



Tutorial session

Abstract

The goal of this tutorial session is to give you a first "hands-on experience" with the BINSEC platform [2]. We are going to exercise with the symbolic execution engine to solve a small reverse-engineering challenge, then, get an idea on how to extend the BINSEC capabilities.

Keywords: GTMFS 2025, reverse engineering, tests, symbolic execution



Frédéric Recoules

SETUP

There are several ways to install BINSEC depending on your operating system and your preferences.

If you are familiar with the OCaml language, you can install BINSEC directly with `opam`.

```
$ opam install --yes curses bitwuzla-cxx unisim_archisec binsec
```

In other cases, we recommend using docker with the provided `Dockerfile`.

```
$ docker build -t binsec/gtmfs:2025 .
```

You will then need to copy the other file in the docker container.

```
$ docker run --name binsec-gtmfs2025 -it binsec/gtmfs:2025
```

On another terminal, you can then do the following.

```
$ docker cp dune-project binsec-gtmfs2025:/home/binsec/  
$ docker cp dune binsec-gtmfs2025:/home/binsec/  
$ docker cp test.ml binsec-gtmfs2025:/home/binsec/  
$ docker cp riddle binsec-gtmfs2025:/home/binsec/  
$ docker cp riddle_arm64 binsec-gtmfs2025:/home/binsec/
```

You can leave or enter the docker at any time.

- from the container

```
$ exit
```

- to restart the container

```
$ docker start binsec-gtmfs2025
```

- to go into the container

```
$ docker attach binsec-gtmfs2025
```

The source code files `dune-project`, `dune` and `test.ml` define a BINSEC plugin. To install it, run the following.

```
$ dune build @install && dune install
```

We should now be ready to start this lab.

We will exercise the Symbolic Execution concept over the small *CTF* puzzle named `riddle`.

(Basic) reverse-engineering What are we looking for? The *Capture The Flag* games can take the form of a reverse-engineering challenges to find an input (password, etc.) that will reach a special state of the program, such as leading to an exploit, a crash or simply printing a victory message. The puzzle `riddle` is one of them.

At first, the command `file` can give some useful information:

```
$ file riddle
riddle: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=898ec83a699452ca99f534bb6b3d64950e930546, for GNU/Linux 3.2.0,
not stripped
```

This puzzle consists of a single ELF x86-64bit executable that contains dynamic function calls.

At this point, we can simply run the executable to see what is happening – a good practice would be to run it in a virtual environment, but here, we swear it does not contain any malicious code.

Note. If you are working on an ARM processor, you can use the `riddle_arm64` to run the program.

```
$ ./riddle
Who is your host?
```

If we fail providing the expected answer, the program terminates with a `Nope!`.

Using the command `strings` shows the human-readable strings of the program.

```
$ strings riddle
[..]
Who is your host?
Nope!
Not so far!
Come on!
Congratulation!
[..]
```

It looks like we want the program to print the `Congratulation!` message.

We could of course try to reverse engineer all the code by hand, but the challenge has been obfuscated with `Tigress` [3] (we provided the obfuscated source code `challenge_obf.c` for reference) and the task is going to be a tough job.

We will use the `BINSEC` platform instead.

1. GETTING STARTED WITH SYMBOLIC EXECUTION

The easiest way to launch the BINSEC symbolic execution is to write a script file. Let us create our first script – arbitrarily called `crackme.ini`, together.

```
load sections .text, .rodata, .data from file
starting from <main>
with concrete stack pointer

replace <fgets@plt> (buf, size, _) by
  for i<64> in 0 to size - 1 do
    @[buf + i] := stdin[i]
  end
  return size
end

replace <puts@plt> (_) by
  return
end

reach <puts@plt> (str) such that @[str, 15] = "Congratulation!"
  then print c string stdin

halt at <exit@plt>
```

The script is a sequence of commands that define both the initial state of the analysis and its goals.

Initial state. Here, BINSEC will initialize the content of the initial memory with the data from the sections `.text` (the executable instructions), `.rodata` (the constants, especially the program strings) and `.data` (the global variables). BINSEC will then choose an arbitrary value for the initial stack pointer and run the execution from the entry of the `main` function.

Writing a function mock. Software interacts with the environment and often depends on several libraries.

By default, BINSEC does not model the environment of the program. Thus, we need to provide the semantics of the functions `fgets` and `puts`. We can do so using the BINSEC intermediate language, called DBA (*Dynamic Bitvector Automata*).

Here, we keep it simple: we skip the `puts` function while the `fgets` function copies bytes from the symbolic array `stdin` to the input buffer.

Note. The `@plt` refers to the Procedure Linkage Table. This table contains the address of the external functions that the program calls.

Target goal. Here, we instruct BINSEC to try to *reach* the function `puts` such that the string that is going to be printed starts with `Congratulation!`. On success, BINSEC will print the content of the symbolic array `stdin` as an ASCII string. We also mark the `exit` function as the end of the execution path.

Now, let us run BINSEC with the following.

```
1 $ binsec -sse -sse-script crackme.ini -sse-depth 100000 riddle
2 [sse:info] Load section .data (0x0000000000004000, 0x1198)
3 [sse:info] Load section .rodata (0x0000000000002000, 0x652)
4 [sse:info] Load section .text (0x00000000000010a0, 0x769)
5 [sse:result] Path 21 reached address 0x00001070 (<puts@plt>) (0 to go)
6 [sse:result] C string stdin : "Village Club Mil&ade\n"
7 [sse:info] SMT queries
8     Preprocessing simplifications
9         total          10503
10        true           513
11        false          5423
12        constant enum  4567
13
14        Satisfiability queries
15            total       22
16            sat         21
17            unsat       1
18            unknown     0
19            time        0.08
20            average     0.00
21
22        Exploration
23            total paths          21
24            completed/cut paths  0
25            pending paths       21
26            stale paths         0
27            failed assertions    0
28            branching points     10523
29            max path depth       65457
30            visited instructions (unrolled) 65457
31            visited instructions (static)  408
```

It seems that BINSEC found the solution. The lines 5 and 6 contains the result of our request. *BINSEC also outputs some statistics about the exploration (number of instruction executed, number of paths, etc.) and the formula discharged to the SMT solver.*

Let us verify that the solution is correct.

```
$ ./riddle
Who is your host?
Village Club Mil&ade
Congratulation!
```

Not that bad! But, can we use BINSEC for another goal than just finding the solution? Of course! In the next section, we will see how to use BINSEC to generate test cases for this challenge.

Starting from a core dump. Still, before moving on to the next topic, there exists a way to simplify the definition of initial state for the BINSEC configuration. The idea is as follows. Instead of an empty, fully symbolic state, we can instruct BINSEC to start from a fully concrete initial state from a real run. To do this, we can use `gdb` to dump the state of the program at the `main` entry.

For now, this approach is only supported for `x86` code.

To do so, we can use the script `make_coredump.sh` (installed with BINSEC).

```
$ make_coredump.sh snapshot riddle
```

Then, we can update the script as follows.

```
starting from core

replace <fgets> (buf, size, _) by
  for i<64> in 0 to size - 1 do
    @[buf + i] := stdin[i]
  end
  return size
end

replace <puts> (_) by
  return
end

reach <puts> (str) such that @[str, 15] = "Congratulation!"
  then print c string stdin

halt at <exit>
```

In short, we can replace all initializations by `starting from core`. We can also remove the `@plt` attribute because the dump contains the actual body of the external functions (resolved by the dynamic linker).

```
1 $ binsec -sse -sse-script crackme.ini -sse-depth 100000 snapshot
```

2. GENERATE TEST CASES

The symbolic execution can be used to resolve reachability queries. It is interesting when you know what you are looking for (e.g. outputting `Congratulation!`).

Yet, it is not always the case and we may require a *comprehensive exploration*.

To this end, we can use the `explore all` command. For instance, we can update the script as follows.

```

load sections .text, .rodata, .data from file
starting from <main>
with concrete stack pointer

replace <fgets@plt> (buf, size, _) by
  for i<64> in 0 to size - 1 do
    @[buf + i] := stdin[i]
  end
  return size
end

replace <puts@plt> (__) by
  return
end

hook <exit@plt> (__) with
  print c string stdin
  halt
end

explore all

```

When running BINSEC, it now returns a collection of inputs that exercise the different behavior of the program.

```

$ binsec -sse -sse-script crackme2.ini -sse-depth 100000 riddle
[sse:info] Load section .data (0x0000000000004000, 0x1198)
[sse:info] Load section .rodata (0x0000000000002000, 0x652)
[sse:info] Load section .text (0x00000000000010a0, 0x769)
[sse:result] C string stdin : "Village Club Mil&ade"
[sse:result] C string stdin : "Village Club Mil&ad"
[sse:result] C string stdin : "Village Club Mil&a"
[sse:result] C string stdin : "Village Club Mil&"
[sse:result] C string stdin : "Village Club Mil"
[sse:result] C string stdin : "Village Club Mi"
[sse:result] C string stdin : "Village Club M"
[sse:result] C string stdin : "Village Club "
[sse:result] C string stdin : "Village Club"
[sse:result] C string stdin : "Village Clu"
[sse:result] C string stdin : "Village Cl"
[sse:result] C string stdin : "Village C"
[sse:result] C string stdin : "Village "
[sse:result] C string stdin : "Village"
[sse:result] C string stdin : "Villag"
[sse:result] C string stdin : "Villa"
[sse:result] C string stdin : "Vill"
[sse:result] C string stdin : "Vil"
[sse:result] C string stdin : "Vi"

```

```

[sse:result] C string stdin : "V"
[sse:result] C string stdin : ""
[sse:info] Empty path worklist: halting ...
[sse:info] SMT queries
      Preprocessing simplifications
      total          41783
      true           513
      false          21062
      constant enum  20208

      Satisfiability queries
      total          21
      sat            20
      unsat          1
      unknown        0
      time           0.07
      average        0.00

      Exploration
      total paths                21
      completed/cut paths        21
      pending paths               0
      stale paths                0
      failed assertions           0
      branching points            41804
      max path depth              65459
      visited instructions (unrolled) 273459
      visited instructions (static)  410

```

Yet, the definition of a behavior may not be the one we want. Here, the BINSEC definition is the fact that the program takes different branches during the execution. Yet, from a test point of view, we may want to define the test cases as the different outputs on the terminal.

This is where the plugin capabilities of BINSEC come into play. The symbolic execution engine of BINSEC can be extended to add some instrumentations along the execution trace and record some data. A plugin can also add new features and extend the script language.

As an example for this lab, we provide a small plugin that generate test cases for the `cram` framework on top of the exploration of BINSEC.


```

load sections .text, .rodata, .data from file
starting from <main>
with concrete stack pointer

replace <fgets@plt> (buf, size, _) by
    gets(buf, size)
    return size
end

replace <puts@plt> (str) by
    puts(str)
    return
end

replace <exit@plt> (__) by
    exit()
end

explore all

```

The plugin defines new *builtins* (wisely named `gets`, `puts` and `exit`) that we can use to mock the external functions.

We can run it with the following command.

```

$ binsec -sse -sse-script crackme.ini -sse-depth 100000 riddle \
    -cram-test -cram-test-output run.t

```

The file `run.t` now contains the different outputs of the `riddle` program.

TO GO FURTHER

The BINSEC website (<https://binsec.github.io/>) is the place to look at to gather information about the academic activities of the team and the latest development of the platform (including other tutorials and reference documents).

You can also explore other reverse-engineering challenges from the different editions of the France CyberSecurity Challenge [1]. You can find a good introduction tutorial to the reverse-engineering here [5]. The community of Root Me [4] also proposes a large selection of challenges of increasing difficulties.

REFERENCES

- [1] ANSSI. Hackropole. <https://hackropole.fr/fr/>, 2024.
- [2] CEA. BINSEC. <https://binsec.github.io/>, 2024.
- [3] Christian Collberg. Tigress. <https://tigress.wtf/introduction.html>, 2024.
- [4] Root Me. Root Me. <https://www.root-me.org/>, 2024.
- [5] Reverse ZIP. Introduction au reverse. <https://reverse.zip/categories/introduction-au-reverse/>, 2024.