The Medusa Project

spv

October 2, 2022

1 Notes

1. This document / paper / book is written from the perspective of the *Medusa* Project's main developer, spv.
2. All software released by the *Medusa* Project is, whenever possible, released under the GNU General Public License, Version 2 (without the ”any later version” clause).
3. This document / paper / book was partially written by me (spv) in order to learn LaTeX.
4. Documentation for *Medusa* Project API's can be found at

   docs.medusa-re.org.
5. A wiki for the *Medusa* Project can be found at wiki.medusa-re.org.
6. When a first-person pronoun is used, and then a name appears in (parentheses), that first-person pronoun refers to the name in parentheses. For example: My (spv) cat is cute.

2 Introduction

Medusa is a project to create a cross platform, free (libre), and general purpose tool for software and hardware research, like reverse engineering, analysis, emulation, development, debugging, and other similar tasks. I started the *Medusa* project originally under the name of xp^{DBG}, as a hobby project to learn about reverse engineering, and because I felt that current development and reverse engineering tools all have their own problems, which I wanted to solve.

2.1 Current Tools' Issues

This subsection is partially paraphrased from the *Medusa* Project's `README.md` file.

- *Cutter*: not very featureful, essentially a *radare2* GUI, and doesn't have debugger and/or emulation support to my knowledge.
- *Ghidra*: personal favorite currently, still doesn't have emulation support or code editing, and is written in *Java* (besides the decompiler), which is one of my least favorite languages*.
- *IDA Pro*: expensive, non-free, does not have emulation support, or code editing.
- *Radare2*: does not have code editing, a GUI, or the level of emulation support which I intend to include in *Medusa*.
- *Binary Ninja*: I honestly do not have a lot of experience with *Binary Ninja*, but to my knowledge, it is not free / open-source software, it isn't a full IDE (like *Medusa* is intended to be), and doesn't have emulation support (like *Medusa* is intended to).

* Since the writing of `README.md`, I have changed my opinion slightly in regards to Java. I still dislike parts of the language (particularly, requiring a VM), but I can appreciate other parts of it.

2.2 *Medusa*'s Solutions

2.2.1 *libmedusa*

For a first example, take *Unicorn*. *Unicorn* is a library/API based on *QEMU*, that provides an interface to control virtualized/emulated CPUs, and general machines. I do appreciate the *Unicorn* project's work, but I think it has some flaws. (or rather, it is not the perfect library for the *Medusa* Project's goals.)

To solve some of *Unicorn*'s issues, the *Medusa* Project has a subproject / subcomponent called *libmedusa*. *libmedusa* is a C++ library with a "standardized" API for interfacing with emulated machines ("soft silicon", as I call it), as well as real machines ("hard silicon", as I call it). *libmedusa* also provides a "standardized" API for interacting with other types of components, such as displays, sound outputs, and other components useful when controlling, say, an emulated Commodore 64. If you wanted to do so, you could provide an implementation of the *libmedusa* API for emulators for the 6502, SID, VIC-II, and other components. (or even real hardware!) Then, other software can interface with an emulated, or even a real C64, without needing to be specifically written to support it.

Another way that this API could be useful is if you (or your company) is developing a new piece of hardware. *Medusa* (or other software) probably doesn't support unreleased hardware, and with other software, say *IDA Pro* (or something with emulation support) it may be difficult to emulate your hardware elegantly for testing purposes. With *libmedusa*, you could implement its API for your particular display, sound output, CPU, etc, (or even re-use existing implementations, if, say, you use a standard CPU ISA, like *ARMv8*), and software can interact with your hardware without needing to be specifically written to support it.

libmedusa doesn't just provide an alternative to *Unicorn*. It provides an all-in-one API that can replace *Unicorn*, *Capstone*, *Keystone*, *LIEF* (*libmedusa* provides an API for parsing formats like ELF), and other libraries.

2.2.2 *barcelona*

This subsubsection is adapted from the *barcelona* subcomponent / subproject's `README.md` file.

barcelona is a subproject of the *Medusa* Project to create a TUI (Terminal User Interface)-based IDE (Integrated Development Environment).

2.2.3 *paris*

This subsubsection is adapted from the *paris* subproject's `README.md`.

paris is a subcomponent of the *Medusa* Project to develop the client/server architecture that is intended to be used in the project. *Medusa* is meant to be modeled after a client/server architecture, where a machine (or machines) operates the server, and handle the bulk of the processing work; and a machine (or machines) runs a client, which connects to the server, and provides a UI to interface with the server.

The server can be the same machine as the client, and it does not need to be over the network (i.e. TCP), it could be a socket, for example.

2.2.4 *rome*

This subsubsection is adapted from the *rome* subcomponent's `README.md` file.

Rome is a subcomponent of the *Medusa* Project to write a modern C++ TUI framework based on *ncurses*. *ncurses* / *curses* is a great framework, but in my (spv) opinion, it feels a bit ancient compared to what could be done with, say, C++. It's a bit low-level, and the code you write with it is, in my opinion, not the best-looking, to put it mildly.

I do understand the historical reason(s) for the *curses* API essentially using (y, x) instead of (x, y), but it doesn't make it any less strange to write.

3 History

3.1 p0laris

The *Medusa* Project's history begins with another one of my projects, *p0laris*. I, at the time, was writing shellcode to execute from within kernel-land on iOS 9, and wanted to debug said shellcode. I did not (and still don't, at the time of writing) own a DCSD-cable, or other similar means to debug the shellcode, so I set out to develop a debugger that I could run on my computer.

3.2 ARMistice

I discussed xp^{DBG} in the *Introduction* section, but before even xp^{DBG}, I created a small project called *ARMistice*. *ARMistice* was a small *Python* script that emulated an *ARMv7* CPU using *Unicorn*, and provided memory editing, assembly, disassembly, and a UI with *ncurses*, *Keystone*, and *Capstone*. This little *Python* script was the genesis for what would eventually become the *Medusa* Project.

3.3 xp^{DBG}

After *ARMistice*, as was stated in the *Introduction* section, the *Medusa* project was started under a different name, xp^{DBG}. xp^{DBG} stands (or rather, stood) for "cross (X) Platform DeBuGger", which references the project's goal to create a cross platform suite for software and hardware research.

3.4 Name change

I changed the name from xp^{DBG} to *Medusa* mainly for branding reasons. I thought of the name (I forget how), and I thought it sounded cool. I also figured that it was early enough in the project's life that it wouldn't be too damaging to do so. I still own the domains xpdbg.com and xpdbg.org, but they (at the time of writing, at least) just contain partially broken versions of the *Medusa* website.

4 Documentation

4.1 *libmedusa*

Note: see 2.2.1 for an introduction to this subsection.
Note: this code is subject to change.

This subsection will document code examples for *libmedusa*, a subcomponent / subproject of the *Medusa* Project.

Let's use, for example `ARMv7Machine`. `ARMv7Machine` is a class that implements the generic `Machine` class, which is a "standardized" interface between software utilizing *libmedusa*, and soft-or-hard silicon machines (specifically, CPUs).

Requirements:

- A computer.
- *libmedusa* installed on said computer. (Exact directions may differ from OS to OS, but in general, clone the *Medusa* Project's *git* repository, cd into `Medusa/medusa_software/libmedusa`, and run `make`, followed by `sudo make install`. This will require *Unicorn*, *Keystone*, *Capstone*, and possibly other components installed. Check `README.md` for a full list of requirements.)
- Your preferred IDE for editing C++ code. (in the future, Medusa will hopefully be able to be used for this purpose. (specifically, the london subproject / subcomponent which will be integrated into the larger tool.))
- A compatible C++ compiler: we use *LLVM / Clang*. [1].

Create a new source file, and `#include <libmedusa/libmedusa.hpp>`, `<libmedusea/Machine.hpp>`, and `<libmedusa/ARMv7Machine.hpp>`.

To create a new `ARMv7Machine`, declare it like you would any other instance of a class.

```
libmedusa::ARMv7Machine armv7_machine;
```

Now, let's map some memory to place the code we'd like to emulate in.

```
/*
 *  a mem_reg_t is a libmedusa type representing a memory region, containing
 *  information about its address, size, and memory protections.
 */
libmedusa::mem_reg_t region;

/*
 *  set region.addr to 0x0, size to 0x10000, and allow RWX in the memory.
 *  this describes a memory region at the beginning of the CPU's memory
 *  space, that is 0x10000 bytes long, and allows all operations, read,
 *  write, and execute.
 */
region.addr = 0x0;
```

[1] llvm.org

```
region.size = 0x10000;
region.prot = XP_PROT_READ | XP_PROT_WRITE | XP_PROT_EXEC;

/*
 *  Machine::map_memory(mem_reg_t region) will map the region into memory.
 */
armv7_machine.map_memory(region);
```

Now, let's copy some *ARMv7* code into memory to be executed.

First, we need to define an array containing our code.

```
uint8_t test_arm_thumb_code[] = {
    0x41, 0x20,             //  movs    r0, #0x41
    0x40, 0xF2, 0x20, 0x40, //  movw    r0, #0x420
    0x40, 0xF2, 0x69, 0x01, //  movw    r1, #0x69
    0xA0, 0xEB, 0x01, 0x00, //  sub     r0, r0, r1
    0x01, 0x44,             //  add     r1, r1, r0
    0x00, 0x00,             //  mov     r0, r0
};
```

The *libmedusa* API uses `std::vector`'s for a lot of work that would normally be done by standard arrays or pointers. So, we must create an `std::vector` containing our *ARMv7* code.

```
vector<uint8_t> tmp_vector(test_arm_thumb_code,
                           test_arm_thumb_code
                         + sizeof(test_arm_thumb_code));
```

Now, let's copy our code `vector` into the `ARMv7Machine`'s memory space.

```
armv7_machine.write_memory(0, tmp_vector);
```

Next, let's set the `pc` register's value to `0x1`. Remember, in 32-bit *ARM*, having the LSB set in the `pc` indicates executing *THUMB* code, rather than *ARM* code.

```
/*
 *  This API should be improved. For now, you have to include the register name
 *  and description, but in the future, that will probably not be necessary.
 *  I'll probably write another version of the 'set_register' and 'get_register'
 *  functions that take just a register name or id later.
 *
 *  reg_t is another libmedusa-specific type. It contains fields describing a
 *  processor register, with fields such as the name, a short description of the
 *  register's function, an ID specific to this register (unique per CPU, so 2
 *  different CPU's can share an ID, as long as it is not shared within the same
 *  CPU), and the register's value.
```

```
 */
libmedusa::reg_t reg;

reg.reg_description = "pc";
reg.reg_name = "pc";
reg.reg_id = 0xf;
reg.reg_value = 0x1;

/*
 *  Machine::set_register(reg_t register) sets the value of a register.
 */
armv7_machine.set_register(reg);
```

Now, let's step through a few instructions, and print the state of all registers after each step.

```
const int number_of_registers_to_step_through = 0x8;
for (int i = 0;
         i < number_of_registers_to_step_through;
         i++) {
    /*
     *  Machine::get_registers() returns an ‘std::vector‘ of all of the CPU’s
     *  registers, as ‘reg_t‘’s.
     */
    vector<libmedusa::reg_t> registers = armv7_machine.get_registers();
    for (libmedusa::reg_t& i : registers) {
        printf("-----\n"
               "%s %s %lx %lx\n",
               i.reg_description.c_str(),
               i.reg_name.c_str(),
               i.reg_id,
               i.reg_value);
    }

    /*
     *  step_instruction is a pre-processor #define for exec_code_step.
     *  Machine::exec_code_step() steps forward one instruction on the CPU.
     */
    armv7_machine.step_instruction();
}
```

Congratulations! You just wrote your first program with *libmedusa*.

4.2 ***libmedusa*** Components

libmedusa's emulation / control framework is based on Components. A Component is a generic class that describes a "thing" that can be controlled, can give output, or both.

For example, a DisplayOutput Component describes a display output: a framebuffer / bitmap. A SoundOutput Component could provide an interface to a *DAC*, or a CPU Component could provide a generic interface to a *CPU*, that could be implemented by a class to interact with, say, an *ARMv7* or *x86(_64)* processor.

Let's implement an example DisplayOutput Component. [2]

First, we need to install *libmedusa*. See the previous subsection for instructions.

Create a class implementing the DisplayOutput generic class in a header file.

```
class ExampleDisplayComponent : public DisplayOutput {
    ...
}
```

Next, #include <libmedusa/libmedusa.hpp>, <libmedusa/Component.hpp>, and the header file that contains your implementation.

4.3 ***rome***

As stated previously, *rome* is a subcomponent / subproject of the *Medusa* Project to create a C++ library / API, as a sort-of replacement for *ncurses*. I thought that *ncurses*'s API felt quite dated, so I started the *rome* subproject.

Let's write some code with *rome*.

Requirements:

- A computer.
- *rome* "installed". Currently, *rome* does not have an install rule in its Makefile, but in the future, git clone the *Medusa* Project *git* repository as usual, then cd Medusa/medusa_software/rome, and make.
- Your preferred IDE for editing C++ code. (in the future, Medusa will hopefully be able to be used for this purpose. (specifically, the london subproject / subcomponent which will be integrated into the larger tool.))
- A compatible C++ compiler: we use *LLVM / Clang*. [3]

[2] This essentially serves as documentation for both the DisplayOutput generic class, and the ExampleDisplayComponent implementation of said generic class.
Most code is stolen/paraphrased from the ExampleDisplayComponent implementation.

[3] See footnote 1.

First, `#include <rome/rome.hpp>`. Now, create a `rome::window`.

```
rome::window window;
```

Now, let's add some text!

```
/*
 *  rome::window::addstr(std::string str, int x, int y) prints a
 *  string `str` to the screen at position `(x, y)`.
 */
window.addstr("This is rome example code.", 3, 1);
```

How about we reverse that text, like a titlebar? (Background becomes foreground, and vice versa.)

```
/*
 *  rome::window::chgattr(int attr,
 *                        int x,
 *                        int y,
 *                        int n,
 *                        int color_pair) will set the terminal attributes
 *  `attr` at the location `(x,y)`, for `n` characters, using the `curses` color
 *  pair `color_pair`. Using a negative number for `n` will change the
 *  attributes up until `abs(n)` characters from the end of the line.
 */
window.chgattr(A_REVERSE, 1, 1, -2, 0);
```

Finally, let's wait for a key, and exit the program.

```
/*
 *  rome::window::getch() returns an `ncurses` `int` key. We discard the actual
 *  key returned, we just want to wait for a key press.
 */
window.getch();
```

When the `rome::window` object is destroyed, the proper / normal terminal state is automatically reset. So, we're done!

Congratulations! You just wrote your first program with *rome*.

5 Links

This section contains useful links for the *Medusa* Project.

1. medusa-re.org the *Medusa* Project's website

2. docs.medusa-re.org documentation for the *Medusa* Project's software, libraries, and API's. (mostly[4] doxygen)
3. wiki.medusa-re.org the *Medusa* Project's wiki
4. gitlab.com/MedusaRE the *GitLab* organization / group for the *Medusa* Project
5. gitlab.com/MedusaRE/Medusa the *GitLab* repository for the *Medusa* Project
6. gitlab.com/MedusaRE/wiki the *GitLab* repository for the *Medusa* Project's wiki
7. https://gitlab.com/MedusaRE/Medusa.git the *git* repository for the *Medusa* Project, hosted by *GitLab*
8. https://gitlab.com/MedusaRE/wiki.git the *git* repository for the *Medusa* Project's wiki, hosted by *GitLab*
9. https://git.medusa-re.org/Medusa.git a "futureproof" *git* repository URL for the *Medusa* Project [5]
10. https://git.medusa-re.org/wiki.git a "futureproof" git repository URL for the *Medusa* Project's wiki [6]

6 Contributors

The *Medusa* Project is a pet-project of mine. As such, most contributions have been made by me. All contributors (including me) are listed below.

- spv (*spv@spv.sh*)

7 Credits

The *Medusa* Project might not have been possible without the work of many others. A non-exhaustive list of individuals/entities that deserve credit for their work is provided below. Note that none of these entities are necessarily associated with the *Medusa* Project, just that their work helped the project.

[4] Read: "currently all".

[5] While the git.medusa-re.org domain is hosted by the *Medusa* Project (or even more specifically, my home server), the *git* repositories you can access from it (namely, `Medusa.git` and `wiki.git`) are not. They are currently hosted by *GitLab*. I simply set up my *Apache* installation to redirect *git.medusa-re.org/** to *https://gitlab.com/MedusaRE/**. This is done so that, in the unlikely case that the *Medusa* Project has to move away from *GitLab*, *git* repository URL's (and remote URL's used by *git* for pushes and such) will not have to be updated.

[6] See footnote 5.

- spv (*spv@spv.sh*) founding the project
- Nguyen Anh Quynh (*aquynh@gmail.com*) *Capstone*, *Keystone*, *Unicorn*
- The *QEMU* Team (qemu.org) *QEMU*
- *LIEF* Project (lief-project.github.io) *LIEF*
- The *GNOME* Project (gnome.org) *GTK*, *GTKMM*