

# Strings & bytes in Python 3

You are building an exploit and, not being a barbarian, have switched from Python 2 to 3. One part of your exploit involves leaking an important address by dumping a chunk of memory. This chunk of memory contains a lot of random data but somewhere in the middle you know that there is a string "Här: " <sup>1</sup> followed by the 4 bytes representing the little endian encoding on the address you are looking for. You copy some old Python 2 code that you have used for a similar situation:

```
def extract_leaked_pointer(leak):
    marker = 'Här: '
    start = leak.find(marker)
    leak_start = start + len(marker)
    leak_data = leak[leak_start:leak_start+4]
    return struct.unpack('<I', leak_data)[0]
```

Sadly, when running this, you get the following error: `TypeError: argument should be integer or bytes-like object, not 'str'`

To solve this, you try to decode the leaked bytes into a string by adding this line to the code:

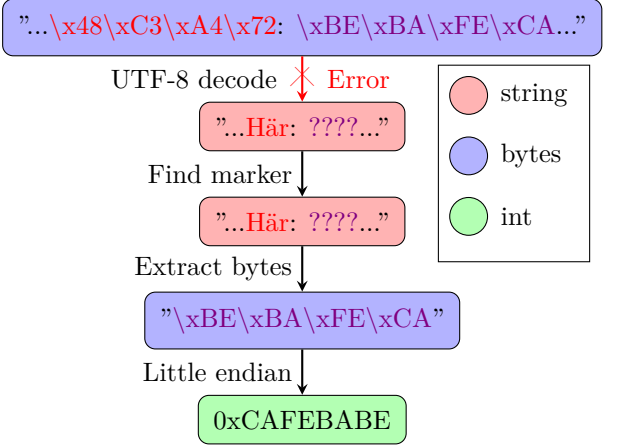
```
leak = leak.decode('utf-8')
```

Unfortunately, this doesn't work either and you are left staring at another error message:

```
UnicodeDecodeError: 'utf-8' codec can't
decode bytes in position 0-1: invalid
continuation byte
```

In anger your desire to develop as hacker, you turn to Twitter and complain about how Python 3 sucks this article to understand how to reason about Python 3, strings, character encodings and arbitrary bytes.

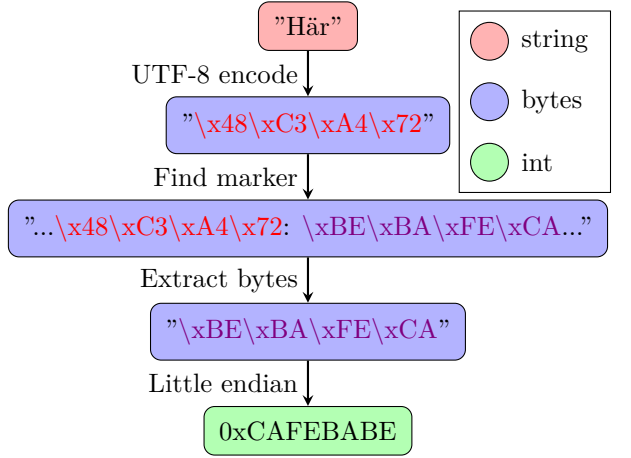
The problem is that you are trying to interpret an arbitrary sequence of bytes as UTF-8 encoded data. This is the equivalent of trying to push a square peg through a round hole. It won't work. The following diagram shows what you are trying to do and where it goes wrong.



Let's remind ourselves about character encodings. The goal is to represent a character such as "A". Computers work with numbers (more precisely bits), not characters, so we translate the character into a number. In the ASCII encoding, this is the number 65. We call this the codepoint. This value is then encoded using a single byte 0x41. This is simple in ASCII because there is a one to one to one mapping between characters, codepoints and bytes and this can be implicitly done without thinking about it. Python strings are not limited to ASCII but can represent the full Unicode range. If we use the UTF-8 encoding and take the character "ä" we instead get:

| Character | Codepoint | Bytes     | Encoding |
|-----------|-----------|-----------|----------|
| A         | 65        | 0x41      | ASCII    |
| A         | 65        | 0x41      | UTF-8    |
| ä         | 228       | 0xC3 0xA4 | UTF-8    |

Specifically, one character maps to a codepoint which is encoded with more than one byte and not every sequence of bytes represents a valid character. This is what causes the problem. To solve this, instead of trying to convert the "haystack" bytes into a string and search for a substring, you convert the "needle" marker into a sequence of UTF-8 bytes and search for that sequence of bytes in the haystack. When it is found, you can extract bytes relative to that offset and process them accordingly, in this case, convert them to a 32-bit number. This slightly modified diagram describes this approach.



Which, translated to Python 3 code, looks like this:

```
def extract_leaked_pointer_python3(leak):
    marker = 'Här: '.encode('utf-8')
    start = leak.find(marker)
    leak_start = start + len(marker)
    leak_data = leak[leak_start:leak_start+4]
    return struct.unpack('<I', leak_data)[0]
```

In short, don't try to convert bytes that don't represent text, into text. Instead convert the text into bytes, use it to extract the relevant bytes and then process them accordingly. Now your exploit works in Python 3 and you can leave another legacy language behind.

<sup>1</sup>Swedish for "here"