garak, LLM vulnerability scanner

garak checks if an LLM can be made to fail in a way we don't want. garak probes for hallucination, data leakage, prompt injection, misinformation, toxicity generation, jailbreaks, and many other weaknesses. If you know nmap, it's nmap for LLMs.

garak's a free tool. We love developing it and are always interested in adding functionality to support applications.

What is garak?

- ☺ garak finds holes in LLM-based tech
  - We'll see more use of language models in technologies, systems, apps and services. We don't know how to secure language models, yet. garak works with language models and any tech using them to determine where the security holes can be with each solution.
- ☺ garak identifies how your LLM can fail
  - With dozens of different plugins, a hundred different probes, and tens of thousands of challenging prompts, garak tries hard to explore many different LLM failure modes.
- ☺ garak reports each failing prompt and response
  - Once garak finds something, the exact prompt, goal, and response is reported, so you get a full log of everything that's worth checking out and why it might be a problem.
- ☺ garak adapts itself over time
  - Each LLM failure found goes into a "hit log". These logs can then be used to train garak's "auto red-team" feature into finding effective exploitation strategies, meaning a more thorough testing and more chance of finding LLM security holes before anyone else does.

Our Features

- ☺ Direct focus on LLM security
  - garak focuses primarily on LLM security. While other tools might look at generic machine learning security, or app security, garak specifically focuses on risks that are inherent in and unique to LLM deployment, such as prompt injection, jailbreaks, guardrail bypass, text replay, and so on.
- ☺ Automated scanning
  - garak has a range of probes but doesn't need supervision - it will run each of these over the model, and manage things like finding appropriate detectors and handling rate limiting itself. You can override many aspects of the config to get a custom scan, but out of the box, it can do a full standard scan and report without intervention.
- ☺ Connect to many different LLMs
  - garak supports a ton of LLMs - including OpenAI, Hugging Face, Cohere, Replicate - as well as custom Python integrations. It's a community project, so even more LLM support is always coming.
- ☺ Structured reporting
  - garak keeps track of everything found, and outputs four kinds of log:
    - 1. Screen output - useful for monitoring scan progress; a precise description of what's happening at any time during the scan, including a list of everything on the schedule

        - 2. Report log - detailing the run down to every single prompt, response, and the evaluation of that response
        - 3. Hit log - describing each time that garak managed to get through and find a vulnerability
        - 4. Debug log - a logfile for troubleshooting and keeping track of garak's operations

- What is LLM security?
    - LLM security is the investigation of the failure modes of LLMs in use, the conditions that lead to them, and their mitigations.

Large language models can fail to operate as expected or desired in a huge number of ways; this means they can be insecure. On top of that, they need to run in software (like PyTorch, ONNX, or CUDA) - and that software can be insecure. Finally, the way that LLMs are deployed and their outputs are used can also fail when the LLM behaves in an unexpected way, which also presents a security risk. LLM security covers all this.

LLM security is broader than just things that are within existing security knowledge and existing LLM/NLP knowledge. LLM security covers not the intersection of Security and NLP, but the union of everything about information security and everything about natural language processing.

- Your first scan
    - Only run garak on systems that you have permission to use! It performs an exhaustive scan with some pretty strong prompts that might be misinterpreted.
- So you've installed garak - great! Let's run a scan to see what's up.
    - To keep things simple, let's try a straightforward target that could run on most machines - gpt2. Hugging Face Hub has a free and openly available version of this, so let's use that. And let's try a relatively straightforward probe, one that tries to get rude generations from the model.
        - First we'd like to see the list of probes available. Open a terminal if you don't have one up already, and run garak by entering the following then pressing enter:
- python -m garak --list_probes
    - You should get a list of probes in garak, with symbols at the side of some lines. You can always read more about garak's options by running python -m garak --help (or sometimes just garak --help, depending on your installation).
- Back to the list of probes. It might look like this.
    - garak LLM security probe v0.9.0.6 (https://github.com/leondz/garak) at 2023-07-24T11:20:16.086762
- probes: art 🌟
- probes: art.Tox
- probes: continuation 🌟
- probes: continuation.ContinueSlursReclaimedSlurs50
- probes: dan 🌟
- probes: dan.Ablation_Dan_11_0
- probes: dan.AntiDAN
- probes: dan.ChatGPT_Developer_Mode_RANTI
- probes: dan.ChatGPT_Developer_Mode_v2
- probes: dan.ChatGPT_Image_Markdown
- probes: dan.DAN_Jailbreak
- probes: dan.DUDE
- probes: dan.Dan_10_0
- probes: dan.Dan_11_0
- probes: dan.Dan_6_0

- probes: dan.Dan_6_2
- probes: dan.Dan_7_0
- probes: dan.Dan_8_0
- probes: dan.Dan_9_0
- probes: dan.STAN
- probes: encoding 🌟
- probes: encoding.InjectAscii85
- probes: encoding.InjectBase16
- probes: encoding.InjectBase2048
- probes: encoding.InjectBase32
- probes: encoding.InjectBase64
- probes: encoding.InjectBraille
- probes: encoding.InjectHex
- probes: encoding.InjectMime 💤
- probes: encoding.InjectMorse
- probes: encoding.InjectQP 💤
- probes: encoding.InjectROT13
- probes: encoding.InjectUU
- probes: glitch 🌟
- probes: glitch.Glitch 💤
- probes: glitch.Glitch100
- probes: goodside 🌟
- probes: goodside.ThreatenJSON
- probes: goodside.WhoIsRiley
- probes: goodside._Davidjl
- probes: knownbadsignatures 🌟
- probes: knownbadsignatures.EICAR
- probes: knownbadsignatures.GTUBE
- probes: knownbadsignatures.GTphish
- probes: leakreplay 🌟
- probes: leakreplay.LiteratureCloze 💤
- probes: leakreplay.LiteratureCloze80
- probes: leakreplay.LiteratureComplete 💤
- probes: leakreplay.LiteratureComplete80
- probes: lmrc 🌟
- probes: lmrc.Anthropomorphisation
- probes: lmrc.Bullying
- probes: lmrc.Deadnaming
- probes: lmrc.Profanity
- probes: lmrc.QuackMedicine
- probes: lmrc.SexualContent
- probes: lmrc.Sexualisation
- probes: lmrc.SlurUsage
- probes: malwaregen 🌟
- probes: malwaregen.Evasion
- probes: malwaregen.Payload
- probes: malwaregen.SubFunctions

- probes: malwaregen.TopLevel
- probes: misleading 🌟
- probes: misleading.FalseAssertion50
- probes: promptinject 🌟
- probes: promptinject.HijackHateHumans 💤
- probes: promptinject.HijackHateHumansMini
- probes: promptinject.HijackKillHumans 💤
- probes: promptinject.HijackKillHumansMini
- probes: promptinject.HijackLongPrompt 💤
- probes: promptinject.HijackLongPromptMini
- probes: realtoxicityprompts 🌟
- probes: realtoxicityprompts.RTPBlank
- probes: realtoxicityprompts.RTPFlirtation
- probes: realtoxicityprompts.RTPIdentity_Attack
- probes: realtoxicityprompts.RTPInsult
- probes: realtoxicityprompts.RTPProfanity
- probes: realtoxicityprompts.RTPSevere_Toxicity
- probes: realtoxicityprompts.RTPSexually_Explicit
- probes: realtoxicityprompts.RTPThreat
- probes: snowball 🌟
- probes: snowball.GraphConnectivity 💤
- probes: snowball.GraphConnectivityMini
- probes: snowball.Primes 💤
- probes: snowball.PrimesMini
- probes: snowball.Senators 💤
- probes: snowball.SenatorsMini
- probes: test 🌟
- probes: test.Blank 💤
- probes: xss 🌟
- probes: xss.MarkdownImageExfil

That's good! The stars 🌟 indicate a whole plugin; if we also garak to run them, it will run all the probes in that category. Except the disabled ones, marked with 💤. We can run disabled probes my naming them directly when we start a garak run, but they won't be selected automatically.

- Under lmrc we can see a probe named "lmrc.Profanity". Let's try this one. LMRC stands for "Language Model Risk Cards", and the probes in here come from a framework for assessing language model deployments. But for now let's just run the profanity probe.
    - To recap - we'll run a Hugging Face model, called gpt2, and use the lmrc.Profanity probe. So, our command line is:

python -m garak --model_type huggingface --model_name gpt2 --probes lmrc.Profanity

    - Typing that in and pressing enter should start the garak run! It will download gpt2 if you don't have it already - there'll be some progress bars about that - and then it will start the scan.
        - If all goes well, you should see a progress bar and then a number of lines each saying PASS or FAIL. Congratulations! You're run your first garak scan. The next section covers how to read these results.
- Reading the results

    - Here's the command line output from a sample garak run:
        - garak gpt2 scan with lmrc.Profanity
- Let's go line-by-line:
    - garak LLM security probe v0.9.0.6 (https://github.com/leondz/garak) at 2023-07-24T11:36:33.959494
        - This line tells us that garak has started, and gives the version number and the time that this run started for reference.
            - 📜 reporting to garak.01dc6fbe-af6f-4dcd-9a74-ae69928cad50.report.jsonl
- Here we're told the name of the file the report will be written to. This file is updated in real-time, so you can have a look inside it to find out what garak's doing (or even what it's planning to do). If you want, you can control the name of the file using the --report_prefix option.
- 🦜 loading generator: Hugging Face 🤗 pipeline: gpt2

Now we can see a few things. First, a parrot 🦜 to show we're loading a "generator" (what garak calls things like LLMs, that take text and give responses. Next we see that one of the Hugging Face generators is being loaded: specifically, the pipeline loader. Finally, we see that garak is going to use the gpt2 model from Hugging Face. This last part is the name of the model on Hugging Face Hub; you can see the webpage for Hugging Face gpt2 here, huggingface.co/gpt2.

- 🦹 queue of probes: lmrc.Profanity
    - The next thing garak is telling us is which probes it's going to use, and the order. Here, just a single probe was specified - lmrc.Profanity - and so the probe queue has just this item. You can read more about lmrc.Profanity by running python -m garak --plugin_info probes.lmrc.Profanity.
        - lmrc.Profanity riskywords.SurgeProfanityAnimalReferences: PASS ok on 20/ 20

This is our first line of results! It says:

- The probe was lmrc.Profanity
    - The detector, used to identify failures, was riskyword.SurgeProfanityAnimalReferences. In this case, this detector was specified by the probe. It's a keyword-based detector
    - The generator (gpt2) passed the test
        - Out of 20 generations, 20 were OK
- Let's skip a line and find a failing entry.
    - lmrc.Profanity riskywords.SurgeProfanityMentalDisability: FAIL ok on 17/ 20 (failure rate: 15%)
- Here, they layout's pretty similar to the message with the passing test, but there are few things to note:
- Because this is from the same probe as the previous entries, it's results over the same generator outputs. The probe has run and got one set of results; multiple detectors run over that same set of results.
    - The detector here is different - it's riskywords.SurgeProfanityMentalDisability, another keyword-based detector from Surge.
- The generator failed this test
- Of the twenty outputs, 17 were OK
- This gives a failure rate of 15%
    - 📜 report closed :) garak.01dc6fbe-af6f-4dcd-9a74-ae69928cad50.report.jsonl
        - At the end of the run, garak has finished writing to the report and so closed it. You can look in this file to see what went wrong (and right). If you're only interested in the failures, have a look in the hit log instead; it has the same name as the report, but with "hitlog" instead of "report".
- ✔️ garak done: complete in 11.90s

And we're done! garak let's you know when the scan's complete, and how long it took.

- Examples
- https://docs.garak.ai/garak/examples
- Components

- https://docs.garak.ai/garak/garak-components

How garak runs

In a typical run, garak will read a model type (and optionally model name) from the command line, then determine which probe and detector plugins to run, start up a generator, and then pass these to a harness to manage the probing; an evaluator deals with the results. There are many modules in each of these categories, and each module provides a number of classes that act as individual plugins.

- *garak/probes/* - classes for generating interactions with LLMs
- *garak/detectors/* - classes for detecting an LLM is exhibiting a given failure mode
- *garak/evaluators/* - assessment reporting schemes
- *garak/generators/* - plugins for LLMs to be probed
- *garak/harnesses/* - classes for structuring testing
- *resources/* - ancillary items required by plugins

The default operating mode is to use the **garak.harnesses.probewise** harness. Given a list of probe module names and probe plugin names, the probewise harness instantiates each probe, then for each probe reads its recommended_detectors attribute to get a list of detector s to run on the output.

Each plugin category (probes, detectors, evaluators, generators, harnesses) includes a base.py which defines the base classes usable by plugins in that category. Each plugin module defines plugin classes that inherit from one of the base classes. For example, **garak.generators.openai.OpenAIGenerator** descends from **garak.generators.base.Generator**.

Larger artefacts, like model files and bigger corpora, are kept out of the repository; they can be stored on e.g. Hugging Face Hub and loaded locally by clients using garak.

Usage

Installation

garak is a command-line tool. It's developed in Linux and OSX.

Friendly install instructions are at https://docs.garak.ai/garak/llm-scanning-basics/setting-up/installing-garak ; the instructions below should work, but you might need to be quite familiar with your OS to use them, because they assume some particular pieces of background knowledge.

Standard quick *pip* install

To use garak, first install it using pip:

pip install garak

Install development version with *pip*

The standard pip version of *garak* is updated periodically. To get a fresher version, from GitHub, try:

python3 -m pip install -U git+https://github.com/leondz/garak.git@main

For development: clone from *git*

You can also clone the source and run *garak* directly. This works fine and is recommended for development.

garak has its own dependencies. You can to install *garak* in its own Conda environment:

```
conda create --name garak "python>=3.9,<3.12"
conda activate garak
gh repo clone leondz/garak
cd garak
python3 -m pip install -r requirements.txt
```

OK, if that went fine, you're probably good to go!

Running garak

The general syntax is:

python3 -m garak <options>

garak needs to know what model to scan, and by default, it'll try all the probes it knows on that model, using the vulnerability detectors recommended by each probe. You can see a list of probes using:

python3 -m garak --list_probes

To specify a generator, use the --model_name and, optionally, the --model_type options. Model name specifies a model family/interface; model type specifies the exact model to be used. The "Intro to generators" section below describes some of the generators supported. A straightfoward generator family is Hugging Face models; to load one of these, set --model_name to huggingface and --model_type to the model's name on Hub (e.g. "RWKV/rwkv-4-169m-pile"). Some generators might need an API key to be set as an environment variable, and they'll let you know if they need that.

garak runs all the probes by default, but you can be specific about that too. --probes promptinject will use only the PromptInject framework's methods, for example. You can also specify one specific plugin instead of a plugin family by adding the plugin name after a .; for example, --probes lmrc.SlurUsage will use an implementation of checking for models generating slurs based on the Language Model Risk Cards framework.

Examples

Probe ChatGPT for encoding-based prompt injection (OSX/*nix) (replace example value with a real OpenAI API key):

```
export OPENAI_API_KEY="sk-123XXXXXXXXXXX"
python3 -m garak --model_type openai --model_name gpt-3.5-turbo --probes encoding
```

See if the Hugging Face version of GPT2 is vulnerable to DAN 11.0:

```
python3 -m garak --model_type huggingface --model_name gpt2 --probes dan.Dan_11_0
```

Contributing

Getting ready

The best and only way to contribute to garak is to start by getting a copy of the source code. You can use github's fork function to do this. Once you're done, make a pull request to the main repo and describe what you've done – and we'll take it from there!

Developing your own plugins

- Take a look at how other plugins do it
- Inherit from one of the base classes, e.g. **garak.probes.base.Probe**
- Override as little as possible
    - **You can test the new code in at least two ways:**
        - **Start an interactive Python session**
- Import the model, e.g. *import garak.probes.mymodule*
- Instantiate the plugin, e.g. *p = garak.probes.mymodule.MyProbe()*
        - **Run a scan with test plugins**
- For probes, try a blank generator and always.Pass detector: *python3 -m garak -m test.Blank -p mymodule -d always.Pass*
- For detectors, try a blank generator and a blank probe: *python3 -m garak -m test.Blank -p test.Blank -d mymodule*
- For generators, try a blank probe and always.Pass detector: *python3 -m garak -m mymodule -p test.Blank -d always.Pass*
- Get *garak* to list all the plugins of the type you're writing, with *–list_probes*, *–list_detectors*, or *–list_generators*

Tests

garak supports pytest tests in garak/tests. You can run these with python -m pytest tests/ from the root directory. Please write running tests to validate any new components or functions that you add. They're pretty straightforward - you can look at the existing code to get an idea of how to write these.

Finding help and connecting with the garak community

There are a number of ways you can reach out to us! We'd love to help and we're always interested to hear how you're using garak.

- GitHub discussions: https://github.com/leondz/garak/discussions
- Twitter: https://twitter.com/garak_llm
- Discord: https://discord.gg/xH3rs3ZH4B

Reporting

By default, garak outputs a JSONL file, with the name garak.<uuid>.report.jsonl, that stores outcomes from a scan. garak provides a CLI option to further structure this file for downstream consumption. The open data schema of AI vulnerability Database (AVID) is used for this purpose.

The syntax for this is as follows:

python3 -m garak -r <path_to_file>

Examples

As an example, let's load up a garak report from scanning gpt-3.5-turbo-0613.

wget
https://gist.githubusercontent.com/shubhobm/9fa52d71c8bb36bfb888eee2ba3d18f2/raw/ef1808e6d3b26002d9b046e6c120d438adf49008/gpt35-0906.report.jsonl
python3 -m garak -r gpt35-0906.report.jsonl

This produces the following output.

📜 Converting garak reports gpt35-0906.report.jsonl
📜 AVID reports generated at gpt35-0906.avid.jsonl

CLI reference for garak

garak LLM security probe v0.9.0.11.post1 (https://github.com/leondz/garak) at 2024-01-26T14:56:34.622801
usage: python -m garak [-h] [--verbose] [--report_prefix REPORT_PREFIX]
[--narrow_output]
[--parallel_requests PARALLEL_REQUESTS]
[--parallel_attempts PARALLEL_ATTEMPTS] [--seed SEED]
[--deprefix] [--eval_threshold EVAL_THRESHOLD]
[--generations GENERATIONS] [--config CONFIG]
[--model_type MODEL_TYPE] [--model_name MODEL_NAME]
[--generator_option_file GENERATOR_OPTION_FILE | --generator_options GENERATOR_OPTIONS]

```
                [--probes PROBES]
                [--probe_option_file PROBE_OPTION_FILE | --probe_options PROBE_OPTIONS | --probe_tags PROBE_TAGS]
                [--detectors DETECTORS] [--extended_detectors]
                [--buff BUFF] [--taxonomy TAXONOMY]
                [--plugin_info PLUGIN_INFO] [--list_probes]
                [--list_detectors] [--list_generators] [--list_buffs]
                [--list_config] [--version] [--report REPORT]
                [--interactive] [--generate_autodan] [--interactive.py]

LLM safety & security scanning tool

options:
  -h, --help            show this help message and exit
  --verbose, -v         add one or more times to increase verbosity of output
                        during runtime
  --report_prefix REPORT_PREFIX
                        Specify an optional prefix for the report and hit logs
  --narrow_output       give narrow CLI output
  --parallel_requests PARALLEL_REQUESTS
                        How many generator requests to launch in parallel for
                        a given prompt. Ignored for models that support
                        multiple generations per call.
  --parallel_attempts PARALLEL_ATTEMPTS
                        How many probe attempts to launch in parallel.
  --seed SEED, -s SEED  random seed
  --deprefix            remove the prompt from the front of generator output
  --eval_threshold EVAL_THRESHOLD
                        minimum threshold for a successful hit
  --generations GENERATIONS, -g GENERATIONS
                        number of generations per prompt
  --config CONFIG       YAML config file for this run
  --model_type MODEL_TYPE, -m MODEL_TYPE
                        module and optionally also class of the generator,
                        e.g. 'huggingface', or 'openai'
  --model_name MODEL_NAME, -n MODEL_NAME
                        name of the model, e.g.
                        'timdettmers/guanaco-33b-merged'
  --generator_option_file GENERATOR_OPTION_FILE, -G GENERATOR_OPTION_FILE
                        path to JSON file containing options to pass to
                        generator
  --generator_options GENERATOR_OPTIONS
                        options to pass to the generator
  --probes PROBES, -p PROBES
                        list of probe names to use, or 'all' for all
                        (default).
  --probe_option_file PROBE_OPTION_FILE, -P PROBE_OPTION_FILE
                        path to JSON file containing options to pass to probes
  --probe_options PROBE_OPTIONS
                        options to pass to probes, formatted as a JSON dict
```

```
  --probe_tags PROBE_TAGS
                only include probes with a tag that starts with this
                value (e.g. owasp:llm01)
  --detectors DETECTORS, -d DETECTORS
                list of detectors to use, or 'all' for all. Default is
                to use the probe's suggestion.
  --extended_detectors  If detectors aren't specified on the command line,
                should we run all detectors? (default is just the
                primary detector, if given, else everything)
  --buff BUFF, -b BUFF  buff to use
  --taxonomy TAXONOMY   specify a MISP top-level taxonomy to be used for
                grouping probes in reporting. e.g. 'avid-effect',
                'owasp'
  --plugin_info PLUGIN_INFO
                show info about one plugin; format as
                type.plugin.class, e.g. probes.lmrc.Profanity
  --list_probes        list available vulnerability probes
  --list_detectors     list available detectors
  --list_generators    list available generation model interfaces
  --list_buffs         list available buffs/fuzzes
  --list_config        print active config info (and don't scan)
  --version, -V        print version info & exit
  --report REPORT, -r REPORT
                process garak report into a list of AVID reports
  --interactive, -I    Enter interactive probing mode
  --generate_autodan   generate AutoDAN prompts; requires --prompt_options
                with JSON containing a prompt and target
  --interactive.py     Launch garak in interactive.py mode
```

See https://github.com/leondz/garak

garak

garak._config

This module holds config values.

These are broken into the following major categories:

- system: options that don’t affect the security assessment
- run: options that describe how a garak run will be conducted
- plugins: config for plugins (generators, probes, detectors, buffs)
- transient: internal values local to a single garak execution

Config values are loaded in the following priority (lowest-first):

- Plugin defaults in the code

- Core config: from garak/resources/garak.core.yaml; not to be overridden
- Site config: from garak/garak.site.yaml
- Runtime config: from an optional config file specified manually, via e.g. CLI parameter
- Command-line options

Code

garak global config

***class*garak._config.GarakSubConfig**

Bases: **object**

***class*garak._config.TransientConfig**

Bases: **GarakSubConfig**

Object to hold transient global config items not set externally

args= ***None***

basedir=

PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/garak/checkouts/latest/docs/source/../../garak')

hitlogfile= ***None***

report_filename= ***None***

reportfile= ***None***

run_id= ***None***

starttime= ***None***

starttime_iso= ***None***

garak._config.load_base_config()→ None

garak._config.load_config(*site_config_filename='garak.site.yaml', run_config_filename=None*)→ None

garak._config.parse_plugin_spec(*spec: str, category: str, probe_tag_filter: str = ''*)→ List[str]

garak._plugins

This module manages plugin enumeration and loading. There is one class per plugin in garak. Enumerating the classes, with e.g. --list_probes on the command line, means importing each module. Therefore, modules should do as little as possible on load, and delay intensive activities (like loading classifiers) until a plugin's class is instantiated.

Code

Functions for working with garak plugins (enumeration, loading, etc)

garak._plugins.configure_plugin(*plugin_path: str, plugin: object*)→ object

garak._plugins.enumerate_plugins(*category: str = 'probes', skip_base_classes=True*)→ List[tuple[str, bool]]

A function for listing all modules & plugins of the specified kind.

garak's plugins are organised into four packages - probes, detectors, generators and harnesses. Each package contains a base module defining the core plugin classes. The other modules in the package define classes that inherit from the base module's classes.

enumerate_plugins() works by first looking at the base module in a package and finding the root classes here; it will then go through the other modules in the package and see which classes can be enumerated from these.

Parameters:
category (*str*) – the name of the plugin package to be scanned; should be one of probes, detectors, generators, or harnesses.

garak._plugins.load_plugin(*path*, *break_on_fail=True*)→ object

load_plugin takes a path to a plugin class, and attempts to load that class. If successful, it returns an instance of that class.

Parameters:

- **path** (*str*) – The path to the class to be loaded, e.g. "probes.test.Blank"
- **break_on_fail** (*bool*) – Should we raise exceptions if there are problems with the load? (default is True)

In garak, attempt objects track a single prompt and the results of running it on through the generator. Probes work by creating a set of garak.attempt objects and setting their class properties. These are passed by the harness to the generator, and the output added to the attempt. Then, a detector assesses the outputs from that attempt and the detector's scores are saved in the attempt. Finally, an evaluator makes judgments of these scores, and writes hits out to the hitlog for any successful probing attempts.

garak.attempt

Defines the Attempt class, which encapsulates a prompt with metadata and results

***class*garak.attempt.Attempt(*status=0*, *prompt=None*, *probe_classname=None*, *probe_params=None*, *targets=None*, *outputs=None*, *notes=None*, *detector_results=None*, *goal=None*, *seq=-1*)**

Bases: **object**

A class defining objects that represent everything that constitutes a single attempt at evaluating an LLM.

Parameters:

- **status** (*int*) – The status of this attempt; ATTEMPT_NEW, ATTEMPT_STARTED, or ATTEMPT_COMPLETE
- **prompt** (*str*) – The processed prompt that will presented to the generator
- **probe_classname** (*str*) – Name of the probe class that originated this Attempt
- **probe_params** (*dict, optional*) – Non-default parameters logged by the probe
- **targets** (*List(str), optional*) – A list of target strings to be searched for in generator responses to this attempt's prompt

- **outputs** (*List(str)*) – The outputs from the generator in response to the prompt
- **notes** (*dict*) – A free-form dictionary of notes accompanying the attempt
- **detector_results** (*dict*) – A dictionary of detector scores, keyed by detector name, where each value is a list of scores corresponding to each of the generator output strings in outputs
- **goal** (*str*) – Free-text simple description of the goal of this attempt, set by the originating probe
- **seq** (*int*) – Sequence number (starting 0) set in **garak.probes.base.Probe.probe()**, to allow matching individual prompts with lists of answers/targets or other post-hoc ordering and keying

as_dict()→ dict

Converts the attempt to a dictionary.

garak.cli

Flow for invoking garak from the command line

garak.cli.main(*arguments=[]*)→ None

garak.probes

garak's probes each define a number of ways of testing a generator (typically an LLM) for a specific vulnerability or failure mode.

- garak.probes
- garak.probes.atkgen
  - Tox
- garak.probes.base
  - Probe
- garak.probes.continuation
  - ContinueSlursReclaimedSlurs80
- garak.probes.dan
  - Ablation_Dan_11_0
  - AntiDAN
  - AutoDANProbe
  - ChatGPT_Developer_Mode_RANTI
  - ChatGPT_Developer_Mode_v2
  - ChatGPT_Image_Markdown
  - DAN_Jailbreak
  - DUDE
  - DanInTheWild
  - DanInTheWildMini
  - Dan_10_0

- Dan_11_0
- Dan_6_0
- Dan_6_2
- Dan_7_0
- Dan_8_0
- Dan_9_0
- STAN
- garak.probes.encoding
- BaseEncodingProbe
- InjectAscii85
- InjectBase16
- InjectBase2048
- InjectBase32
- InjectBase64
- InjectBraille
- InjectEcoji
- InjectHex
- InjectMime
- InjectMorse
- InjectNato
- InjectQP
- InjectROT13
- InjectUU
- InjectZalgo
- braille()
- load_payloads()
- morse()
- rot13()
- garak.probes.gcg
- GCG_Probe
- garak.probes.glitch
- Glitch
- Glitch100
- garak.probes.goodside
- Davidjl
- Tag
- ThreatenJSON
- WhoIsRiley
- garak.probes.knownbadsignatures
- EICAR

- GTUBE
- GTphish
- garak.probes.leakreplay
- GuardianCloze
- GuardianComplete
- LiteratureCloze
- LiteratureCloze80
- LiteratureComplete
- LiteratureComplete80
- NYTCloze
- NYTComplete
- garak.probes.lmrc
- Anthropomorphisation
- Bullying
- Deadnaming
- Profanity
- QuackMedicine
- SexualContent
- Sexualisation
- SlurUsage
- garak.probes.malwaregen
- Evasion
- Payload
- SubFunctions
- TopLevel
- garak.probes.misleading
- FalseAssertion50
- garak.probes.packagehallucination
- Python
- garak.probes.promptinject
- HijackHateHumans
- HijackHateHumansMini
- HijackKillHumans
- HijackKillHumansMini
- HijackLongPrompt
- HijackLongPromptMini
- constructor()
- load_data_all_attacks()
- garak.probes.realtoxicityprompts
- RTPBlank

- RTPFlirtation
- RTPIdentity_Attack
- RTPInsult
- RTPProfanity
- RTPSevere_Toxicity
- RTPSexually_Explicit
- RTPThreat
- load_local_data()
- local_constructor()
- rtp_challenging
- garak.probes.replay
- Repeat
- RepeatExtended
- garak.probes.snowball
- GraphConnectivity
- GraphConnectivityMini
- Primes
- PrimesMini
- Senators
- SenatorsMini
- garak.probes.test
- Blank
- garak.probes.xss
- MarkdownImageExfil

Details for each probe
https://reference.garak.ai/en/latest/garak.probes.html

garak.detectors

- garak.detectors
- garak.detectors.base
- Detector
- HFDetector
- StringDetector
- TriggerListDetector
- garak.detectors.always
- Fail
- Pass
- garak.detectors.continuation

- Continuation
- garak.detectors.dan
- AntiDAN
- DAN
- DANJailbreak
- DUDE
- DevMode
- MarkdownLink
- STAN
- garak.detectors.encoding
- DecodeApprox
- DecodeMatch
- garak.detectors.goodside
- Glitch
- PlainJSON
- RileyIsnt
- garak.detectors.knownbadsignatures
- EICAR
- GTUBE
- GTphish
- garak.detectors.leakreplay
- StartsWith
- garak.detectors.lmrc
- Anthro
- QuackMedicine
- garak.detectors.malwaregen
- AnyCode
- garak.detectors.misleading
- MustContradictNLI
- MustRefuteClaimKW
- MustRefuteClaimModel
- garak.detectors.mitigation
- MitigationBypass
- garak.detectors.packagehallucination
- PythonPypi
- garak.detectors.perspective
- Attack_On_Author
- Attack_On_Commenter
- Flirtation
- Identity_Attack

- Identity_Attack_Experimental
- Incoherent
- Inflammatory
- Insult
- Insult_Experimental
- Likely_To_Reject
- Obscene
- Profanity
- Profanity_Experimental
- Severe_Toxicity
- Severe_Toxicity_Experimental
- Sexually_Explicit
- Spam
- Threat
- Threat_Experimental
- Toxicity
- Toxicity_Experimental
- Unsubstantial
- constructor()
- detect()

- garak.detectors.promptinject
  - AttackRogueString
- garak.detectors.replay
  - RepeatDiverges
- garak.detectors.riskywords
  - LDNOOBW
  - OfcomOffensiveDiscriminatory
  - OfcomOffensiveGeneral
  - OfcomOffensiveMentalHealth
  - OfcomOffensiveRaceEthnic
  - OfcomOffensiveSexGender
  - OfcomOffensiveSexual
  - SurgeProfanityAnimalReferences
  - SurgeProfanityBodilyFluids
  - SurgeProfanityMentalDisability
  - SurgeProfanityPhysicalAttributes
  - SurgeProfanityPhysicalDisability
  - SurgeProfanityPolitical
  - SurgeProfanityRacialEthnic
  - SurgeProfanityReligious

- SurgeProfanitySexual
- SurgeProfanitySexualOrientationGender
- garak.detectors.snowball
- DontStartNo
- DontStartYes
- garak.detectors.specialwords
- SlursReclaimedSlurs
- garak.detectors.toxicity
- ToxicCommentModel
- garak.detectors.xss
- MarkdownExfilBasic
- MarkdownExfilContent

Details for each detector

https://reference.garak.ai/en/latest/detectors.html#

garak.generators

- garak.generators
- load_generator()
- garak.generators.base
- Generator
- garak.generators.cohere
- CohereGenerator
- garak.generators.function
- Multiple
- Single
- garak.generators.ggml
- GgmlGenerator
- garak.generators.guardrails
- NeMoGuardrails
- garak.generators.huggingface
- HFInternalServerError
- HFLoadingException
- HFRateLimitException
- InferenceAPI
- InferenceEndpoint
- Model
- OptimumPipeline
- Pipeline

- garak.generators.langchain
  - LangChainLLMGenerator
- garak.generators.octo
  - InferenceEndpoint
  - OctoGenerator
- garak.generators.openai
  - OpenAIGenerator
- garak.generators.nemo
  - NeMoGenerator
- garak.generators.nvcf
  - NvcfGenerator
- garak.generators.replicate
  - InferenceEndpoint
  - ReplicateGenerator
- garak.generators.rest
  - RESTRateLimitError
  - RestGenerator
- garak.generators.test
  - Blank
  - Repeat

Details for each generator

https://reference.garak.ai/en/latest/generators.html

garak.buffs

- garak.buffs
- garak.buffs.base
  - Buff
- garak.buffs.encoding
  - Base64
  - CharCode
- garak.buffs.low_resource_languages
  - LRLBuff
- garak.buffs.lowercase
  - Lowercase
- garak.buffs.paraphrase
  - Fast
  - PegasusT5

Details for each buff

https://reference.garak.ai/en/latest/buffs.html

garak.harnesses

- garak.harnesses
- garak.harnesses.base
  - Harness
- garak.harnesses.probewise
  - ProbewiseHarness
- garak.harnesses.pxd
  - PxD

Details for each harness

https://reference.garak.ai/en/latest/harnesses.html

garak.evaluators

- garak.evaluators
- garak.evaluators.base
  - Evaluator
  - ThresholdEvaluator
  - ZeroToleranceEvaluator
- garak.evaluators.maxrecall
  - MaxRecallEvaluator

Details for each evaluator

https://reference.garak.ai/en/latest/evaluators.html

garak.report

Defines the Report class and associated functions to process and export a native garak report

***class*garak.report.Report(*report_location*, *records=[]*, *metadata=None*, *evaluations=None*, *scores=None*)**

Bases: **object**

A class defining a generic report object to store information in a garak report (typically named *garak.<uuid4>.report.jsonl*).

Parameters:

- **report_location** (*str*) – location where the file is stored.
- **records** (*List[dict]*) – list of raw json records in the report file
- **metadata** (*dict*) – report metadata, storing information about scanned model

- **evaluations** (*pd.DataFrame*) – evaluation information at probe level
- **scores** (*pd.DataFrame*) – average pass percentage per probe
- **write_location** (*str*) – location where the output is written out.

export()

Writes out output in a specified format.

get_evaluations()

Extracts evaluation information from a garak report.

load()

Loads a garak report.