# Concolic analysis with *angr*

Laboratory for the class "Security Verification and Testing" (01TYASM/01TYAOV)
Politecnico di Torino – AY 2023/24
Prof. Riccardo Sisto

*prepared by:*
Cataldo Basile (cataldo.basile@polito.it)

v. 1.2.1 (13/12/2023)

## Contents

## Purpose of this laboratory

The purpose of this laboratory is to start using *angr*, a very powerful tool for the concolic analysis of binaries.

The *angr* documentation is overall excellent. A large community uses, works with, and contributes to *angr*. However, they are mainly researchers and CTF players (especially the Shellphish team, who have developed it and used to participate in the Cyber Grand Challenge). For this reason, the documentation and examples cannot be considered user-friendly or industry-ready. The learning curve of *angr* is relatively high. Hence, it is rarely used for testing purposes in corporate scenarios.

This laboratory aims to provide a set of basic examples that can allow approaching *angr* and appreciate its features, hoping that this tool could enter, sooner or later, the toolchain of the security testing experts.

The executables and templates needed for this laboratory are available in the file:

`lab05_material.zip`

that is available on the Portal.

### Installing *angr* and downloading documentation

The installation of *angr* requires attention as it overwrites some standard Python packages. Therefore, it is strongly suggested to be installed in a virtual environment. This document reports all the steps for installing and configuring *angr* for the VM prepared for the laboratories, which may require additional packages compared to a Kali full VM.

To avoid the mistakes originating from copy-and-paste from a pdf file, we have provided the file `useful_commands.txt`, available with the material provided with the lab, where you can find the strings of the commands to execute to install *angr*.

First, the following packages need to be installed, as reported here:

```
sudo apt-get update
sudo apt-get install python3-dev libffi-dev build-essential virtualenvwrapper
```

After installing `virtualenvwrapper`, you have to set the environment variables and execute the reconfiguration with (the last shell script may be located elsewhere in your machine, just type the following command to discover the actual position `sudo find / -iname "virtualenvwrapper.sh"`):

```
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/Devel
source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
```

Add these three lines in `/.zshrc` if you want to add support for the virtual environments permanently.

The *angr* tool can be installed with this command:

```
mkvirtualenv --python=$(which python3) angr && pip install angr
```

Finally, it is strongly suggested to experience with *angr* using an interactive Python environment. We suggest IPython, which can be installed with the following:

```
sudo apt install ipython3
```

We suggest cloning the repository containing the *angr* documentation in a folder of your choice. It provides excellent examples to continue your *angr* study:

```
git clone https://github.com/angr/angr-doc.git
```

### Working with virtual environment

> **NOTE**
>
> If you correctly installed *angr*, you will automatically find an environment name `angr`, which is the one to use for the rest of the exercise. Nonetheless, should you need to create more virtual environments, here are the basic instructions to manage them. Moreover, before doing your exercises, you will need to move the `angr` environment.

To see all the virtual environments you have created on your machine, just use the following command

```
workon
```

To create a virtual environment named `environ` use the `mkvirtualenv` command:

```
mkvirtualenv environ_name
```

To enter the `environ` virtual environment, use the `workon` command:

```
workon environ_name
```

To exit from a virtual environment, type:

```
deactivate
```

Alternatively, you can install a virtual environment with `venv`. More instructions here:

https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/#creating-a-virtual-environment

## Further documentation

Here are a few more links with introductory material about *angr* and the API references:

- https://docs.angr.io/en/latest/
- https://github.com/angr/angr-doc
- https://docs.google.com/presentation/d/1eOOW6-roopNHg8J4DJSD1V0EWzKFHxAzFKfbyRcTt2o/edit#slide=id.p

# 1 Using *angr* for static and dynamic analysis

Before continuing your laboratory exercises, remember to enter the *angr* virtual environment.

## 1.1 Creating an analysis project

The use of *angr* requires that the binaries are properly wrapped with ad-hoc Python objects, which can be instantiated as explained below (inside an IPython interactive environment):

```
$ ipython3
In [x]: import angr
In [x]: p = angr.Project('filename_of_the_binaries')
```

**ATTENTION**

If you use the '~' character in the path of the binaries, *angr* will not properly work as it does not escape it correctly.

Several options can be used when the project is instantiated, as reported in the official documentation here:

https://docs.angr.io/en/latest/api.html#angr.Project

Moreover, even if it is not reported in the documentation, we have found several examples documenting the use of the `load_options={"auto_load_libs": False}`. This option will indicate *angr* that the libraries must not be loaded when the project is created (it is a sort of lazy loading).

Now, write the code to create a project that wraps the binaries of the application `crackme2` without opening and processing the libraries.

```
→
```

The object `p` will be the base of all the subsequent *angr*-based analysis and hacking steps.

Note that you can extract useful information from a Project object, for instance:

```
In [x]: p.entry #the entry point of the binaries
In [x]: p.filename #the path to the binaries
```

Explore the attributes and functions associated with the project with the IPython autocompletion:

```
In [x]: p.
```

then pressing Tab, a menu will appear to show all the available options.

## 1.2 Getting ArchInfo data

The ArchInfo module extracts information about the platform for which the binaries have been conceived.

Try to find additional information by typing:

```
In [x]: p.arch.
```

then pressing the TAB key to enable autocompletion.

Annotate here the attributes that you consider interesting for gathering information useful for performing binary analysis and add a description explaining why you consider it relevant:

```
→
```

More data about ArchInfo are available here https://api.angr.io/projects/archinfo/en/latest/api.html

### 1.3 The *angr* Loader

The loader is the component that can prepare the binaries for execution. The *angr* loader (CLE) performs more or less the same operations the OS would have performed when loading the binary in memory. The advantage is that CLE is much more general. It understands different types of binaries for different platforms and processes them to be symbolically executed and debugged.

More information about CLE is available here:

https://docs.angr.io/projects/cle/en/latest/quickstart.html

CLE provides several methods and attributes. One of the most interesting attributes is the *main object*, which is the element the loader would start in case of execution of the binaries. You can save it to a variable for ease of access using the following:

```
In [x]: main_obj = p.loader.main_object
```

It is interesting, for binary analysis purposes, knowing the address where the binaries of the main object start and finish:

```
In [x]: hex(main_obj.min_addr)
In [x]: hex(main_obj.max_addr)
```

Moreover, you can navigate ELF data (as you already did in the past laboratory with `rabin2` or `objdump`):

```
In [x]: p.loader.shared_objects
In [x]: p.loader.all_elf_objects
%In [x]: p.loader.find_object_containing(address)
```

Using the autocompletion applied to the `p.loader` object and the `main_object`, try to find the attributes that provide information about the protections applied to the binaries (e.g., DEP, ASLR, PIE, RELRO) and report the *angr* command and a description of the protection:

```
→
```

### 1.4 The *angr* project Factory

As the name clearly says, the factory associated with the project allows for the creation of objects useful for analysing binaries with *angr*, including:

- *states*, objects able to contain all the information about a running application (e.g. blocks to be executed, the value of all the registers, the memory, I/O of the execution platform)

  https://github.com/angr/angr-doc/blob/master/docs/states.md
  https://docs.angr.io/en/latest/core-concepts/states.html

- *simulation managers*, objects that can perform a concolic execution of the binaries. These objects transform an *angr* state into a new one after executing a code block. For this purpose, they need to be instantiated with an initial simulation state

  https://docs.angr.io/en/latest/core-concepts/pathgroups.html#

- *basic blocks*, which are pieces of binaries with no jumps that may be disassembled by *angr* modules (starting from a given point in memory)

  An ugly visualisation of a disassembled block can be obtained with the following code (better ways to show disassembled code can be found in Section 3.0.1)

```
import angr
p = angr.Project("/bin/true",auto_load_libs=False)
block = p.factory.block(p.entry)
block.pp()
```

## 1.5 The Simulation Manager

The Simulation Manager is the component in charge of dynamically executing the application under analysis, probably the core of the innovation proposed by *angr*. It uses symbolic techniques and resorts to concrete values when asked by the user or when symbolic values "collapse" into concrete ones.

### 1.5.1 Symbolic execution

As a first sample exercise, we propose a step-by-step presentation of the operations you must do to use *angr* to crack one of the applications you have already seen in the laboratory n.4, namely the `crackme2` program. The crack is successful if you discover the password that allows you to bypass the authentication (which you can also see with strings; you know that already from lab03; however, the purpose is learning *angr*).

Suppose you have statically analysed the code of the application, you should have already done it with Radare2 for the past laboratory; nonetheless, we report here the commands to look for the information

```
$ r2 crackme2
[xxx] aaaa
[xxx] sf main
[xxx] VV
```

You can also use `pdf` instead of `VV` to see the disassembled code if you want to copy the memory address value.

After reading the disassembled or decompiled code or navigating with the arrow keys into the CFG, you know the exact point the application will reach when the crack is successful. That is, you want to force the Simulation Manager to reach the branch where the comparison of the input password with the fixed value is evaluated to True.

Now determine and write down the address of the branch you want the Simulation Manager to enter:

→

Then, execute the following instructions in an Ipython interactive environment:

```
In [x]: import angr
In [x]: p = angr.Project('./crackme2', load_options={"auto_load_libs":
    False})
```

Before simulating the execution, you have to instantiate the Simulation Manager associated with the project. However, it needs to know where to start. One of the best alternatives to start is, of course, the entry state of the binaries, which can be obtained by using the *angr* factory, as in

```
In [x]: es = p.factory.entry_state()
```

An *angr* state reports all the information about the platform (registers, memory, stdin, stdout, etc.) so that the following instructions (basic block) can be properly executed.

You can instantiate the Simulation Manager and execute it with the following commands:

```
In [x]: sm = p.factory.simulation_manager(es)
In [x]: sm.explore(find=address of the branch)
```

Write down the actual command to execute until the branch where the password is correct:

```
→
```

You will find the following output:

```
<SimulationManager with 2 active, 9 deadended, 1 found>
```

**ATTENTION**

You cannot use the `explore` method of the Simulation Manager more than once, its behaviour may be not the one you expect. If you intend to use explore more than once, you need to allocate a fresh Simulation Manager.

This string reports that the simulation Manager has symbolically executed the `crackme2` binaries. It has exited from the program for nine explored branches without reaching the required state (*deadended*). In one branch, the program reached the address we have passed (*found*) Moreover, there are still two active states from which one may want to continue the execution (it is not our case, however). The found state can be accessed using the Simulation Manager `found` list:

```
In [x]: found = sm.found[0]
```

The `posix.dumps` function allows accessing the standard input and output.

Now, we can see the output of the application when the found state is reached.

```
In [x]: found.posix.dumps(0) # stdin
In [x]: found.posix.dumps(1) # stdout
```

Note that the symbolic execution explores all the found branches in parallel. This fact is evident as the Simulation Manager has two active branches.

```
In [x]: sm.active[0]
In [x]: sm.active[1]
```

### 1.5.2 Starting the simulation from any state

The entry point is just a very convenient initial state for the simulation. However, you can instantiate a Simulation Manager from any state. One of the most valuable things in practice is to execute until you reach a specific condition, then step-wise, continue the dynamic analysis of the application.

An elementary (and silly) example can be continuing the execution from the point where we stopped the execution at the found state. By executing this command:

```
In [x]: found.posix.dumps(1)
```

you can note that the `printf` still needs to be executed.

To this purpose, we instantiate a new Simulation Manager at the found state:

```
sm1 = p.factory.simulation_manager(found)
```

then, we step-wise execute the binaries with

```
In [x]: sm1.step()
```

Continue stepping until you see the correct output (note that when the output is the expected one, the application has reached the address we have determined in the found state).

```
In [x]: sm1.active[0]
In [x]: sm1.active[0].posix.dumps(1)
```

Note from the addresses of the active states that *angr* does not execute single assembly instructions; it executes symbolically relevant basic blocks (i.e. until something happens at symbolic execution like calling functions, variables that change their symbolic state, etc.).

You can use additional Simulation Managers to try to give meaning to the two active states the Simulation Manager uses. Can you guess something about the two *active* states?

→

### 1.5.3 Pruning unwanted execution branches

The symbolic execution may require much time and computation with small non-trivial examples. If a preliminary analysis ensures that you can exclude some branches (for instance, because you already know from the manual inspection of the static disassembled code), you better use the `avoid` construct (you have to restart the simulation manager at the entry state):

```
In [x]: sm = p.factory.simulation_manager(entry_state)
In [x]: sm.explore(find=address of the branch, avoid=address of the branch
   to avoid)
```

Write the command to set the exploration that excludes the branch where the input password is not the correct one:

→

If you did it correctly, the output would be:

```
<SimulationManager with 1 active, 1 found, 10 avoid>
```

Given the complexity of the example binaries, *angr* only needs a few hundred milliseconds more to explore the avoided branches (as they immediately reach an exit/return and become deadended). Thus you cannot appreciate how important it is to instruct *angr* to avoid branches when you already know they are not leading to the part of the application you want to inspect. However, you can easily reduce the simulation time from months to minutes in some not-so-complex binaries.

### 1.5.4 Conditional execution with PIE binaries

In this second example, the loading of the `crackme3` application memory layout is randomised, as you can check with the *angr* loader (or using rabin2). Moreover, *angr* will inform you of these characteristics when you load the project.

To this purpose, we have to use an additional feature of *angr* to find the correct memory point to stop the execution.

Determine the offset of the branch to reach with a disassembler, e.g. radare2. This value will not be an absolute memory address; it is just the offset from the beginning of the code segment.

Annotate here the offset:

→

Now you can then use the information from the loader to compute the base address by means of:

```
In [x]: p.loader.min_addr
```

Annotate here the base address:

→

Now that you have the real address, repeat the steps in Section 1.5 using new information to find the input that needs to crack the application.

Write down the Python instructions to crack the `crackme3` application:

→

Find the input with `posix.dumps(0)` on the found state. Annotate the found key here:

→

Now, check that the string is valid by executing the application on a terminal:

```
echo "found_input" | ./crackme3
```

Try recreating the simulation manager and exploring it; did you find the same value? Guess something about *angr* internals.

### 1.5.5 Working with known output

Imagine now you have analysed the code of an application statically. Instead of concentrating on the memory address to reach, you know the output the application produces when it reaches the point of your interest. This is typical when some bugs have been triggered, and you only want to execute those application parts that led to the bugs.

You can use *angr* to symbolically execute the application until the desired output is produced. Instead of passing the address, we can take advantage of the `find` parameter, which also accepts the name of a Python function to execute at each execute state (in this case, it is a lambda Python function and the check to perform is simple):

```
In [x]: sm.explore(find=lambda s: b"the output you want" in s.posix.dumps(1))
In [x]: print(sm.found[0].posix.dumps(0)) #now print the input
```

Crack the `crackme2` and `crackme3` programs again by following this approach. Report in the space below the commands that you have to type in IPython.

For `crackme2`:

→

For `crackme3`:

→

### 1.5.6 Managing inputs as command line arguments

Claripy is the interface of the SMT Solver used as the backend. If you did not change anything in the default *angr* configuration, the solver is Z3.

Stating complex conditions is essential for advanced use of *angr*. For instance, these conditions are important to determine when the Simulation Manager must stop and what it has to avoid.

To start learning Claripy, we suggest following the tutorial in the official *angr* documentation:

https://docs.angr.io/advanced-topics/claripy

https://docs.angr.io/appendix/ops

Meanwhile, let's try a simple example showing the importance of mastering Claripy.

The analysis that finds valid inputs from standard input can also be performed on any other input types, like registers and variables. It is simply necessary to create a symbolic variable and then consider it during the execution.

The simplest example to prove this *angr* feature is to use command line arguments.

Consider then the application './fairlight', available in the *angr* documentation at

```
angr-doc/examples/securityfest_fairlight
```

Try it with (make it executable if needed with `chmod`):

```
fairlight password
```

To this purpose, we have first to generate a symbolic variable that will represent the argument to pass with the following commands (you have to import Claripy, the *angr* interface to the solver):

```
In [x]: import angr, claripy
In [x]: p = angr.Project('./fairlight', load_options={"auto_load_libs":
    False})
In [x]: argv1 = claripy.BVS("argv1", size_in_bits_of_argv1)
In [x]: initial_state = p.factory.entry_state(args=["./fairlight", argv1])
```

For `size_in_bits_of_argv1` guess the typical value of buffers, check the disassembled binaries to find the correct value or write a script that progressively increases the size until *angr* finds a valid state.

Write the command to instantiate the Simulation Manager and execute it until id does not find the branch in the main function that returns an affirmative output. Then save the found state in the `found` variable:

```
→
```

To show the correct value of the symbolic variable, you need to call the SMT solver, which will evaluate the conditions associated with the branches, then print the value in as bytes:

```
In [x]: found.solver.eval(argv1, cast_to=bytes)
```

### 1.5.7 Further reading

For a deeper look at the Simulation Manager, have a look at this document:

https://github.com/angr/angr-doc/blob/master/CHEATSHEET.md

and at this link:

https://ekse.gitbooks.io/angr-dev/content/docs/paths.html

## 2 Exercises

This section includes five exercises you can solve using *angr* to practice with the basic features of this powerful tool.

### Exercise 1

| | |
|---|---|
| Purpose: | bypass the constraint in the `func` function and make the program print the string "This is the answer". |
| Suggested tools: | *angr*, strings/rabin/objdump. |
| Hints: | look for the strings, find the one you want to see printed and avoid the other one. |

As a first exercise, you must exploit the same buffer overflow you have already seen in one of the past labs (the same example application you saw during class).

The binary file is available in the lab05 material file:

`/lab05_material/crackme1`

You also have access to the source code.

### Exercise 2

| | |
|---|---|
| Purpose: | this is another example of password guessing. |
| Suggested tools: | angr |
| Hints: | Claripy. Use the template provided, which will guide you through the solution. |
| Source: | PSU angr CTF (https://angr.oregonctf.org/solve/) |

In this case, you need to model the registers to solve this exercise. Claripy serves this purpose: *angr* can guess the password if you pass Claripy the proper registers' bit size. You must also use a state that forces the execution of the Simulation Manager just after the `scanf` using a `blank_state`. Indeed, *angr* cannot manage cases when multiple inputs are received from the standard input.

The binary file is available in the lab05 material file:

`/lab05_material/angr_symbolic_registers`

The template solution is available in the following file:

`/lab05_material/template_solution_angr_symbolic_registers`

### Exercise 3

| | |
|---|---|
| Purpose: | this is another example of password guessing. |
| Suggested tools: | angr |
| Hints: | use Claripy again, but it will be more complex now. Use the template provided, which will guide you through the solution. Remember to store the symbolic objects in the memory of the initial state |
| Source: | PSU angr CTF ()https://angr.oregonctf.org/solve/) |

In this case, you need to model the memory to solve this exercise. All the considerations from the previous exercise also hold in this case. You only have to store the symbolic objects in the initial state of the four inputs.

The binary file is available in the lab05 material file:

`/lab05_material/angr_symbolic_memory`

The template file is

`/lab05_material/template_solution_angr_symbolic_memory`

### Exercise 4 (Advanced)

| | |
|---|---|
| Purpose: | Guess the password, again. |
| Suggested tools: | angr |
| Hints: | again Claripy, but you have to constrain its default behaviour a bit. Use the template provided, which will present you with many details and suggestions, and guide you through the solution. |
| Source: | PSU angr CTF (https://angr.oregonctf.org/solve/) |

In this case, more than the default behaviour of Claripy is needed to solve this challenge. You need to look at the name of the 'checking function' and explain to Claripy how to manage the function names properly. The template will provide more detailed information on how to solve the exercise.

The binary file is available in the lab05 material file:

`/lab05_material/angr_constaints`

The template file is

`/lab05_material/template_solution_angr_constraints`

### Exercise 5 (Advanced)

| | |
|---|---|
| Purpose: | modify the program control flow to make the program print the content of `flag.txt`. |
| Suggested tools: | *angr*, cyclic, Ghidra or radare2. |
| Hints: | The program already implements a win-function that executes "cat flag.txt" |
| Source: | ROP Emporium |

**ATTENTION**

`angrop` does not work with the last version of angr as one required class (a logger) has been renamed.
One very quick (and dirt) workaround to allow you to use `angrop` is to open with a text editor the file
`/home/aldo/.virtualenvs/angr/lib/python3.10/site-packages/angr/misc/loggers.py`
then make a copy of the CuteFormatter class and rename it as CuteHandler.

This exercise is the first ROP exercise presented in lab03. You will need to:

- pay attention to the alignment, as indicated in the Warning shown in the text of the past lab;
- find the size of the padding, e.g., with cyclic, like in the original exercise;
- find the name of the win-function (the symbol).

Then, you have to use *angr* to implement the ROP attack.

*angr* can also be used to generate the ROP chains, though this is not a completely automated task. *angrop* provides an interface to find gadgets and others that expose methods that build chains to perform high-level operations.

https://github.com/angr/angrop

If you want to learn how to use *angrop*, you should first study how to exploit binaries with ROP. Therefore, it is suggested to first solve this challenge (and the ROP Emporium ones) without *angrop*, then do the same solutions using the features provided by this tool.

https://ropemporium.com/

Compared to the tools that find gadgets statically (pwntools, ropper, onegadget), `angrop` provides additional semantic information that you may find useful when building a ROP chaing (involved registers, length in bytes of the gadget, etc.).

# 3 Extras

This section reports additional problems that you can solve with *angr* but are not mandatory for this laboratory and, in general, for this course. Correctly managing these features would require too much effort to resort to additional utilities provided with *angr*, and, often, skills in Python, we cannot assume. For this reason, you can consider these sections as additional material to start your study, should you plan to improve your knowledge of *angr*. In short, it is an optional section.

You can install the utilities:

```
git clone https://github.com/angr/angr-dev
cd angr-dev
git clone https://github.com/axt/bingraphvis
pip install -e ./bingraphvis
git clone https://github.com/axt/angr-utils
pip install -e ./angr-utils
```

### Analyses

You can access the results of the different analyses *angr* provides. As before, you can start from a project in IPython and try all the available analyses available at a project:

```
p.analyses.
```

then using autocompletion to see all the available functions. Not all the *angr* analyses are documented, but their names are usually self-explanatory.

#### 3.0.1 Control Flow Graphs

*angr* also generates the control flow graph; an approximated version can be obtained with `CFGFast()` a more accurate one with `CFG` and `CFGEmulated`.

More documentation and examples are available here:

https://docs.angr.io/built-in-analyses/cfg

Methods to plot a CFG are available as utils:

https://github.com/axt/angr-utils

A very comprehensive and general example is here:

https://github.com/axt/angr-utils/blob/master/examples/plot_cfg/plot_cfg_example.py

#### 3.0.2 Function identifiers

You can extract the names of the functions with the following code that uses the `Identifiers()` analysis:

```
id = p.analyses.Identifier()
for funcInfo in id.func_info:
    print(hex(funcInfo.addr), funcInfo.name)
```

### Exploitation

*angr* can also be used for exploiting binaries. As an instance, in the angr-doc you can find a challenge from the Insomni'hack 2016 CTF

```
angr-doc/examples/insomnihack_aeg
```

To perform the required buffer overflow attacks, the objective is to find a state where the Program Counter is completely under the control of the user, which translates into finding a completely symbolic Program Counter, as in the script linked below:

https://github.com/angr/angr-doc/blob/master/examples/insomnihack_aeg/solve.py

As you will immediately notice, this is a much more advanced topic. Moreover, this can be considered the very first example of Automatic Exploit Generation, as the script automatically finds the BOF vulnerability and generates the correct payload to spawn a shell.

Another example of buffer overflow is the CADET_00001 available in the *angr* documentation

```
angr-doc/examples/CADET_00001
```

## 3.1 Additional resources

A more comprehensive and detailed tutorial is available on an external resource. It is divided into four parts that start from basic concepts to very complex ones:

- https://blog.notso.pro/2019-03-25-angr-introduction-part1/
- https://blog.notso.pro/2019-03-26-angr-introduction-part2/
- https://blog.notso.pro/2019-04-03-angr-introduction-part2.1/
- https://blog.notso.pro/2019-04-10-angr-introduction-part3/

This paper also presents some useful material:

- https://www.usenix.org/system/files/conference/ase18/ase18-paper_springer.pdf

The list of examples that have some context to introduce them is here:

- https://github.com/angr/angr-doc/blob/master/docs/examples.md
- https://github.com/angr/angr-doc/blob/master/docs/more-examples.md