Trace memory references in your ELF PIE

poc-code: http://github.com/ltlollo/instr Lorenzo Benelli

Dear fellow cooks, have you ever wondered which positions of memory is your freshly baked x86 64 ELF executable accessing? Here, follow this simple three-step recipe to find out how to check that, using binary instrumentation!

Ingredients (for one executable):

- *1* good disassembler (I suggest Capstone®)
- *1* good assembler (I suggest Keystone®)
- *5* memory pages at least, *4*KiB (*4.096*kB) each.
- *1* function that dumps its input onto a file.

Step one: Find the code

If the binary is not stripped, you can easily find its functions offsets and sizes, by looking inside the elf's sections: Locate the *section header table* in your elf's header. In the section headers find one with type `SHT_SYMTAB` named *`.symtab`* and one with type `SHT_STRTAB` named *`.strtab`*. In the *`.symtab`*, the entries with type `STT_FUNC`, are your functions, while their names are in *`.strtab`*.

Step two: Instrument

Write a piece of position independent code that stores its input (`rax`) somewhere (I'll call it *`rax_dump`*). Personally, I like to place it after a page that I know I can write to, that, when full, I can dump its content on disk. Disassemble the code you found before and look for instructions of the form `op reg, [expr]`, `op reg, reg:[expr]`, `op [expr], reg`, or `op reg:[expr], reg`. For each of them, generate a tiny gadget, using `lea rax, [expr]` to fetch the address, and append it after the *`rax_dump`* you previously wrote. Finally, replace the oringinal instruction with a jump to your new code, and you are all set.

```
  ...
raxr1110x80:    ;(before)
  add r12, [rax+r11*1+0x70]
  ...
```

```
  ...
raxr1110x80:   ;(after)
  jmp dump_raxr1110x80
  ...
rax_dump:
  ...
dump_raxr1110x80:
  push rax
  lea rax, [rax+r11*1+0x70]
  call rax_dump
  pop rax
  add r12, [rax+r11*1+0x70]
  jmp raxr1110x80+JMP_SZ
```

A couple of caveats: If your instruction is rip-relative remember to skip it or recompute its destination, and offset your expressions by 8 if it uses `rsp`.

Also the instruction you are replacing might be smaller than a jump, so you may have to copy a bunch.

If you do so, remember to recompute the jumps internal to the original function.

Step three: Reassemble

Adding our new stuff to the executable is not as easy as appending it, we also need to tell the kernel where to map it into memory using program header entries. So we are going to add a new copy of our original program header table with three new mappings: one read only, with offset and address of the table itself (this so that the linker can also see it), one R/W (so we can store some addresses), and one R/E pointing to the code we just generated. Beware of mixing all these ingredients after the latest `vaddr+memsz` to avoid a confict with the *bss*, and that `vaddr-offset` must be 0 mod 4096, or just follow grandma's tip: *keep all offsets and sizes in the program headers page-aligned.*

If you also wish to call some flushing code before the program shuts down, you'll need to append two additional sections (and the respective R/W mappings as program headers entries). A copy of *`.fini_array`* with the virtual address of your flushing code appended, and a copy of *`.rela.dyn`* with a new `R_X86_64_RELATIVE` symbol pointing its `r_offset` and `r_addend` to the file offset and address of your finalizer. Don't forget to update all the `r_offsets` of the other `R_X86_64_RELATIVE` symbols you moved with the *`.fini_array`* and update the `DT_FINI_ARRAY` and `DT_FINI_ARRAYSZ` address and offset in the *`.dynamic`* section. Finally, update the *program header table* offset in your elf's header (and in the `PT_PHDR` program header), with its new virtual address, et voilà, your binary is ready to reveal its delicious secrets!

Are you a professional chef? Then make sure to check out these professional tools for instrumentation needs: **Pin**, **DynamicRIO**