Concolic analysis with *angr*

Laboratory for the class "Security Verification and Testing" (01TYASM/01TYAOV)
Politecnico di Torino – AY 2021/22

v. 1.1 (21/12/2020)

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. There is a large community of people using, working with, and contributing to *angr*. However, they are mainly researchers and CTF players (expecially the shellphish team who has developed it and used to participate the Cyber Gran Challenge). For this reason, the documentation and examples cannot be considered user-friendly or industry-ready. The learning curve of *angr* is very high, hence, it is rarely used for testing purposes in corporate scenarios.

This laboratory is intended to provide you a set of basic examples that can allow you to approach *angr* and appreciate its features, hoping that this tool could enter, sooner or later, in the tool chain of the security testing experts.

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

```
lab05_material.zip
```

that you can find on the Portal.

Installing *angr* and downloading documentation

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

To avoid the mistakes originated from copy-and-paste from a pdf file, we have provided the file `useful_commands.txt`, available withthe 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
```

The *angr* tool is actually 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 enviroment. We suggest IPython, which can be installed with:

```
sudo apt install ipython3
```

ATTENTION

Unfortunately, the last version of IPython may have problems with the autocompletion (incompatibility with a required package) which is very useful to explore the features of *angr* without the need to navigate the documentation. If (and only if) you note problems, the only known workaround to date (not effective in 100% of the cases) is to downgrade the autocompletion library to a past version with:

```
sudo pip install jedi==0.17.2
```

We suggest you to clone the repository containing the *angr* documentation in a folder of your choice. You will find excellent examples to continue your *angr* study:

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

Working with virtual environment

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

```
mkvirtualenv environ
```

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

```
workon environ
```

To exit from a virtual environment, type:

```
deactivate
```

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

```
workon
```

> 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.

In alternative, you can install 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 a few more links with introductory material about *angr* and the API references:

- https://angr.io/api-doc/
- 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

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 (insiede an IPython interactive environment):

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

There are several options that can be used when the project is instantiated, as reported in the official documentation here:

http://angr.io/api-doc/angr.html#module-angr.project

Moreover, not reported in the documentation, we have found several examples that document the use of the `load_options={"auto_load_libs": False}`, to indicate *angr* that it must not automatically load the libraries 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 next *angr*-based analysis and hacking steps.

Note that you can basic extract 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 to 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 extract information about the platform for which the binaries have been conceived.

Also in this case, try to find additional information by typing:

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

then pressing the TAB key to enable the 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 http://angr.io/api-doc/archinfo.html

1.3 The *angr* Loader

The loader is the component able to prepare the binaries for the 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 the binaries for different platforms and processes them to be symbolically executed and debugged.

More information about CLE are available here:

`http://angr.io/api-doc/cle.html`

CLE provides several methods and attributes. One of the most interesting information 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 by means of:

```
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)
```

Note that these values are the presented also for binaries are compiled with PIE enabled.

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_objemainct_containing(address)
```

By means of the autocompletion applied to the `p.loader` object and the `main_object`, try to find the attributes that information about the protections applied to the binaries (e.g., DEP, ASLR, PIE) 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 to 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)

  `http://angr.io/api-doc/angr.html#module-angr.sim_state`

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

  `http://angr.io/api-doc/angr.html#module-angr.sim_manager`

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

  http://angr.io/api-doc/angr.html#module-angr.sim_state

  An ugly visualization of a disassembled block can be obtained with the following code (better ways to show disassembled code can be found in Section 3.1.1)

  ```
  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, which is 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 have to do to use *angr* to crack one of the applications that 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 the lab03, however, the purpose is learning *angr*).

Suppose you have statically analyzed 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 disassmbled code if you want to copy the memory address value.

After having read the disassembled or decompiled code or navigated with the arrow keys 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 enters:

→

Then, execute the following instructions in a 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 to 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

```
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 next 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>
```

This string reports that the simulation Manager has symbolically executed the `crackme2` binaries. For nine of the explored branches, it has exited from the program 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 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 is visible by the the fact that the Simulation Manager has two active branches.

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

Try to give a meaning to the two active states the Simulation Manager is using. Can you guess something?

→

1.5.2 Starting the simulation from any state

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

A very simple (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` has not been yet 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 found).

```
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 changes their symbolic state, etc.)

1.5.3 Pruning unwanted execution branches

The symbolic execution may require a lot of time and computation also with small non-trivial examples. If you are already sure that you can exclude some branches (for instance, because you already know from the manual inspection of the static disassembled code), you better do it with 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 setup the exploration that excludes the branch where the input password is not the correct one:

→

If you did it correctly, the output will 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 the immediately reach an exit/return and become deadended). Thus you cannot appreciate how important 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, in some not-so-complex binaries you can easily reduce the simulation time from months to minutes.

1.5.4 Conditional execution with PIE binaries

In this second example, the loading of the `crackme3` application memory layout is randomized, as you can check with the *angr* loader (or using rabin2). Moreover, *angr* will inform 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 with a disassmbler, e.g. radare2, the offset of the branch to reach. 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 "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

Image now you have analyzed 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 has been triggered and you do not necessarily want to execute the entire application.

You can use *angr* to symbolically execute the application until the desired output is produced. Instead of passing the address, we can take advantage from 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 again the `crackme2` and `crackme3` programs 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 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 to follow the tutorial in the official *angr* documentation:

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

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

In the meanwhile, let’s try with a simple example that shows the importance of mastering Claripy.

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

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

Consider thus 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 to first 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 to 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 to 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 to 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 that you can solve using *angr* to practice with the basic features of this powerful tool.

Exercise 1

Purpose:	you have to 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, avoid the other one.

As a first exercise, you have to 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 to this purpose, *angr* can guess the password if you pass Claripy the proper 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 the 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:	again Claripy but a bit more complex. Use the template provided which will guide you through the solution. Remember to store the in the memory of the initial state the addresses of the four inputs
Source:	PSU angr CTF ()https://angr.oregonctf.org/solve/)

In this case, you need to model the registers to solve this exercise. All the considerations from the previous exercise also hold in this case. You only have to store the in the memory of the initial state the addresses 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 a lot of details, give suggestions, and guide you through the solution.
Source:	PSU angr CTF (https://angr.oregonctf.org/solve/)

In this case, the default behaviour of Claripy is not enough to solve this challenge. You need to have a look at the name of the checking function and explain Claripy how to properly manage the function names.

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 in order 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

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 other ones that expose methods that build chains to perform high-level operations.

https://github.com/angr/angrop

If you want to learn how 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 doing 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.

3.1 Installing development parts and add-ons

ATTENTION

We have experienced problems with IPython autocompletion after the installation of `angr-dev` within the virtual environment. If you are learning *angr* the autocompletion is very important, as the documentation and the features of *angr* are vast. If you experience this issue, use the *angr* docker that can be obtained with:

```
docker build -t angr - < angr-dev/Dockerfile
docker run -it angr
```

The *angr* framework provides additional utilities to process the data it can extract from the analysis. To this purpose, you have to install the development first with:

```
git clone https://github.com/angr/angr-dev
cd angr-dev
echo "deb http://deb.debian.org/debian/ sid main" >> /etc/apt/sources.list
sudo apt update
sudo apt install openjdk-8-jdk
sudo apt install libgcc1:i386
sudo apt autoremove
./setup.sh -i -e angr
```

In case your system will continue to complain about the missing `libgcc1:i386` package (which has been substituted by `libgcc1:i386`), you can simply delete it in the `setup.sh` file.

Then, you can install the utilities:

```
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
```

After this installation you may have to reconfigure the virtual environments with

```
source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
```

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

3.1.1 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 aviailable 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

3.1.2 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
```

In order to perform the required buffer overflow attacks, the objective is finding 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 a 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
```

3.2 Additional resources

A more comprehensive and detailed tutorial is available on an external resource. It is divided in 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/

A paper is also presenting 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