microsurf

Nicolas Dutly

Sep 02, 2022

CONTENTS:

1 Requirements **3**

2 Installing microsurf (from source) **5**

3 Quickstart **7**
- 3.1 Introduction . . . 7
- 3.2 Usage . . . 8
- 3.3 Emulation root directory (`rootfs` argument) . . . 8
- 3.4 Specifying arguments (`args` argument) . . . 9
- 3.5 Producing secrets (`rndGen` argument) . . . 10
- 3.6 Selective tracing (`sharedObjects` argument) . . . 10
- 3.7 Validating `BinaryLoader` Arguments . . . 11
- 3.8 Specifying Detection Modules . . . 11
- 3.9 Executing the analysis . . . 11

4 Advanced Usage (Key-Bit Dependency Estimation) **13**

5 Secret Generators **15**
- 5.1 Introduction . . . 15
- 5.2 Existing Generators . . . 15
- 5.3 Writing your own secret generator . . . 15

6 Secret Dependent Control Flow **19**
- 6.1 Introduction . . . 19
- 6.2 Secret Dependent Control Flow Detection . . . 19
- 6.3 The `CFLeakDetector` module . . . 20
- 6.4 References . . . 20

7 Secret Dependent Memory Accesses **21**
- 7.1 Introduction . . . 21
- 7.2 Secret Dependent Memory Access Detection . . . 21
- 7.3 The `DataLeakDetector` module . . . 22
- 7.4 References . . . 22

8 Compiler-induced Constant Time Violations **23**
- 8.1 Introduction . . . 23
- 8.2 Examples . . . 23
- 8.3 References . . . 23

9 Module Documentation **25**
- 9.1 General Modules . . . 25

9.2 Detection Modules 26
9.3 Secret Generators 26

10 Frequently Asked Questions **27**
10.1 Emulation Errors 27
10.2 Does the lack of reported leaks imply that my binary is safe ? 27

Index **29**

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 3.10. It might work on other python version, check your version with

```
python --version
```

If your python version differs, follow this guide to install the required 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
```

The framework has been tested with Ubuntu 22.04 LTS x86-64. It has not been tested on M1 (ARM) chips.

CHAPTER

TWO

INSTALLING MICROSURF (FROM SOURCE)

1. Acquire the repository:

If the code was acquired as a zip archive:

unzip it and navigates to the resulting directory (containing USAGE.pdf)

If access has been granted to repository or the repository has been made public:

```
git clone https://github.com/Jumpst3r/microsurf.git
cd microsurf
```

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

Using the default python version:

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

Using a custom interpreter path:

```
virtualenv --python=/usr/bin/python3.9  env
```

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.

3.1 Introduction

Microsurf is a framework for finding side-channels in compiled binaries. Similarly to other tools, it enables the detection of:

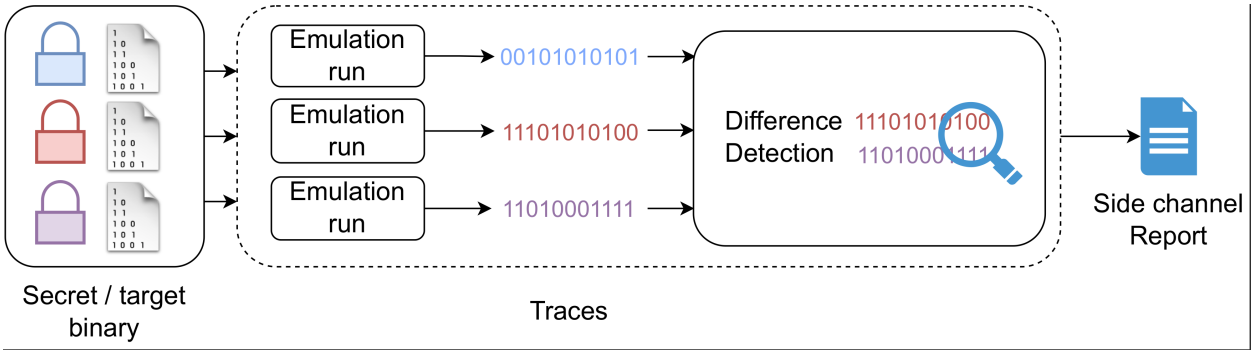
- Secret-dependent memory accesses.
- Secret-dependent control flow operations.

These are of importance when developing constant-time (or data-oblivious) code, since they are the source of leakages such as secret-dependent execution timing. Some flaws caused by such leakages may be exploited remotely.

The advantage of Microsurf is that it supports a wide set of target architectures (`x86`, `x86-64`, `arm`, `mips`, `powerPC`, `riscv`) without requiring manual secret annotations.

3.1.1 How it works

Microsurf works by emulating a target binary with different secrets:



During emulation, we gather traces which contain information on accessed memory locations and control flow. We then detect differences in traces recorded with different secrets to report secret-dependent operations.

We control possible sources of randomness during emulation to ensure that differences are indeed secret-dependent.

3.2 Usage

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:

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.

3.3 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 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.

If you have access to the source code, you can simply compile your binary using a pre-defined toolchain and use the included `sysroot` directory as an emulation root directory. Don't forget to move your target libraries to the corresponding directory.

Hint: For evaluation purposes, we provide a valid emulation root directory for OpenSSl under `x86-64` (`docs/rootfs/jail-openssl-1.1.1dev-x8664`).

3.4 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.

You can also mark partial arguments 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 could be:

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

3.5 Producing secrets (rndGen 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*)

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. Common use-cases such as RSA and ECDSA on-disk key generators are already included.

3.6 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 not be traced, unless the name is passed to the sharedObjects parameter.

For OpenSSL, the BinaryLoader object would look as follows:

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

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(
```

(continues on next page)

(continued from previous page)

```
    path=binpath,
    args=opensslArgs,
    rootfs=rootfs,
    rndGen=hex_key_generator(keylen=128, nbTraces=8),
    sharedObjects=sharedObjects
)
```

Hint: The `nbTraces` argument specifies how many traces should be gathered. Increasing the number of traces increases the probability of finding new leaks.

3.7 Validating `BinaryLoader` Arguments

To test whether emulation is properly supported on the target binary, we can call the following function:

```
binaryLoader.configure()
```

The return value will be non-zero in case emulation failed. For a list of common errors, please consult the *FAQ*.

3.8 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)
```

3.9 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
```

(continues on next page)

(continued from previous page)

```
# 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. Refer to *advanced* usage to learn how to estimate key-bit dependencies.

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

ADVANCED USAGE (KEY-BIT DEPENDENCY ESTIMATION)

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.

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

The resulting reports will include a list of leaks detected by the framework. For example, when running the Camellia example, one of the leaks will pertain to the hexadecimal key parsing in OpenSSL:

Runtime Addr	offset	Detection Module	Comment	Symbol Name	Object Name	Source Path
0x7fffb7fddbc9	0x97dbc9	Secret dep. mem. operation (R/W)	none	OPENSSL_hexchar2int	libcrypto.so.1.1	/home/nicolas/cryptolibs/openssl_x86_64/crypto/o_st

If we were to want to estimate how much this leak affects the confidentiality of cryptographic material, we could use the key-bit estimation technique offered by microsurf. To do so, we would run the analysis for a second time, but passing the runtime address of the leak observed in the first run as an argument to the `SCDetector` class:

```
scd = SCDetector(modules=[
            # Secret dependent memory R/W detection
            DataLeakDetector(binaryLoader=binLoader),
            # Secret dependent control flow detection
            CFLeakDetector(binaryLoader=binLoader)
            ],
            addrList=[0x7fffb7fddbc9]
        )
scd.exec()
```

The result of the second run will be an in-depth analysis of the specified address. Microsurf will produce a heatmap which visualizes the estimated key-bit dependencies:

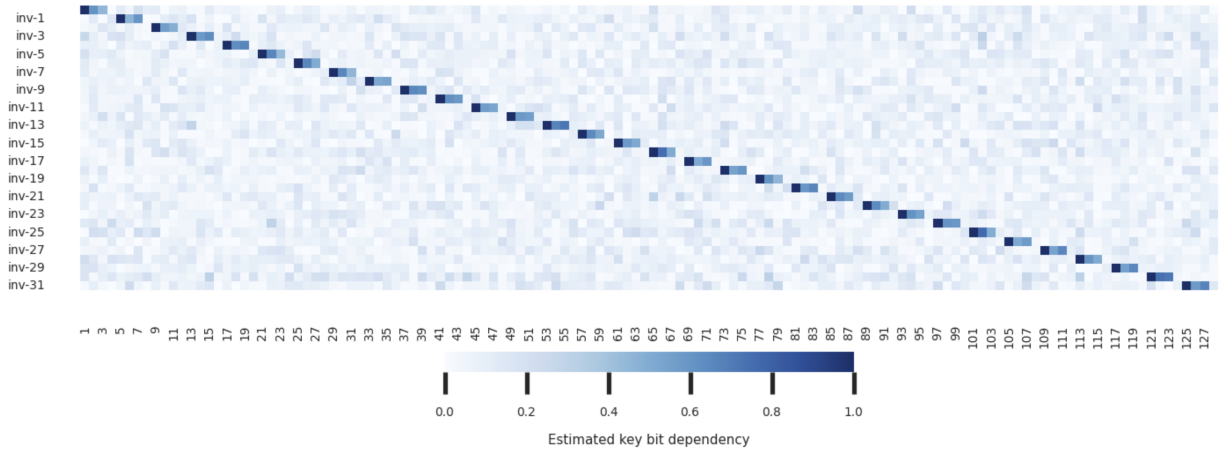
Leaks for OPENSSL_hexchar2int

Runtime Addr	offset	MI score	Detection Module	Comment	Symbol Name	Object Name	Source Path
0x7fffb7fddbc9	0x17dbc9	0.381062	Secret dep. mem. read detector	none	OPENSSL_hexchar2int	libcrypto.so.1.1	/home/nicolas/cryptolibs/openssl_x86_64/crypto/o_str.c:105

Leaking instruction

```
movsx eax, byte [rax + rdi]
```

Key bit dependencies (estimated):



CHAPTER

FIVE

SECRET GENERATORS

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

5.1 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 acomodate 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.

5.2 Existing Generators

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

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

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

5.3 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*)

__call__(**args*, ***kwargs*)

Call self as a function.

Return type str

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,
            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.

Hint: The existing generators provided in the framework are a bit more complex, since they ensure that the same secrets are used for secret-dependent memory detection and control flow operations.

5.3.1 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, nbTraces=10),
```

(continues on next page)

(continued from previous page)

```
        sharedObjects=sharedObjects,
    )
```

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

CHAPTER

SIX

SECRET DEPENDENT CONTROL FLOW

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

6.1 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*.

6.2 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()
```

6.3 The `CFLeakDetector` module

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

6.4 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

SEVEN

SECRET DEPENDENT MEMORY ACCESSES

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

7.1 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.

7.2 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()
```

7.3 The `DataLeakDetector` module

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

7.4 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

EIGHT

COMPILER-INDUCED CONSTANT TIME VIOLATIONS

8.1 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].

8.2 Examples

8.2.1 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.

8.3 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.

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*, *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.
- **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=1000*, *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.

Args: binaryLoader: A BinaryLoader instance miThreshold: 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).

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.

Args: binaryLoader: A BinaryLoader instance miThreshold: 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: 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

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

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

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

To get the most information, make sure to set the environment variable `DEBUG` (`export DEBUG=1`). This will cause *Microsurf* to be more verbose, which might help tracking down any problems.

10.1.1 `QlErrorCoreHook: _hook_intr_cb : not handled`

This is an issue with the underlying emulator (Qiling): Not all interrupts are handled, and some are yet to be added. In the meanwhile, if you wish to improve Qiling, feel free to open a pull request.

Compiling with an older toolchain usually resolves the problem.

10.1.2 `Application returned a non zero exit code`

The emulated application returned a non-zero exit code. This might be due to an incorrect list of arguments provided to the application.

Sometimes, the emulation root directory might not be set up correctly and the runtime linker might have problems finding certain shared objects. In that case, the output will often contain a line such as `error while loading shared libraries: 'libcrypto.so.3'`

make sure that all shared objects are where the binary expects them to be. If the target architecture matches the host architecture, you can check where an application expects shared objects to be located by running `ldd my-binary.elf`.

10.2 Does the lack of reported leaks imply that my binary is safe ?

No, it does not. A common trait of dynamic detection frameworks is that they can only reason about observed behavior. It may be possible that the generated secrets did not trigger every code path.

Nevertheless, dynamic approaches are useful as constant-time debugging tools, especially given that they are much more scalable and easy to use compared to formal tools that may provide guarantees in the absence of leakages.

INDEX

Symbols

__call__() (*microsurf.utils.generators.SecretGenerator method*), 15

B

BinaryLoader (*class in microsurf.pipeline.Stages*), 25

C

CFLeakDetector (*class in microsurf.pipeline.DetectionModules*), 26

D

DataLeakDetector (*class in microsurf.pipeline.DetectionModules*), 26

H

hex_key_generator (*class in microsurf.utils.generators*), 10, 15, 26

R

RSAPrivKeyGenerator (*class in microsurf.utils.generators*), 15, 26

S

SCDetector (*class in microsurf*), 25
SecretGenerator (*class in microsurf.utils.generators*), 15