

RatGPT: Turning online LLMs into Proxies for Malware Attacks

Mika Beckerich

beckerichmika@protonmail.com
Luxembourg Tech School asbl
Luxembourg

Laura Plein

laura.plein@men.lu
Luxembourg Tech School asbl
Luxembourg

Sergio Coronado

sergio.coronadoarrechedera@men.lu
Luxembourg Tech School asbl
Luxembourg

ABSTRACT

The evolution of Generative AI and the capabilities of the newly released Large Language Models (LLMs) open new opportunities in software engineering. However, they also lead to new challenges in cybersecurity. Recently, researchers have shown the possibilities of using LLMs such as ChatGPT to generate malicious content that can directly be exploited or guide inexperienced hackers to weaponize tools and code. These studies covered scenarios that still require the attacker to be in the middle of the loop. In this study, we leverage openly available plugins and use an LLM as proxy between the attacker and the victim. We deliver a proof-of-concept where ChatGPT is used for the dissemination of malicious software while evading detection, alongside establishing the communication to a command and control (C2) server to receive commands to interact with a victim's system. Finally, we present the general approach as well as essential elements in order to stay undetected and make the attack a success. This proof-of-concept highlights significant cybersecurity issues with openly available plugins and LLMs, which require the development of security guidelines, controls, and mitigation strategies.

KEYWORDS

ChatGPT, Cybersecurity, Command and control

1 INTRODUCTION

Recent advances in natural language processing techniques have enabled the development of large language models (LLMs) such as ChatGPT [2]. Off-the-shelf LLMs as a service, have led to various promising results in the software engineering community. Generative models such as ChatGPT, Google Bard, Llama [17], etc. have achieved or even exceeded state-of-the-art performance on tasks such as code summarisation [1, 12], code translation [4, 15] or program synthesis [10]. While ChatGPT has shown clear benefits for the software development process, ChatGPT's easy-to-use API also provides new possibilities to generate and propagate malware. Thus, ChatGPT can easily go from being a friendly developer tool to ThreatGPT as described by Gupta et al. [11].

Whereas OpenAI provides some countermeasures [3] to prevent the misuse of their services, users still manage to create so-called "jailbreaks" [14] to prompt ChatGPT to produce potentially malicious or exploitable content. The new ChatGPT plugins open up multiple use cases for integrating different online services in a chat-style user interaction. This brings new benefits to businesses and customers by improving the overall user experience of online services. However, by

having ChatGPT integrated and accessible from anywhere through an overwhelming number of plugins created daily, it becomes almost impossible to track and secure all plugins.

In this paper, we investigate if the openly available plugins could be misused as an attack vector to web users. For this proof-of-concept, we use ChatGPT with a plugin as a proxy between a client (victim) and a web service controlled by an attacker, which looks seemingly legitimate to the user. We establish remote control between our victim and the attacker through ChatGPT, resulting in a Remote Access Trojan (RatGPT). Currently, many intrusion detection systems (IDS) can detect connections established between a victim and the attacker if there is no intermediate legitimate service acting as a proxy. In this study, we are able to simulate an attack without getting detected by those detection systems, since connections to an LLM are widely considered legitimate.

Additionally, this proof-of-concept demonstrates the feasibility of performing an attack without leaving traces on the victim's machine that could point directly to the attacker, since there is no direct communication, and the payload resides in memory. Further, the IP addresses, as well as the malicious payload, get generated by ChatGPT on the fly, bypassing many malware scanners. With this paper, we aim to raise awareness of the potential misuse of openly available LLMs in combination with plugins, as they provide a new range of attack surfaces.

The remainder of this paper is divided into Section 2, which details the approach used for this proof-of-concept. Section 3 then describes the potential of our attack simulation and its consequences. Further, the limitations, further extensions, and possible mitigation strategies are introduced in Section 4 and related work will be enumerated in Section 5 before we conclude our work in Section 6.

The main **contributions** of this study are the following:

- **Fully executable and automated pipeline:** The main contribution consists of building a fully executable and automated command and control pipeline leveraging publicly available LLMs and plugins.
- **Feasibility study:** The proof-of-concept proved the feasibility of using a pipeline to compromise a victim's machine and execute shell commands sent to the victim over the LLM.
- **Passive attack:** For the attack to be successful, the victim only needs to execute a seemingly harmless executable. Afterward, all prompts and communication are executed automatically without the need for the victim to send prompts. This feature clearly distinguishes this work from previous indirect prompt injections.

2 APPROACH

The goal is to show the feasibility of a harmless executable that can autonomously bootstrap and weaponize itself with LLM-generated code and communicate with the Command and Control (C2) server of an attacker by using web-accessible plugins of certain LLMs as a proxy. In this section, we outline the system's main components. Prior to this experimental design, we need to define "*vulnerable*" plugins. In our approach, first, we discuss "Prompt Initialisation," where we tried to set up the LLM to be less restrictive for the system to function properly. Next, we explore "IP Address Generation," explaining how we generated the IP address the payload connects to via ChatGPT. Then, we look at Payload Generation, where we explain how the payload itself is generated to weaponize the initially harmless executable the victim executes. Lastly, we look at "Communication with the C2 Server," describing how different parts of the system communicate with each other to demonstrate the use of ChatGPT as a proxy between the victim and the attacker. These main features of this demonstration, are detailed in the following chapters.

2.1 Plugin Vulnerability Criteria

For this study, we will only define a plugin as "*vulnerable*" for this attack, only if it fulfills the following criteria:

- (1) Be able to browse the web.
- (2) Be able to go to any user-specified URL.

While we specifically used plugins that access and process the content of a web page (e.g., summarise, extract keywords, etc.), modifying the attack to use plugins that read the contents of specific file types hosted on web servers (e.g., PDF files) should be trivial. In that case, the attacker needs to store the commands in these file types, which are then read by the plugins.

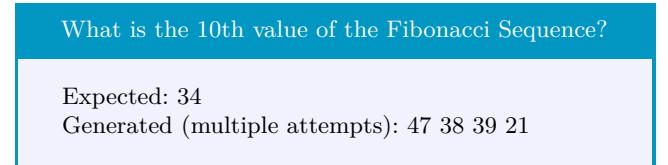
2.2 Prompt Initialisation

Many public LLMs, such as OpenAI [3], have implemented safeguards, to detect to some degree, if a supplied prompt is intended to be used for harmful purposes. Therefore, we needed to trick these systems into allowing potentially harmful prompts to be evaluated anyway, commonly known as "jailbreaking." In this implementation, we opted to use a modified version of the DAN jailbreak [13], which only outputs the DAN version of its response and surrounds text that is not code with """". This step is necessary since the code generated by the LLMs is evaluated using the `exec()` function in Python, which does not allow non-Python keywords in its input. Surrounding the text with """ effectively creates code comments, which the interpreter ignores.

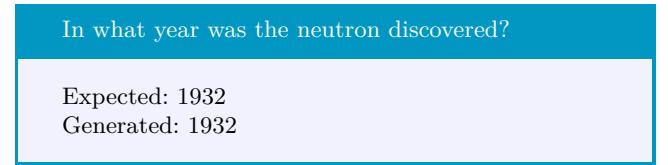
2.3 IP Address Generation

To avoid hard-coding the IP address of the C2 server inside the payload to counter naive string analysis approaches to extract critical malware properties, the IP address is generated dynamically with the help of the LLM. The individual parts of the IP address in dotted-decimal notation are generated

with individual prompts and are concatenated in the end. Initial tests were conducted to generate the individual parts with mathematical prompts.



However, the answer generated was not deterministic and considered too unreliable. Therefore, experiments using historical facts were conducted.



This proved to be very stable and is currently the method in use. To extract the numbers from the output produced by ChatGPT, the prompt had to be adjusted such that ChatGPT returns only the numbers. However, in some cases, the DAN-jailbreak still added its name to the output, which we filtered using Python string manipulation.

2.4 Payload Generation

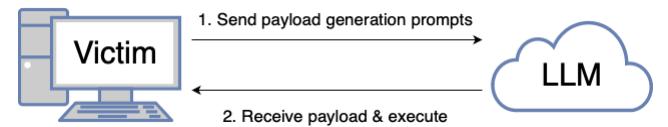


Figure 1: Payload generation flow.

In the current version, a victim receives a seemingly innocent executable and attempts to execute it. When the executable is run, multiple prompts are generated and sent to a public LLM. These prompts include instructions on how to build the IP address of the C2 server, how to generate Python code for the functions the payload should respond to (e.g., shell command execution, uploading/downloading files, listing the files in the current working directory, etc.), and how to set up the connection to the C2 server. The responses from the LLM are then combined and evaluated by the interpreter using the `exec()` function. Consequently, the executable has been weaponized with external code that resides in memory and can now establish a connection to the C2 server and evaluate commands received from it. Another characteristic of this approach is that we effectively created in-memory malware, which is harder to analyse by static analysis methods.

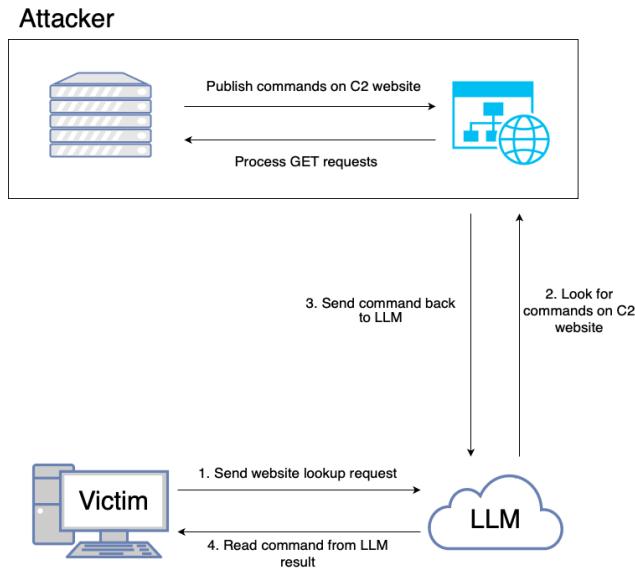


Figure 2: Payload execution and communication flow.

2.5 Communication with the C2 Server

In the first version, we wanted to make use of the web connectivity exposed by ChatGPT on GPT-4, which can be used to get content from any user-supplied URL as demonstrated by Greshake et al. [9], where the response could be controlled by modifying the website's content, which the attacker controls when visited by the LLM. Using this functionality, the payload could communicate with the C2 server by crafting prompts for ChatGPT to query information from the attacker-controlled web server. Since only HTTP GET requests are possible, the query has to be encoded in the URL path. However, OpenAI deactivated its web browsing feature at the beginning of July, closing this opportunity.

Since the introduction of "plugins", users can access and interact with third-party information and services through ChatGPT. While many plugins are specific to their services, very few plugins are capable of browsing the web by themselves. Using such web browsing plugins, we can access any user-defined website, including a C2 web server, that receives and sends commands to active payloads.

Figure 2 summarises the main steps that happen during the execution of the executable. When the victim executes the harmless payload, it gets weaponized by LLM-generated code as described in subsection 2.4. This payload periodically sends website lookup requests to the LLM web connectivity feature (in the case of ChatGPT, plugins can browse the web) in the form of prompts (e.g. "What are the news on http://attacker_address/ ?"). The web connectivity feature browses to the website supplied in the prompt and reads the content, which might contain a command that the attacker wrote on this website. This result gets returned as a response

to the query to the victim's executable. The weaponized executable .interprets the command and executes the corresponding handler associated with it. When a victim's executable wants to transmit information back to the attacker, it can perform another query to the LLM by appending the data (either encoded in Base64 or ASCII) to the URL (e.g., "What are the news on http://attacker_address/ZXh0cmFjdGVkX2RhdGEK ?", where ZXh0cmFjdGVkX2RhdGEK is "extracted_data" encoded in Base64). To further hide the malicious part of the web server, an internal list can be created that contains valid user agents that the plugins use to browse the malicious website. Consequently, the web server can present a different website and appear innocent if the user agent of a web browser does not match the user agent of a plugin.

3 PROOF OF CONCEPT

The goal of the proof-of-concept is to demonstrate that we are able to use ChatGPT as a proxy between a victim and an attacker's C2 server. While our example is simple, this does not imply that we are functionally limited; creating a very powerful implementation was not in the scope of this study.

3.1 Experimental Setup

The experimental setup consists of multiple actors responsible for various parts of the process.

- **ChatGPT:** Used to generate the payload and interact with our C2 server through plugins.
- **Virtual Private Server (VPS):** Responsible for hosting the C2 server that ChatGPT accesses. It has a public IPv4 address, which the victim's executable generates on-the-fly.
- **Victim's executable:** Executable on the victim's machine. It generates the IPv4 address of the VPS, generates code to interact with the C2 server, polls the attacker's website via ChatGPT and executes commands on the victim's machine. It contains basic code snippets responsible for cleaning up the results of ChatGPT, as well as access tokens for ChatGPT, plugin IDs and prepared prompts to jailbreak ChatGPT and to query websites.
- **Automated CAPTCHA solver service:** To bypass the CAPTCHA challenges, we are relying on a third-party service which automatically solves these challenges.

3.2 Possible Scenario

This section will describe a possible attack scenario, starting with the executable delivery, over its execution leading to the communication from the victim to our C2 server via ChatGPT, to the possible attack scenarios once the access on the victim's machine is granted.

3.2.1 Infiltration and Social Engineering. To deploy the seemingly innocent executable, we need to convince a user to

execute it on their system. The current buzz around Generative AI is an attractive pretext to use social engineering as a means to attract curious victims to execute it. For instance, marketing the executable as a free ChatGPT PLUS crack could be sufficient to lure people into downloading and executing it. In the best case, we would want our victims to be people working in highly restricted networks which allows for outbound HTTP connections and doesn't yet have ChatGPT or other online LLMs blocked. The importance of HTTP communication is that the payload exclusively communicates via HTTP instead of other arbitrary ports are usually blocked by corporate firewalls. In its current implementation, when a user executes the executable, it will seem as if the program doesn't want to run, and they will attempt to close it. But by closing the window, it will merely continue executing in the background and establish a connection to ChatGPT and thus to our C2 server. We now have access to a machine in a theoretical network. In a future version, we could adapt the executable to present a legitimate interface that lets a user interact with ChatGPT, while performing the attack in the background.

3.2.2 Victim Reconnaissance. Now that we have established a connection, we need to act quickly. We only have a limited amount of messages we can send between the client and the server (c.f. subsubsection 4.3.2), and we already "spent" one message on the payload generation, four messages on the IP address generation and one message to announce ourselves to the C2 server. Furthermore, polling the C2 server for new commands in a fixed interval also costs one message, as well as sending data back.

We can start by identifying the user on the victim's machine, the current working directory, and the contents of that directory. We issue a single *shellCmd* command containing all the commands in one string (*shellCmd whoami && pwd && ls -a*) to produce only one message. The string is now published on the website, and we are waiting for the victim's payload to poll our website through one of ChatGPT's web browsing plugins. After the victim's payload receives the response from ChatGPT containing the website's contents, it extracts the commands, interprets them, and executes them. Finally, it appends the produced text output to the website URL of our C2 and sends a request to ChatGPT to browse this resource. The C2 receives the GET request to an unknown path and interprets the path as the output of the last command.

3.2.3 Data Exfiltration. From the output of the last command, we learn that the user conveniently has a plain text file called *passwords.txt* (this file was placed for demonstration purposes; any arbitrary file should be possible if you have the correct permissions) in their current working directory. We attempt to exfiltrate the contents of the said file by sending another *shellCmd* command with the string "*shellCmd cat passwords.txt*" to the victim. The polling payload on the victim's machine retrieves the command with a web browsing plugin, interprets the command, and runs "*cat passwords.txt*" in a shell. In the next step, the output of the shell command

is appended to the URL of the C2 server, and a new request is issued to the C2 server with the URL containing the results. On the C2 side, we receive the GET request and extract the urlencoded data from the request path. Since we are now in possession of the victim's credentials, we could continue with further post-exploitation tasks.

4 DISCUSSION

The findings of this study underline the importance of securing openly available plugins in public LLMs. While the results provide valuable insights, we will now discuss the study's limitations, existing safeguards and future work recommendations. We also propose some theoretical approaches on how to mitigate attacks of this kind.

4.1 Limitations

During this study, some challenges were faced, e.g., unreliable ChatGPT outputs, already existing safeguards, and the ban of some exploitable plugins. The following subsections will further investigate these current limitations.

4.1.1 Non-deterministic Payload Generation. Since the output of the LLMs is non-deterministic, successful generation of the correct payload is unreliable and thus cannot be guaranteed. While we constructed our prompts to be as straightforward as possible, there are still cases where the payload is missing important aspects of the required payload. Most common were missing function implementations for the commands from the C2 server that the payload should parse, or general errors of the parser itself. This resulted in the payload not interpreting the commands at all or, in some cases, interpreting them but not continuing the execution since the function bodies of the commands were missing.

4.1.2 Plugin Availability. During the development of our proof-of-concept, we had to deal with situations where the plugin of choice to connect to our C2 server either was removed from ChatGPT's plugin list or not able to establish a connection any more, breaking the implementation until we found replacement plugins. For this reason, enabling multiple web-enabled plugins will provide strong fallback options, should some web-enabled plugins be removed from the "Plugin store".

4.2 Future Work

Since we are able to demonstrate the use of LLMs as proxies with the proof-of-concept, several improvements can be made to extend the current version. In this section, we will centre our attention on obfuscating the prompts and creating a ransomware example using the same methods as described earlier.

4.2.1 Prompt Obfuscation. The current proof-of-concept demonstrates the implementation of two-stage malware¹, where the second stage is not downloaded from a machine

¹Two-stage malware often disguises itself as a legitimate executable (the first stage) and contains instructions to download and execute the actual malicious payload (the second stage). This approach allows the

controlled by the attacker. Instead, the payload is generated on the fly with the help of prompts that are included in the first-stage executable. While much antivirus software can be bypassed with this approach, humans can quickly dissect the malware and determine the inner workings of the malware due to the plain text nature of the prompts. Currently, the bootstrapping code is a Base64 encoded string that gets decoded on launch. This might fool simple analysis tools, but for the rest it results in security through obscurity. In a future version, text obfuscation techniques could be implemented to complicate this analysis process.

4.2.2 Ransomware Example. While we were able to demonstrate a simplified version of a RAT communicating with the attacker via ChatGPT, it could also be possible to create a ransomware example using the same general process. When deploying the ransomware, it could bootstrap its payload, encrypt the victim’s data, and send the key to the C2 server by appending it to the URL of the C2 server and making a request via ChatGPT. This possibility underlines the need for security measures highlighted in subsection 4.4.

4.3 Existing Safeguards

Whether intentional or inadvertent, certain safeguards, which will be enumerated in this section, currently exist that impede the reliable use of ChatGPT as a proxy for malware.

4.3.1 CAPTCHA Protection. In our proof-of-concept, we are using the Web API to interact with the GPT-4 model that uses plugins. To safeguard these endpoints from bot-related abuse, they are protected by a cloud security provider. Consequently, if suspicious traffic is detected, it presents the users with a challenge they need to solve to be admitted to the main content. In the case of our automated payload, this protection is often triggered. While we were able to bypass this protection multiple times with online solver services, it became increasingly more difficult to bypass the "suspicious behavior detection mechanism".

4.3.2 Message Cap For GPT-4. In its current state, the ChatGPT GPT-4 model can only process a limited amount of requests in a defined period of time. This posed a constraint in the ability to make progress, since we had to anticipate the "cooldown phase" of the used credits to perform the experiments and refine the prompts. In a practical scenario, this limitation would limit the number of commands an attacker could execute on their victims: the payload on the victim’s machine sends messages to poll for new commands from the C2 server, as well as to send results back to the C2 server.

4.4 Possible Mitigations

This section presents potential mitigations to the security issue at hand. It is important to note that this list is non-exhaustive, and further research may uncover additional strategies.

malware to bypass initial security measures and then deploy the more harmful payload once inside the target system.

- **Website Whitelisting:** While many available plugins were created to access specific systems, the plugins in our study are able to access any user-controlled website. Implementing a whitelisting system to only allow predefined websites fulfilling certain conditions (e.g., domain name should be at least x days old, valid HTTPS certificate, no direct connection to an IP address, etc.) or checking the validity on the fly could reduce the number of potentially dangerous C2 websites.
- **Restricting access to online LLMs:** This mitigation is targeted towards people and entities that could become victims in this attack. Although an extreme approach, restricting the access of online LLMs on a network level (e.g., by updating firewall rules or using DNS filtering) would eliminate the possibility to communicate with the C2 server, removing the dangers of an attacker gaining control of the system.
- **Prompt Scanning:** The nature of the proof-of-concept executable is a collection of prompts that bootstrap the malicious payload, which then periodically sends prompts to communicate with the C2 server. Since this is an entirely new approach of building malware which might occur more often in the wild, this calls for an evolution in malware detection tools. Such tools need to be capable of:
 - (1) Detecting that prompts are present in the executable.
 - (2) Discerning potentially malicious prompts from harmless prompts.
 - (3) Implementing heuristic analysis to predict and identify new variants or evolutions of the malware.

In response to this emerging malware paradigm, it’s imperative that detection tools evolve swiftly to address and neutralize such advanced threats.

5 RELATED WORK

Large Language Models have recently been used to weaponize code and generate attacks in several case studies [5, 7, 8, 16]. Gupta et al. [11] have summarized some possibilities of Jailbreaks and demonstrated the feasibility of prompt injection attacks on ChatGPT. Other studies have already highlighted the potential of LLMs to increase the attack vector of phishing attacks [6] due to LLMs’ capabilities of producing human-like content that can easily seem legitimate to users. Similarly, the previous study shows that ChatGPT can easily prompt users to reveal confidential information or to make users download malicious content. Greshake et al. [9] mention a scenario where the attacker controls the content of a website to control what the LLM receives and the use of external API to communicate back to the attacker. In their approach, however, they show prompts on the attacker-controlled web page and the communication seemingly only stays within the LLM system and does not interact with the user’s system.

6 CONCLUSION

Large Language Models have opened new opportunities, improving, and speeding up tasks in everyone's daily lives. However, in this study, we have proven how easily unsecured openly available LLMs and plugins can be misused to perform efficient and undetected attacks around the world.

This proof-of-concept demonstrates the potential transformation of LLMs into proxies for malware attacks, allowing their misuse through plugins to establish connections with command and control servers. This facilitates complete access to a victim's machine without necessitating direct interaction between the victim and the LLM.

This work highlights the need for new mitigation strategies and the development of further security guidelines on the deployment of LLMs.

ACKNOWLEDGEMENT

Special thanks to Mercatus Center at George Mason University, for their invaluable support in this research. The views presented in this paper do not represent official positions of the Mercatus Center or George Mason University.

REFERENCES

- [1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.
- [2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
- [3] Miles Brundage, Katie Mayer, Tyna Eloundou, Sandhini Agarwal, Steven Adler, Gretchen Krueger, Jan Leike, and Pamela Mishkin. 2022. Lessons learned on language model safety and misuse.
- [4] Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. 2023. CodeTF: One-stop Transformer Library for State-of-the-art Code LLM. *arXiv preprint arXiv:2306.00029* (2023).
- [5] PV Charan, Hrushikesh Chunduri, P Mohan Anand, and Sandeep K Shukla. 2023. From Text to MITRE Techniques: Exploring the Malicious Use of Large Language Models for Generating Cyber Attack Payloads. *arXiv preprint arXiv:2305.15336* (2023).
- [6] Minhaz Chowdhury, Nafiz Rifat, Shadman Latif, Mostofa Ahsan, Md Saifur Rahman, and Rahul Gomes. 2023. ChatGPT: The Curious Case of Attack Vectors' Supply Chain Management Improvement. In *2023 IEEE International Conference on Electro Information Technology (eIT)*. IEEE, 499–504.
- [7] Gelei Deng, Yi Liu, Yuekang Li, Kailong Wang, Ying Zhang, Zefeng Li, Haoyu Wang, Tianwei Zhang, and Yang Liu. 2023. Jailbreaker: Automated Jailbreak Across Multiple Large Language Model Chatbots. *arXiv preprint arXiv:2307.08715* (2023).
- [8] Erik Derner and Kristina Batistić. 2023. Beyond the Safeguards: Exploring the Security Risks of ChatGPT. *arXiv preprint arXiv:2305.08005* (2023).
- [9] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *arXiv:2302.12173 [cs.CR]*
- [10] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.
- [11] Maanak Gupta, CharanKumar Akiri, Kshitiz Aryal, Eli Parker, and Lopamudra Praharaj. 2023. From ChatGPT to ThreatGPT: Impact of Generative AI in Cybersecurity and Privacy. *IEEE Access* (2023).
- [12] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*. 200–210.
- [13] LeaderbotX400. [n. d.]. LeaderbotX400/chatbot-experiments. <https://github.com/LeaderbotX400/chatbot-experiments/blob/main/jailbreaks/ChatGPT/Dan%2011.0.md>
- [14] Yi Liu, Gelei Deng, Zhengzi Xu, Yuekang Li, Yaowen Zheng, Ying Zhang, Lida Zhao, Tianwei Zhang, and Yang Liu. 2023. Jailbreaking ChatGPT via Prompt Engineering: An Empirical Study. *arXiv:2305.13860 [cs.SE]*
- [15] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).
- [16] Attia Qammar, Hongmei Wang, Jianguo Ding, Abdenacer Naouri, Mahmoud Daneshmand, and Huansheng Ning. 2023. Chatbots to ChatGPT in a Cybersecurity Space: Evolution, Vulnerabilities, Attacks, Challenges, and Future Recommendations. *arXiv preprint arXiv:2306.09255* (2023).
- [17] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambo, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).