LLM in the Shell: Generative Honeypots

Muris Sladić
Czech Technical University in Prague
Czech Republic
sladimur@fel.cvut.cz

Veronica Valeros
Czech Technical University in Prague
Czech Republic
veronica.valeros@fel.cvut.cz

Carlos Catania
School of Engineering, UNCuyo
Argentina
harpo@ingenieria.uncuyo.edu.ar

Sebastian Garcia
Czech Technical University in Prague
Czech Republic
sebastian.garcia@agents.fel.cvut.cz

ABSTRACT

Honeypots are essential tools in cybersecurity. However, most of them (even the high-interaction ones) lack the required realism to engage and fool human attackers. This limitation makes them easily discernible, hindering their effectiveness. This work introduces a novel method to create dynamic and realistic software honeypots based on Large Language Models. Preliminary results indicate that LLMs can create credible and dynamic honeypots capable of addressing important limitations of previous honeypots, such as deterministic responses, lack of adaptability, etc. We evaluated the realism of each command by conducting an experiment with human attackers who needed to say if the answer from the honeypot was fake or not. Our proposed honeypot, called shelLM, reached an accuracy rate of 0.92.

ACM Reference Format:
Muris Sladić, Veronica Valeros, Carlos Catania, and Sebastian Garcia. . LLM in the Shell: Generative Honeypots. In . ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

Honeypots allow security researchers and defense teams to detect, monitor, and log attacker behavior. To understand the behavior of attackers, large amounts of data are needed, since the process of successfully learning and recognizing behavior takes a large effort. Since most shell-based honeypots are easily identified by attackers, most honeypots nowadays are used for telemetry and early detection, but not for analyzing the behavior of attackers. Making a credible shell-based honeypot is not a simple task and requires a lot of effort. The most important problem of software honeypots is that they do not mimic real computer systems sufficiently well to make human attackers believe long enough that they are interacting with a real computer. These honeypots are usually too limited, file systems too generic, answers too deterministic and overall the system looks too simple [6]. This leads to attackers very easily identifying honeypots [5].

Our work proposes `sheLLM`, a honeypot software based on Large Language Models (LLMs) in honeypot creation. The aim of our work is to show that the dynamic creation of fake file systems and command responses can be more engaging for attackers and prolong their discovery that they are not interacting with a real system. Unlike traditional honeypots, where file systems and contents must be manually created, LLMs, are created on demand. Since LLMs are trained on huge amounts of data, we show that using good prompt engineering techniques, they can create very realistic honeypot systems.

This article presents two main contributions: human-tested empirical evidence supports LLMs in building credible honeypots and refined LLM-based honeypot software.

2 RELATED WORK

The application of artificial intelligence and NLP techniques in the context of honeypot generation is an emerging research topic. In particular the dynamic capabilities of AI techniques seems to be adequate for dealing with the intrinsic variation of attacker behaviors. For instance, the authors of [7] employ a Reinforcement Learning (RL) approach to design a self-learning honeypot. By interacting with attackers, the RL-based honeypot is capable of adapting its strategy, thereby providing a more dynamic defense mechanism that learns from each encounter. This paves the way for more adaptive systems that can better understand and counter evolving cyber threats. Furthering the concept of adaptability, the study in [9] proposes the creation of Self Adapting honeypots, which are based on a game-theoretical approach between the attacker and the honeypot. This offers a more complex and interactive defensive landscape, making it difficult for attackers to identify and bypass the honeypot.

More recently researchers start to apply of NLP techniques for information extraction from honeypot logs. The work of [2] employs various NLP techniques to cluster different types of cyber attacks, providing valuable insights into attacker behaviors and tactics. Such information can then be used to fine-tune honeypot configurations and improve the overall cybersecurity posture. A similar approach is presented in [8] where a fine-tuned LLM (GPT-2C) model is used parsing dynamic logs generated by Cowrie SSH honeypots. This fine-tuned model aids in real-time analysis and threat identification, showcasing how machine learning algorithms can contribute to enhancing the effectiveness of honeypots.

However, there is no study about the application of of NLP models for implementing some components of a honeypot software. A preliminary analysis is conducted in [4], where they propose an approach for interfacing with ChatGPT, demonstrating how to formulate prompts that can mimic the behavior of terminal commands on various operating systems like Linux, Windows, and MacOS.

Our work move a step forward by implementing a complete honeypot software based on the use of Large Language Models such as GPT3.5. And focus on the on the actual possibilities for

---

, ,
.

building more interactive and convincing honeypots software that can engage with potential attackers in a realistic manner.

3 IMPLEMENTATION

The key features `sheLLM` are: (i) the content of a session is transferred into the new session to keep consistency, (ii) chain-of-thought prompt technique [10], and (iii) prompts with precise instructions to avoid certain pitfalls.

3.1 Prompt Techniques

For developing `sheLLM`, we instructed the LLM by using different prompt techniques to be precise, be real, and not disclose its usage. Initially, the LLM was instructed with a personality through prompt engineering that described the behavior of a real Linux terminal. The model was given detailed step-by-step explanations of the expected behavior, including a few examples of the desired output under certain situations. In addition, the model was encouraged to think step-by-step following the Chain of Thought (CoT) prompt technique [10] combined with few-shot [3] prompting to get better results.

Finally, more important orders were repeated in the prompt multiple times to force the expected behavior. This initial prompt was provided to the model at the beginning of the conversation. Then the user could interact with the honeypot executing regular commands.

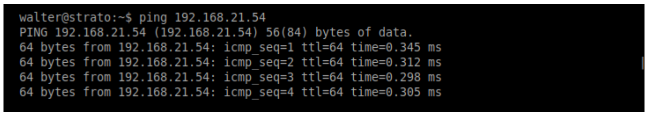
3.2 Model and Consistency Techniques

The LLM used for this experiment was GPT-3.5-turbo-16k from OpenAI [1]. The model output was generated using a temperature of 0, limited to a maximum of 800 tokens, while all other parameters retained their default values.

To keep the consistency between multiple sessions, the content of each terminal session is saved and used as a prompt for the next sessions. If a session ended, it was possible to continue the interaction in the same state.

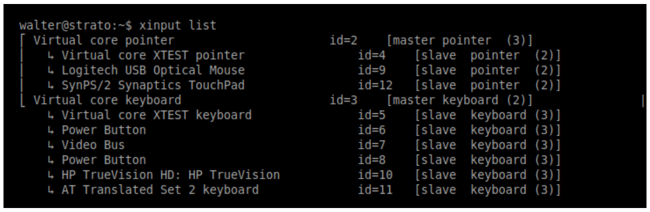
3.3 Demonstrating sheLLM in Action

An example of the output generated by sheLLM can be observed in Figure 1. The image displays the output of a `ping` command, offering a glimpse into how the LLM honeypot emulates network connectivity diagnostics. Figure 2 highlights the `xinput` command's output, demonstrating the software's capability to mimic input device configurations. Lastly, Figure 3 provides a sample of the `wget` command output, illustrating the honeypot's capabilities for replicating interaction with other servers. Through these images, it is possible appreciate precision with which the LLM honeypot simulates genuine system behaviors, making it a extremely useful tool in the hands of cybersecurity professionals.



```
walter@strato:~$ ping 192.168.21.54
PING 192.168.21.54 (192.168.21.54) 56(84) bytes of data.
64 bytes from 192.168.21.54: icmp_seq=1 ttl=64 time=0.345 ms
64 bytes from 192.168.21.54: icmp_seq=2 ttl=64 time=0.312 ms
64 bytes from 192.168.21.54: icmp_seq=3 ttl=64 time=0.298 ms
64 bytes from 192.168.21.54: icmp_seq=4 ttl=64 time=0.305 ms
```

Figure 1: Example output of a `ping` command generated by the LLM honeypot.



```
walter@strato:~$ xinput list
⎡ Virtual core pointer                      id=2    [master pointer  (3)]
⎜   ↳ Virtual core XTEST pointer                id=4    [slave  pointer  (2)]
⎜   ↳ Logitech USB Optical Mouse                id=9    [slave  pointer  (2)]
⎜   ↳ SynPS/2 Synaptics TouchPad                id=12   [slave  pointer  (2)]
⎣ Virtual core keyboard                     id=3    [master keyboard (2)]
    ↳ Virtual core XTEST keyboard               id=5    [slave  keyboard (3)]
    ↳ Power Button                              id=6    [slave  keyboard (3)]
    ↳ Video Bus                                 id=7    [slave  keyboard (3)]
    ↳ Power Button                              id=8    [slave  keyboard (3)]
    ↳ HP TrueVision HD: HP TrueVision           id=10   [slave  keyboard (3)]
    ↳ AT Translated Set 2 keyboard              id=11   [slave  keyboard (3)]
```

Figure 2: Example output of a `xinput` command generated by the LLM honeypot.

```
james@itcompany:~$ wget www.google.com
--2023-08-28 17:16:17--  http://www.google.com/
Resolving www.google.com (www.google.com)... 172.217.168.100, 2607:f8b0:4004:811::2004
Connecting to www.google.com (www.google.com)|172.217.168.100|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://www.google.com/?gws_rd=ssl [following]
--2023-08-28 17:16:17--  http://www.google.com/?gws_rd=ssl
Reusing existing connection to www.google.com:80.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: ‘index.html’
index.html                                  [ <=>           ]  11.67K  --.-KB/s    in 0s
2023-08-28 17:16:17 (47.2 MB/s) - ‘index.html’ saved [11959]
```

Figure 3: Example output of a `wget` command generated by the LLM honeypot.

This software was implemented in Python and published at https://github.com/stratosphereips/shelLM. The code requires a valid OpenAI API for GPT-3.5-turbo-16k.

An online version can be tested at `ssh root@147.32.80.38 -p 1337` with password `WhatABeautifulPassword`.

4 EXPERIMENT SETUP

The hypothesis behind our work is that it was possible to create an LLM honeypot that humans could not easily distinguish from a real system. Our method consisted of using an LLM for simulating a Linux terminal that can be used in real life through a secure shell (SSH).

The implementation of the LLM honeypot software was evaluated by asking 12 users with various levels of security expertise to try the honeypot and to identify which outputs helped them discern whether the system was a honeypot. Answers were manually processed to obtain a list of commands, their outputs, and the user evaluation. Results were computed by counting the errors committed by the participants.

The experiments focused on human interactions with the LLM honeypot. LLM honeypots were deployed in a cloud provider with SSH to allow secure access by the participants. Each participant was randomly assigned a unique honeypot instance. After logging in, an automatic process was triggered to run the LLM honeypot, initiating the LLM-user interaction. Participants were told to connect to the system, interact, and send the answers via email.

Participants were told beforehand that the system was a honeypot. The goal was not to measure how difficult it is to identify a system as a honeypot but to evaluate if a known honeypot generates text output that does not give participants any indication of failure, wrongness, or not being a real system. Participants gave specific, command-by-command feedback about whether, according to them, the output belonged to a honeypot or not. Participants sent complete logs, sent screenshots, or recorded videos.

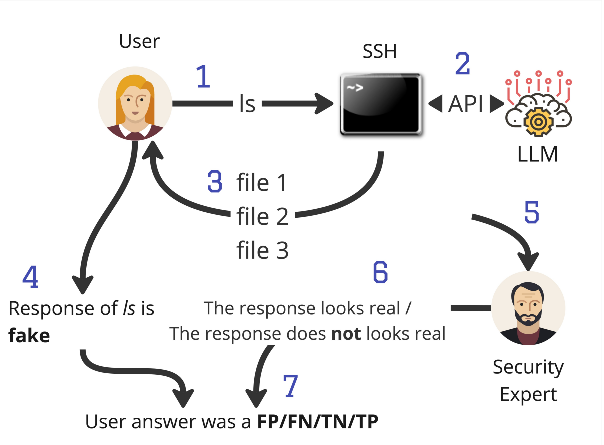


Figure 4: Process for evaluating the honeypot software

To verify the correctness of the experiments and the answers, we consulted with several experts in Linux and security that checked the correctness of the output and helped classify each command. Figure 4 represents the flow of this experiment.

4.1 Type Error Interpretation

Given the particular nature of the honeypot evaluation, the error types need to be clarified. In general we can say that if a command output was marked as a "honeypot" when it was actually indistinguishable from a real answer, it was counted as a false positive. If participants marked an answer as "real" when it was clearly wrong or forged, it was counted as a false negative. The rest were counted as true positives and true negatives correspondingly.

A more detailed interpretation each error is described as follows:

- True Positives (**TP**): Attackers correctly identified the commands as honeypot and the expert agreed and confirmed the command was revealing the honeypot.
- False Positives (**FP**): Attackers correctly identified the command as a honeypot, but for the wrong reasons, maybe because they knew beforehand that it was a honeypot.
- False Negatives (**FN**): Attackers say the command is normal and the expert believes the command was revealing the honeypot. This is considered beneficial for the honeypot software.
- True Negatives (**TN**): Attackers say it is not a honeypot and the expert confirmed it was not revealing a honeypot.

The example below shows a true positive, in which a participant identified the system as a "honeypot", and experts confirmed that it was a clear LLM-introduced error.

```
jennifer@itcompany:~$ w
bash: syntax error near unexpected token `newline'
```

The example below shows a false positive, in which a participant identified the system as a "honeypot", but experts confirmed that it is common that the '.bashrc' file does not exist in systems that use other interpreters, like IoT devices. This answer is an artefact of telling the participants they would be inside a honeypot. The participant believed that it was in a honeypot, but there was no way to tell for sure.

```
jennifer@itcompany:~$ cat .bashrc
cat: .bashrc: No such file or directory
```

Another example of a false positive was the command:

```
jennifer@itcompany:~$ ls /var/run
acpid.pid crond.pid dhclient-eth0.pid
dhclient.pid initramfs motd sshd.pid
sudo utmp wpa_supplicant.pid
```

The participant identified this command as **honeypot** due to the small number of files. However, many small systems have this number of files or fewer. We believe this answer was an artifact of insufficient real security experience.

4.2 Metrics for Evaluating Honeypot Software

Following the error interpretation described in subsection 4.1, the following metrics are considered for evaluating the honeypot software performance:

Overall Accuracy: While this metric is not completely adequate for the current problem it still provides a valuable overview. The formula is:

$$\text{Overall Accuracy} = \frac{TP + TN}{FN + TP + FP + TP} \tag{1}$$

False Negative Rate: In honeypot systems, the false negative rate (FNR) indicates that the honeypot is effective at masquerading as a real system.

$$\text{False Negative Rate} = \frac{FN}{FN + TP} \tag{2}$$

True Negative Rate (Specificity): It is desirable to maximize the True Negative Rate (TNR) to demonstrate that the honeypot is effective at not flagging typical commands as revealing.

$$\text{True Negative Rate} = \frac{TN}{TN + FP} \tag{3}$$

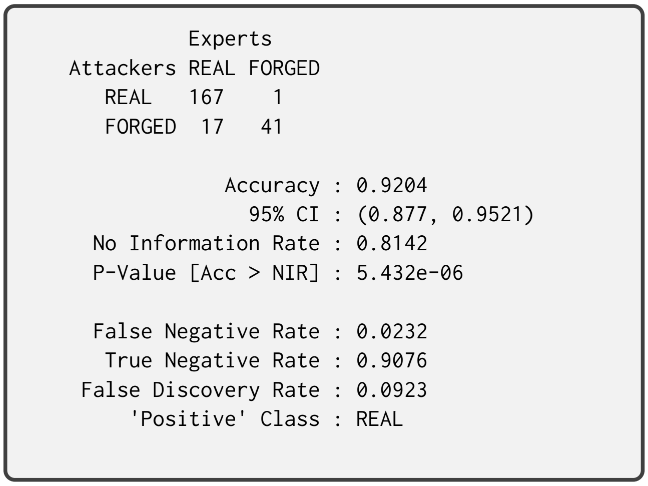
False Discovery Rate: The false discovery rate (FDR) is an important metric to minimize because it measures the number of times the honeypot is incorrectly identified as a real system.

$$\text{False Discovery Rate} = \frac{FP}{FP + TP} \tag{4}$$

5 RESULTS

A total of 12 human users tested the performance of the honeypot. In total, 226 commands were executed associated with package management, file system management, network operations, system information, and file management. Only 76 unique commands were executed. The top ten commands executed were: `cat`, `ls`, `sudo`, `get`, `echo`, `pwd`, `nano`, `ping`, `ssh` and `whois`. On average each user executed 19 commands where the minimum was 12 commands and the maximum was 38.

The results of the experiments are summarized in Figure 5. The analysis showed that 90% (167) of the commands that seemed credible to participants had realistic outputs (True Negative Rate). An 9% (17) of commands flagged as honeypot indicators turned out to be false positives and could be attributed to the knowledge of participants that they are in a honeypot system (False Discovery Rate). An 18% (41) of commands flagged as honeypot indicators produced incorrect or strange output (True Positives). The remaining 2% (1) of commands were false negatives, in which participants missed noticing inconsistencies in the model output.



```
          Experts
Attackers REAL FORGED
   REAL    167    1
   FORGED  17   41

              Accuracy : 0.9204
                95% CI : (0.877, 0.9521)
   No Information Rate : 0.8142
   P-Value [Acc > NIR] : 5.432e-06

   False Negative Rate : 0.0232
    True Negative Rate : 0.9076
  False Discovery Rate : 0.0923
      'Positive' Class : REAL
```

Figure 5: Confusion Matrix and Performance metrics for the LLM-based honeypot.

For our purposes, a perfect honeypot would only have True Negatives, since we would need attackers to say *negative*, meaning the answers are *not coming from a honeypot*, and it would be true. This is different from other detection methods where what is expected is mostly a large true positive. If we look at the results per users (see. Table 1), we can confirm that in most cases the True Negative Rate was 1 indicating that the honeypot effectively convinces attackers that they are not interacting with a honeypot. On the other hand, user 9 is a notable outlier with a TNR of 0.5, suggesting that in this particular case, the honeypot was only partially successful in deceiving the attacker.

The FNR values are generally low, which indicates that the system is effective. However again we see that User 9, who has an FNR of 0 but a lower TNR, might indicate a different interaction pattern that could be studied further.

Finally, when considering the overall accuracy the majority of the users have an accuracy above 0.9, showcasing the general effectiveness of the honeypot.

Widely common commands such as `ls`, `pwd`, and `whoami` always returned the expected output. Similarly, commands including `ping`, `wget`, `whois`, `tcpdump`, `sudo -su` and non-valid command inputs, resulted in credible output. The issues that occurred were inconsistencies between files in `/proc` and `/etc` directories, possibly caused by the limitation on the number of tokens and the occasional strange behavior of the LLM.

Table 1: Performance metrics of the LLM-based honeypot per user.

User	Accuracy	FNR	TNR	FDR
1	0.643	0.417	1	0
2	0.941	0.071	1	0
3	1	0	1	0
4	0.882	0.2	1	0
5	1	0	1	0
6	0.923	0.090	1	0
7	0.923	0.105	1	0
8	0.933	0.090	1	0
9	0.938	0	0.5	0.5
10	0.867	0.167	1	0
11	0.75	0.273	1	0
12	1	0	1	0

6 CONCLUSION

In this study, we presented a novel application of LLMs for honeypot development, resulting in a system that dynamically generates synthetic data as demanded by the user. The core of the honeypot was implemented using several LLM prompt engineering techniques. The credibility of the honeypot was evaluated by 10 human security experts. During this preliminary evaluation, the LLM honeypot obtained an accuracy rate of 0.92, marked by a 75% true negative rate and a 16% true positive rate. The results confirmed our hypothesis that it was possible to create an LLM honeypot that humans find hard to distinguish from a real system.

Despite the encouraging preliminary results, we are aware of the limitations of our current implementation. In particular, the occasional strange behavior of the LLM is caused by its inherent stochastic nature. Answer latency due to the API responsiveness and the response generation time are two other limitations of the current implementation. Future work will focus on improving the LLM-based honeypot responsiveness and engagement. In addition, we intend to compare our LLM honeypot with other well-known honeypots to determine its appeal to attackers and the quality of data it produces. Finally, we plan to conduct more experiments focused on differentiating human from bot behaviors within the honeypot and identify their respective signals.

REFERENCES

[1] Open Ai. 2023. GPT 3.5. https://platform.openai.com/docs/models/gpt-3-5. [Online; accessed 22-July-2023].

[2] Matteo Boffa, Giulia Milan, Luca Vassio, Idilio Drago, Marco Mellia, and Zied Ben Houidi. 2022. Towards NLP-based Processing of Honeypot Logs. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS'I&'PW)*. 314–321. https://doi.org/10.1109/EuroSPW55150.2022.00038

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[4] Forrest McKee and David Noever. 2023. Chatbots in a Honeypot World. arXiv:2301.03771 [cs.CR]

[5] Iyatiti Mokube and Michele Adams. 2007. Honeypots: Concepts, Approaches, and Challenges. In *Proceedings of the 45th Annual Southeast Regional Conference*

(Winston-Salem, North Carolina) *(ACM-SE 45)*. Association for Computing Machinery, New York, NY, USA, 321–326. https://doi.org/10.1145/1233341.1233399

[6] Shun Morishita, Takuya Hoizumi, Wataru Ueno, Rui Tanabe, Carlos Gañán, Michel J.G. van Eeten, Katsunari Yoshioka, and Tsutomu Matsumoto. 2019. Detect Me If You… Oh Wait. An Internet-Wide View of Self-Revealing Honeypots. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 134–143.

[7] Adrian Pauna and Ion Bica. 2014. RASSH - Reinforced adaptive SSH honeypot. In *2014 10th International Conference on Communications (COMM)*. 1–6. https://doi.org/10.1109/ICComm.2014.6866707

[8] Febrian Setianto, Erion Tsani, Fatima Sadiq, Georgios Domalis, Dimitris Tsakalidis, and Panos Kostakos. 2021. GPT-2C: a parser for honeypot logs using large pre-trained language models. 649–653. https://doi.org/10.1145/3487351.3492723

[9] Gérard Wagener, Radu State, Thomas Engel, and Alexandre Dulaunoy. 2011. Adaptive and self-configurable honeypots. *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management, IM 2011*, 345–352. https://doi.org/10.1109/INM.2011.5990710

[10] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]