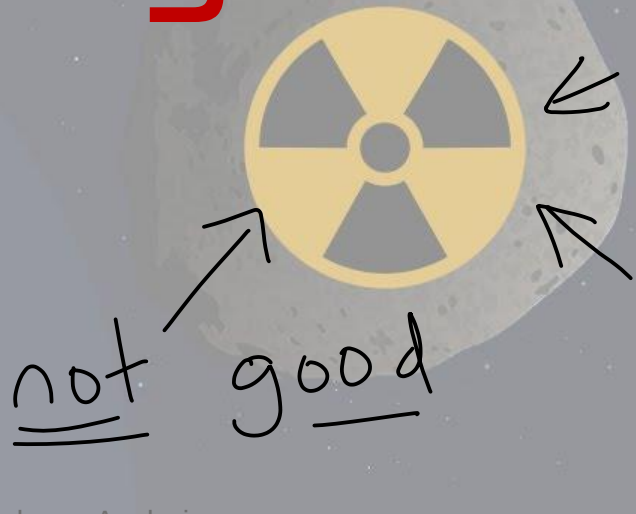
BAMA 2020

Challenge 3



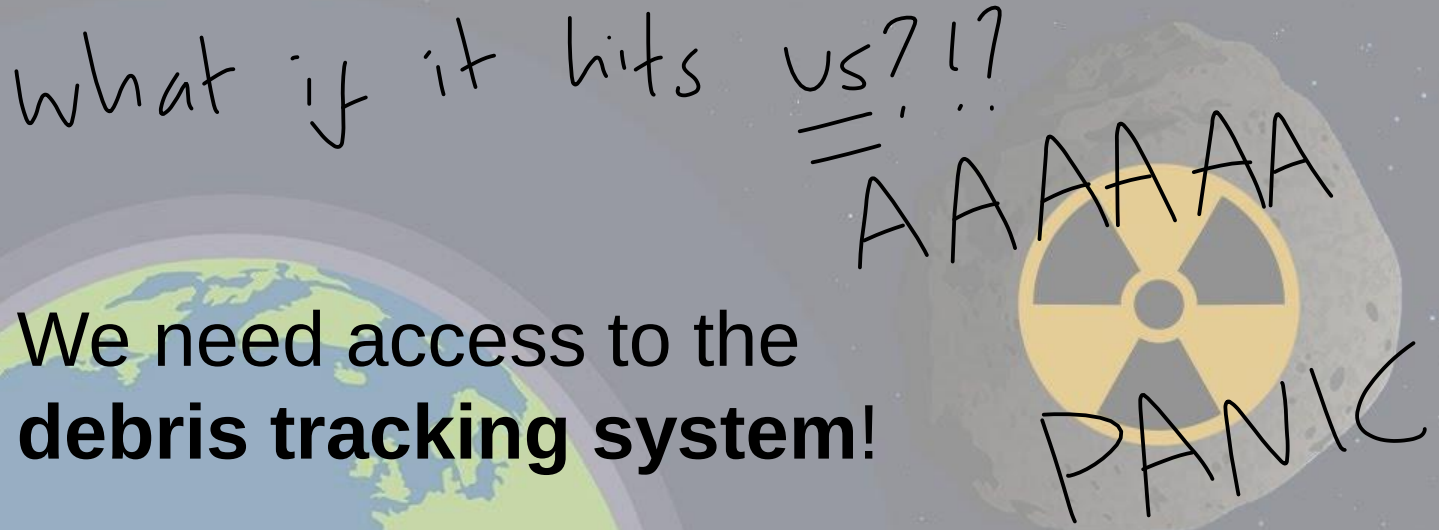
Thus far

- We have dived into the armatronic binary
  - Obtained the arming key! ✓ WIN
  - Unpacked the inner binary!
  - Crunched that password!

where is the BOOM?!?

Oh no!

- What happens if the debris hits satellites?



- We need access to the **debris tracking system**!

VUSec

new binary!

What next?

```
$ debris_accessctl --help

Space debris access control system

--msgfile/-m filename
```

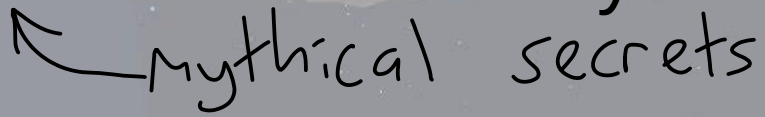
Provide authorization

What next?

`debris_accessctl` <filename>

To gain access to the system, an authorization file must be provided with very specific contents!

(Sources indicate that the authorization file also probably contains documentation for the system!)



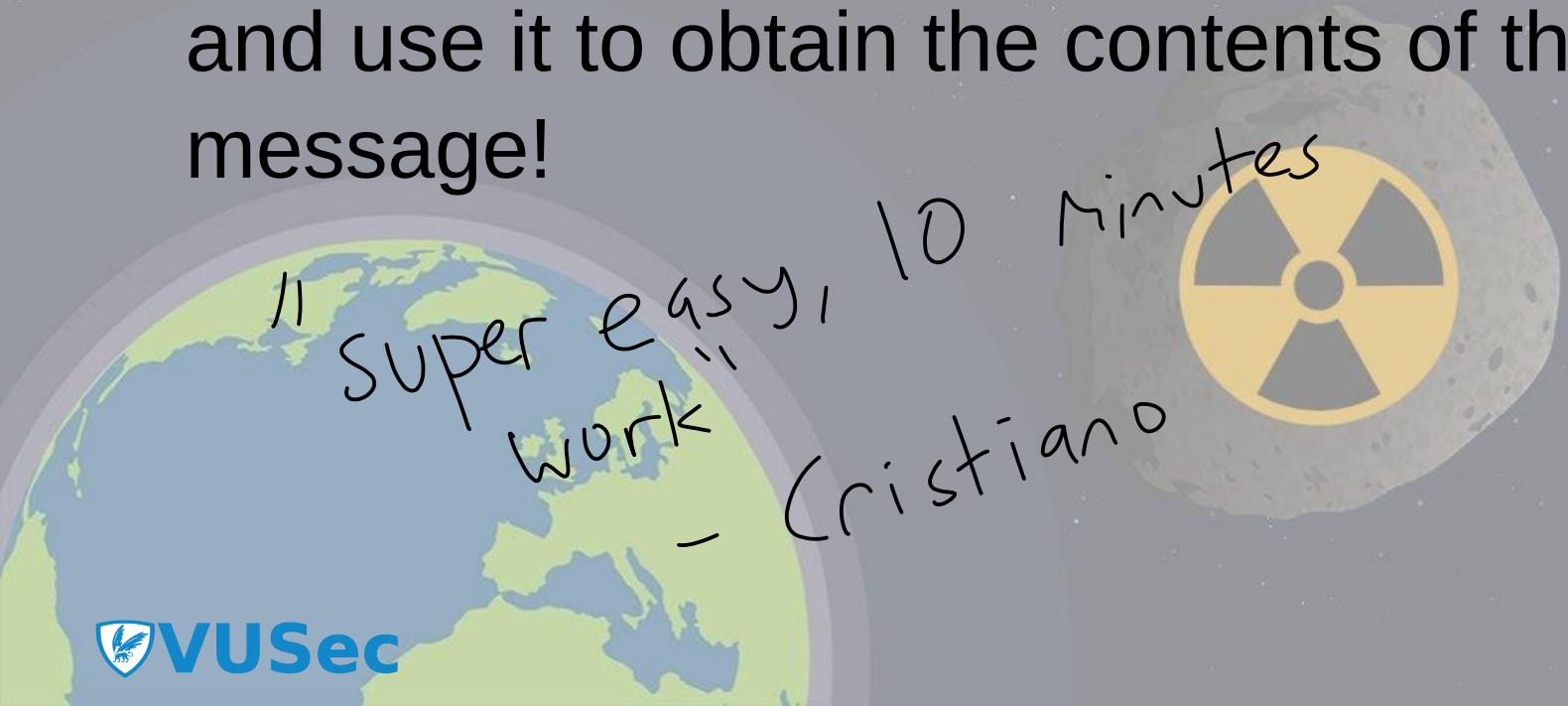
Countermeasures

Engineers have applied obfuscation, encryption and other techniques to make static analysis difficult.

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

Your task

- Write a **dynamic taint analysis engine**, and use it to obtain the contents of the message!



VUSec

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
    --
  ./debris_accessctl -m input_file
```

pin

your tool

see the knob

binary command line

Analysis Steps

- The secret authorization data has been split in 4 parts, which require different approaches to be recovered.
- Increasingly more complex forms of DTA are required in each step:
  - Step 1 → Direct cmp operations.
  - Step 2 → Comparisons through library functions.
  - Step 3 → Evasion through arithmetic operations.
  - Step 4 → Obfuscated cmp operations.

Key Insights

- Each input character is used only once by the analyzed binary.
- Improving your taint engine (by handling more steps of the analysis) will allow you to recover more characters.

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 the message is tainted, we will observe some cmp instructions between tainted values and untainted constants (‘1’, ‘.’, ’1’ etc.).
- By changing the input so that “1.1” ends up in buffer positions 111-113, we can make execution continue to the verification of the rest of the message.

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 2
Library Funct. 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. In the example above:
  - The first function argument is tainted, and it is compared to an untainted string.
  - We could change the message to read "HTTP/" at position 106 and see if execution will continue past the check.

Step 3
Arithmetic Instructions

```
eax = msg[5];  // eax tainted
ebx = eax + 1; // taint not propagated
                  on arith instructions.
               // ebx not tainted!

ebx -= 1;
if (ebx == 'a') {
        // continue processing input…
}
```

- Your code needs to propagate taint on arithmetic instruction.
- Analysis can otherwise be duped by introduction of simple arithmetic, as shown in the example.

Step 3
Arithmetic Instructions

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

- The comparison of msg[5] to ‘a’ will be missed because taint was not propagated properly.
- You need to instrument code so that taint is propagated on arithmetic operations (add, sub, xor, etc.).
- Recover characters that are compared after having their taint “washed” by an arithmetic operation.

Step 4
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 analyze! E.g. msg[5] is actually compared against ‘a’.

Step 4
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 – hard! Focus on retrieving the authorization data.

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.
  - 0xc0ffee4 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.

(implement ALL the instructions!)

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 (please!)

- Do not attempt static analysis ← waste of time
  - You do not need to work out what the binary is doing (yet).
- Write scripts/code for everything
  - Any language is fine (Python/Go/Rust/ocaml/…)
- Focus on the task
  - No need to overgeneralize, you just need the message!
- Don’t forget the other command-line flags
  - You might need some other flags (listed in `--help`?)

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 -> 1 point
- Bonus points:
  - The usual bonus points for the first students to submit.
  - Extra bonus points if you implement bit-level tainting!

VUSec

Submission Guidelines

You need to deliver a **zip file** containing:

- A plain text file ‘auth1’ containing the (parts of the) authorization data you recovered.
- A plain text file ‘**README**’ describing what you did and how to run your code.
- Your code+scripts, which should generate auth1.

Submission Guidelines

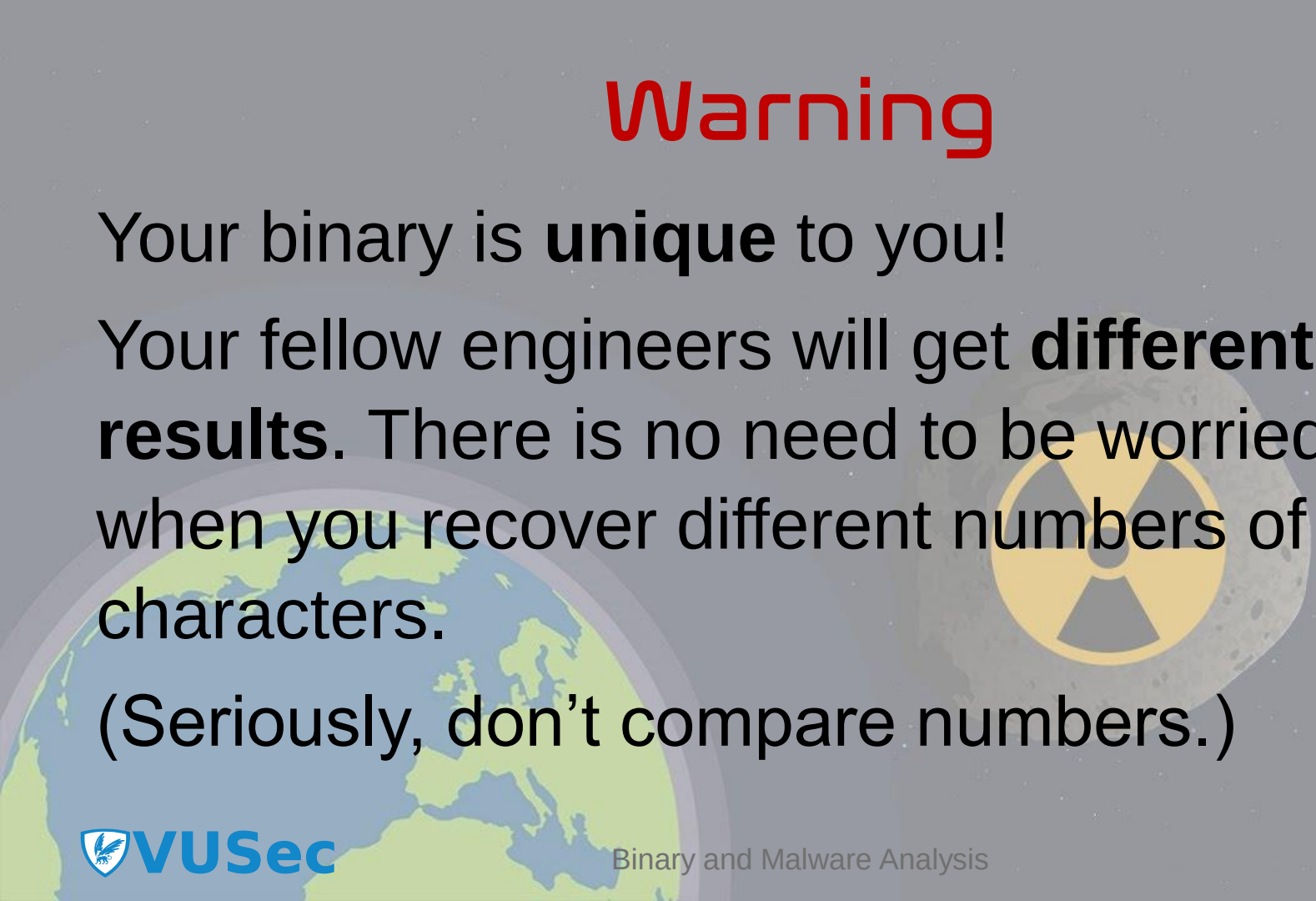
- Submission will be through **Canvas**.
- **Deadline: Thursday, 30th April 2020, 23:59 CEST**
- Delay penalties: 1pt/24h delayed

Warning

Your binary is **unique** to you!

Your fellow engineers 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!