microsurf

Nicolas Dutly

Aug 13, 2022

CONTENTS:

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

CHAPTER

ONE

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
```

CHAPTER

TWO

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

THREE

QUICKSTART

This page will walk you through the basics of using Microsurf with an applied example, testing for side channel vulnerabilities in OpenSSL's Camellia-128 implementation.

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 general workflow is as follows:

1. Create a new `BinaryLoader` instance, this will allow you to configure general settings relating to the target binary:

class `microsurf.`**`BinaryLoader`**(*path*, *args*, *rootfs*, *rndGen*, *x8664Extensions=['DEFAULT', 'AESNI', 'SSE', 'SSE2', 'SSE3', 'AVX']*, *sharedObjects=[]*, *deterministic=True*, *resultDir='results'*)

The BinaryLoader class is used to tell the framwork where to located the target binary, shared libraries and to specify emulation and general execution settings.

Parameters

- **path** (`Path`) – The path to the target executable (ELF-linux format, ARM/MIPS/X86/RISCV).
- **args** (`List[str]`) – List of arguments to pass, '@' may be used to mark one argument as secret.
- **rootfs** (`str`) – The emulation root directory. Has to contain expected shared objects for dynamic binaries.
- **rndGen** (*`SecretGenerator`*) – The function which will be called to generate secret inputs.
- **x8664Extensions** (`List[str]`) – List of x86 features, ["DEFAULT"] for all (supported) extensions. Must be subset of: ["DEFAULT", "AESNI", "NONE"]
- **sharedObjects** (`List[str]`) – List of shared objects to trace, defaults to tracing everything. Include binary name to also trace the binary.
- **deterministic** (`bool`) – Force deterministic execution.

- **resultDir** (str) – Path to the results directory.

Most optional arguments can be left as is, we'll dive into more details later on. Let's see how you would use the `BinaryLoader` class if you would like to test the OpenSSL implementation of *Camellia-128*.

Note: A full working example can be found here.

3.1 Emulation root directory (`rootfs` argument)

For dynamic binaries you will have to provide a root directory (the `rootfs` 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`.

Hint: If a particular shared library is not found in the emulation root, microsurf will issue a warning with the name of the concerned library.

3.2 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",
```

(continues on next page)

(continued from previous page)

```
        "-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.

You can also mark partial arrguments as secret dependent, for example in the mbedTLS aes driver program, the expected arguments are:

```
./crypt_and_hash 0 input.bin output.bin AES-128-ECB SHA512␣
↪hex:5e1defa4a22621eca5ab3ec051feb3a8
```

In that case, the argument list to pass to microsurf would be:

```
args = [
        "0",
        "input.bin",
        "output.bin",
        "AES-128-ECB",
        "SHA512",
        "hex:@",
    ]
```

3.3 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:

`class microsurf.utils.generators.`**`hex_key_generator`**`(`*keylen*`)`

Generates a hexadecimal secret string. Not saved to file (directly substituted in the argument list).

Args: keylen: The length of the key in bits.

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

A list of secret generators is given *here*, along with a guide on how to write your own generators.

3.4 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). Only canonical library names are needed, no need to pass the exact file name.

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

For OpenSSL, the BinaryLoader object would look as follows:

```
from microsurf.pipeline.Stages import BinaryLoader
from microsurf.utils.generators import getRandomHexKeyFunction

rootfs = 'path-to-rootfs'
binpath = rootfs + "openssl"

opensslArgs = [
    "camellia-128-ecb",
    "-e",
    "-in",
    "input.bin",
    "-out",
    "output.bin",
    "-nosalt",
    "-K",
    "@",

]
sharedObjects = ['libcrypto'] # only trace libcrypto.so

binLoader = BinaryLoader(
    path=binpath,
    args=opensslArgs,
    rootfs=rootfs,
    rndGen=getRandomHexKeyFunction(128),
    sharedObjects=sharedObjects
)
```

3.5 Specifying Detection Modules

There are currently two detection modules which can be used:

1. The secret dependent memory read detection *module*
2. The secret dependent control flow detection *module*

They both take a number of arguments - through most can be left to default values. For further information consult the pages dedicated to the two modules.

The only required argument is the `binaryLoader`, which is used to pass the previously created `BinaryLoader` object:

```
from microsurf.pipeline.DetectionModules import CFLeakDetector, DataLeakDetector

binLoader = BinaryLoader(...)
data_leak_detection = DataLeakDetector(binaryLoader=binLoader)
cf_leak_detection = CFLeakDetector(binaryLoader=binLoader)
```

3.6 Executing the analysis

Having created our required detection modules, we are now ready to execute the side channel detection pipeline. To do so, we can create a `SCDetector` object and pass along the list of detection modules:

```
from microsurf.pipeline.DetectionModules import CFLeakDetector, DataLeakDetector
from microsurf.pipeline.Stages import BinaryLoader
from microsurf import SCDetector

# Create the binary loader as described before
binLoader = BinaryLoader(...)
data_leak_detection = DataLeakDetector(binaryLoader=binLoader)
cf_leak_detection = CFLeakDetector(binaryLoader=binLoader)

scd = SCDetector(modules=[
        data_leak_detection,
        cf_leak_detection,
    ])

scd.exec()
```

This will search for any data and control flow side channels in the target application.

Note: Per default, the `SCDetector` will execute a quick analysis. Key bit dependency estimation is not performed by default. To

Warning: If you know that your target binary has non-deterministic behavior, pass `deterministic=True` when creating the `BinaryLoader` object. Not doing so might trigger false positives. For more common pitfalls, refer to the *FAQ*

By default a report will be created in the `reports` directory. If not present, it will be created. If you wish to continue working and processing results with python, you can access the underlying pandas dataframe like so:

```
result_dataframe = scd.DF
```

A detailed documentation for all high level modules can be found *here*.

CHAPTER

FOUR

SECRET DEPENDENT CONTROL FLOW

This page gives an introduction to secret dependent control flow, assuming no prior technical knowledge.

4.1 Introduction

Secret-dependent control-flow is a possible source of variable execution time. This can be exploited through timing attacks. Note that not all secret dependent control flow operation lead to exploitable timing differences: This depends on the resolution available to the attacker.

Secret dependent control flow operations should be eliminated from high-level code, as timing attacks are practical [1,2] and can compromise cryptographic material in a remote setting.

Classical examples of secret dependent control flow include (but are not limited to):

- Secret dependent branching (for example the conditional subtraction in a naive Montgomery modular multiplication)
- Secret dependent loop iterations: These have also been shown to produce secret dependent execution timing. Brumley et al. [2] discovered a secret dependent iteration count in the Montgomery ladder primitive used to perform scalar multiplication in OpenSSL `0.9.8o`.

Warning: secret dependent control flow has also been shown to be caused by certain compiler *optimizations*.

4.2 Secret Dependent Control Flow Detection

In the *Microsurf* framework, secret dependent control flow detection can be implemented with the `CFLeakDetector` module. Passing this module to the detectio pipeline will flag any observed secret dependent control flow operations:

```
scd = SCDetector(modules=[
        DataLeakDetector(binaryLoader=binLoader),
    ]
    )
scd.exec()
```

4.3 The `CFLeakDetector` module

The exact documentation for the `CFLeakDetector` module is given bellow.

`class microsurf.`**`CFLeakDetector`**`(`*, *binaryLoader*, *miThreshold=0.2*, *flagVariableHitCount=False*`)`

The CFLeakDetector class is used to collect traces for analysis of secret dependent conctrol flow.

Parameters

- **`binaryLoader`** (*`BinaryLoader`*) – A BinaryLoader instance
- **`miThreshold`** (`float`) – The treshold for which to produce key bit estimates. Values lower than 0.2 might produce results which do not make any sense (overfitted estimation).
- **`flagVariableHitCount`** (`bool`) – Include branching instruction which were hit a variable number of times in the report. Doing so will catch things like secret dependent iteration counts but might also cause false positives. Usually these are caused by a secret dependent branch earlier in the control flow, which causes variable hit rates for subsequent branching instructions. Fixing any secret dependent branching and then running with flagVariableHitCount=True is advised.

4.4 References

[1] Brumley, D. and Boneh, D., 2005. Remote timing attacks are practical. Computer Networks, 48(5), pp.701-716.

[2] Brumley, B.B. and Tuveri, N., 2011, September. Remote timing attacks are still practical. In European Symposium on Research in Computer Security (pp. 355-371). Springer, Berlin, Heidelberg.

CHAPTER

FIVE

SECRET DEPENDENT MEMORY ACCESSES

This page gives an introduction to secret dependent memory accesses, assuming no prior technical knowledge.

5.1 Introduction

Secret dependent memory accesses can lead to side channels which can be used to recover secrets. Commonly, these attacks are possible because reading data from cache is faster than reading data from memory. Past research [1,2] has shown that cache attacks are practical and can be used to compromise cryptographic material.

A classical example of a secret dependent memory access is using the secret (or parts of it) to index a table:

```
int SBOX[256] = {...};
int main(int argc, char **argv){
    int secret = atoi(argv[1]);
    int sub = SBOX[secret];
    return sub
}
```

In the given example, a secret integer is read from a user input and substituted using a table. To retrieve the substited values, the secret is used as an index. While this looks like a toy example, it is important to note that many ciphers (such as AES) are substitution based. Many frameworks revert back to a table-based, leaking implementation on non-mainline architectures. Even on `x86`, some frameworks may choose to leverage a table-based implementation in the absence of crypto or SIMD extensions.

5.2 Secret Dependent Memory Access Detection

In the *Microsurf* framework, secret dependent memory access detection can be implemented with the `DataLeakDetector` module. Passing this module to the detectio pipeline will flag any observed secret dependent memory accesses:

```
scd = SCDetector(modules=[
        DataLeakDetector(binaryLoader=binLoader),
    ]
    )
scd.exec()
```

5.3 The `DataLeakDetector` module

The exact documentation for the `DataLeakDetector` module is given bellow.

`class microsurf.`**`DataLeakDetector`**`(`*binaryLoader*, *miThreshold=0.2*, *granularity=1*`)`

The DataLeakDetector class is used to collect traces for analysis of secret dependent memory accesses.

Parameters

- **`binaryLoader`** (*`BinaryLoader`*) – A BinaryLoader instance.
- **`miThreshold`** (`float`) – The treshold for which to produce key bit estimates (if key bit estimates are requested). Values lower than 0.2 might produce results which do not make any sense (overfitted estimation).
- **`granularity`** (`int`) – Resultion of the detection algorithm (in bytes). The default value of one flags any memory accesses which differ by at least one byte. This value can be increased to simlulate detection of cross-cache line leaks.

5.4 References

[1] Liu, F., Yarom, Y., Ge, Q., Heiser, G. and Lee, R.B., 2015, May. Last-level cache side-channel attacks are practical. In 2015 IEEE symposium on security and privacy (pp. 605-622). IEEE.

[2] Yarom, Y., Genkin, D. and Heninger, N., 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. Journal of Cryptographic Engineering, 7(2), pp.99-112.

CHAPTER

SIX

SECRET GENERATORS

This page describes how secrets are generated for use with the Microsurf framework.

6.1 Introduction

Cryptographic frameworks expect secrets in various different formats. Sometimes they can be directly passed has hexadecimal strings in a command line argument list. Sometimes a utility will expect a path to a file containing the secret in a binary format.

Microsurf is designed to accomodate both cases. Secret generators are classes which produce a specific secret, either to be included directly in a list of command line arguments or to be written to a file. The path to the file will then be passed as a command line arguments.

6.2 Existing Generators

A number of common cases are covered with predefined secret generators. This section documents the classes.

class `microsurf.utils.generators.`**`RSAPrivKeyGenerator`**(*keylen*)

Generates RSA privat keys with PEM encoding, PKCS8 and no encryption. The key are written to disk (*/tmp/microsurf_key_gen_**.key*).

Args: keylen: The length of the private key in bits.

class `microsurf.utils.generators.`**`DSAPrivateKeyGenerator`**(*keylen*)

Generates DSA privat keys with PEM encoding, PKCS8 and no encryption. The key are written to disk (*/tmp/microsurf_key_gen_**.key*).

Args: keylen: The length of the private key in bits.

class `microsurf.utils.generators.`**`DSAPrivateKeyGenerator`**(*keylen*)

Generates DSA privat keys with PEM encoding, PKCS8 and no encryption. The key are written to disk (*/tmp/microsurf_key_gen_**.key*).

Args: keylen: The length of the private key in bits.

class `microsurf.utils.generators.`**`ECDSAPrivateKeyGenerator`**(*keylen*)

Generates ECDSA privat keys with PEM encoding, PKCS8 and no encryption (SECP256K1). The key are written to disk (*/tmp/microsurf_key_gen_**.key*).

Args: keylen: The length of the private key in bits.

class `microsurf.utils.generators.`**`hex_key_generator`**(*keylen*)

Generates a hexadecimal secret string. Not saved to file (directly substituted in the argument list).

Args: keylen: The length of the key in bits.

`class microsurf.utils.generators.hex_file(keylen)`

Generates a binary file. Good for use when evaluating constant time proprieties of hashing functions.

Args: keylen: The length of the file in bits.

6.3 Writing your own secret generator

What if the primitive you wish to evaluate expects a different format ? Worry not, as you can implement your own secret generator to acomodate your needs.

`class microsurf.utils.generators.SecretGenerator(keylen, asFile)`

Template class used to implement custom secret generators.

Parameters

- **keylen** (int) – Length of the key.
- **asFile** (bool) – Whether the class implements an on-disk generator.

`__call__(*args, **kwargs)`

The __call__ function defines the behavior implemented when calling the secret generator. This function must create a fresh secret every time it is called

Returns path to file for on-disk secret or encoded secret.

Return type A string representation of the secret

`getSecret()`

Returns a numerical representation of the secret. The key-bit dependency analysis will be performed on the return value of this function.

Return type int

In order to write a custom secret generator you must choose one of two operational modes:

1. Directly including the encoded secret as part of the list of arguments.
2. Saving the secret to a file and passing the path to generated secrets to the list of arguments.

This choice is reflected in the value `asFile`.

An example of an on-disk RSA key generator is given below:

```
class RSAPrivKeyGenerator(SecretGenerator):
    """
    Generates RSA privat keys with PEM encoding, PKCS8 and no encryption. The key are␣
↪written to disk (`/tmp/microsurf_key_gen_**.key`).

        Args:
            keylen: The length of the private key in bits.
    """
    def __init__(self, keylen:int):
        # we pass asFile=True because our secrets are loaded from disk (RSA priv key)
        super().__init__(keylen, asFile=True)

    def __call__(self, *args, **kwargs):
        self.pkey = rsa.generate_private_key(
            public_exponent=65537,
```

(continues on next page)

(continued from previous page)

```
            key_size=self.keylen
        )
        kbytes = self.pkey.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )
        tempfile.tempdir = '/tmp'
        keyfile = tempfile.NamedTemporaryFile(prefix="microsurf_key_gen", suffix=".key").
↪name
        with open(keyfile, 'wb') as f:
            f.write(kbytes)
        return keyfile

    def getSecret(self) -> int:
        return self.pkey.private_numbers().p
```

Hint: The numerical result of the getSecret method will be used to estimated key bit dependencies. This can be used to analyze selective leakages on parts of the secret (say a single coefficient in RSA) or to force a custom leakage model by returning a masked version of the secret.

6.3.1 Using the secret generator

The secret generator has to be passed to the `BinaryLoader` as an argument:

```
binLoader = BinaryLoader(
        path=binpath,
        args=opensslArgs,
        rootfs=jailroot,
        rndGen=RSAPrivKeyGenerator(2048),
        sharedObjects=sharedObjects,
    )
```

Note that a keysize has to be passed. The resulting object is a then called during emulation to create different secrets.

CHAPTER

SEVEN

COMPILER-INDUCED CONSTANT TIME VIOLATIONS

7.1 Introduction

Mainstream compiler were not build to enforce constant time proprieties, they are meant to provide the best possible performance. This is not always compatible with constant time proprieties.

To complicate things further: If a binary is formally proven to be leakage free with one compiler on one architecture, then this cannot be extrapolated to other architectures, compilers or even compiler versions.

For an extensive study on compiler-induced behavior, consult [1].

7.2 Examples

7.2.1 Function inlining

Inlining is an optimization which removes the overhead of creating and destroying function frames by embedding (small-ish) functions into the caller.

By doing so, the compiler may inadvertently introduce a secret dependent jump.

7.3 References

[1] Simon, L., Chisnall, D. and Anderson, R., 2018, April. What you get is what you C: Controlling side effects in mainstream C compilers. In 2018 IEEE European Symposium on Security and Privacy (EuroS&P) (pp. 1-15). IEEE.

CHAPTER

EIGHT

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:

8.1 SCDetector module

`class microsurf.SCDetector`(*modules*, *itercount=20*, *addrList=None*, *getAssembly=False*)

The SCDetector class is used to perform side channel detection analysis.

Parameters

- **`modules`** (`List[Detector]`) – List of detection modules to run.
- **`itercount`** (`int`) – Number of traces per module to collect when estimating key bit dependencies.
- **`addrList`** (`Optional[List[int]]`) – List of addresses for which to perform detailed key bit dependency estimates. If None, no estimates will be performed. If an empty list is passed, estimates will be generated for all leaks. To selectively perform estimates on given leaks, pass a list of runtime addresses as integers. The runtime addresses can be taken from the generated reports (Run first with addrList=None and then run a second time on addresses of interest as found in the report.)

`exec()`

Perform the side channel analysis using the provided modules, saving the results to ‘results’.

8.2 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

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.

8.3 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*, *possibleLeaks=None*, *granularity=1*)

Creates a Memory trace collection object. The secrets of the individual traces must be random.

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

`buildDataFrames``()`

Build a dictionary of dataframes, indexed by leak adress. T[leakAddr] = df with rows indexing the executions and columns the addresses accessed. The first column contains the secret.

`class microsurf.pipeline.tracetools.Trace.`**`PCTrace`**(*secret*)

Represents a single Program Counter (PC) Trace object.

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

Parameters

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

`add`(*range*)

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 `range` – the start / end PC of the instruction block (as a tuple)

`class microsurf.pipeline.tracetools.Trace.`**`PCTraceCollection`**(*traces*, *possibleLeaks=None*, *flagVariableHitCount=False*)

Creates a PC trace collection object. The secrets of the individual traces must be random.

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

todo

CHAPTER

NINE

MODULE DOCUMENTATION

This page gives detailed documentation about the different modules.

9.1 General Modules

9.1.1 BinaryLoader

class microsurf.pipeline.Stages.**BinaryLoader**(*path*, *args*, *rootfs*, *rndGen*, *x8664Extensions=['DEFAULT', 'AESNI', 'SSE', 'SSE2', 'SSE3', 'AVX']*, *sharedObjects=[]*, *deterministic=True*, *resultDir='results'*)

The BinaryLoader class is used to tell the framwork where to located the target binary, shared libraries and to specify emulation and general execution settings.

Parameters

- **path** (Path) – The path to the target executable (ELF-linux format, ARM/MIPS/X86/RISCV).
- **args** (List[str]) – List of arguments to pass, '@' may be used to mark one argument as secret.
- **rootfs** (str) – The emulation root directory. Has to contain expected shared objects for dynamic binaries.
- **rndGen** (*SecretGenerator*) – The function which will be called to generate secret inputs.
- **x8664Extensions** (List[str]) – List of x86 features, ["DEFAULT"] for all (supported) extensions. Must be subset of: ["DEFAULT", "AESNI", "NONE"]
- **sharedObjects** (List[str]) – List of shared objects to trace, defaults to tracing everything. Include binary name to also trace the binary.
- **deterministic** (bool) – Force deterministic execution.
- **resultDir** (str) – Path to the results directory.

9.1.2 `SCDetector`

`class microsurf.`**`SCDetector`**`(`*modules*, *itercount=20*, *addrList=None*, *getAssembly=False*`)`

The SCDetector class is used to perform side channel detection analysis.

Parameters

- **`modules`** (`List[Detector]`) – List of detection modules to run.
- **`itercount`** (`int`) – Number of traces per module to collect when estimating key bit dependencies.
- **`addrList`** (`Optional[List[int]]`) – List of addresses for which to perform detailed key bit dependency estimates. If None, no estimates will be performed. If an empty list is passed, estimates will be generated for all leaks. To selectively perform estimates on given leaks, pass a list of runtime addresses as integers. The runtime addresses can be taken from the generated reports (Run first with addrList=None and then run a second time on addresses of interest as found in the report.)

9.2 Detection Modules

9.2.1 `DataLeakDetector`

`class microsurf.pipeline.DetectionModules.`**`DataLeakDetector`**`(`*binaryLoader*, *miThreshold=0.2*, *granularity=1*`)`

The DataLeakDetector class is used to collect traces for analysis of secret dependent memory accesses.

Parameters

- **`binaryLoader`** (*`BinaryLoader`*) – A BinaryLoader instance.
- **`miThreshold`** (`float`) – The treshold for which to produce key bit estimates (if key bit estimates are requested). Values lower than 0.2 might produce results which do not make any sense (overfitted estimation).
- **`granularity`** (`int`) – Resultion of the detection algorithm (in bytes). The default value of one flags any memory accesses which differ by at least one byte. This value can be increased to simlulate detection of cross-cache line leaks.

9.2.2 `CFLeakDetector`

`class microsurf.pipeline.DetectionModules.`**`CFLeakDetector`**`(`*, *binaryLoader*, *miThreshold=0.2*, *flagVariableHitCount=False*`)`

The CFLeakDetector class is used to collect traces for analysis of secret dependent conctrol flow.

Parameters

- **`binaryLoader`** (*`BinaryLoader`*) – A BinaryLoader instance
- **`miThreshold`** (`float`) – The treshold for which to produce key bit estimates. Values lower than 0.2 might produce results which do not make any sense (overfitted estimation).
- **`flagVariableHitCount`** (`bool`) – Include branching instruction which were hit a variable number of times in the report. Doing so will catch things like secret dependent iteration counts but might also cause false positives. Usually these are caused by a secret dependent branch earlier in the control flow, which causes variable hit rates for subsequent branching

instructions. Fixing any secret dependent branching and then running with flagVariableHitCount=True is advised.

9.3 Secret Generators

CHAPTER

TEN

FREQUENTLY ASKED QUESTIONS

This page covers common pitfalls that could be encountered during the usage of the microsurf library.

10.1 Emulation Errors

todo

10.2 Secret Dependent Memory Accesses

todo

10.3 Secret Dependent Control Flow

todo

PYTHON MODULE INDEX

INDEX