Project Detail

Overview

As a binary emulation framework, Qiling has done a nice job in emulation. But Qiling has limited ability to analyze binaries, users have to rely on other tools for useful information like function addresses then hardcode them, which is inconvenient.

My entire project aims to bridge Qiling with other static analysis software, thus providing users with high-level concepts like stack frame, CFG and even symbolic execution. The robustness and usability of the project should also be improved to make everything tidy, which may involve some refactoring.

To be specific, I will use r2libr to integrate the functionality of radare2 into Qiling. If time permits, rizin can also be supported by rz-pipe or their SWIG bindings which may be introduced in the future.

Motivation and Goals

Goal 1: Improve user experience to eliminate hard-coded value

Many hardcoded values, most of them being addressed and opcodes, can be seen in the official examples, which is confusing and error-prone. By adding binary analysis functionalities to Qiling, the hardcoded values could be eliminated.

Goal 2: Provide APIs at binary layer

Qiling can emulate multiple platforms, architecture and file formats with the support of capstone and unicorn. The functionalities are divided into different layers (memory, loader, OS), it may be better to have a **binary layer** that can connect with other layers and provide intuitive APIs.

Goal 3: Explore use case symbolic execution and deobfuscation

With the high-level analysis information like CFG, it is possible for Qiling to do symbolic execution and deobfuscation. Demos and documents should be made as proof of concept.

Existing Work

It is necessary to make it clear that Qiling's IDA plugin, which can make Qiling a debugger backend for IDA, has little to do with this project since we are doing almost the opposite.

I have found two popular Python projects which utilize radare2/rizin to do basic analysis.

- Quark-engine uses rz-pipe as one approach to analyzing APK and DEX files
- Uefi_r2 uses rz-pipe to analyze UEFI firmware

There are also some reverse engineering frameworks in Python that can be learned.

- Barf-project is a binary analysis framework with high level analysis functionalities
- Angr is a platform-agnostic binary analysis framework
- Miasm is a comprehensive reverse engineering framework
- Pwntools is a CTF exploitation framework

Design

These are some key questions I want to answer before starting this project. I will discuss the details with my mentors during the summer.

How to bridge with Radare2/Rizin?

Radare2/Rizin is a command-line reverse engineering framework with a shell-like UI, where users enter commands then get results. Multiple language bindings are officially provided to let users get results by calling the same commands in different languages. The result is always returned as string, but most commands can print in JSON format, which can be easily converted to Python dict. So Qiling can get the necessary information and store them as custom objects, on top of which more high level methods could be built.

How to achieve modularity and maintainability?

Although this project only considers Radare2/Rizin, it is possible to bridge other static analysis software like Ghidra and IDA in the future. Considering extensibility, there will be an abstract class to define which properties and methods should be implemented by each extension, namely r2 and rizin.

The basic concepts like symbol and function, shown as JSON in Radare2/Rizin, will be converted to classes in Python. To be specific, dataclass can be used to organize collections of data. It should be noted that these concepts are platform-independent, which means they are more of simple encapsulations than complete ABI like elftools.

The newly added directory structure should be as follows:

```
qiling/bin
├── __init__.py
├── base.py  # classes of symbol, section, function, etc.
├── analyze.py  # classes of basic blocks, instr, CFG, etc
qiling/extensions/r2
├── __init__.py
└── r2.py
```

I have already made a draft version, the code is like below:

```
@dataclass(unsafe_hash=True)
class QlBinFunction:
    name: str
```

```python
    offset: int
    size: int
    signature: str

@dataclass(unsafe_hash=True)
class QlBinSection:
    name: str
    size: int
    vsize: int
    perm: int
    paddr: int
    vaddr: int

@dataclass(unsafe_hash=True)
class QlBinString:
    name: str
    vaddr: int
    paddr: int
    size: int
    length: int
    section: str
```

```python
class QlBinAnalyzer(ABC):
    """
    An abstract base class for concrete static binary analyzers.
    To extend a new binary analyzer, just derive from this class and implement
    all the methods marked with the @abstractmethod decorator.
    """

    def __init__(self):
        super().__init__()

    @property
    @abstractmethod
```

```
def sections(self) -> Set[QlBinSection]:
    raise NotImplementedError

@property
@abstractmethod
def strings(self) -> Set[QlBinString]:
    raise NotImplementedError

@property
@abstractmethod
def symbols(self) -> Set[QlBinSymbol]:
    raise NotImplementedError

@abstractmethod
def addr_of_sym(self, name: str) -> int | None:
    raise NotImplementedError

@abstractmethod
def addr_of_fcn(self, name: str) -> int | None:
    raise NotImplementedError

@abstractmethod
def addr_of(self, name: str) -> int | None:
    raise NotImplementedError
```

How to improve user experience?

Qiling has advanced emulation of CPU(registers), OS and memory, which makes it powerful and flexible. But sometimes users do not care about those details, it will be more convenient if they can manipulate the execution from a binary perspective.

For instance, users must hardcode the address to hook functions, since the address should be passed to unicorn.

```
ql.hook_address(callback=start_afl, address=ba + 0x1275)
```

After the binary analyzer is introduced, we can just use

```
ql.hook_address(callback=start_afl, address=ql.analyzer.addr_of("main"))
```

At first, the binary analyzer can be implemented as a relatively independent module to avoid breaking the existing code. When time is ripe, more methods can be added to Qiling's core class so users can do the same thing more conveniently.

```
ql.hook_function("main", callback=start_afl)
```

The method looks like ql.os.set_api, but the latter could only hook system APIs. It will be beneficial to have an API to hook all functions without specifying their addresses.

Moreover, many APIs could be added on top of ql.mem, so users could modify binary data without manually dealing with memory.

How to utilize high level analysis results?

At HITB Sec Conf 2021, the mentor presented a demo to use Qiling for symbolic execution, but he did not complete it because the code has gone missing.

Esilsolve, which is a Python symbolic execution framework using radare2, can be used as an extension for symbolic execution. Its API design is similar to the current Qiling API so it will not be difficult to integrate esilsolve into Qiling.

I also find angr's top level interfaces well-designed, like the loader and basic blocks factory. Their official document provides many examples, most of which are CTF demos. I can try to solve these CTF problems using Qiling and make them new examples.

Another use case is to reimplement the deflat functionality in Qiling's IDA plugin.

Timeline

This is the timeline that I think is suitable. If, for any reason, a task is taking too long or cannot be completed, I will contact the mentor and after discussing the reason and explanation, suitable action will be taken. Also if any task needs to be rescheduled, that can be done after proper discussion from the mentor.

I. Community Bonding Period [May 20 - June 12]

I'll familiarize myself with the mentors, understand the codebase more deeply and discuss my project with mentors.

II. Week 1-2 [June 13 - June 26]

- Implement binary information analysis using r2
- Make sure the results are compatible with existing loader code

III. Week 3-4 [June 27 - July 10]

- Implement function analysis using r2
- Write unit tests and update examples with newly added API

IV. Week 5-6 [July 11 - July 24]

- Integrate new code with existing mem and regs to improve reusability
- Write unit tests and update examples with newly added API

When Phase 1 Evaluation deadline comes, Qiling should have been able to get binary information from radare2 and build easy-to-use new APIs at binary level.

V. Week 7-8 [July 25 - August 7]

- Implement basic blocks and CFG interfaces like angr

VI. Week 9-10 [August 8 - August 21]

- Implement symbolic execution demo using esilsolve
- Add new API, write unit tests and update examples

VII. Week 11-12 [August 22 - September 4]

- Rewrite deflat functionality using radare2
- Add new API, write unit tests and update examples

VIII. Final Week [September 5 - September 19]

- Implement similar things using rizin
- Buffer period in case any task takes a longer time

At the final stage, users should be able to get high-level analysis results to help their work of reverse engineering, binary emulation, or even symbolic execution. The whole project should be more robust and developer-friendly with the improvements of tests and documents.