Cybersecurity ML projects

That aren't spam detection

The full build guide // @hinumpy

This is the companion to the carousel. Four projects, four free datasets, no request forms and no waiting on an access committee. Every one of them is something a security team would actually recognize.

Security ml is one of the only spaces left where the bar for a student project is still low and the impression it makes is still high. Almost nobody your age has built one of these. That is the whole opportunity.

Read the setup and the metrics section first. They are short and they apply to all four projects. Then pick one. Do not try to build all four.

> **IF YOU ONLY BUILD ONE** Start with project 3, the phishing url classifier. It is the shortest, and the leakage check inside it is the single most impressive thing on this list. It is the one that makes an interviewer sit up.

Before you start

ENVIRONMENT

```
python -m venv venv
source venv/bin/activate      # windows: venv\Scripts\activate
pip install pandas scikit-learn matplotlib ucimlrepo
```

Python 3.10 or newer. Everything here runs on a laptop. Nothing on this list needs a gpu, and nothing on this list needs you to touch a live malicious file.

A NOTE ON SAFETY, BECAUSE IT WILL COME UP

None of these projects involve downloading, running, or handling live malware. Every dataset here ships pre-extracted features or already-captured logs. That is not a compromise, it is how the actual research field works. If someone asks you about it in an interview, that answer is a point in your favour.

The metrics section, read this twice

This is the part that separates a security ml project from a kaggle notebook, and it applies to all four projects below.

Accuracy is meaningless here. In any real security dataset the overwhelming majority of traffic, files, urls and log lines are benign. A model that predicts "benign" every single time and never detects anything will score in the high nineties. That number is worthless and a security engineer will spot it instantly.

Report these instead:

- **False positive rate.** Of everything that was actually benign, what fraction did you flag? This is the number that decides whether a tool is usable.
- **Precision at a fixed alert budget.** Pick a realistic number of alerts a human could review in a day, say 100. Take your top 100 highest-confidence detections. How many were real? That is the question a SOC actually asks.
- **Per-class recall.** Aggregate recall hides everything. Your model probably catches DDoS easily and misses infiltration completely. Break it down by attack type and show the breakdown.

The framing to use out loud: "an analyst has to look at every alert this produces. Here is how many false alarms per day my model would generate." If you can say that sentence about your own project you are already ahead of most people applying next to you.

```
from sklearn.metrics import classification_report, confusion_matrix
import numpy as np

# per-class breakdown, not one accuracy number
print(classification_report(y_test, preds, digits=4))

# false positive rate
tn, fp, fn, tp = confusion_matrix(y_test, preds).ravel()
print("fpr:", fp / (fp + tn))

# precision at an alert budget of 100
scores = model.predict_proba(X_test)[:, 1]
top = np.argsort(scores)[-100:]
print("precision@100:", y_test.values[top].mean())
```

Project 1 // attack traffic classifier

Time: 4 to 5 hours · difficulty: intermediate · most credible

You take five days of real captured network traffic and classify each flow as benign or as a specific attack: brute force, DoS, DDoS, port scan, botnet, web attack, infiltration, heartbleed.

THE DATASET

CICIDS2017, from the Canadian Institute for Cybersecurity at UNB. Free, public, no request form. It ships full packet captures plus a MachineLearningCSV bundle that is ready for pandas.

https://www.unb.ca/cic/datasets/ids-2017.html

> **THE WHOLE PROJECT** Do not use the original csv files. Use the corrected ones.

Researchers audited this dataset and found real problems in how the traffic was generated, how flows were constructed, how features were extracted and how everything was labeled. The tool that produced the original csvs mishandled TCP, terminating a flow after a single FIN packet and ignoring RST packets entirely. Their corrected pipeline reconstructed and relabeled more than twenty percent of the original flows.

So if you train on the raw csv that every tutorial uses, a meaningful chunk of what your model learns is somebody else's bug. The corrected dataset and the fixed extraction tool are both public:

https://intrusion-detection.distrinet-research.be/WTMC2021/tools_datasets.html

https://github.com/GintsEngelen/CICFlowMeter

BUILD IT

- Download the corrected flow dataset. Skim the changelog on that page, it tells you exactly what was wrong and when it was fixed.
- Load and clean. There are NaN and infinity values in the flow duration and rate features. Drop them and note how many you dropped.
- Drop the identifier columns before training: source ip, destination ip, port, timestamp, flow id. **Leaving them in is how people accidentally get 99% — the model just memorises the attacker's ip.**
- Split by time, not randomly. Train on the earlier days, test on the later ones. A random split lets near-identical flows from the same attack burst land on both sides.
- Start with a random forest or gradient boosting. Tree models are the standard baseline on flow features and they train in minutes.
- Evaluate with the metrics section above. Per-attack-class recall, false positive rate, precision at an alert budget.

THE PART THAT ACTUALLY MATTERS

Run it twice. Once on the original labels, once on the corrected labels. Put the two results side by side in your readme.

That comparison is the entire project. It shows you read past the tutorial, it shows you can evaluate a dataset instead of just consuming one, and it gives you something specific and true to talk about for ten minutes in an interview. Almost nobody does this.

HOW YOU KNOW YOU ARE DONE

- You can name which attack class your model is worst at, and give a reason why
- You have no ip addresses or ports in your feature set
- You can state your false positive rate as a number of alerts per day

STRETCH GOAL

A small matplotlib chart of per-class recall, original labels versus corrected, dropped straight into your readme.

RESUME BULLET TEMPLATE

Fill in your own real numbers. Run it and count.

"built a network intrusion classifier on [n] labeled flows from CICIDS2017 using corrected ground-truth labels; evaluated with per-attack-class recall and a fixed false-positive budget rather than aggregate accuracy, achieving [x]% recall on [attack class] at [y] false alerts per day"

Project 2 // malware detector

Time: 4 to 6 hours · difficulty: intermediate · biggest dataset

You predict whether a file is malicious or benign from its static properties alone. No execution, no sandbox, no risk.

THE DATASET

EMBER2024. It covers files first uploaded to VirusTotal between september 2023 and december 2024, with exactly 50,500 files sampled from each week. The training set is 2,626,000 files and the test set is 606,000, plus a separate challenge set of 6,315 malicious files. It includes seven types of labels, so you can do malicious-versus-benign, malware family classification, or behaviour prediction.

https://github.com/FutureComputing4AI/EMBER2024

Features and labels are hosted on HuggingFace and the repo has a download helper. It ships extracted features, not executables.

> **THE WHOLE PROJECT** The split is chronological, and that is the entire point.

The first 52 weeks are training. The last 12 weeks are test. So your model is evaluated on malware that is genuinely newer than anything it was trained on, which is exactly the situation a real detector is in every single day. Malware authors keep changing things, and a model that looked great last quarter quietly stops working.

This is called concept drift and it is one of the defining problems in security ml. This dataset hands it to you for free.

BUILD IT

- Start with a subset. Do not pull 2.6 million files on your first run. Take a few weeks of training data, get the pipeline working end to end, then scale up.
- Vectorize the features using the helper in the repo, then train LightGBM. It is the standard baseline for this task and it handles the feature scale well.
- Now do the thing that makes it a project: **Score your model week by week across the test period** instead of reporting one number for the whole test set.
- Plot detection rate against week. Look at whether it degrades.
- Then retrain including some recent data and plot it again. Show the difference.

THE PART THAT ACTUALLY MATTERS

The drift curve. One line chart, weeks along the bottom, detection rate up the side. If it slopes down, you have just demonstrated model decay on real data with your own hands.

The conclusion you write underneath it is the valuable part: "this model would need retraining every [n] weeks to hold performance." That is a maintenance argument, not a demo, and it is what the job is actually about.

HOW YOU KNOW YOU ARE DONE

- You have a per-week performance chart, not a single test score
- You can say how fast your model degrades and what you would do about it
- You also ran it on the challenge set and reported that separately

STRETCH GOAL

Use the family labels instead of the binary ones and find which malware families your detector misses most.

RESUME BULLET TEMPLATE

"trained a static malware classifier on [n] PE feature vectors from EMBER2024 using a chronological train/test split; measured concept drift across [n] weeks of unseen samples, showing a [x] point detection drop and establishing a [n]-week retraining cadence"

Project 3 // phishing url classifier

Time: 2 to 3 hours · difficulty: beginner · start here

You flag a phishing site from its url and page structure alone. This is the shortest project here and it contains the best lesson.

THE DATASET

PhiUSIIL, on the UCI machine learning repository. 235,795 urls, 134,850 legitimate and 100,945 phishing, across 54 features. About 14 MB. One pip install and one function call.

https://archive.ics.uci.edu/dataset/967/phiusiil+phishing+url+dataset

```
pip install ucimlrepo

from ucimlrepo import fetch_ucirepo
ds = fetch_ucirepo(id=967)
X = ds.data.features
y = ds.data.targets

# Careful: label 1 = legitimate, 0 = phishing. That is backwards
# From what most people assume. Check before you interpret anything.
```

> **THE WHOLE PROJECT** You will hit 99.9% accuracy in twenty minutes. That is the problem, not the achievement.

BUILD IT

- Load the data. Drop the FILENAME column, it is an identifier and carries no signal.
- Train a random forest. Take three minutes. Watch it score near-perfectly.
- **Stop and be suspicious.** Near-perfect accuracy on a real-world security problem is almost always a data problem, not a modelling win. Phishing detection is genuinely hard. The companies that do it for a living do not have this solved.

- Pull feature importances. **One feature will dominate everything else** — across published work on this dataset, URLSimilarityIndex consistently ranks as the most influential feature in every model people train.
- Drop it. Retrain. Record the honest number.
- Then work through the next few dominant features the same way and see how far the score falls.

THE PART THAT ACTUALLY MATTERS

The write-up. One honest paragraph in your readme: here is what i scored, here is why i did not believe it, here is what i found, here is what the model scores without it.

This demonstrates the single most valuable instinct in applied ml, which is distrusting your own good result. Most people never learn it. Some people never learn it professionally. If you can walk an interviewer through this, they will remember you, and it takes an afternoon.

> **DO NOT DO THIS** Do not put 99.9% on your resume or in a caption. Anyone who knows this dataset will know exactly what happened, and you will have advertised that you did not check. The lower honest number with the explanation is worth far more.

HOW YOU KNOW YOU ARE DONE

- You can name the features that were doing the work and explain why they leak
- You have two scores in your readme, the naive one and the honest one
- The honest one is lower and you are comfortable saying so out loud

STRETCH GOAL

Build a lexical-only version that uses nothing but the raw url string, no page content features at all. It will score much lower and it is much closer to what has to run at real-world speed.

RESUME BULLET TEMPLATE

"built a phishing url classifier on 235k labeled urls; identified and removed [n] leaking features that inflated accuracy to [x]%, and reported a corrected [y]% with per-feature ablation analysis documented in the repo"

Project 4 // log anomaly detector

Time: 5 to 6 hours · difficulty: intermediate · closest to the real job

You surface the abnormal sessions inside millions of lines of system logs. This is what a security operations centre does all day, and it is the least-built project on this list.

THE DATASET

LogHub, which collects system log datasets for log analytics research. Two to start with:

- **HDFS** — hadoop distributed filesystem logs, sliced into traces by block id, each trace labeled normal or anomaly. The sessions are already defined for you, which makes this the easier starting point.
- **BGL** — logs from the BlueGene/L supercomputer at Lawrence Livermore National Labs. Alert and non-alert messages are marked in the first column: a dash means non-alert, anything else is an alert.

https://github.com/logpai/loghub

Direct downloads:

https://zenodo.org/record/3227177/files/HDFS_1.tar.gz

https://zenodo.org/record/3227177/files/BGL.tar.gz

> **THE WHOLE PROJECT** Logs are not rows. They are sequences. Classify a line at a time and you have already lost.

A single log line is almost never anomalous by itself. The anomaly is in the pattern: an event that normally appears once appearing forty times, or two events arriving in an order they never arrive in. If you feed lines into a classifier independently you throw away the only signal that matters.

BUILD IT

- Parse the raw lines into **event templates**. "deleting block blk_-1608999687919862906" and "deleting block blk_7503483334202473044" are the same event with a different id. Collapse them into one template.
- Group into sequences. For HDFS, group by block id, the sessions come pre-defined. For BGL, use fixed time windows, sixty seconds is a common choice, and label a window anomalous if any line inside it is.
- Turn each sequence into a count vector: how many times each event template appears in that window.
- Start with the boring model. Isolation forest or a PCA-based detector on the count vectors. Get a baseline before you reach for anything deep.

- Evaluate on the false positive rate above everything else. An anomaly detector that flags five percent of a million-line log has generated fifty thousand alerts and is useless.

THE PART THAT ACTUALLY MATTERS

The windowing decision. How you group lines into sequences changes your results more than which model you pick, and you should test that claim rather than assert it.

Run your detector with three different window sizes and show what happens. That comparison is your project. "i tried thirty, sixty and three hundred second windows and here is the tradeoff" is a much better answer than any model architecture you could name.

HOW YOU KNOW YOU ARE DONE

- You can explain your windowing choice and defend it with your own numbers
- Your false positive rate translates to an alert volume a human could actually handle
- You tried both HDFS and BGL and can say why the same approach worked differently on each

STRETCH GOAL

A small dashboard that takes a log file and shows the flagged windows with the actual lines inside them, so a person could triage without leaving the screen.

RESUME BULLET TEMPLATE

"built a log anomaly detector over [n] million lines of HDFS and BGL system logs using template parsing and session-based windowing; compared [n] windowing strategies and reduced false positives from [x] to [y] per million lines at constant recall"

The shipping checklist

A project nobody can run is not a project. Before you call any of these done:

- A readme with a chart or a screenshot **at the top**. Recruiters spend seconds on a repo. Give them something to look at before they read a word.
- One command to run it. Python main.py. If it takes six steps, write a shell script.
- Your metrics table, **and not just accuracy**. False positive rate, per-class recall, precision at an alert budget.

- A sentence saying which dataset version you used and where you got it. For CICIDS2017 in particular, say explicitly that you used the corrected labels.
- Pinned versions in requirements.txt. Pip freeze > requirements.txt.
- Three sentences at the bottom on what you would do differently. Interviewers quote this section back to you more than any other.
- Do not commit the datasets. They are large and they are already public. Link them instead.

What to do once it is built

- Record a thirty second screen recording of it running. The actual thing, not a slideshow of code.
- Put the repo in your linkedin featured section, not buried in a projects list nobody scrolls to.
- Write one post about the moment your result looked too good and what you found. That story is more interesting than the build and it proves you did it yourself.
- Add the resume bullet with your real numbers filled in.

One last thing

Security is one of the few areas where a student project can still be genuinely uncommon. The reason so few people build these is not that they are hard, it is that nobody tells them the datasets are sitting right there, free, no form.

You do not need all four. One of these, finished, documented, with an honest metric and a paragraph explaining a result you did not trust, will do more for you than four half-built repos.

Pick one. Finish it properly. Then come back for the next one.

Go build something <3