Collision-Based Attacks Against Whiteboxes with QBDI

Paul HERNAULT (@0xAcid)

quarkslab
www.quarkslab.com

2 / 52

Table of Contents

About this talk

acid@hyères: $ whoami

- Paul HERNAULT, engineer at **Quarkslab**
- RE, Vulnerability research, fuzzing

Goal of the talk

- Start from zero knowledge about whitebox cryptography
- End up with a fully working attack
- With (almost) no requirements[1] in : maths, crypto or RE
- For more info : `https://blog.quarkslab.com/introduction-to-whiteboxes-and-collision-based-attacks-with-qbdi.html`

[1] Some parts will be overlooked, due to the complexity, and time constraint

How to approach the subject ?

A walkthrough using a public whitebox

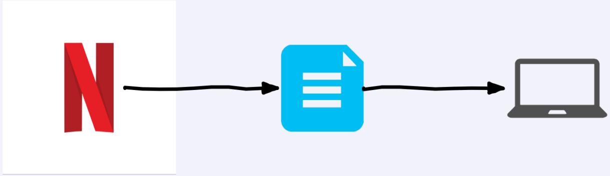
- **Understanding** the need of whitebox cryptography
- **Analyzing** how it is implemented
- **Identifying** weaknesses
- **Building** an attack and **breaking** a whitebox

Table of Contents

An example : Digital Rights Management

Netflix : Sharing content

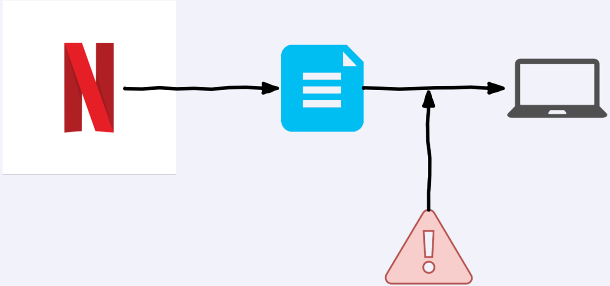
- We want to share content



An example : Digital Rights Management

Netflix : Stealing content

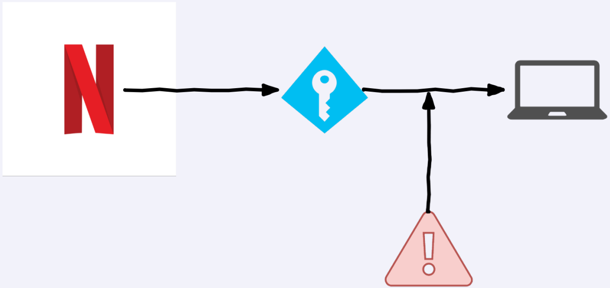
- But an attacker could steal it



An example : Digital Rights Management

Netflix : protecting the content

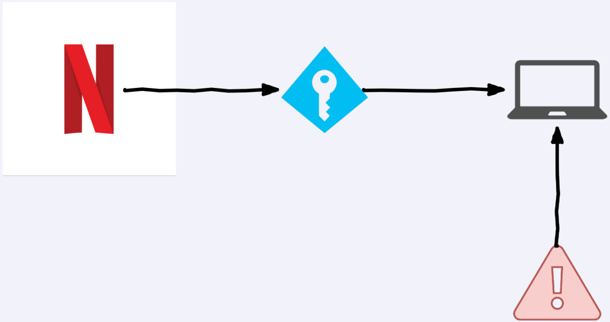
- Protect the content with encryption



An example : Digital Rights Management

Netflix : Reverse engineering

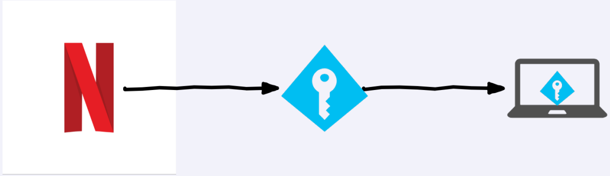
- But an attacker could find the key



An example : Digital Rights Management

Netflix : Protecting the protection

- Protect the key



An example : Digital Rights Management

Whitebox cryptography: protecting cryptographic assets

- Whitebox cryptography is:
  - Protecting a cryptographic key in a hostile environnement
    - *debugging, memory access, code patching*
  - Tries to make the key non-extractable

Table of Contents

Discovering the sample : GH2019 whitebox

Quick analysis

- Most recent public sample I could find
- A single exported function : `void encrypt (uint8_t *buffer)`
  - **input:** data to encrypt (in `buffer`)
  - **output:** encrypted data (in `buffer`)

Reverse-engineering is hard :(

- **Reverse engineering is long and tedious**
  - We will observe the target behaviour instead of reading ASM
- Where to start ? A quick trick
  - We are looking at a cryptographic algorithm
    - Visualizing memory accesses helps a lot understanding them
  - We will use **QBDI** for that!

About QBDI

QBDI : QuarkslaB Dynamic binary Instrumentation

- Cross-platform, cross-architecture instrumentation framework
- Based on LLVM, written in C/C++ ; Python-scriptable
- Think of it as a next-gen fast and scriptable debugger

QBDI Usage

- Profiling
- Binary tracing (trace recording/replay)
- Deobfuscation [2]
- Closed-source Fuzzing [3]
- Crypto (This talk!)

[2] Romain Thomas : Android Native Library Analysis with QBDI
[3] Acid@Kyoto : Fuzzing Binaries using Dynamic Instrumentation - 5th France Japan CyberWorkshop

Tracing the binary with QBDI

- Having fun with pyQBDI !

3 steps to execute the binary with python [4]

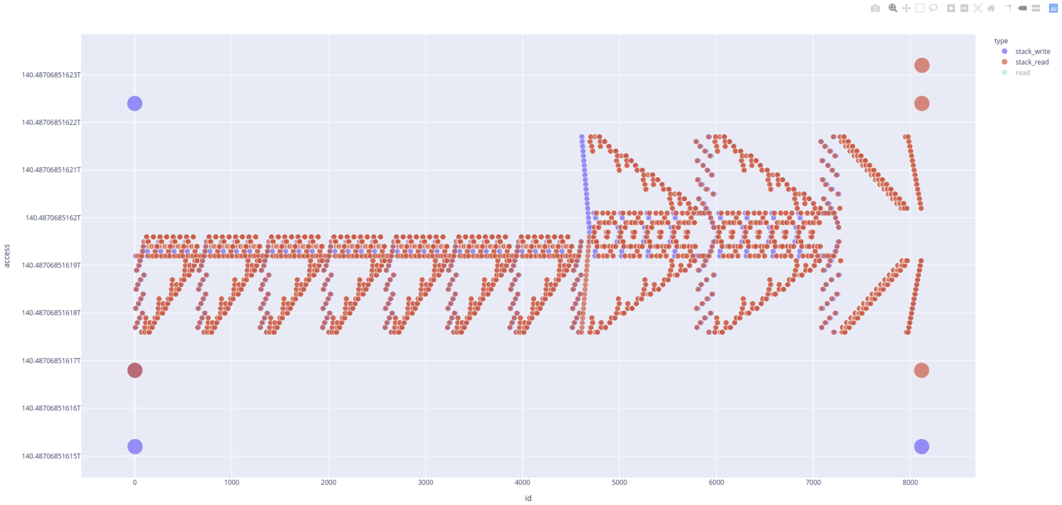
1. import ctypes ; LoadLibrary("whitebox.so")
2. import pyqbdi ; vm = pyqbdi.VM()
3. vm.addMemAccessCB()
4. vm.call(encrypt_ptr, [data_ptr])

Plotting the data

- From the execution, we collect a trace, that we can plot
  - x-axis: **time** of execution
  - y-axis: **address** accessed

[4] This is simplified. But scripting with pyQBDI is **really** easy.

Visualizing Memory accesses



Understanding memory accesses

What can we grab from the graph ?

- Repeating patterns on stack accesses
  - **10 “rounds”** of operation
- A crypto algorithm, with 10 round of operation... ?
  - **AES-128** ?

Crypto is hard :(

- A AES-128 whiteboxed binary
- **Cryptography is hard and painful**
  - How can we extract the key without a PhD ?

Table of Contents

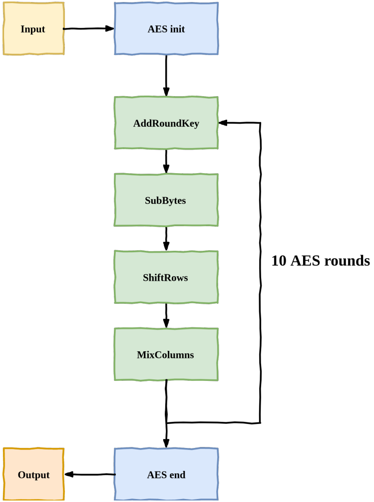
Reminders about AES

AES cheatsheet [5]

- Symmetric cryptographic algorithm
- Size of key : 128-bits (16-bytes)
- 10 rounds of operations composed of 4 specific operations
  - **AddRoundKey**
  - **SubBytes**
  - **ShiftRows**
  - **MixColumns**

[5] Here we talk about AES-128.

Reminders about AES [6]



[6] It is not an accurate AES, but it is easier to explain that way

Reminders about AES

The AES operations

- **AddRoundKey**
  - Simple XOR between the initial state and the round key
- **SubBytes**
  - Substitue bytes with others
- **ShiftRows**
  - Shifting bytes from the state
- **MixColumns**
  - Mixing bytes together

What about whiteboxes ?

From a plain AES to a whiteboxed AES

- Obviously, we want to somewhat hide **AddRoundKey**
- We will merge all operations into **lookup tables**
  - Key will never appear **in plain** in memory
- A round is basically **"a big lookup"** (more or less)
  - You can see before the round, or after the round. not in between.
  - An attacker cannot look at **AddRoundKey** and grab the key.

What about whiteboxes ?

Is AES protected ?

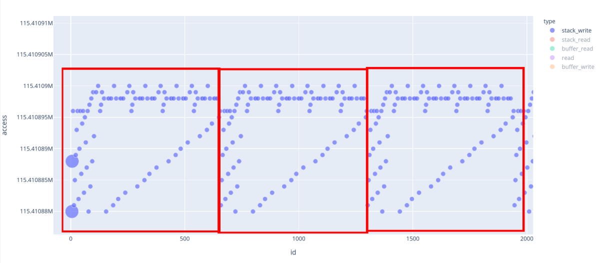
- Is that enough to protect the key ? [7]
- AES is strong because there are many rounds
- If you can isolate rounds... it becomes weak
  - You can revert a round
  - Compute the inverse functions of the 4 operations

[7] Note that, we are protecting **round keys** not the key per se

What about whiteboxes ?

Is AES protected ?

- But we **can** isolate rounds !



Round isolated == key extracted ?

Accessing the state

- With QBDI, we can identify rounds
  - We can observe the **intermediate state** of AES after a round
- So could we recover the key ?
  - Compute **InvMixColumns** then **InvShiftRow** then **InvSBytes**
- For the last operation, we know that

$$\textbf{AddRoundKey}(state\_pre\_round, round\_key) = state\_pre\_round \oplus round\_key = state\_post\_round$$

- Just need to XOR both states to recover the round key !

Is the whitebox broken ?

Is that all we need ?

- Exposing intermediate state in clear is the equivalent of exposing the round key
- Whiteboxes are designed to be protected against that using **InternalEncodings** [8]
  - Intermediate states are encoded in an unkwown way (merged into the lookup tables)
  - We never see any plain state

[8] Each round has its own encoding. The magic of maths allows to still conserve a real AES.

Table of Contents

Considering internal encodings

So what now ?

- Key is hidden in lookup tables
  - Cannot observe it in memory
- Intermediate states are encoded
  - Cannot revert a round

Math is hard :(

- With a bit of math, we could find the encodings
  - Could view plaint state –> Could reverse rounds
- **Math is not that hard, but i forgot everything i learnt**
  - We will break the whitebox, without recovering encodings
  - We need to figure out 2 properties about AES
  - Those properties stand true both for a plain AES and a whitebox AES

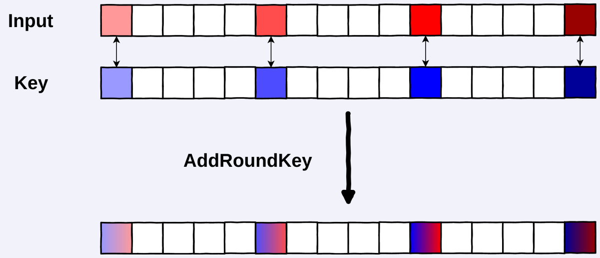
Observation #1: input-output dependency on a single round

- For the following, we will use **one round of encryption** only

Observation #1: input-output dependency on a single round

Merging input and key

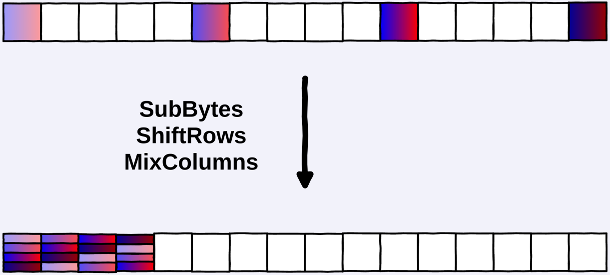
- **AddRoundKey** creates a relation between **input** and **key**
  - Bytes are XORed together



Observation #1: input-output dependency on a single round

Creating bytes relationship

- **MixColumns** is the last operation. It **mixes** bytes together
  - But **not all** bytes are mixed together



Observation #1: input-output dependency on a single round

What do we get from this ?

- **One output byte depends only on 4 bytes of the input/key**
- There is a **defined relationship** between the **parameters** and the **output**
  - It allows us to **split AES** in **4** independent parts

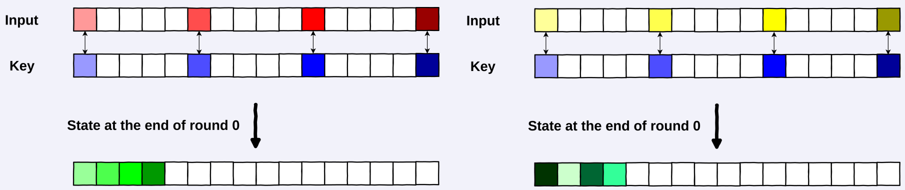
Observation #2: State collision and encodings

- For the following, we will use **one round of encryption** only
- For the following, we will modifies **bytes** that have a **relationship**
- For the following, we use **a plain AES**

Observation #2: State collision and encodings

Observation does not help. Comparison does.

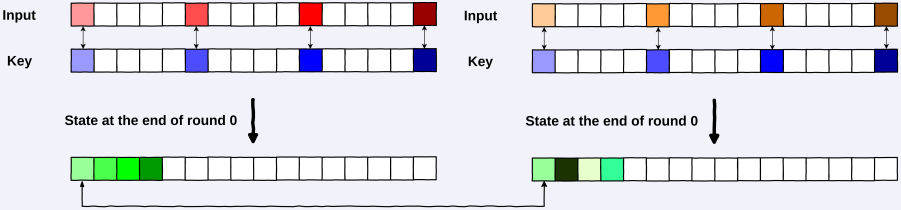
- A round of AES on 2 different plaintexts will **always** give 2 different ciphertexts



Observation #2: State collision and encodings

Observation does not help. Comparison does.

- A round of AES on 2 different plaintexts will **always** give 2 different ciphertexts
  - However, sometimes the ciphertexts have **a few identical bytes**

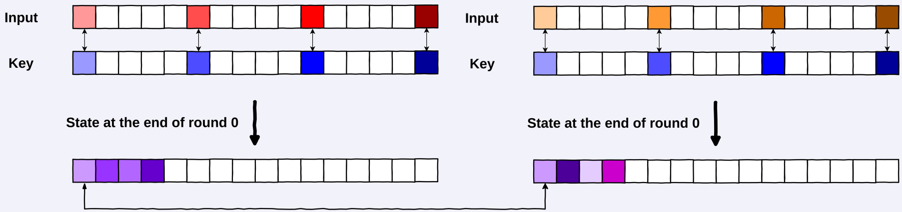


We call this **a collision**

Observation #2: State collision and encodings

Byte collision and internal encodings

- Now let's consider the **same key**, **same plaintexts**
  - But with an **encoded intermediate state**, using the bijection:



- Bytes are differents, but the **collision still exists** !

Observation #2: State collision and encodings

What do we get from this ?

- **If we are able to observe a collision on encoded states, that collision also exists on plain states.**
- **Note**: This collision exists for a **specific** pair of inputs, and a **specific** key
  - **Not all keys** would generate a **collision** for **this pair of inputs**

Table of Contents

Observation recap before diving in

- This is true for both a plain and a whiteboxed AES

Observation #1: input-output dependency on a single round

- **One output byte depends only on 4 bytes of the input/key**

Observation #2: State collision and encodings

- **If we are able to observe a collision on encoded states, that collision also exists on plain states.**

Diving in

Where to go from here ?

- Thanks to QBDI, we can compute

$$\textbf{Whitebox}(inputA, inputB) == collision$$

- This collision also exists on AES without internal encodings

$$\textbf{AES}(inputA, inputB, whitebox\_key) == collision$$

- But we don't know the key :(
  - We can bruteforce now !

Collision-based attacks : How-to

The process

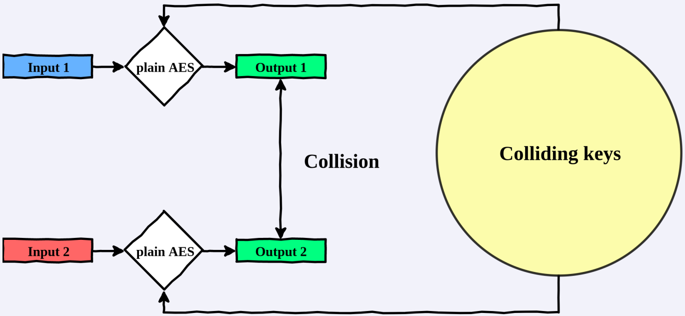
1. **Find 2 plaintexts** that have a **collision** after a **round of whiteboxed AES**
2. Run those **2 plaintexts** in a **plain AES**. **Bruteforce** the key
   - We operate on 32 bits ! [9]
3. If we find a collision... We found the key !
   - Almost...

[9] It is still slow, check the blogpost to reduce optimize the research to 2^17 instead of 2^32

Collision-based attacks : How-to

A set of potential keys

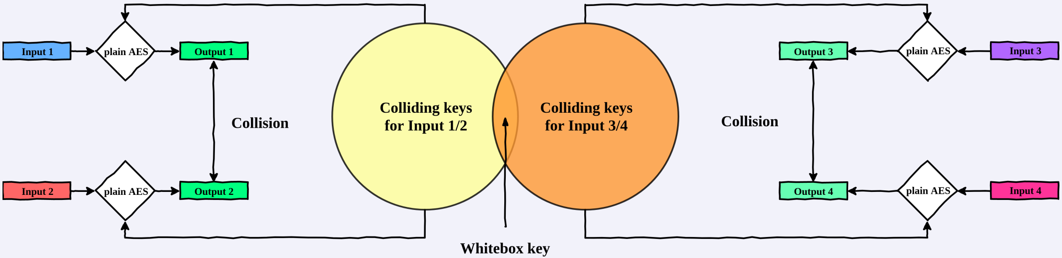
- A few keys generate a collision for a pair of plaintext
  - How can we identify the good one ?



Collision-based attacks : How-to

Reducing the set

- Find other collisions !



Validating the key

Is that the good key ?

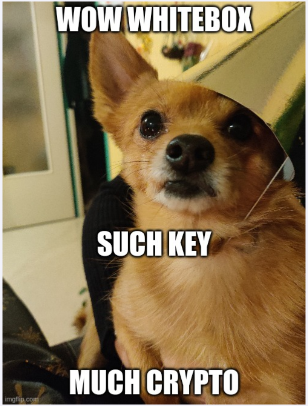
- We recovered : **GH19{AES is FUN}** (looks good!)
  - Validate it with a python AES

```
data = [0]*16

cipher = AES.new("GH19{AES is FUN}", AES.MODE_ECB)
cipher.encrypt(data)
# [200, 48, 81, 207, 15, 188, 94, 26, 143, 211, 192, 201, 176, 229, 73, 159]

pyqbdi_vm.call(encrypt_ptr, data)
# [200, 48, 81, 207, 15, 188, 94, 26, 143, 211, 192, 201, 176, 229, 73, 159]
```

- Success \o/

Validating the key

Good boy [10]



[10] https://www.instagram.com/goupix_the_dog/

Table of Contents

Everybody is whitebox-fighting

What was required to break the whitebox

- Observing intermediate rounds
  - We used **QBDI**, you could use **gdb** or **Intel PIN**
- Figuring out a few stuff about AES
  - Playing around with data, you can quickly discover the described observations

What was not needed

- Hardcore reverse-engineering
- Head-scratching maths
- Cosmic cryptography skills

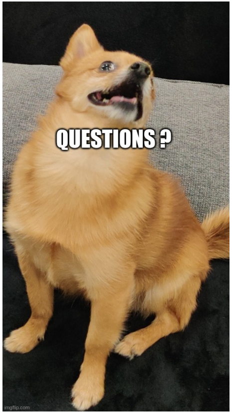
Last words

About the talk

- This attack is not a novelty
  - Has been known for ~15 years (yet still efficient)
- There are countermeasures to this attack
  - But it is not the only attack ! (See DCA and DFA attacks)
- A lot of things have been overviewed, to fit in time
  - Refer to the blogpost [11]

[11] https://blog.quarkslab.com/introduction-to-whiteboxes-and-collision-based-attacks-with-qbdi.html

Questions ? [12]



[12] https://www.instagram.com/goupix_the_dog/

Questions?

SECURING EVERY BIT OF YOUR DATA

52 / 52