microsurf

Nicolas Dutly

Mar 29, 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*, *sharedObjects=[]*, *jail=None*, *resultsDir='results'*)

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.
- **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.
- **resultsDir** (`str`) – Directory to which the markdown report will be saved, created if not not already existing.

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",
```

(continues on next page)

(continued from previous page)

```
        "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(
        ...
        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**

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

CHAPTER

THREE

ADVANCED USAGE

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

In the *Quickstart* section, we went over how to use the microsurf library in an end-to-end manner. This is very useful for the average user, as it only requires two basic steps:

1. Create a `SCDetector` object.
2. Call the `.exec()` function on the created object.

However, from a research perspective, it is sometimes required to be able to have fine-grained control on some aspects of the process. Fortunately, microsurf is a *framework*, and as such, provides you with the tools you need to build your own multi-arch, custom side channel detection pipeline.

To do so, you will still need to create a `SCDetector` object (refer to the *Quickstart* for more information). Once the object is created, you will have access to a number of lower level functions, which are documented below, along with the `SCDetector` constructor:

3.1 SCDetector module

class microsurf.**SCDetector**(*binPath*, *args*, *randGen*, *deterministic*, *asFile*, *sharedObjects=[]*, *jail=None*, *resultsDir='results'*)

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.

- **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.
- **resultsDir** (str) – Directory to which the markdown report will be saved, created if not not already existing.

exec(*report=False*)

Executes the complete side channel analysis pipeline with sensible defaults (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.

isDeterministic(*traceCollection*)

Determines whether the memory reads are deterministic given a MemTraceCollectionFixed object

Parameters **traceCollection** (*MemTraceCollectionFixed*) – A MemTraceCollectionFixed object with at least two traces.

Return type bool

Returns True or False, depending on whether the execution is deterministic.

recordTracesFixed(*n*, *pcList=None*, ***kwargs*)

Record memory accesses resulting from repeated execution with the same secret.

By default, it will target:

- For dynamic binaries only the shared objects which were passed to the SCDetector constructor
- For static binaries, the entire binary.
- Determinism will be fixed if the appropriate parameter was passed to the SCDetector constructor.

The last point can be modified by passing deterministic=True or deterministic=False

Parameters

- **n** (int) – Number of traces to collect
- **pcList** (Optional[List]) – Specifc PCs to trace. Defaults to None.

Return type *MemTraceCollectionFixed*

Returns A MemTraceCollectionFixed object representing the set of traces collected.

recordTracesRandom(*n*, *pcList=None*, ***kwargs*)

Record memory accesses resulting from repeated execution with random secrets.

By default, it will target:

- For dynamic binaries only the shared objects which were passed to the SCDetector constructor
- For static binaries, the entire binary.
- Determinism will be fixed if the appropriate parameter was passed to the SCDetector constructor.

The last point can be modified by passing deterministic=True or deterministic=False

Parameters

- **n** (int) – Number of traces to collect
- **pcList** (Optional[List]) – Specifc PCs to trace. Defaults to None.

Return type *MemTraceCollectionRandom*

Returns A MemTraceCollectionRandom object representing the set of traces collected.

3.2 Elf tools module

The microsurf framework also exposes a number of function which allow you to parse ELF files (using pyelftools under the hood). These can be useful to add context to detected side channels, if the binary is compiled with debug symbols (or not stripped).

microsurf.utils.elf.**getCodeSnippet**(*file*, *loc*)

Returns a list of source code lines, 3 before and 3 after the given offset for a given ELF file.

Note that only DWARF <= v4 is supported.

Parameters

- **file** (str) – Path to the ELF file
- **loc** (int) – Offset value

Return type List[str]

Returns source code lines, 3 before and three after

Raises **FileNotFoundError** – If the given ELF file does not exist

microsurf.utils.elf.**getfnname**(*file*, *loc*)

Returns the symbol of the function for a given PC

Parameters

- **file** (str) – Path to elf file (relative or absolute)
- **loc** (int) – PC value

Returns the symbol name of the function for a given PC if the symbols are not stripped, None otherwise.

Raises **FileNotFoundError** – If the given file does not exist.

3.3 Traces module

In microsurf, traces are represented as objects:

class microsurf.pipeline.tracetools.Trace.**MemTrace**(*secret*)

Represents a single Memory Trace object.

Initialized with a secret, trace items are then added by calling the .add(ip, memaddr) method.

Parameters

- **secret** – The secret that was used when the trace
- **recorded.** (*was*) –

add(*ip*, *memaddr*)

Adds an element to the current trace. Note that several target memory addresses can be added to the same PC by calling the function repeatedly.

Parameters

- **ip** – The instruction pointer / PC which caused
- **read** (*the memory*) –
- **memaddr** – The target address that was read.

remove(*keys*)
Removes a set of PCs from the given trace

Parameters **keys** (List[int]) – The set of PCs to remove

class microsurf.pipeline.tracetools.Trace.**MemTraceCollection**(*traces*)
A generic MemorTraceCollection object.

Parameters **traces** (list[*MemTrace*]) – The traces that make up the collection.

get(*indices*)
Returns all memory traces which contain the specified PCs

Parameters **indices** (List[int]) – List of PCs

Return type List[*MemTrace*]

Returns List of memory traces which contain the

specified PCs

remove(*indices*)
Remove a set of PCs/IPs from all traces in the collection.

Parameters **indices** (List[int]) – List of PCs to remove.

class microsurf.pipeline.tracetools.Trace.**MemTraceCollectionFixed**(*traces*)
Creates a Memory trace collection object. The secrets of the individual traces must be fixed.

Parameters **traces** (list[*MemTrace*]) – List of memory traces

class microsurf.pipeline.tracetools.Trace.**MemTraceCollectionRandom**(*traces*)
Creates a Memory trace collection object. The secrets of the individual traces must be random.

Parameters **traces** (list[*MemTrace*]) – List of memory traces

prune()
Automatically prunes the trace collection: Iterates pairwise over all traces, if they have differing secrets but the same list of memory accesses for a given PC, remove the PC from both traces.

Calling .prune() populates the field .possibleLeaks which contains: Every PC for which different secrets resulted in different memory accesses. Note that these may not automatically be directly secret dependent and may be due to inherent non-determinism.

Return type None

PYTHON MODULE INDEX

m

INDEX