CHALLENGE III

VUSec

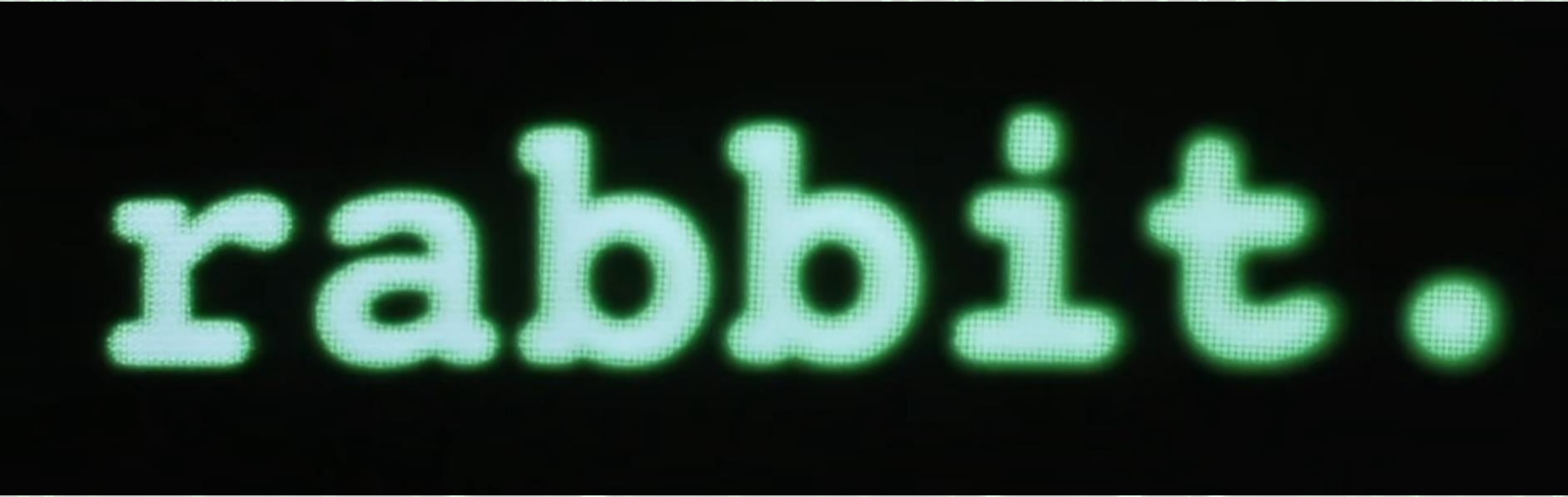
Thus Far

- We investigated potentially stolen intellectual property!
  - Found a secret key…
  - Unpacked a contained binary…
  - Played a hidden game!

WHAT'S NEXT

- We found a strange video…
- There must be more!
- Let's dive deeper…
- …but how?!


white


rabbit.

WHAT'S NEXT

- Probably a different white rabbit…

WHAT'S NEXT

- Probably a different white rabbit…
- A new binary!

WHAT'S NEXT

- Probably a different white rabbit…
- A new binary!
- `./rabbit`

RABBIT

We cannot tell you too much;
They might find it out as well…

RABBIT

```
$ ./rabbit --help
```

RABBIT

```
$ ./rabbit --help
```

Let's just say it…
"Connects" you to more knowledge…

RABBIT

```
$ ./rabbit -i <identity_file>
```

RABBIT

```
$ ./rabbit -i <identity_file>
```

- You need to prove that we were right to choose you

RABBIT

```
$ ./rabbit -i <identity_file>
```

- You need to prove that we were right to choose you
- Find out our secret message

RABBIT

```
$ ./rabbit -i <identity_file>
```

- You need to prove that we were right to choose you
- Find out our secret message
- Use it to show you understand…

RABBIT

```
$ ./rabbit -i <identity_file>
```

- You need to prove that we were right to choose you
- Find out our secret message
- Use it to show you understand…
- **That you are one of us**

COUNTERMEASURES

We had to apply obfuscation, encryption, and more to make static analysis difficult – to avoid "them"…

Thankfully, ~~we~~ you have other tricks up our sleeves!

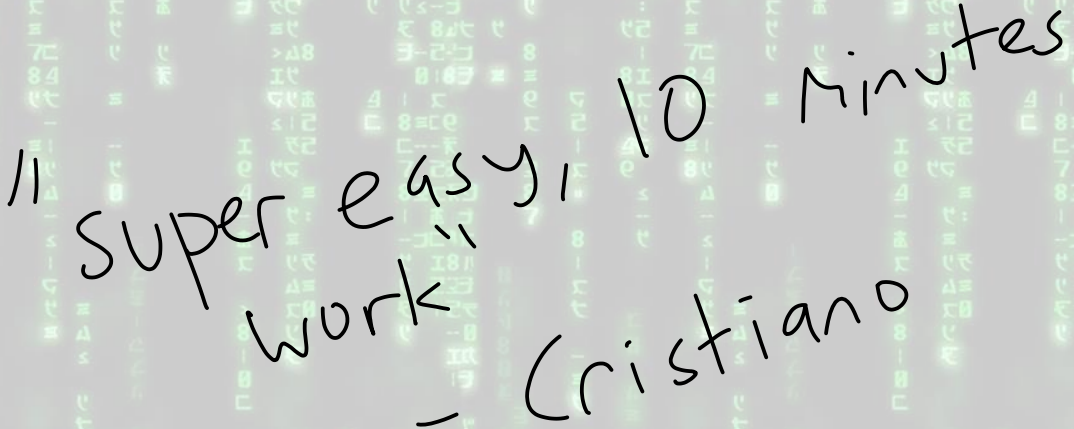
YOUR TASK

YOUR TASK

- **Write a dynamic taint analysis engine, and use it to obtain our secret message!**

YOUR TASK

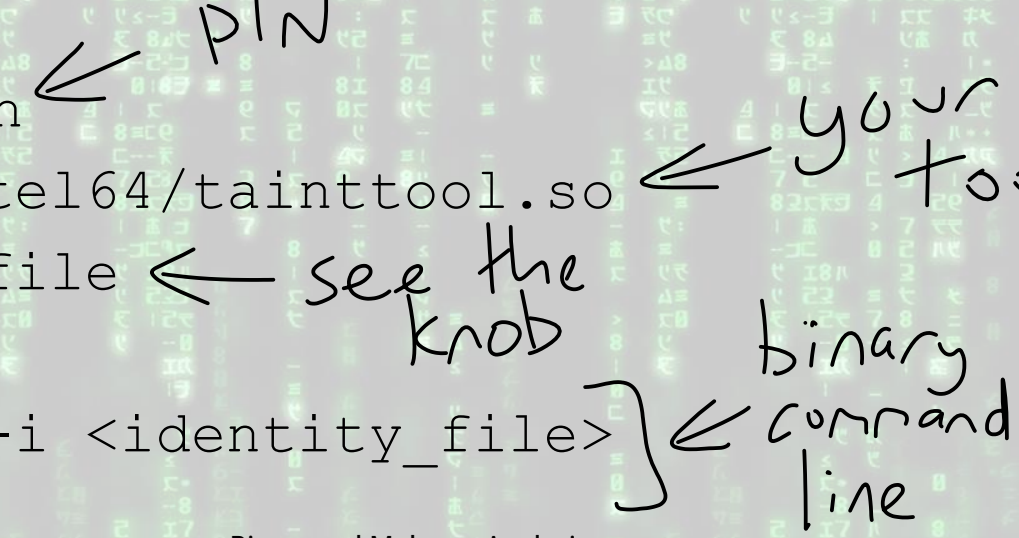
- **Write a dynamic taint analysis engine, and use it to obtain our secret message!**



DYNAMIC ENGINE

- You must write your own PIN tool, based on the framework on Canvas (as discussed during the lecture).
- You can run your PIN tool using something like this:

```
$PIN_ROOT/pin
  -t obj-intel64/tainttool.so
  -i input_file
  --
  ./rabbit -i <identity_file>
```



THE ANALYSIS

ANALYSIS STEPS

- The secret data has been split into 4 parts

ANALYSIS STEPS

- The secret data has been split into 4 parts
- Each requires a different approach to recover

ANALYSIS STEPS

- The secret data has been split into 4 parts
- Each requires a different approach to recover
- They increase in complexity

ANALYSIS STEPS

1. Direct cmp operations

ANALYSIS STEPS

1. Direct cmp operations
2. Comparisons through library functions

ANALYSIS STEPS

1. Direct cmp operations
2. Comparisons through library functions
3. Evasion through arithmetic operations

ANALYSIS STEPS

1. Direct compare operations
2. Comparisons through library functions
3. Evasion through arithmetic operations
4. Obfuscated compare operations

KEY INSIGHTS

1. Each input character is **used only once**

KEY INSIGHTS

1. Each input character is **used only once**
2. You can “progressively” improve:

    Implementing more steps will recover more characters

THE STEPS

Step 1

Direct comparisons

```
if ((msg[111]==‘1’) && (msg[112]==‘.’) && (msg[113]==‘1’) {
    /* version 1.1 -> perform further parsing */
}
```

Step 1
Direct comparisons

```
if ((msg[111]=='1') && (msg[112]=='.') && (msg[113]=='1') {
    /* version 1.1 -> perform further parsing */
}
```

- The parser will continue **only** if `msg` contains string "1.1"
- If `msg` is tainted, we will observe some `cmp` instructions between tainted values and untainted constants ('1', '.', '1' etc)
- By changing the input such that "1.1" ends up in buffer positions 111-113, we can progress the execution

Step 1
Direct comparisons

- Instrument CMP to find the values compared with the input data
- This will let you unearth the part of the data that is only copied around and compared

Step 1
Direct Comparisons

- Instrument CMP to find the values compared with the input data
- This will let you unearth the part of the data that is only copied around and compared
- *It's not quite so simple, but more on that later…*

Step 11

Library Func Comparisons

```
if (strncmp(&msg[106], "HTTP/", 5) == 0) {
   /* continue parsing an HTTP message */
}
```

Step 11

Library Func Comparisons

```
if (strncmp(&msg[106], "HTTP/", 5) == 0) {
   /* continue parsing an HTTP message */
}
```

- Use the known semantics of libc functions (such as string functions) to recover further fragments of the data:
  - The first argument is tainted, and compared against an untainted string
  - We could change the message to read "HTTP/" at position 106 and see if execution will continue past the check

Step 111

Arithmetic Comparisons

```
eax = msg[5];  // eax tainted
ebx = eax + 1; // taint not propagated
               //    on arith instructions
ebx -= 1;      // ebx not tainted!
if (ebx == 'a') {
   // continue processing input…
}
```

Step 1)))

Arithmetic Comparisons

```
eax = msg[5];  // eax tainted
ebx = eax + 1; // taint not propagated
               //    on arith instructions
ebx -= 1;      // ebx not tainted!
if (ebx == 'a') {
   // continue processing input…
}
```

- **Your code needs to propagate taint on arithmetic instruction**
  (XOR, ADD, SUB, etc…)
- **Analysis can otherwise easily be duped**

Step IV

Obfuscated Comparisons

```
eax = msg[5]; // eax tainted
ebx = eax + 1; // after step 3, ebx will also be tainted
if (ebx == 'b') {
    // continue processing input…
}
```

- **Comparisons may be obfuscated so that they are harder to analyse!**
- `msg[5]` **is actually compared against** `'a'`

Step IV

Obfuscated Comparisons

```
eax = msg[5]; // eax tainted
ebx = eax + 1; // after step 3, ebx will also be tainted
if (ebx == 'b') {
    // continue processing input…
}
```

- Two approaches to obfuscated comparisons:
  - Brute force all ASCII characters until you have a match
  - Keep track of operations on tainted values, then back-track and compute the real input value expected by the program
- Don’t try to generalize the back-tracking! Focus on retrieving the secret!

CAVEATS AND CONTEXT

Step 1 Caveat

```
mov ebx, input
xor ebx, 0xc0ffee
cmp ebx, 0xc0ffe4
```

- This is an obfuscated equivalent of (`input == 0xa`)
- When still working on Step 1, this sequence of instructions may be misinterpreted:
  - `ebx` will be tainted; `xor` does not affect its taint!
  - `cmp` will be compare it against untainted `0xc0ffe4`
  - `0xc0ffe4` will be regarded as part of the hidden message!

Step 1 Caveat

```
mov ebx, input
xor ebx, 0xc0ffee
cmp ebx, 0xc0ffe4
```

- To ignore such sequences during Step 1, you may want to "wash" taint on arithmetic operations on tainted values
- If you remove the taint after the `xor` instruction:
  - `ebx` will be not be tainted when `cmp` is executed
  - Value `0xc0ffee4` will be ignored

Context of Analysis

- `CMP` instructions are **everywhere**
- Code and external library functions may end up using `CMP` instructions on tainted values
  - e.g. if an internal `print` function compares a tainted value to -1, this doesn't necessarily mean that -1 is expected in the input
- I.e. the context of `CMP` instructions is important!

Focus on the Goal

- Don't try instrumenting `libc` functions!
  They use complex instructions and SSE registers
- Instead, propagate taint (as in Step 2) according to the known semantics of `libc` functions

- Don't try implementing lots of x86 instructions!
  Limit your instrumentation to the minimum needed

PERFORMANCE

- Make sure your analysis functions get **inlined**
- Avoid doing work that you don't need
- Do you need more than one pass?

PERFORMANCE

- Make sure your analysis functions get **inlined**
- Avoid doing work that you don't need
- Do you need more than one pass?

(you can do it in one run!)

BE SMART

- Do not attempt static analysis
  - You do not need to work out what the binary is doing
- Write scripts/code for everything
  - Any language is fine (Python/Go/Rust/ocaml/…) as long as you document/we can run it easily
- Focus on the task
  - No need to overgeneralise – you just need the secret!
- Don't forget the other command-line flags…

yet...

Grading

- Direct `CMP` comparisons -> 2 points
- Comparisons via library functions -> 2 points
- Obfuscation using arithmetic -> 2 points
- Comparisons needing arithmetic -> 2 points
- High performance tool -> 1 point
- Readable scripts/good automation/convenience -> 1 point
- Bonus points:
  - The usual bonus points for the first students to submit.

Submission Guidelines

You need to deliver a zip file containing:

- **A plain text file `secret.txt` containing the (parts of the) secret data you recovered**
- **A plain text file `README.md` describing what you did (and why), and how to run your code**
- **Your code & scripts – which should generate `secret.txt`**
- **Any other files you used/are necessary for your stuff to run!**

Submission Guidelines

A friendly reminder:

- Make sure you include **all** necessary files
- Use (environment) variables for paths or hardcoded variables/addresses
- We assume pin 3.22 ($PIN_ROOT)
  - If you use anything else please specify in your submission and give a (good) reason
- Use file extensions (.sh/.py/etc.)
- Make it clear on how to run your scripts and which one does what

Submission Guidelines

- Submission will be through **Canvas**.
- **Deadline: Sunday the 1st of May, 2022, at 23:59 CEST**
- Delay penalties: 1pt/24h delayed

WARNING

Your binary is **unique** to you!

WARNING

Your binary is unique to you!

Everyone will get different results

WARNING

Your binary is **unique** to you!

Everyone will get **different results**

There is no need to be worried when you recover different numbers of characters

WARNING

Your binary is **unique** to you!

Everyone will get **different results**

There is no need to be worried when you recover different numbers of characters

(Seriously, don’t compare numbers)

GOOD LUCK