid C identifier, then `ld` gives us 2 symbols, representing the start and end address of a section.

These 2 features can be used to create arrays of structures splitted in multiple object files. Let's see how we can exploit that.

```c
struct cmd {
    const char *cmd;
    int (*fn)(void *);
};

#define shell_cmd(name, func) \
        static struct cmd __cmd_ ## name \
        __attribute__ ((section("cmds"), used)) = \
        { .cmd = #name, .fn = func }

extern struct cmd __start_cmds[];
extern struct cmd __stop_cmds[];

#define array_size(t) (sizeof(t) / sizeof(*t))

int main(int argc, char **argv)
{
        for (size_t i = 0; i < __stop_cmds - __start_cmds; ++i) {
                struct cmd *cmd = __start_cmds + i;
                if (!strcmp(argv[1], cmd->cmd)) {
                        (*cmd->fn)(NULL);
                }
        }
}

static int do_ls(void *arg)
{
        (void)arg;
        return printf("%s\n", __func__);
}
shell_cmd(ls, do_ls);
```

That way we can register any commands in our files, support automated help message generation, completion, etc...

3.3 Types of binaries

When we are debugging code, we need to take the file format into account. It can change the behavior (and complexity) of your debugger.

Here is the 3 cases that we think are mandatory:

- simple binaries (elf with type `ET_EXEC`)
- PIE binaries (type `ET_DYN` with an entry point)
- optimized binaries (PIE or not)

`ET_EXEC`: the simplest, the loaded address of your segments are known statically (on modern systems, you need to add -no-pie to `CFLAGS` and `LDFLAGS`.

PIE: simple, but with a twist, memory mappings are not known statically, you'll need to discover them.

Optimized binaries: This should be simple, but it *will* break your code. You will have no guaranties that prologue and epilogue will be present and gathering a backtrace will be difficult.

3.4 info_regs

The command `info_regs` should display all the relevant registers. For example, here is the state of a program at the beginning:

```
dbg> info_regs
rip: 0x7f68e49bbf30
rsp: 0x7fffb1cc0140
rbp: 0x0
eflags: 0x200
orig_rax: 0x3b
rax: 0x0
rbx: 0x0
rcx: 0x0
rdx: 0x0
rdi: 0x0
rsi: 0x0
r8: 0x0
r9: 0x0
r10: 0x0
r11: 0x0
r12: 0x0
r13: 0x0
r14: 0x0
r15: 0x0
cs: 0x33
ds: 0x0
es: 0x0
fs: 0x0
gs: 0x0
ss: 0x2b
fs_base: 0x0
gs_base: 0x0
```

Don't forget to be able to read/write registers, for the next features that will need register manipulation (almost *all* the other features).

Potentially useful apis:

- ptrace
- libunwind

3.5 info_memory

The command `info_memory` should display all the memory mappings of the debugged program. There is multiple ways to gather these maps, but the simplest way is to read `/proc/$PID/map`:

```
$ cat /proc/$$/maps
00400000-004c6000 r-xp 00000000 fe:01 1448097            /usr/bin/bash
006c5000-006c6000 r--p 000c5000 fe:01 1448097            /usr/bin/bash
006c6000-006ca000 rw-p 000c6000 fe:01 1448097            /usr/bin/bash
006ca000-006d8000 rw-p 00000000 00:00 0
00d4c000-00ebc000 rw-p 00000000 00:00 0                  [heap]
7f7fcd01f000-7f7fcd02a000 r-xp 00000000 fe:01 1443977    /usr/lib/libnss_files-2.26.so
```

```
7f7fcd02a000-7f7fcd229000 ---p 0000b000 fe:01 1443977    /usr/lib/libnss_files-2.26.so
7f7fcd229000-7f7fcd22a000 r--p 0000a000 fe:01 1443977    /usr/lib/libnss_files-2.26.so
7f7fcd22a000-7f7fcd22b000 rw-p 0000b000 fe:01 1443977    /usr/lib/libnss_files-2.26.so
7f7fcd22b000-7f7fcd231000 rw-p 00000000 00:00 0
...
7f7fce4c9000-7f7fce4cb000 rw-p 00000000 00:00 0
7f7fce4cb000-7f7fce4f0000 r-xp 00000000 fe:01 1444042    /usr/lib/ld-2.26.so
7f7fce513000-7f7fce6ae000 r--p 00000000 fe:01 1462825    /usr/lib/locale/locale-archive
7f7fce6ae000-7f7fce6b2000 rw-p 00000000 00:00 0
7f7fce6ef000-7f7fce6f0000 r--p 00024000 fe:01 1444042    /usr/lib/ld-2.26.so
7f7fce6f0000-7f7fce6f1000 rw-p 00025000 fe:01 1444042    /usr/lib/ld-2.26.so
7f7fce6f1000-7f7fce6f2000 rw-p 00000000 00:00 0
7ffc8f843000-7ffc8f864000 rw-p 00000000 00:00 0          [stack]
7ffc8f9bf000-7ffc8f9c2000 r--p 00000000 00:00 0          [vvar]
7ffc8f9c2000-7ffc8f9c4000 r-xp 00000000 00:00 0          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0  [vsyscall]
```

Don't forget to keep the relevant informations for later.

3.6 Breakpoints

Now, let's talk about breakpoints!

A software breakpoint is an instruction that triggers the exception 3 (`#BP`). It will be signaled to the process via a `SIGTRAP` signal. Since when a process is ptraced, signals are available to the debugger via `wait()`, we can intercept every breakpoint, and do something with it.

The instruction for breakpoint is `int3`, and is encoded as `0xCC`.

Pay attention to the fact that on a breakpoint, `%rip` is on the *next* intruction, you'll need to update `%rip` to execute your instruction properly.

For the moment, your breakpoint should be simple, `break [ADDR]` should be enough.

To be able to write a breakpoint, you'll need to save the instruction that you overwrite, and be able to identify which breakpoint caused the `SIGTRAP`.

3.7 Continue execution

In order to implement the `continue` command, you'll need to use `ptrace(PTRACE_CONT, pid, 0, sig)`.

3.8 Single step

The `step_instr` command will allow to single step into your program. You need to use `ptrace(PTRACE_SINGLESTEP, pid, 0, sig)`. The debugger will be alerted via `wait()`. Pay attention that the signal return will be `SIGTRAP`, like for the breakpoints. In order to differenciate it, you will need to look into the siginfo struct to see which one is it (see `ptrace(PTRACE_GETSIGINFO)`).

3.9 Read memory

Let's implement an `examine` command that can read into the debugged program memory.

We need to be able to have multiple formats, for example:

- `x`: hexadecimal

- `d`: decimal
- `i`: instructions (look into capstone for a disassembler)
- `s`: string

the basic syntax could be:

```
examine $format size start_addr
```

Pay attention to the error checking here.

In order to read from the debugged program memory, you can use multiple ways:

- `ptrace(PEEK_DATA/PEEK_TEXT)`: only for 4bytes read you'll have to loop to read more
- `process_vm_readv`: fast, this allows to copy any size. Pay attention that the permission of the pages are taken into account. In that case, you'll need to use something else
- `/proc/$PID/mem`: this allows to read/write the memory of a process.

3.10 Backtrace

The `backtrace` command will allow you to print the call trace at the current `%rip`.

For this, you need to be able to read registers and memory.

If the program is not optimized, it'll use `%rbp` and the stack to track all stack frames.

If your program is optimized, you'll need to use the content of the `.eh_frame` section of the binary. Or use libunwind!

3.11 Temporary Breakpoints

Temporary breakpoints are breakpoints that can be hit only 1 time. You'll need them for lots of features later. If you want a command for it you can call it *tbreak* for example.

3.12 Next instruction

`next_instr` command is the last of the core part of this project. It works like `step_instr` but will not "enter" functions at `call` instructions, just step over them. In order to implement this, you will need to put a temporary breakpoint on the next instruction (just after the call). For this, you'll need to disassemble the code to know the instruction's size.

A simple optimisation could be to check if we are on a call or not, and put a breakpoint, or just single step over it if is not necessary.

4 Features

For this next part, now that you have a basic debugger, you are able to experiment and explore more advanced features.

Most of the features here are independant. For every feature, we will list the other features that would be smart to implement before. Because there are lots of ways to do all these features, this order is just a suggestion.

4.1 Attach a process

Let's try your work on an external running process.

`attach <pid>` needs to stop your program and trace it. Have a look to `ptrace()` with `PTRACE_ATTACH` request. Use `/proc/<pid>/exe` in order to find the corresponding binary.

4.2 Breakpoints on functions

`breakf <funtion>` command allows you to set a breakpoint on the first matching symbol found. You must look for the symbol in the binary first, and if not found, in the libraries. The symbols are in the dynamic GNU hash table (`DT_GNU_HASH` in `/usr/include/elf.h`[3]).

Library support is harder, it is not mandatory to validate this step (but still counts).

In order to set your breakpoint, use the `struct r_debug` in `link.h`.

4.3 Breakpoints on syscalls

`breaks <syscall|syscall number>` command allows you to set a breakpoint on a syscall. It is easier to implement `breaks` with a syscall number. Once again, `ptrace()` is your best friend, even more with `PTRACE_SYSCALL` request.

4.4 List of the breakpoints

`break_list` command must print a table of all breakpoints with their types, their ID and the their addresses.

4.5 Delete a breakpoint

`break_del <number>` command allows you to delete the breakpoint matching the given ID. If there is any breakpoint associated with the given ID, an error message must be printed out to the user.

4.6 Disassemble

`disassemble <address> <number>` command allows you to disassemble `number` instructions starting at address `address`. (capstone *will* help a lot)

4.7 Finish

`finish` command continues execution until the end of the current function. In other words, you have to set a temporary breakpoint on the function return address on the stack.

4.8 List

`list` command allows you to print the source code current line.

After fighting against ELF, it is time to play with DWARF[4] and more specifically with `.debug_line` sections.

[3] http://www.muppetlabs.com/~breadbox/software/ELF.txt
[4] http://www.dwarfstd.org/doc/DWARF4.pdf

4.9 Step line

`step_line` allows you to step over a C line of code. See `.debug_line` for this.

4.10 Next source line

`next_line` is like `next_instr` but for C line of code.

4.11 Breakpoint on a line

`breakl <line> <file>` command let you set a breakpoint on a line of C code.

4.12 Backtrace with function names

Display the backtrace with function names. You can simply look into the symbol tables.

4.13 Backtrace with function parameters

Let's be awesome and display also the function parameters in backtraces. This will be harder. You'll need to look into the `DWARF` file format. Especially the sections `.debug_pubname`, `.debug_abbrev` and `.debug_info`.

4.14 Handling libraries

Be able to debug code in dynamic libraries. (See `.dynamic`, `struct r_debug`)

4.15 Handling 32bit binaries

Be able to debug 32bit binaries. Warning, this will need lots of abstractions for all the ELF handling.

4.16 follow fork mode

Add a mode that allow the debugger to attach automatically when the binary is forking.

4.17 Hardware Breakpoints

registers `%dr0` to `%dr7` allows to put hardware breakpoints. You can read more about them in the Intel Software Developper Manual. In order to set them, you can use `ptrace()` with ``PTRACE_PEEKUSER` and `PTRACE_POKEUSER`.

This allows to have breakpoints, that can break on read/write in memory, or on an instruction. Remember that there is only 4 of them though.

You are free for the command format.

4.18 Memory breakpoints

There are lots of ways to break when reading/writing in memory (HW breakpoints, `SIGSEGV` ...). Let's implement this.

4.19 Checkpoint/Restore

Let's go back in time! With the command `checkpoint`, you will save the process context, and restore it with `restore`.

This looks really hard, but if you think about executing the `fork()` syscall in the debugged program, this will look much simpler.

4.20 Drop libunwind

libunwind really helps to display call chains and whatnot. That's nice, but in order to finish the nightmare, you can choose to drop it completely.

4.21 Other OSes support

Add support for Mac OS X or Windows and enjoy!

5 Conclusion

Ça va bien se passer