QuarkslaB Dynamic Loader (QBDL)

Romain Thomas & Adrien Guinet

Quarkslab

Table of Contents

What is QBDL?

QuarkslaB Dynamic Loader: a cross-platform dynamic loader library

In a nutshell

- A simple-to-use **system abstraction** to load dynamically linked binaries
- Load binaries in foreign systems or **lightweight sandboxes** (e.g. Miasm/Triton/Unicorn)
- Support for **PE/MachO/ELF** binaries
- Written in C++ with Python bindings, and **documentation** :)

URL / install

```
https://github.com/quarkslab/QBDL
pip install pyqbdl
```

QBDL by example

Run a MachO binary from a Python process under Linux

```
import pyqbdl
import lief
import ctypes

class TargetSystem(pyqbdl.engines.Native.TargetSystem):
    def __init__(self):
        super().__init__(pyqbdl.engines.Native.memory())
        self.libc = ctypes.CDLL("libc.so.6")

    def symlink(self, loader: pyqbdl.Loader, sym: lief.Symbol) -> int:
        ptr = getattr(self.libc, sym.name[1:], 0)
        return ctypes.cast(ptr, ctypes.c_void_p).value

loader = pyqbdl.loaders.MachO.from_file("mybin.macho", pyqbdl.engines.Native.arch(),
    TargetSystem())
main_type = ctypes.CFUNCTYPE(ctypes.c_int, ctypes.c_int, ctypes.c_voidp)
main_ptr = main_type(loader.entrypoint)
main_ptr(0, ctypes.c_void_p(0))
```

Why QBDL?

Why a dynamic loader library?

A solution that is covering multiple of our **needs**:

- **Run** and **debug/instrument** very simple iOS/Android binaries under Linux:
  - for reverse engineering needs
  - also to debug our own cross-platform libraries (e.g. whiteboxes)
- Load all kinds of binaries in **Triton**'s memory space
- Extend **Miasm** with MachO support

Why QBDL?

Related work

- `https://github.com/malisal/loaders`: small, self-contained implementations of various binary formats loaders
- `maloader` [1]: a userland Mach-O loader for linux
- `https://github.com/taviso/loadlibrary`: a library that allows native Linux programs to load and call functions from a Windows DLL
- `https://github.com/polycone/pe-loader`

[1] `https://github.com/shinh/maloader`

The *novelty*: a *target system abstraction*

The need

- Binaries can be loaded in various contexts:
  - In a **native process**, by mapping and writing memory directly in the current memory space.
  - In a *lightweight* **sandbox**: Unicorn, Miasm, Triton, ...
- We don't want to rewrite loaders for each of these cases!

$\Rightarrow$ We need to **abstract the targeted system**!

The *novelty*: a *target system abstraction*

Target memory & system abstraction

```
class TargetMemory {
  virtual uint64_t mmap(uint64_t hint, size_t len) = 0;
  virtual bool mprotect(uint64_t addr, size_t len, int prot) = 0;
  virtual void write(uint64_t addr, const void *buf, size_t len) = 0;
  virtual void read(void *dst, uint64_t addr, size_t len) = 0;
};

class TargetSystem {
  TargetSystem(TargetMemory &mem);
  virtual uint64_t symlink(Loader &loader, LIEF::Symbol const &sym) = 0;
};
```

The *novelty*: a *target system abstraction*

Native implementation

```
class NativeTargetMemory: public TargetMemory {
  uint64_t mmap(uint64_t hint, size_t len) override {
    return mmap(hint, len, PROT_READ|PROT_WRITE, MAP_ANONYMOUS, -1, 0);
  }
  bool mprotect(uint64_t addr, size_t len, int prot) override {
    return mprotect(addr, len, prot) == 0;
  }
  void write(uint64_t addr, const void *buf, size_t len) override {
    memcpy((void*)addr, buf, len);
  }
  void read(void *dst, uint64_t addr, size_t len) override {
    memcpy(dst, (void*)addr, len);
  }
};
class NativeTargetSystem: public TargetSystem {
  uint64_t symlink(Loader&, LIEF::Symbol const &sym) override {
    return dlsym(RTLD_DEFAULT, sym.name());
  }
};
```

What is **not** QBDL?

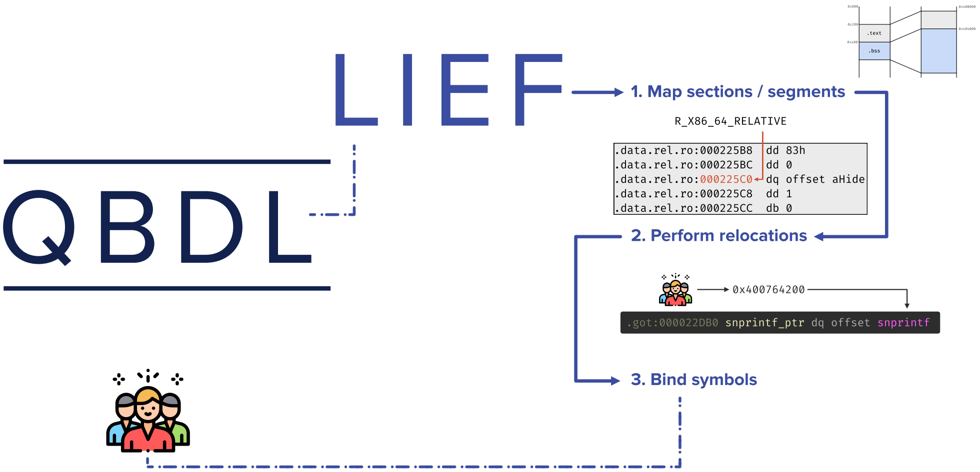
Non goals of the library

- Provide full operating system (re)implementations, like Wine [2] or Darling [3]
- Get the best performance out of all dynamic linkers
- Supports architectures where pointer values are bigger than 64 bits

[2] https://www.winehq.org/
[3] https://darlinghq.org/

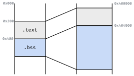
Table of Contents

QBDL & LIEF


LIEF
QBDL
1. Map sections / segments
.text
.bss
R_X86_64_RELATIVE
.data.rel.ro:000225B8 dd 83h
.data.rel.ro:000225BC dd 0
.data.rel.ro:000225C0 dq offset aHide
.data.rel.ro:000225C8 dd 1
.data.rel.ro:000225CC db 0
2. Perform relocations
0x400764200
.got:000022DB0 snprintf_ptr dq offset snprintf
3. Bind symbols

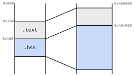
QBDL & LIEF

1. Map sections / segments



`write() / mmap() / mprotect()`

QBDL & LIEF



1. Map sections / segments

2. Perform relocations

R_X86_64_RELATIVE

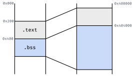
```
.data.rel.ro:000225B8  dd 83h
.data.rel.ro:000225BC  dd 0
.data.rel.ro:000225C0  dq offset
.data.rel.ro:000225C8  dd 1
.data.rel.ro:000225CC  db 0
```

`write() / mmap() / mprotect()`

`write_ptr() / read_ptr()`

QBDL & LIEF

1. Map sections / segments



`write() / mmap() / mprotect()`

2. Perform relocations

R_X86_64_RELATIVE

```
.data.rel.ro:000225B8  dd 83h
.data.rel.ro:000225BC  dd 0
.data.rel.ro:000225C0  dq offset
.data.rel.ro:000225C8  dd 1
.data.rel.ro:000225CC  db 0
```

`write_ptr() / read_ptr()`

3. Bind symbols

0x400764200

```
.got:000022DB0 snprintf_ptr dq offset snprintf
```

`symlink() / write_ptr()`

Table of Contents

Demo 1

Demo 1: Triton Integration

Demo 2

Demo 2: Android Whitebox Attack

Table of Contents

Conclusion

Conclusion

`https://github.com/quarkslab/QBDL`

Thank you

Contact information:

Email: contact@quarkslab.com

Phone: +33 1 58 30 81 51

Website: https://www.quarkslab.com

Quarkslab

Table of Contents

Run simple iOS binaries under Linux

Context

- Consider simple iOS test binaries of a library
  - basically do `printf` and `exit(1)`
- Painful to test and debug on real hardware (need jailbroken devices and working debuggers)

Run simple iOS binaries under Linux

QBDL to the rescue

- Cross-compile QBDL for arm64
- Make a simple tool that loads the MachO file and jump to `main`
  - Resolve `libSystem` symbols to Linux's `libc`. Good enough for what we want.
- Compile it to ARM64, run with qemu userland
- Profit?

Welcome to ABI hell

It would have been easy but...

- Apple's ARM64 ABI is different from the SystemV one [4]
- Among these differences, variadic functions ABI is different (remember `printf`?)
- Introducing `__attribute__((darwin_abi))` in Clang: `https://reviews.llvm.org/D89490`

Wrapper example

```
__attribute__((darwin_abi)) int darwin_aarch64_printf(const char *format, ...) {
  va_list args;
  va_start(args, format);
  const int ret = vprintf(format, args);
  va_end(args);
  return ret;
}
```

[4] `https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms`