microsurf

Nicolas Dutly

Mar 19, 2022

CONTENTS:

CHAPTER ONE

INSTALLATION

tentative ! eventually the package should be published to pypi at release. The repository will probably have to be renamed too.

1.1 Requirements

Microsurf has been tested on python 3.9 and python 3.6. It might work on other python version. If you want to install the package in a virtual environment, you will need a tool that allows you to do so:

```
pip3 install virtualenv
```

1.2 Installing microsurf

1. Clone the repository:

```
git clone https://github.com/Jumpst3r/msc-thesis-work.git
cd msc-thesis-work
```

2. Create a virtual environment (optional, highly recommended)

```
virtualenv env
source env/bin/activate
```

3. Install the microsurf package:

```
pip install -e .
```

Note: This installs the framework in *editable* mode, meaning you can edit the source code without having to reinstall it after making changes.

CHAPTER

TWO

QUICKSTART

Note: Make sure you *installed* the framework if you wish to follow along.

In the general case, testing a target binary for side channels requires a number of items which are highly dependent on the binary, these include:

- The arguments to be passed to the binary
- Which input is considered to be secret
- How to generate secret inputs

As a user of the `microsurf` library, you have to specify these items. Fortunately, doing so is straightforward. The `SCDetector` is used to specify these items:

`class microsurf.SCDetector`(*binPath*, *args*, *randGen*, *deterministic*, *asFile*, *leakageModel*, *sharedObjects=[ ]*, *jail=None*)

The SCDetector class can be used to detect secret dependent memory accesses in generic applications

Parameters

- **binPath** (`str`) – Path to the target binary
- **args** (`list[str]`) – List of arguments to pass to the binary. For a secret argument, substitute the value of the argument with @ (for example, ['–encrypt', '<privkeyfile>'] would become ['–encrypt', '@']). Only one argument can be defined as secret.
- **randGen** (`Callable`[[], `str`]) – A function that generates random bytes in the format expected by the target binary. The SCDetector class will save these bytes to a temporary file and substitute the secret placeholder ('@') with the path to the file
- **deterministic** (`bool`) – Force deterministic execution by hooking relevant syscalls
- **asFile** (`bool`) – Specifies whether the target binary excepts the secret to be read from a file. If false, the secret will be passed directly as an argument
- **jail** (`Optional`[`str`]) – Specifies the a directory to which the binary will be jailed during emulation. For dynamic binaries, the user must ensure that the appropriate shared objects are present. Optional for static binaries, defaults to a tmp directory.
- **leakageModel** (`Callable`[[`str`], `Any`]) – (Callable[[str], Any]): Function which applies a leakage model to the secret. Example under microsurf.pipeline.LeakageModels
- **sharedObjects** (`list[str]`) – List of shared libraries to trace. For example ['libssl.so.1.1', 'libcrypto.so.1.1']. Defaults to None, tracing only the target binary. Only applicable to dynamic binaries.

exec(*report=False*)
Executes the side channel analysis (secret dependent memory accesses).

Parameters **report** – Generates a markdown report. Defaults to False.

Returns A list of leak locations, as integers (IP values). For dynamic objects, the offsets are reported.

2.1 Example: OpenSSL Camellia-128

Let us consider a practical example: Imagine that we want to test for side channels in openssl's Camellia-128 encryption algorithm on `x86-32`. Furthermore, let us assume that the binary is dynamically compiled.

What follows is a closer explanation of the non-trivial arguments that are needed by the `SCDetector` class.

A full working example can be found here.

2.1.1 Emulation root directory (`jail` argument)

For dynamic binaries you will have to provide a root directory (the `jail` argument), in which the binary will be emulated. The structure of the directory might look as follows:

```
jail-openssl-x8632/
  lib/
    ld-linux.so.2
    libc.so.6
    libcrypto.so.1.1
    libssl.so.1.1
  input.bin
  openssl
```

Note: As a user of the framework, you have to ensure that all required shared objects are present in the correct location. Also create make sure that any input files your binary expects are present. For dynamic x86 binaries you can check which shared objects are expected to be where by running `ldd mybinary.bin`.

```
jailroot = '/path/jail-openssl-x8632/'
scd = SCDetector(
        ...
        jail=jailroot,
        ...
    )
```

2.1.2 Specifying arguments (`args` argument)

If we want to encrypt a file `input.bin` using Camellia-128, we would run

```
openssl camellia-128-ecb -e -in input.bin -out output.bin -nosalt -K hexdata
```

where `hexdata` would be a 128bit hexadecimal key (for example: `96d496ea1378bf4f6e1f377606013e25`).

In the command above, the data we pass to the `-K` argument is secret. We can signal this to the microsurf framework by replacing the key with an '@' character:

```
opensslArgs = [
        "camellia-128-ecb",
        "-e",
        "-in",
        "input.bin",
        "-out",
        "output.bin",
        "-nosalt",
        "-K",
        "@",
    ]
scd = SCDetector(
        ...
        args=opensslArgs
        ...
    )
```

The '@' will be replaced with the data produced by the `randGen` function. If `isFile` is true, then the framework will assume that the target binary loads the secret from a file. In that case it will replace the '@' with the path to a temporary file, whose content is generated by the `randGen` function.

2.1.3 Producing secrets (`randGen` argument)

A secret often has to adhere to some specific format in order to be processed by the target binary. Since microsurf cannot guess that, it is the end user's job to specify such a function. In our example, the `-K` flag expects a 128bit key specified as a hex string. Since this is a fairly common requirement, it is already implemented in the `microsurf` framework. Other generators are also included :

`microsurf.utils.generators.`**`genRandInt`**`()`

Generates a random integer in [0,300). Useful for testing.

Return type `str`

Returns The string representation of the generated integer.

`microsurf.utils.generators.`**`getRandomHexKeyFunction`**`(`*keylen*`)`

get a generator which creates random hexadecimal keys of a given length, using the urandom module.

Returns A function which generates string representation of the created keys.

The `SCDetector` class will validate that the function generates different secrets at each call.

```
from microsurf.utils.generators import getRandomHexKeyFunction
scd = SCDetector(
        ...
```

(continues on next page)

(continued from previous page)

```
        randGen=getRandomHexKeyFunction(128),
        ...
    )
```

Note: The `randGen` parameter takes a **callable** object. The framework will validate whether it produces sufficiently random output when called.

2.1.4 Secret leakage model (`leakagemodel` argument)

To estimate how *much* information is leaked, the `SCDetector` class needs a leakage model.

A leakage model is a transformation which is applied to the secret. An example model is the hamming weight of the secret key:

`class microsurf.pipeline.LeakageModels.`**`hamming`**`(`*secret*`)`

Computes the hamming distance of the secret

Parameters **`secret`** (`str`) – A base 10 or base 16 string representation of the secret

Return type `ndarray`

Returns a numpy array containing the hamming distance of the secret.

Custom leakage models can be passed to the `SCDetector` object.

2.1.5 Selective tracing (`sharedObjects` argument)

To selectively trace shared objects, a list of names can be passed to the `sharedObjects` argument. Note that this only works for dynamic libraries. For example:

```
sharedObjects = ['libssl', 'libcrypto']
```

will ignore every other shared library (`libc` etc).

Note: The binary specified in `binPath` will always be traced, no need to pass it to the `sharedObjects` parameter.

2.2 Execution and results

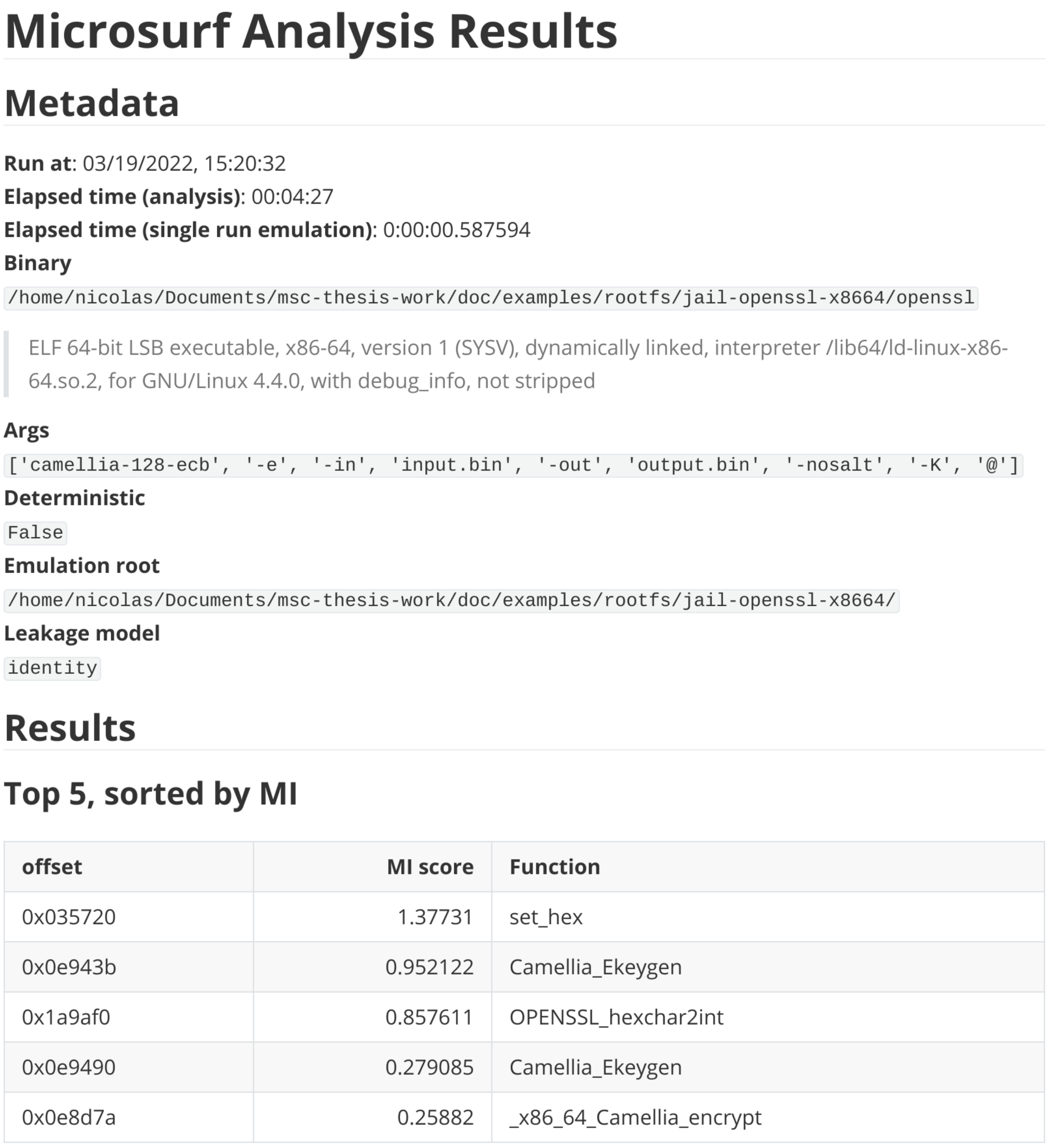
Once an `SCDetector` object has been initialized with all the arguments needed, analysis can be started by calling the `.exec()` function:

The results will look be split in several columns:

```
0003018c - [MI = 0.22]   at set_hex                     openssl
0013f5cf - [MI = 0.21]   at OPENSSL_hexchar2int         libcrypto.so.1.1
00095610 - [MI = 0.12]   at _x86_Camellia_encrypt       libcrypto.so.1.1
00095ef0 - [MI = 0.13]   at Camellia_Ekeygen            libcrypto.so.1.1
000955f5 - [MI = 0.11]   at _x86_Camellia_encrypt       libcrypto.so.1.1
```

The first column is the relative offset within the shared object at which the leak was identified. The second column gives the estimated mutual information score optained by applying the specified leakage model. The next column provides the function name (if the symbols are available). The last column gives the name of the (shared) object.

The `exec` function takes an optional argument `report`. If set to `True`, the results will be saved in markdown table format:



Microsurf Analysis Results

Metadata

Run at: 03/19/2022, 15:20:32
Elapsed time (analysis): 00:04:27
Elapsed time (single run emulation): 0:00:00.587594
Binary
`/home/nicolas/Documents/msc-thesis-work/doc/examples/rootfs/jail-openssl-x8664/openssl`

> ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 4.4.0, with debug_info, not stripped

Args
`['camellia-128-ecb', '-e', '-in', 'input.bin', '-out', 'output.bin', '-nosalt', '-K', '@']`
Deterministic
`False`
Emulation root
`/home/nicolas/Documents/msc-thesis-work/doc/examples/rootfs/jail-openssl-x8664/`
Leakage model
`identity`

Results

Top 5, sorted by MI

offset	MI score	Function
0x035720	1.37731	set_hex
0x0e943b	0.952122	Camellia_Ekeygen
0x1a9af0	0.857611	OPENSSL_hexchar2int
0x0e9490	0.279085	Camellia_Ekeygen
0x0e8d7a	0.25882	_x86_64_Camellia_encrypt

PYTHON MODULE INDEX

INDEX