IT security

(VIHIAC07)

Lab exercise

Input validation

Author:
András Gazdag and Dorottya Futóné Papp

February 13, 2025

Contents

1 Educational Objectives

In this exercise, you will be tasked with writing input validation code for a proprietary file format. The goal of this exercise is to demonstrate the necessity for making sure that data entering an application cannot cause malfunction. You will be given the specification of a custom image file format, the CrySyS Image File Format (CIFF) and a naive implementation of the image viewer application. Your task will be to add the necessary checks to the application, such that malformed input is not displayed by the application. After completing the exercise, you will be capable of extracting from a specification both implicit and explicit assumptions about input data, as well as implementing code which checks whether those assumptions hold for any given input.

2 Background Material

2.1 Data Processing

In order to create applications with fewer bugs and lesser severity levels for compromises, the *attack surface* of the application has to be considered. This notion focuses on what is valuable for the application and how it interacts with its users and other processes.

The attack surface consists of:

- Data that is valuable for the application and/or sensitive, as well as the corresponding code segments protecting said data.
- Execution paths through which data and/or commands enter and leave the application as inputs and outputs, as well as the corresponding code protecting said paths.

Considering the attack surface is the first step in implementing input validation for applications. By default, the programmer must assume that all data entered by users can also be entered by the attacker and is therefore untrusted.

There are three main approaches available for dealing with untrusted inputs:

1. **Normalization / canonicalization**: Conversion to the input's simplest known and anticipated form.
   For example, converting all input strings to a pre-specified encoding scheme (e.g., Unicode) is a conversion because the binary representation of the string changes (original syntax is lost, semantics stay the same). Another example is XML documents, for which an empty tag can be written as both `<tag></tag>` and `<tag/>`. Modifying the input such that only one of these formats is used in all documents is a conversion to an equivalent form.

2. **Filtering**: Removal of elements from the input based on some criteria.
   For example, removing `<` and `>` characters from web form inputs in web applications is a filtering step aimed at mitigating e.g., cross-site scripting. Filtering must always be performed after normalization / canonicalization!

3. **Validation**: Checking if the data is reasonable considering both its structure and semantics. Similarly to filtering, validation must only be performed after normalization and/or canonicalization. For example, if input is a string that is supposed to be a date in the format `YYYY-MM-DD`, the validator needs to perform both:

    - **Syntactical checks** (e.g., the input string must contain exactly two `-` characters, splitting the string into three integers; the integers must be padded if their values take up fewer characters than the format implies, etc.).
    - **Semantic checks** (e.g., none of the three integers must be negative; the second integer has to have a value between 1 and 12 corresponding to the twelve months; the third integer has to have a value in the range of how many days there are in each month, etc.).

In order to write secure and robust code, all three approaches must be implemented as necessary.

2.2 Python Properties and Decorators

The Python language by default does not have access modifiers like `private` or `protected`. Instead, Python programmers use a naming convention: if an attribute's name starts with an underscore, e.g., `class._x`, then the attribute is considered private; otherwise, it is public. Such a convention, however, does not provide data encapsulation for Python classes.

One way of providing data encapsulation is the use of getter and setter methods, as shown in the example below:

```python
class Example:
    def __init__(self, v):
        """
        Constructor
        """
        self._value = v

    def get_value(self):
        """
        Getter method
        """
        return self._value

    def set_value(self, v):
        """
        Setter method
        """

        # ...
        # input validation
        # ...

        self._value = v
```

However, the drawback of this approach is that computations done with the encapsulated attribute are hard to read. Consider the case when the `_value` attribute contains an integer. In order to do addition for two `_value`s from two instances of the `Example` class, one would need to write:

```python
a = Example(1)
b = Example(2)
result = a.get_value() + b.get_value()
```

The more complex the computation, the less readable the code becomes. Alternatively, the class could implement an `add(self, Example)` method to better conform with object-oriented programming guidelines. However, this scenario does not help with readability either.

The solution to this issue is the use of *properties*.
The construct both allows programmers to follow naming conventions and to implement data encapsulation.
Properties can be created using the special `property()` function which takes as arguments three functions (a getter, a setter, and a deleter) and a documentation string.
The constructed property can be used in the place of an attribute.

```python
class Example:
    def __init__(self, v):
        """
        Constructor
        """
        self._value = v

    def get_value(self):
        """
        Getter method
        """
        return self._value

    def set_value(self, v):
        """
        Setter method
        """

        # ...
        # input validation
        # ...

        self._value = v

    value = property(get_value, set_value, None, "")


a = Example(1)
b = Example(2)
result = a.value + b.value
```

When the property is read, the configured getter method is invoked.
When the property is written, the configured setter method is invoked.

The `property()` function is a so-called *decorator* function: a function which takes as an argument another function, modifies its execution, and returns it.
Decorator functions can be directly invoked as shown in the previous example. However, using the special symbol `@`, individual functions and methods can also be decorated without the need to explicitly invoke the decorator function.
The previous example can be rewritten as shown below.

```python
class Example:
    def __init__(self, v):
        """
        Constructor
        """
        self._value = v

    @property
    def value(self):
        """
        Getter
        """
        return self._value

    @value.setter
    def value(self, v):
        """
        Setter
        """

        # ...
        # input validation
        # ...

        self._value = v
```

The method `value(self)` has been decorated as the getter for the property with the same name.
The class also contains the method `value(self, v)` (notice the method overloading!) which is decorated to be the setter for the property `value`.
The property can be used in the same way as before.

2.3 Python Built-in Decorators

Python has three built-in decorators which can be used with the `@` character. These are:

- **`property`** - Creates properties as discussed before.
- **`staticmethod`** - Creates a static method, one that does not have access to any `self` variables.
- **`classmethod`** - Modifies the method such that its first parameter is not `self` but the class object instead.

Decorators can also be defined by the programmer (out of scope for this laboratory exercise). For more information on decorators, see this primer on Python decorators.

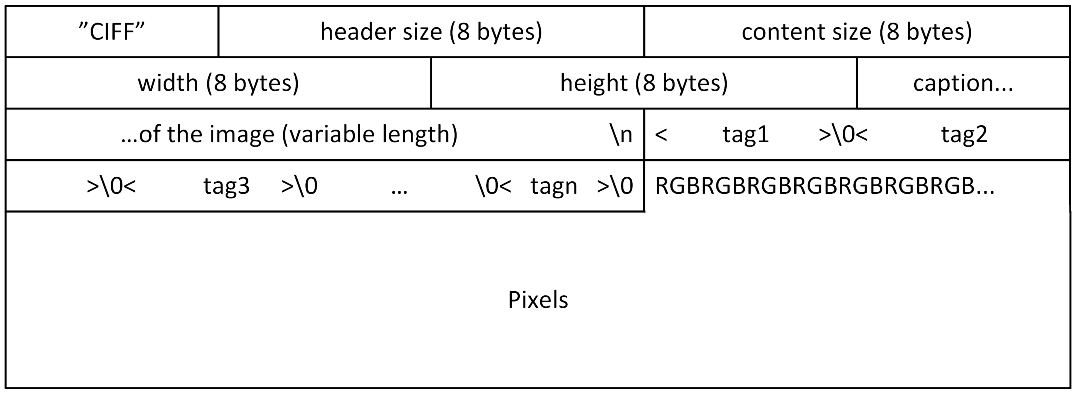

Figure 1. CIFF file format

2.4 CrySyS Image File Format

The CrySyS Image File Format (CIFF) is a custom, uncompressed image format. Because it is custom, there are no existing libraries or modules implementing its parsing. The overview of the file format is shown in the figure above.

2.5 File Specification

Files conforming to the CIFF specification start with a header. The file header consists of the following parts:

1. **Magic:** 4 ASCII characters spelling `CIFF`.
2. **Header size:** An 8-byte-long integer, its value is the size of the header (all fields included), i.e., the first `header_size` number of bytes in the file make up the whole header.
3. **Content size:** An 8-byte-long integer, its value is the size of the image pixels located at the end of the file. Its value must be `width * height * 3`.
4. **Width:** An 8-byte-long integer giving the width of the image. Its value can be zero; however, no pixels must be present in the file in that case.
5. **Height:** An 8-byte-long integer giving the height of the image. Its value can be zero; however, no pixels must be present in the file in that case.
6. **Caption:** Variable-length ASCII encoded string ending with `\n`. It contains the caption of the image. As `\n` is a special character for the file format, the caption cannot contain this character.
7. **Tags:** Variable number of variable-length ASCII encoded strings, each separated by `\0` characters. The strings themselves must not be multi-line. There must be a `\0` character after the last tag as well.

The header is followed by the actual pixels of the image in RGB format, with each component taking up 1 byte. The image must contain exactly the content size amount of bytes making up the pixels.

2.6 Naive Python Implementation for CIFF Images

You will be given a naive Python implementation capable of handling CIFF images conforming to the specification. It consists of two files, `view.py` and `ciff.py`. The former implements the graphical user interface, while the latter is the class representing a CIFF image. You need to add the necessary checks to the latter class. Below are some hints for understanding the code:

- The class has several properties, one for each part of the header as well as for the list of pixels. These properties have both getters and setters as well.
- The property `is_valid` is used to determine whether an instance of the `CIFF` class contains data of a valid CIFF image. The underlying getter function must return a boolean value.
- The class also contains a static parsing method decorated with `@staticmethod`, `CIFF.parse_ciff_file()`. This method is called when a CIFF file is to be parsed and must return an instance of the `CIFF` class. The method relies on the `struct` Python module's `unpack` function to interpret a sequence of bytes as dictated by the format specifier. The documentation of the module can be found here, and its format specifiers must be read before the laboratory exercise.

2.7 Debugging Python Code

There are two main approaches for debugging Python applications.

Firstly, most IDEs, e.g., IDLE, have built-in debugging features. A good summary of IDLE's capabilities can be found at this tutorial, which is mandatory to read before beginning the lab exercise.

The second approach involves a Python module designed for debugging the code and can be used in scenarios where no graphical user interface or a development environment is available. The module is called `pdb`, the Python Debugger, and has to be imported before use.

Once imported, a breakpoint can be set with the `pdb.set_trace()` function. The function stops execution at its invocation and allows the developer to inspect variables, invoke in-scope methods and functions, as well as execute simple Python instructions.

The most notable commands of `pdb` are the following:

- `p` - print, used to inspect variables,
- `s` - step into, used to step into called functions and methods,
- `n` - next, executes code until the next line,
- `u` - until, executes code until the next line with a greater line number is reached (especially useful for skipping over loops in the code), and
- `c` - continue, executes code until the next breakpoint or termination.

3 Setting Up the Environment

3.1 Using a Virtual Machine

Using the VirtualBox application, import the released OVA file and start the VM. No further preparation is required using the provided VM.

3.2 Using a Home Environment

If you are not using the provided virtual machine (VM), follow the included Python tutorial to set up the virtual environment. Then, install and verify the following dependencies:

- Python 3.7 or later (installed system-wide)
- python-tk 3.13 – stable version: 3.13.1 (installed system-wide)
- pillow 11.1.0 (installed within the virtual environment)

3.3 Setting up initial code

Download or git clone the initial implementation from here: https://software.crysys.hu/it-security/input-validation

4 Tasks

At the start of the lab practice, we first interpret the specification and list the necessary validation checks to be implemented!

- After reading each byte, the implementation must check whether it has reached the end of the file (EOF). To do this, it should check if the read operation returned an empty string.
- **Magic**: The characters must form the word `CIFF`. The naive implementation currently decodes the four characters in a single operation, but if it receives a test vector with only three characters, it would crash.
- **Header Size**: $\in [38; 2^{64} - 1)$, meaning it must be at least 38. A smaller maximum value could be chosen if we wanted to impose restrictions, but for now, we use the given range. The value of 38 is derived from calculating the shortest possible valid header size (an empty caption with no tags). The value of $2^{64} - 1$ represents the largest number that can be found in the *header size* field. The naive implementation reads this field as a signed integer (using the `q` format string in the `struct` module instead of `Q`), causing it to interpret $2^{64} - 1$ as -1.
- **Content Size**: $\in [0; 2^{64} - 1)$, since the file may contain no pixels at all or the maximum possible number of pixels. The value of this field must match `width * height * 3`. The naive implementation reads this field as a signed number (`struct` format string).
- **Width**: $\in [0; 2^{64} - 1)$, subject to the same considerations as *content size*. The naive implementation reads this field as a signed number (`struct` format string).

- **Height**: $\in [0; 2^{64} - 1)$, subject to the same considerations as *content size*. The naive implementation reads this field as a signed number (`struct` format string).
- **Caption**: Can only contain ASCII characters. The naive implementation attempts to decode the read bytes as such, but lacks error handling (`try-except` in case a byte cannot be decoded as an ASCII character). To ensure that the caption does not contain a `\n` character, the file should be treated as if the caption has ended upon encountering the first such character (the naive implementation does this).
- **Tags**: Can only contain ASCII characters, and error handling is missing for this field as well. Each read character should be checked to ensure that none are `\n`. The naive implementation reads tags including `\0` at the end but does not display them in the graphical interface. It must be verified that the last tag also ends with `\0`!
- **Pixels**: The file must contain exactly as many pixels as indicated by the *content size* field. This can be implemented by regularly checking whether EOF has been reached: the file is invalid if the end is reached too soon or if EOF is missing after the expected number of pixels.

Example Code for Checking Magic Characters:

```python
# read the magic bytes
magic = ciff_file.read(4)
# read may not return the requested number of bytes
# TODO: magic must contain 4 bytes. If not, raise Exception
if len(magic) != 4:
    raise Exception("Invalid magic: length")
bytes_read += 4
# decode the bytes as 4 characters
try:
    new_ciff.magic = magic.decode('ascii')
except Exception as e:
    raise Exception("Invalid magic: non-ASCII")
# TODO: the magic must be "CIFF". If not, raise Exception
if new_ciff.magic != "CIFF":
    raise Exception("Invalid magic: value")
```

To complete the lab, modifications can be made in two places in the `ciff.py` file:

1. In the method belonging to the `CIFF.is_valid` property, where validation checks can be implemented after reading, or
2. In the `CIFF.parse_ciff_file()` method, where missing checks related to reading and format string errors must be fixed. The complete validation can also be implemented here.

There is a commented-out `try`-`except` block in the naive implementation of the `CIFF.parse_ciff_file()` method.
This block sets the `CIFF.is_valid` property to `false` whenever an exception occurs.
If you uncomment this `try`-`except` block, it will be enough to raise exceptions at the appropriate places, and the code will automatically classify the input as invalid.

To aid the implementation, several `TODO:` comments are present in the `CIFF.parse_ciff_file()` method, providing brief descriptions of the necessary validations.
Example files for testing can be found in the `test-vectors` folder.

The final implementation should be documented using the `moodle_submission.py` script. This script calls the `CIFF.parse_ciff_file()` method for each test vector and prints whether the code recognized the given file as valid or invalid based on the `CIFF.is_valid` property. If the code crashes for an input, this will also be indicated in the output.
Once the work is complete, run the `moodle_submission.py` script and use the output to fill in the Moodle questionnaire!
This script can also be executed via the GUI interface created by `view.py`.