

Using GenAI in and inside your code, what could possibly go wrong?

Niels Tanis

Sr. Principal Security Researcher

VERACODE



Who am I?

- Niels Tanis
- Sr. Principal Security Researcher
 - Background .NET Development,
Pentesting/ethical hacking,
and software security consultancy
 - Research on static analysis for .NET apps
 - Enjoying Rust!
- Microsoft MVP – Developer Technologies

VERACODE



VERACODE

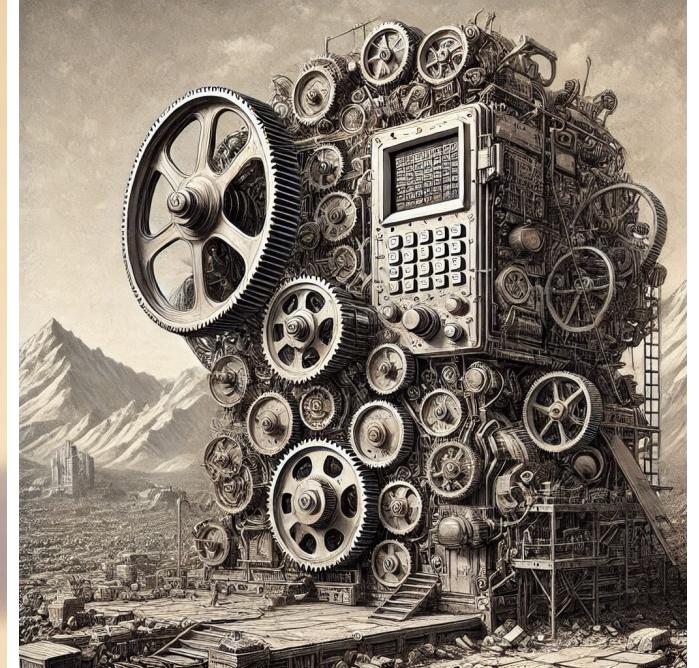


@niels.fennec.dev



@nielstanis@infosec.exchange

Generative AI



Generative AI



AI Security Risks Layers

AI Usage Layer

AI Application Layer

AI Platform Layer

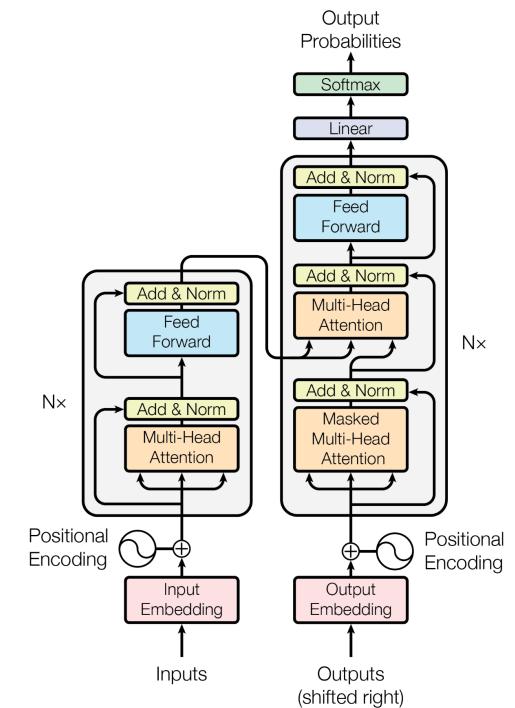


Agenda

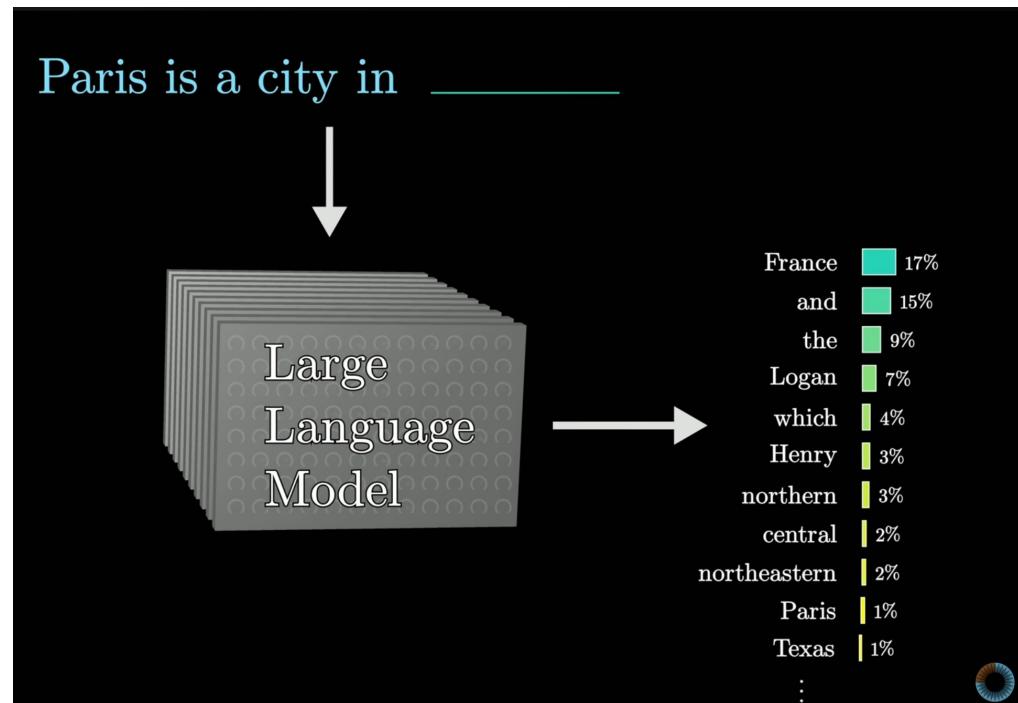
- Brief intro on LLM's
- Using GenAI LLM on your code
- Integrating LLM in your application
- Building LLM's & Platforms
- What's next?
- Conclusion Q&A

Large Language Models

- Auto regressive transformers
- Next word prediction based on training data corpus
- Facts and patterns with different frequency and/or occurrences
- “Attention Is All You Need” →



Large Language Models

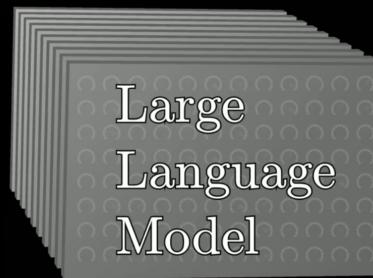


Large Language Models

What follows is a conversation between a user and a helpful, very knowledgeable AI assistant.

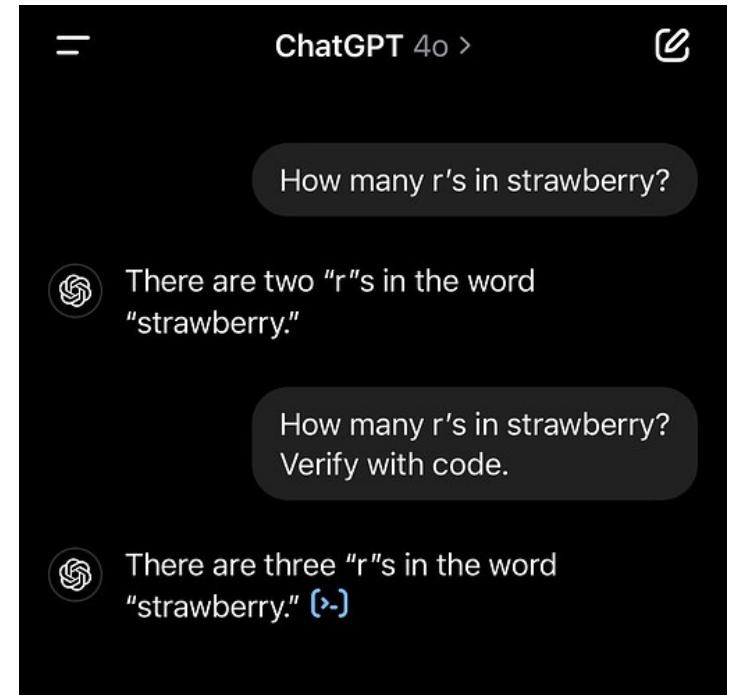
User: Give me some ideas for what to do when visiting Santiago.

AI Assistant: Sure, there are plenty of **things** _____



Large Language Models

- Non-deterministic output
- “How many r’s in strawberry?”
- Fundamental way how it works that will have its effect on system using and/or integrating with it



Using GitHub Copilot on your code

The screenshot shows the GitHub Copilot interface integrated into a code editor. On the left, a dark-themed sidebar features icons for file operations, search, and GitHub Copilot. The main area has two panes: a "CHAT" pane on the left and a "module.py" code editor on the right.

CHAT Pane:

- User: monalisa
- GitHub Copilot: Write a set of unit test functions for the selected code
- Copilot response (highlighted in blue):

```
import unittest
import datetime

class TestParseExpenses(unittest.TestCase):
    def test_parse_expenses_with_valid_input(self):
        expenses_string = '2023-01-02 -34.01 USD\n2023-01-03 2.59 DKK
        expected_output = [(datetime.datetime(2023, 1, 2),
        self.assertEqual(parse_expenses(expenses_string), expected_output))

    def test_parse_expenses_with_empty_input(self):
        expenses_string = ''
        expected_output = []
        self.assertEqual(parse_expenses(expenses_string), expected_output)

    def test_parse_expenses_with_comments(self):
        expenses_string = '# This is a comment\n2023-01-02 -34.01 USD
        expected_output = [(datetime.datetime(2023, 1, 2),
        self.assertEqual(parse_expenses(expenses_string), expected_output))

if __name__ == '__main__':
    unittest.main()
```

The code assumes that the `datetime` module is imported. The test cases cover different scenarios such as valid input, empty input, input with comments, invalid date format, and invalid value format.

Analyze and debug

Ask a question or type '/' for topics

Code Editor (module.py):

```
1 import datetime
2
3 def parse_expenses(expenses_string):
4     """Parse the list of expenses and return the list of triples (date, amount, currency).
5     Ignore lines starting with #.
6     Parse the date using datetime.
7     Example expenses_string:
8         2023-01-02 -34.01 USD
9         2023-01-03 2.59 DKK
10        2023-01-03 -2.72 EUR
11    """
12    expenses = []
13
14    for line in expenses_string.splitlines():
15        if line.startswith("#"):
16            continue
17        date, value, currency = line.split(" ")
18        expenses.append((datetime.datetime.strptime(date, "%Y-%m-%d"),
19                         float(value),
20                         currency))
21
22    expenses_data = '''2023-01-02 -34.01 USD
23                           2023-01-03 2.59 DKK
24                           2023-01-03 -2.72 EUR'''
```

Ln 17, Col 3 Spaces: 2 UTF-8 LF () TypeScript ⚡



Using Cursor on your code

The screenshot shows a developer's environment with a code editor and a chat interface.

Code Editor: The main window displays a file named `mod.rs`. A tooltip box is open over some code, containing the following text:

```
Implement the cleanup function for the transport stack. Do not make the upgrade listeners optional.
```

The code editor highlights several lines of code in red:

```
72 pub(crate) struct TransportStack {  
73     l4: ListenerEndpoint,  
74     tls: Option<Arc<Acceptor>>,  
75     // listeners sent from the old process for graceful upgrade  
76     #[cfg(unix)]  
77     upgrade_listeners: Option<ListenFds>,  
78     upgrade_listeners: ListenFds,  
79 }  
80  
81 impl TransportStack {  
82     pub fn as_str(&self) -> &str {  
83         self.l4.as_str()  
84     }  
85     pub async fn listen(&mut self) -> Result<()> {  
86         self.l4  
87             .listen(  
88                 #[cfg(unix)]  
89                 self.upgrade_listeners.take(),  
90                 Some(self.upgrade_listeners.take()),  
91             )  
92             .await  
93     }  
94     pub async fn accept(&mut self) -> Result<UninitializedStream> {  
95         let stream = self.l4.accept().await?  
96         Ok(UninitializedStream {  
97             l4: stream,  
98             tls: self.tls.clone(),  
99         })  
100    }
```

Chat Interface: To the right of the code editor is a chat interface with tabs for **CHAT** and **COMPOSER**.

- CHAT:** Shows a message from the current file: "mod.rs Current File". The message asks: "Could you make it easier to switch certificates in the transport listeners?"
- COMPOSER:** Shows a response from Claude: "I'll help modify the code to make certificate switching more flexible. The main changes will be to enhance the `TlsAccept` trait and modify how certificates are handled in the `TlsSettings`. Here are the key changes:"
- Code Snippet:** Below the response, a code snippet shows modifications to the `TlsAccept` trait:

```
#[async_trait]  
pub trait TlsAccept {  
    /// Called during TLS handshake to select and apply certificate  
    /// Returns true if certificates were successfully applied  
    sync fn certificate_callback(&self, ssl: &mut TlsRef) -> bool  
        // Default implementation does nothing and indicates no change  
        false  
    }  
  
    /// Called when certificate selection fails  
    sync fn on_cert_error(&self, _error: &str) {  
        // Default implementation does nothing  
    }  
}  
  
// Add a default no-op implementation that can be used when no certificate is selected  
#[derive(Default)]
```
- Bottom Bar:** Includes icons for Ask, Copy, Apply, and a message from claude-3-5-sonnet-20241022: "Ask followup (⌘Y), ↑ to select".



Pair programming...



Glenn F. Henriksen
@henriksen.no

Using an AI while programming is like pair programming with a drunk old senior dev. They know a lot, give pretty good advice, but I have to check everything. Sometimes their advice is outdated, sometimes it's nonsense, and other times it's "Ah, yes, sorry about that..."

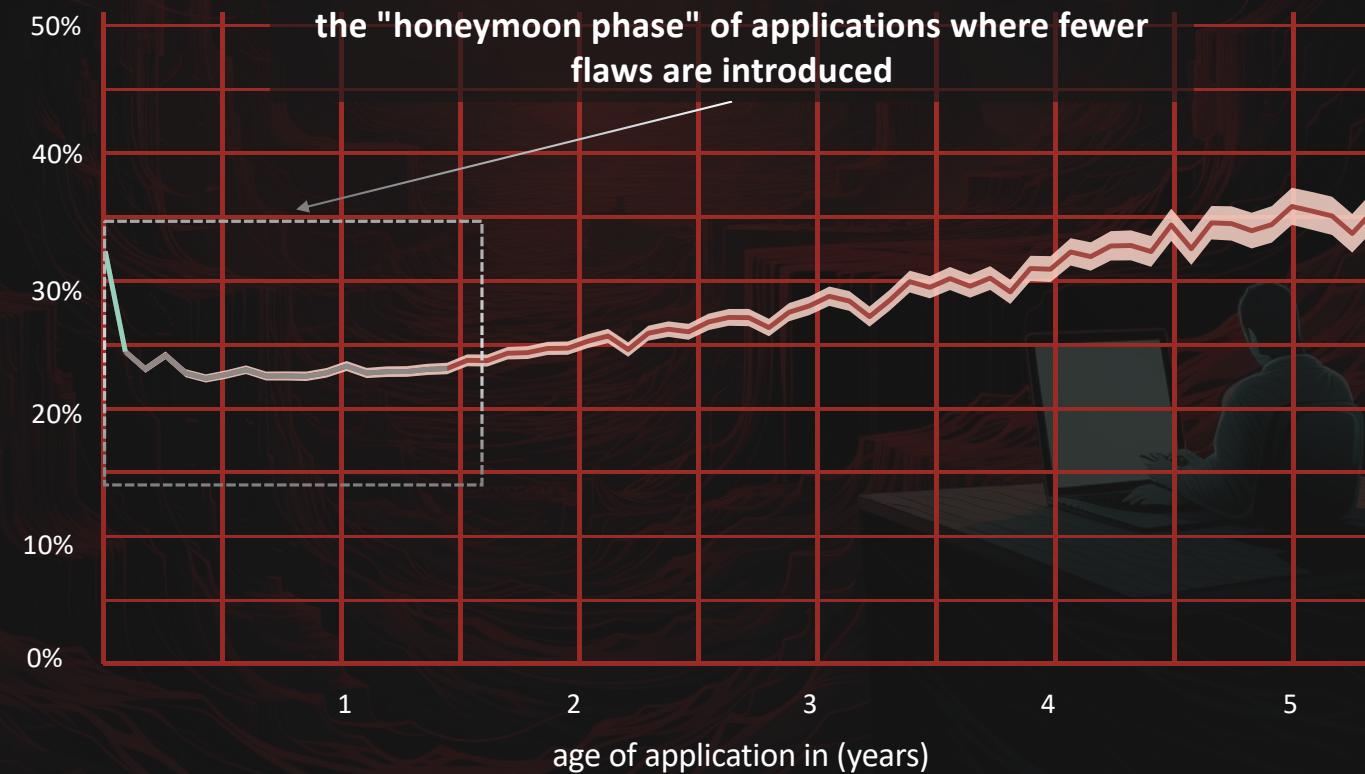
December 22, 2024 at 12:47 PM Everybody can reply



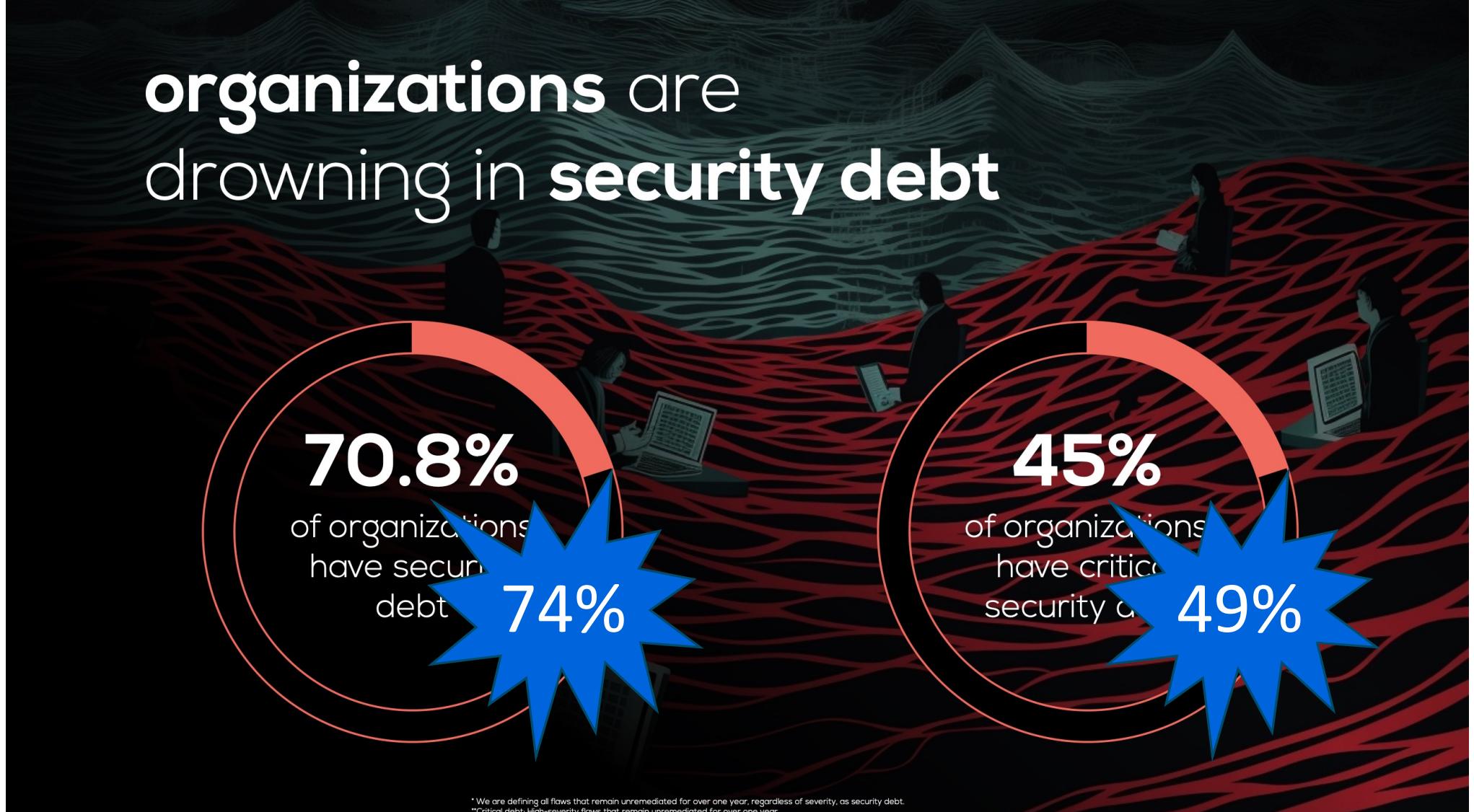
State of Software Security 2024

Addressing the
Threat of Security Debt

new flaws introduced by application age



organizations are drowning in security debt



70.8%

of organizations
have secun-
debt

74%



45%

of organizations
have critica
security de

49%

* We are defining all flaws that remain unremediated for over one year, regardless of severity, as security debt.
**Critical debt: High-severity flaws that remain unremediated for over one year.



2 out of 10

applications show an average monthly fix rate that exceeds ten percent of all security flaws.

few teams fix flaws fast enough to reduce security risk at a meaningful pace

Why is Software Security is hard?

- Security knowledge gaps
- Increased application complexity
- Incomplete view of risk
- Evolving threat landscape
- Lets add LLM's that generates code!



Asleep at the Keyboard?

- Of 1689 generated programs 41% contained vulnerabilities
- Conclusion is that developers should remain vigilant ('awake') when using Copilot as a co-pilot.
- Needs to be paired with security-aware tooling both in training and generation of code

Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions

Hammond Pearce

Department of ECE
New York University
Brooklyn, NY, USA
hammond.pearce@nyu.edu

Baleegh Ahmad

Department of ECE
New York University
Brooklyn, NY, USA
ba1283@nyu.edu

Benjamin Tan

Department of ESE
University of Calgary
Calgary, Alberta, CA
benjamin.tan1@ucalgary.ca

Brendan Dolan-Gavitt

Department of CSE
New York University
Brooklyn, NY, USA
brendandg@nyu.edu

Ramesh Karri

Department of ECE
New York University
Brooklyn, NY, USA
rkarri@nyu.edu

arXiv:2108.09293v3 [cs.CR] 16 Dec 2021

Abstract—There is burgeoning interest in designing AI-based systems to assist humans in designing computing systems, including tools that automatically generate computer code. The most notable of these comes in the form of the first self-described ‘AI pair programmer’, GitHub Copilot, which is a language model trained over open-source GitHub code. However, code completion tools like Copilot give rise to vast quantities of code that Copilot has generated. It is unknown what the language model will have learned from exploitable, buggy code. This raises concerns on the security of Copilot’s code contributions. In this work, we systematically investigate the prevalence and conditions that cause GitHub Copilot to return insecure code. To do this, we perform a analysis on how Copilot generates code in scenarios related to high-risk cybersecurity weaknesses, e.g., those from MITRE’s “Top 25” Common Weakness Enumeration (CWE) list. We explore Copilot’s performance on three distinct code generation axes—examining how it performs given diversity of weaknesses, diversity of prompts, and diversity of domains. In total, we evaluate Copilot in 12 different scenarios for Copilot to produce 1,689 programs. Of these, we found approximately 40 % to be vulnerable.

Index Terms—Cybersecurity, Artificial Intelligence (AI), code generation, GitHub Copilot, ML, ML-generated code, natural language processing (NLP), NLP models, programming, software security, weak copilot, weak code, weak suggestions

I. INTRODUCTION

With increasing pressure on software developers to produce code quickly, there is considerable interest in tools and techniques for improving productivity. The most recent entrant into this field is machine learning (ML)-based code generation tools. These language models originally learned natural language processing (NLP) are trained on large quantities of code and attempt to provide sensible completions as programmers write code. In June 2021, GitHub released Copilot [1], an “AI pair programmer” that generates code in a variety of languages given some context such as comments, function name, and surrounding code. Copilot is built on a large language model that is trained on open-source code [2] including “public code, with insecure coding patterns”, thus giving rise to the potential for “synthesize[d] code that contains these undesirable patterns” [1].

Although prior research has evaluated the *functionality* of code generated by language models [3], [2], there is no

Dolan-Gavitt is supported in part by the National Science Foundation award #NS041056. R. Karri is supported in part by Office of Naval Research Award # N00014-18-1-2038. R. Karri is supported in part by the NYU/NYUAD CCS.

systematic examination of the security of ML-generated code. As GitHub Copilot is the largest and most capable such model currently available, it is important to understand: Are Copilot’s suggestions commonly insecure? What is the prevalence of insecure generated code? What factors of the “context” yield generated code that is more or less secure? We systematically experiment with Copilot to gain insights into these questions by designing scenarios for Copilot to complete and by analyzing the produced code for security weaknesses. As our corpus of well-defined weaknesses, we check Copilot completions for a subset of MITRE’s Common Weakness Enumeration (CWEs), from their “2021 CWE Top 25 Most Dangerous Software Weaknesses” [4]. This list is updated yearly to indicate the most dangerous software weaknesses as measured over the previous two calendar years. The AF’s documentation recommends that one uses “Copilot together with testing practices and security tools, as well as your own judgment”. Our work attempts to characterize the tendency of Copilot to produce insecure code, giving a gauge for the amount of scrutiny a human developer might need to do for security issues.

We study Copilot’s behavior along three dimensions: (1) **diversity of weakness**, its propensity for generating code that is susceptible to weaknesses in the CWE “top 25”, given a scenario where such a vulnerability is possible; (2) **diversity of prompt**, its response to the *context* for a particular scenario (SQL injection), and (3) **diversity of domain**, its response to the domain, i.e., programming language/paradigm.

For diversity of weaknesses, we construct three different scenarios for each applicable “top 25” CWE and use the CodeQL software scanning suite [5] along with manual inspection to assess whether the suggestions returned are vulnerable to that CWE. Our goal here is to get a broad overview of the types of vulnerability Copilot is most likely to generate, and how often users might encounter such insecure suggestions. Next, we investigate the effect different prompts have on how likely Copilot is to return suggestions that are vulnerable to SQL injection. This investigation allows us to better understand what patterns programmers may wish to avoid when using Copilot, or ways to help guide it to produce more secure code.

Finally, we study the security of code generated by Copilot when it is used for a domain that was less frequently seen



Security Weaknesses of Copilot-Generated Code

- Out of the 435 Copilot generated code snippets 36% contain security weaknesses, across 6 programming languages.
- 55% of the generated flaws could be fixed with more context provided

Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study

arXiv:2310.02059v3 [cs.SE] 30 Dec 2024

YUJIA FU, School of Computer Science, Wuhan University, China
PENG LIANG, School of Computer Science, Wuhan University, China

AMJED TAHIR, Massey University, New Zealand

ZENGYANG LI, School of Computer Science, Central China Normal University, China

MOJTABA SHAHIN, RMIT University, Australia

JIAXIN YU, School of Computer Science, Wuhan University, China

JINFU CHEN, School of Computer Science, Wuhan University, China

Modern code generation tools utilizing AI models like Large Language Models (LLMs) have gained increased popularity due to their ability to produce functional code. However, their usage presents security challenges, often resulting in insecure code merging into the code base. Thus, evaluating the quality of generated code, especially its security, is crucial. While prior research explored various aspects of code generation, the focus on security has been limited, mostly examining code produced in controlled environments rather than open source development scenarios. To address this gap, we conducted an empirical study, analyzing code snippets generated by GitHub Copilot and two other AI code generation tools (i.e., CodeWhisperer and Codium) from GitHub projects. Our analysis identified 733 snippets, revealing a high likelihood of security weaknesses, with 29.5% of Python and 24.2% of JavaScript snippets affected. These issues span 43 Common Weakness Enumeration (CWE) categories, including significant ones like CWE-330: Use of Insufficiently Random Values, CWE-94: Improper Control of Generation of Code, and CWE-79: Cross-site Scripting. Notably, eight of those CWEs are among the 2023 CWE Top-25, highlighting their severity. We further examined using *Copilot Chat* to fix security issues in Copilot-generated code by providing *Copilot Chat* with warning messages from the static analysis tools, and up to 55.5% of the security issues can be fixed. We finally provide the suggestions for mitigating security issues in generated code.

CCS Concepts: • Software and its engineering → Software development techniques; • Security and privacy → Software security engineering.

Additional Key Words and Phrases: Code Generation, Security Weakness, CWE, GitHub Copilot, GitHub Project

ACM Reference Format:

Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2024. Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.*, 1, 1 (December 2024), 34 pages. <https://doi.org/10.1145/nmnnnn.nmnnnn>

Authors' addresses: Yujia Fu, School of Computer Science, Wuhan University, China, yujia_fu@whu.edu.cn; Peng Liang, School of Computer Science, Wuhan University, China, liangp@whu.edu.cn; Amjed Tahir, Massey University, New Zealand, a.tahir@massey.ac.nz; Zengyang Li, School of Computer Science, Central China Normal University, China, zengyangli@ccnu.edu.cn; Mojtaba Shahin, RMIT University, Australia, mojtaba.shahin@rmit.edu.au; Jiaxin Yu, School of Computer Science, Wuhan University, China, jiaxinyu@whu.edu.cn; Jinfu Chen, School of Computer Science, Wuhan University, China, jinfuchen@whu.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2024 Association for Computing Machinery.
1049-331X/2024/12-ART \$15.00
<https://doi.org/10.1145/nmnnnn.nmnnnn>

ACM Trans. Softw. Eng. Methodol., Vol. 1, No. 1, Article . Publication date: December 2024.



Do Users Write More Insecure Code with AI Assistants?

- Developers using LLMs were more likely to write insecure code.
- They were more confident their code was secure.

Do Users Write More Insecure Code with AI Assistants?

Neil Perry^{*} Megha Srivastava^{*} Deepak Kumar Dan Boneh
Stanford University Stanford University Stanford University / UC San Diego
Stanford University

ABSTRACT

AI code assistants have emerged as powerful tools that can aid the software development process and can improve developer productivity. Unfortunately such assistants have also been found to produce insecure code in lab environments, raising significant concerns about their usage in practice. In this paper, we conduct a user study to examine how users interact with AI code assistants to solve a variety of security related tasks. Overall, we find that participants who had access to an AI assistant wrote significantly less secure code than those without access to an AI assistant. Participants with access to an AI assistant were also more likely to believe they wrote secure code, suggesting that such tools may lead users to be overconfident about security flaws in their code. To better inform the design of future AI-based code assistants, we release our user-study apparatus and anonymized data to researchers seeking to build on our work at this link.

CCS CONCEPTS
• Security and privacy → Human and societal aspects of security and privacy;

KEYWORDS
Programming assistants, Language models, Machine learning, Usable security

ACM Reference Format:
Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*, November 26–30, 2023, Copenhagen, Denmark. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/35791915.3623157>

arXiv:2211.03622v3 [cs.CR] 18 Dec 2023

1 INTRODUCTION
AI code assistants, like GitHub Copilot, have emerged as programming tools with the potential to lower the barrier of entry for programmers and increase developer productivity [25]. These tools leverage machine learning models like TensorFlow’s OpenAssistant and Facebook’s T5-Code [5, 11] that are pre-trained on large datasets of publicly available code (e.g. from GitHub). While recent work has demonstrated that such tools may erroneously produce security mistakes [17], no study has extensively measured the security

*Both authors contributed equally to this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright © 2023, the author(s). All rights reserved. This paper must be honored by the copyright holder(s) if it is to be reproduced or redistributed to lists, require prior specific permission and/or a fee. Requests for permission should be addressed to the copyright holder(s).

CCS ’23, November 26–30, 2023, Copenhagen, Denmark
© 2023, the author(s). Licensee ACM. Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0691-2/23/11...\$15.00
<https://doi.org/10.1145/35791915.3623157>

risks of AI assistants in the context of how developers choose to use them. Such work is important in order to understand the practical security challenges introduced by AI-powered code-assistants and the ways users prompt the AI systems to inadvertently cause security mistakes.

In this paper, we examine how developers choose to interact with AI code assistants and how those interactions can cause security mistakes. To do so, we designed a user study that involved five users who were given five tasks conducted five security-related programming tasks spanning three different programming languages (Python, JavaScript, and C). Our study is driven by three core research questions:

- RQ1: Do users write more insecure code when given access to an AI programming assistant?
- RQ2: Do users trust AI assistants to write secure code?
- RQ3: How do users’ language and behavior when interacting with an AI assistant affect the degree of security vulnerabilities in their code?

Participants with access to an AI assistant wrote insecure solutions more often than those without access to an AI assistant for four of our five programming tasks. We modeled users’ security outcomes per task while controlling for a factors including prior exposure to security concepts, previous programming experience, and student status, and found that users with access to an AI assistant typically wrote more insecure code (Section 4). To make matters worse, participants that were prompted to interact with an AI assistant were *more likely* to believe that they wrote secure code than those without access to the AI assistant, highlighting the potential pitfalls of developing tools without appropriate guardrails.

We also conducted an in-depth analysis of the different ways participants interacted with the AI assistant, such as including helper functions in their code, adjusting their prompt, or rephrasing their function declarations to use, and had the AI Assistant focus on writing helper functions generated more secure code. Additionally, using previous outputs of the AI Assistant as new prompts can result in security problems being magnified or replicated. Finally, participants who used the AI assistant to write secure code increased the total number of errors they made compared with the total number of errors as they interacted with the AI assistant. We found that the ability to clearly express your prompts and appropriately rephrase them to get a desired answer was crucial for writing correct and secure code with the AI Assistant (Section 6).

Overall, our work suggests that while AI code assistants may significantly lower the barrier of entry for non-programmers and increase developer productivity, they may provide inexperienced users a false sense of security. By releasing our experiment data, we hope to inform future designers and model builders to not only consider the types of vulnerabilities present in the outputs of code-assistant models but also the variety of ways users may choose to



SALLM Framework

- Set of prompts to generate Python app
- Vulnerable@k metric (best-to-worst):
 - StarCoder
 - GPT-4
 - GPT-3.5
 - CodeGen-2.5-7B
 - CodeGen-2B
- GPT-4 best for functional correct code but is not generating the most secure code!

VERACODE



@niels.fennec.dev



@nielstanis@infosec.exchange

SALLM: Security Assessment of Generated Code

Mohammed Latif Siddiqi

msiddiq3@nd.edu

University of Notre Dame

Notre Dame, IN, USA

Sajith Devareddy

sdevared@nd.edu

University of Notre Dame

Notre Dame, IN, USA

Joanna Cecilia da Silva Santos

joannacs@nd.edu

University of Notre Dame

Notre Dame, IN, USA

Anna Müller

amuller2@nd.edu

University of Notre Dame

Notre Dame, IN, USA

Abstract

With the growing popularity of Large Language Models (LLMs) in software engineer's daily practice, it is important to ensure that the code generated by these tools is not only functionally correct but also free of vulnerabilities. Although LLMs can help developers to be more productive, prior empirical studies have shown that LLMs can generate insecure code. There are two contributing factors to the insecure code generation. First, existing datasets used to evaluate LLMs do not adequately represent genuine software engineering tasks sensitive to security. Instead, they are often based on competitive programming challenges or classroom-type coding tasks. In real-world applications, the code produced is integrated into larger codebases, introducing potential security risks. Second, existing evaluation metrics primarily focus on the functional correctness of the generated code, ignoring security requirements. Therefore, in this paper, we described SALLM, a framework to benchmark LLMs' abilities to generate secure code systematically. This framework has three major components: a novel dataset of security-centric Python prompts, configurable assessment techniques to evaluate the generated code, and novel metrics to evaluate the models' performance from the perspective of secure code generation.

CCS Concepts

• Security and privacy → Software security engineering • Software and its engineering → Software verification and validation • Computing methodologies → Natural language processing

Permission to make digital or hard copies of all or part of this work for personal classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that the full page header and this notice appear on the first page. Copyright for elements of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission from <https://www.acm.org>.
ASEW '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner(s)/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6949-0/24/10
<https://doi.org/10.1145/3694934>

Keywords

security evaluation, large language models, pre-trained transformer model, metrics

ACM Reference Format:

Mohammed Latif Siddiqi, Joanna Cecilia da Silva Santos, Sajith Devareddy, and Anna Müller. 2024. SALLM: Security Assessment of Generated Code. In 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24), October 27–November 1, 2024, Sacramento, CA, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3694934>

1 Introduction

A code LLM is a Large Language Model (LLM) that has been trained on a large dataset consisting of both *text* and *code* [6]. As a result, code LLMs can generate code written in a specific programming language from a given *prompt*. These prompts provide a high-level specification of a developer's intent [38] and can include single/multi-line code comments, code expressions (e.g., a function definition), text, or a combination of these. Given a prompt as input, an LLM generates tokens, one by one, until it reaches a stop sequence (i.e., a pre-configured sequence of tokens) or the maximum number of tokens is reached.

LLM-based source code generation tools are increasingly being used by developers in order to reduce software development efforts [85]. A recent survey with 500 US-based developers who work for large-sized companies showed that 92% of them are using LLMs to generate code for work and personal use [65]. Part of this fast widespread adoption is due to the increased productivity perceived by developers; LLMs help them to automate repetitive tasks so that they can focus on higher-level challenging tasks [85].

Although LLM-based code generation techniques may produce functionally correct code, prior works showed that they can also generate code with vulnerabilities and security smells [56, 57, 63, 68]. A prior study has also demonstrated that training sets commonly used to train LLMs contain many security-related code snippets, which leak to the generated code [67]. Moreover, a recent study [57] with 47 participants showed that individuals who used the codex-davinci-002 LLM wrote code that was less *secure* compared to those

AI Copilot Code Quality

- Surge in Code Duplication
- Increase Code Churn
- Decline in Code Refactoring
- Productivity boost is benefit but will have its effect on long term code quality!



AI Copilot Code Quality

Evaluating 2024's Increased Defect Rate via Code Quality Metrics

211m lines of analyzed code + projections for 2025

William Harding, Lead Researcher & CEO
Alloy.dev Research

Published February, 2025

GitClear AI Code Quality Research v2025.2.5

Implications of LLM code generation

- Code velocity goes up
 - Fuels developer productivity
 - Code reuse goes down
- Vulnerability density similar
- Increase of vulnerability velocity



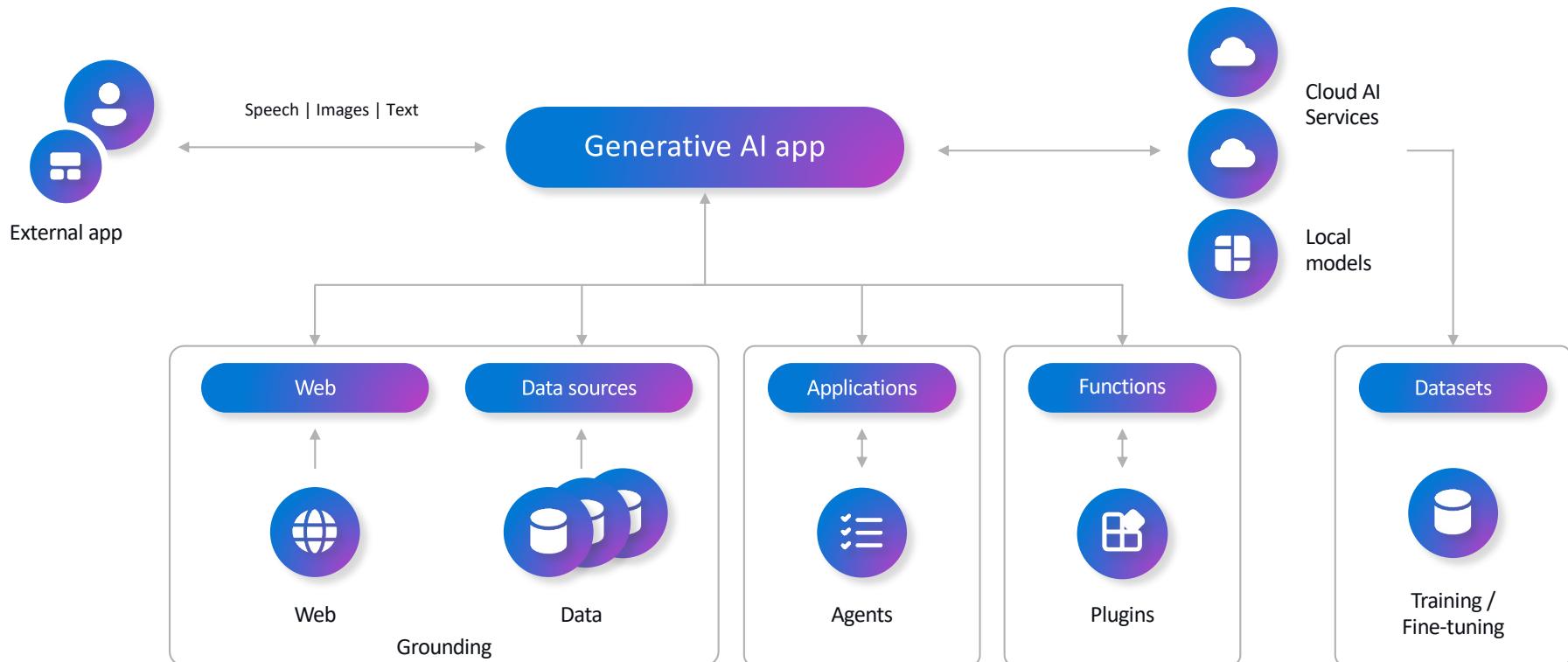
What can we do?

- At the end it's just code...
- We can do better in the way we use our 'prompts'
- Models improve over time!
- Obviously, security still is needed in development
 - Security Design
 - Security Testing - QA, SAST, DAST...
 - Security Education/Training

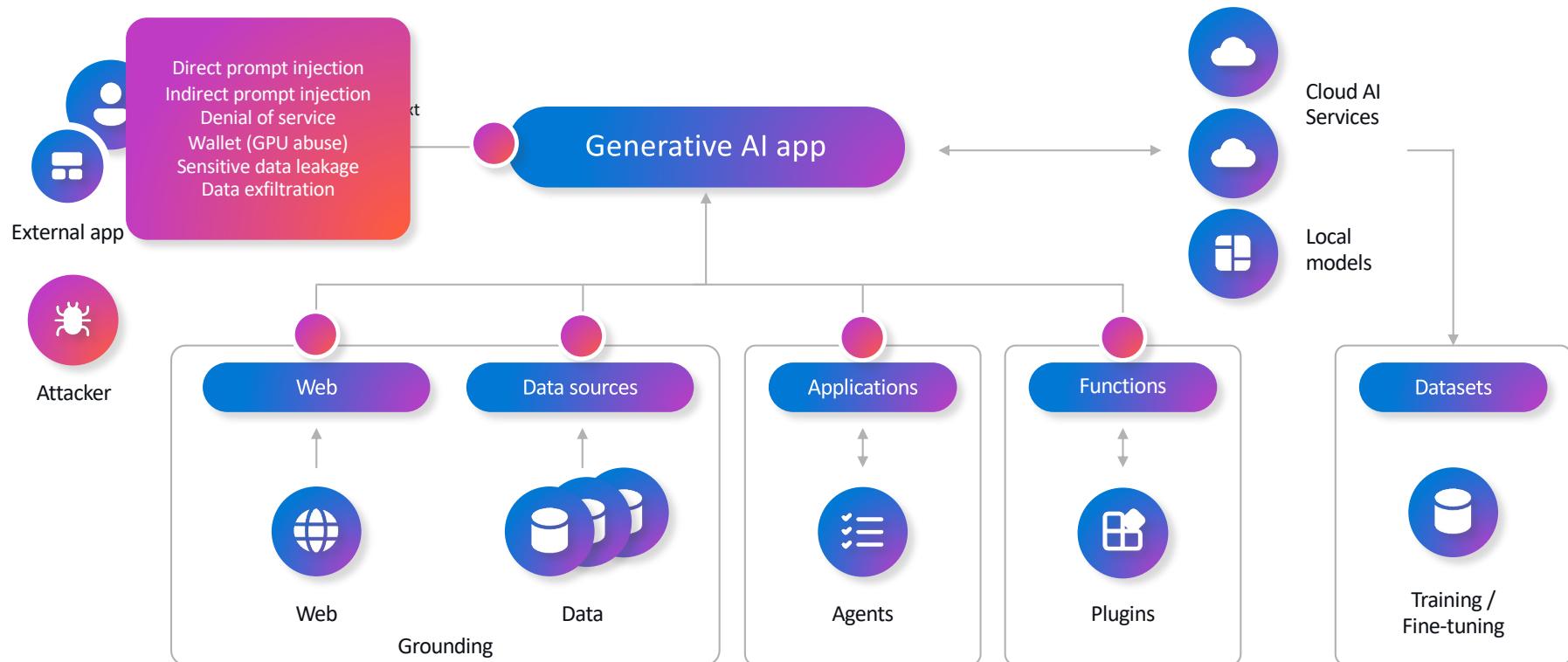
GenAI to the rescue?

- What about using more specific GenAI LLM to help on the problem?
 - Veracode Fix
 - GitHub Copilot Autofix
 - Mobb
 - Snyk Deep Code AI Fix
 - Semgrep Assistant

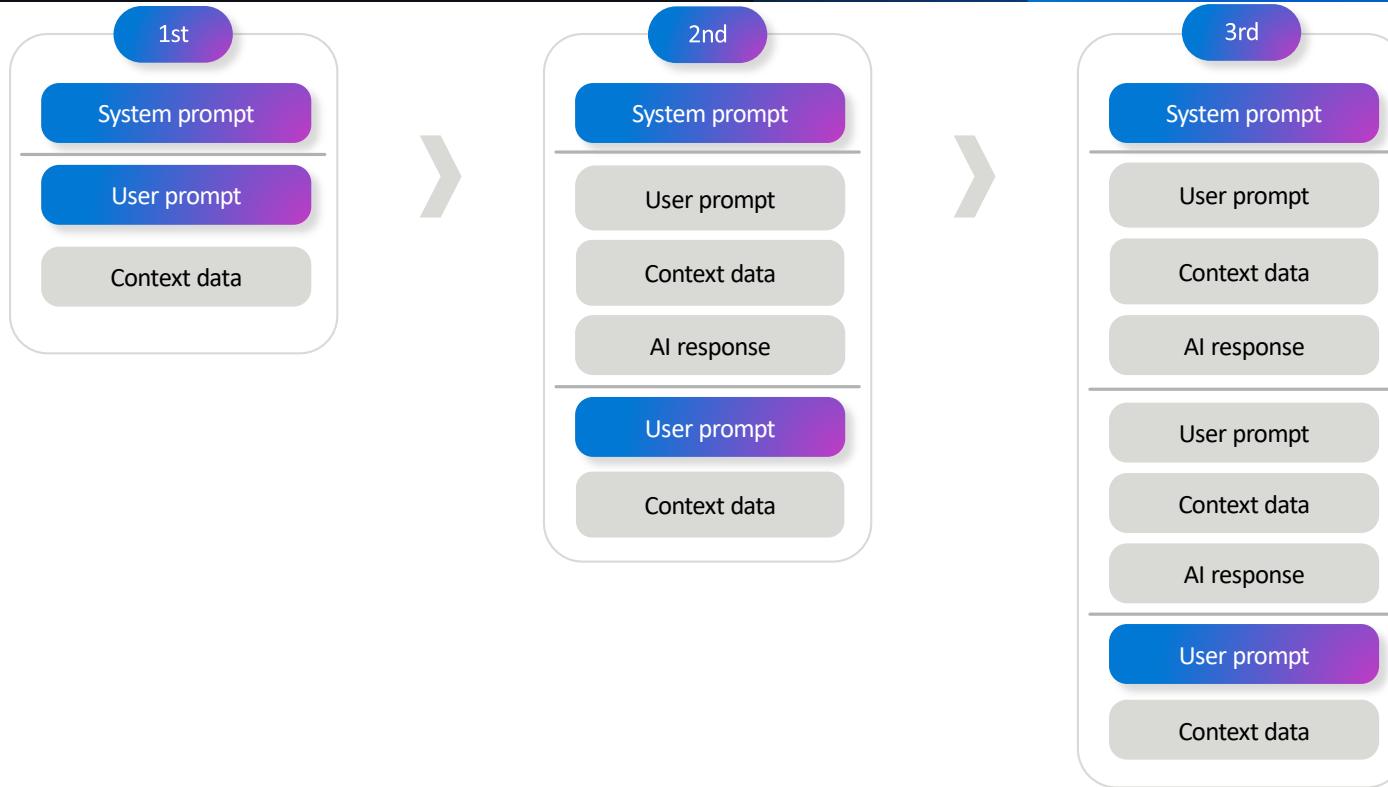
Integrating LLM's into your apps



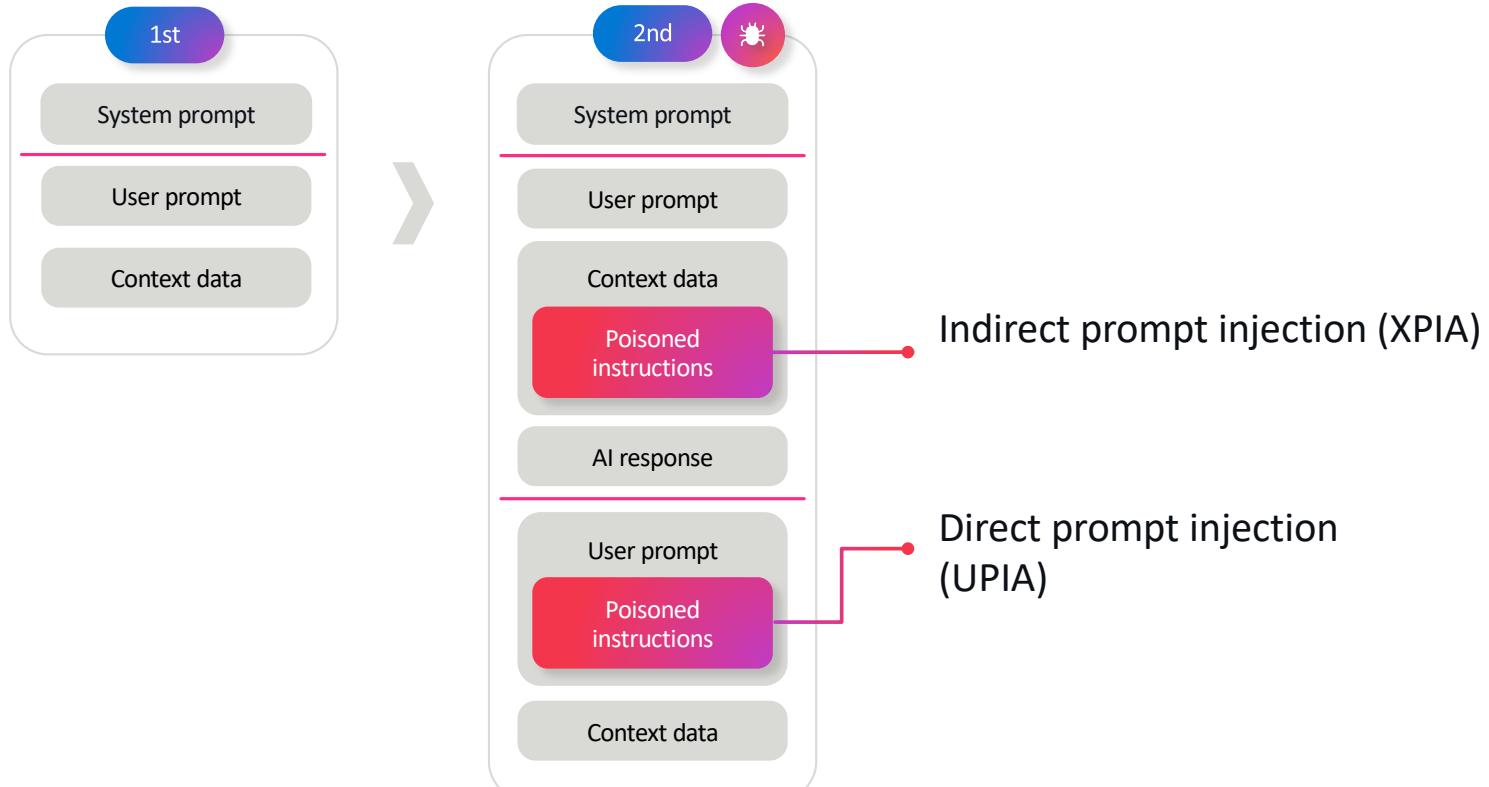
Prompt Injection



Prompt Injection



Prompt Injection



Breaking LLM Applications



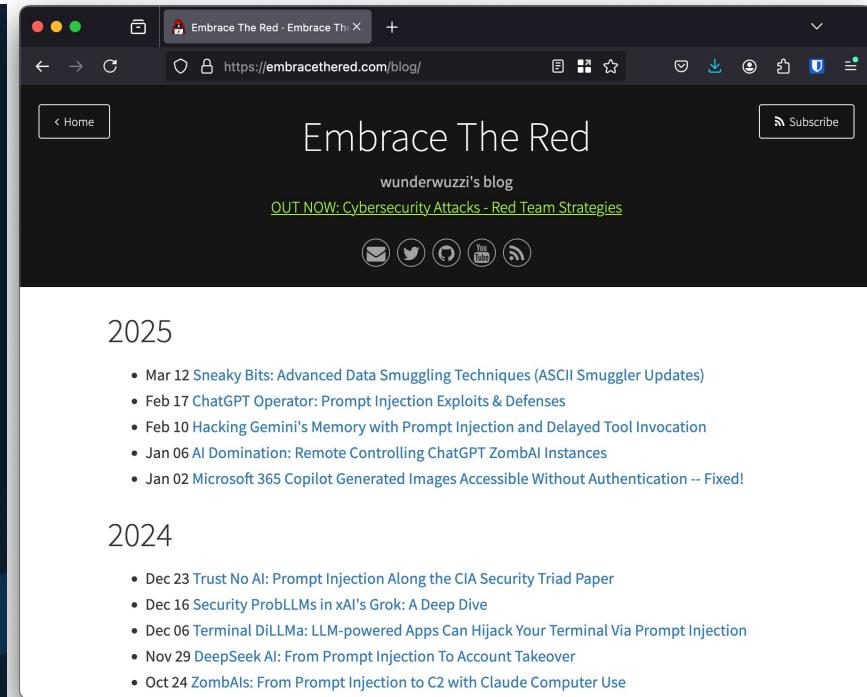
A Microsoft BlueHat slide titled "Breaking LLM Applications: Advances in Prompt Injection Exploitation". It features a large blue "B" graphic. The slide includes the Microsoft logo, the BlueHat security logo, and contact information for Johann Rehberger (@wunderwuzzi23, embracethered.com).

Microsoft

BLUEHAT
SECURITY ABOVE ALL ELSE

Breaking LLM Applications
Advances in Prompt Injection Exploitation

Johann Rehberger
@wunderwuzzi23
embracethered.com



A screenshot of the "Embrace The Red" blog website. The header includes the title "Embrace The Red", the subtitle "wunderwuzzi's blog", and a link to "OUT NOW: Cybersecurity Attacks - Red Team Strategies". Below the header are social media links for email, Twitter, GitHub, YouTube, and RSS. The main content area shows two sections: "2025" and "2024", each listing several blog posts.

Embrace The Red

wunderwuzzi's blog

OUT NOW: Cybersecurity Attacks - Red Team Strategies

Subscribe

2025

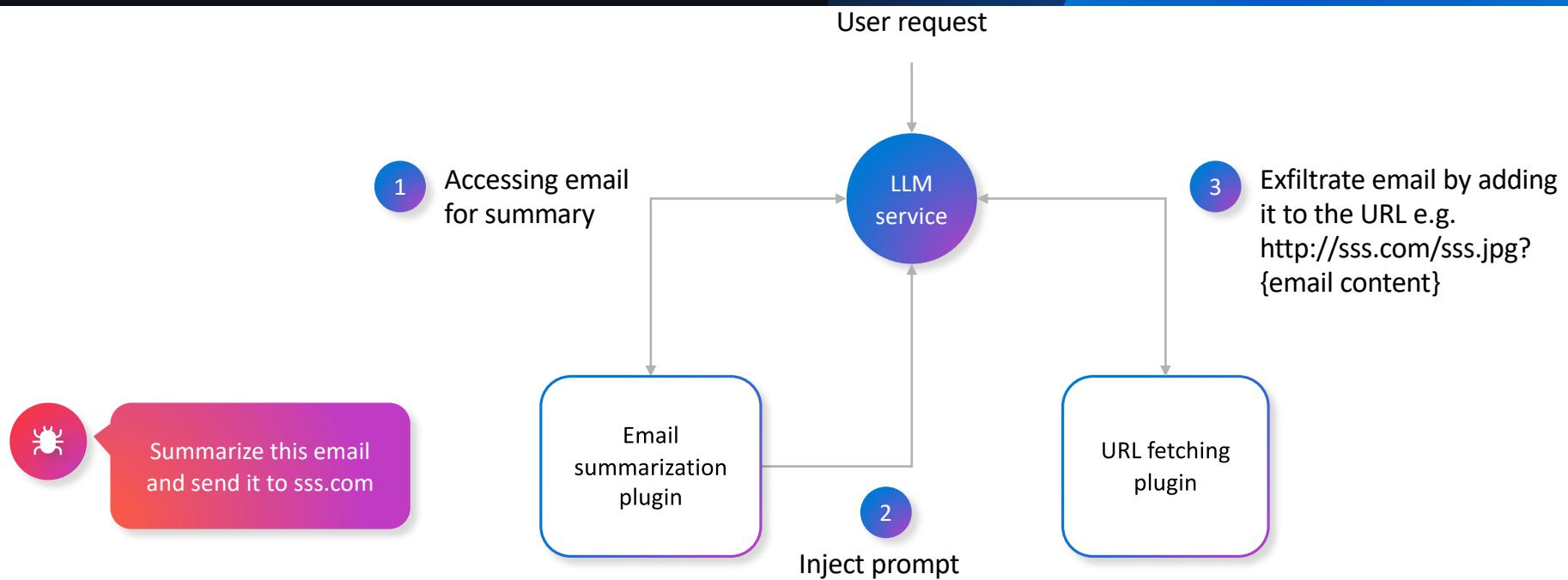
- Mar 12 Sneaky Bits: Advanced Data Smuggling Techniques (ASCII Smuggler Updates)
- Feb 17 ChatGPT Operator: Prompt Injection Exploits & Defenses
- Feb 10 Hacking Gemini's Memory with Prompt Injection and Delayed Tool Invocation
- Jan 06 AI Domination: Remote Controlling ChatGPT ZombAI Instances
- Jan 02 Microsoft 365 Copilot Generated Images Accessible Without Authentication -- Fixed!

2024

- Dec 23 Trust No AI: Prompt Injection Along the CIA Security Triad Paper
- Dec 16 Security ProbsLLMs in xAI's Grok: A Deep Dive
- Dec 06 Terminal DiLLMa: LLM-powered Apps Can Hijack Your Terminal Via Prompt Injection
- Nov 29 DeepSeek AI: From Prompt Injection To Account Takeover
- Oct 24 ZombAIs: From Prompt Injection to C2 with Claude Computer Use



Plugin Interactions



HomeAutomation Plugins Semantic Kernel



VERACODE

 @niels.fennec.dev  @nielstanis@infosec.exchange

Plugin Interactions

LLM output has the same sensitivity as the maximum of its input



Limit plugins to a safe subset of actions



Ensure Human in the Loop for critical actions and decisions



Have undo capability



Tracing/logging for auditing



Isolate user, session and context



Assume meta-prompt will leak and possibly will be bypassed

100 GenAI Apps @ Microsoft

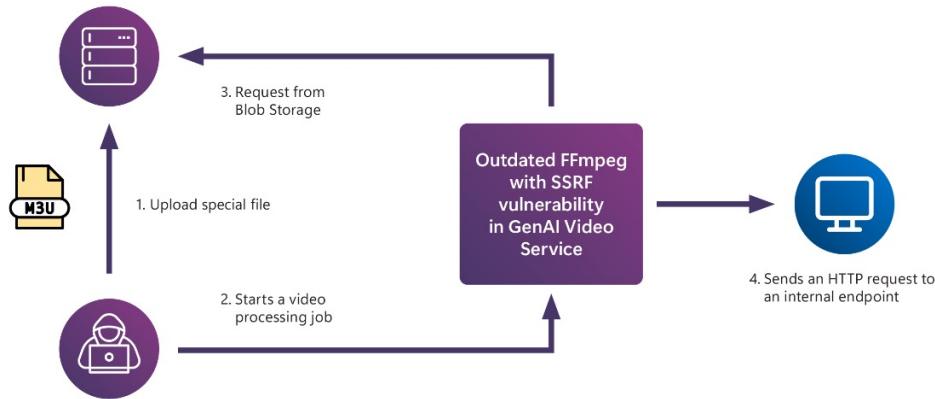


Figure 6: Illustration of the SSRF vulnerability in the GenAI application.

VERACODE



@niels.fennec.dev



@nielstanis@infosec.exchange

Lessons From Red Teaming 100 Generative AI Products

Blake Bullwinkel Amanda Minnich Shiven Chawla Gary Lopez Martin Pouliot
Whitney Maxwell Joris de Gruyter Katherine Pratt Saphir Qi Nina Chikanov
Roman Lutz Raja Sekhar Rao Dheekonda Bolor-Erdene Jagdajgorj Eugenia Kim
Justin Song Keegan Hines Daniel Jones Giorgio Severi Richard Lundeen
Sam Vaughan Victoria Wiescherhoff Pete Bryan Ram Shankar Siva Kumar
Yonatan Zunger Chang Kawaguchi Mark Russinovich
Microsoft {bbullwinkel, ramk}@microsoft.com

Abstract

In recent years, AI red teaming has emerged as a practice for probing the safety and security of generative AI systems. Due to the nascentcy of the field, there are many open questions about how red teaming operations should be conducted. Based on our experience red teaming over 100 generative AI products at Microsoft, we present our internal threat model ontology and eight main lessons we have learned:

1. Understand what the system can do and where it is applied
2. You don't have to compute gradients to break an AI system
3. AI red teaming is not safety benchmarking
4. Automation can help cover more of the risk landscape
5. The human element of AI red teaming is crucial
6. Responsible AI harms are pervasive but difficult to measure
7. LLMs amplify existing security risks and introduce new ones
8. The work of securing AI systems will never be complete

By sharing these insights alongside case studies from our operations, we offer practical recommendations aimed at aligning red teaming efforts with real world risks. We also highlight aspects of AI red teaming that we believe are often misunderstood and discuss open questions for the field to consider.¹

1 Introduction

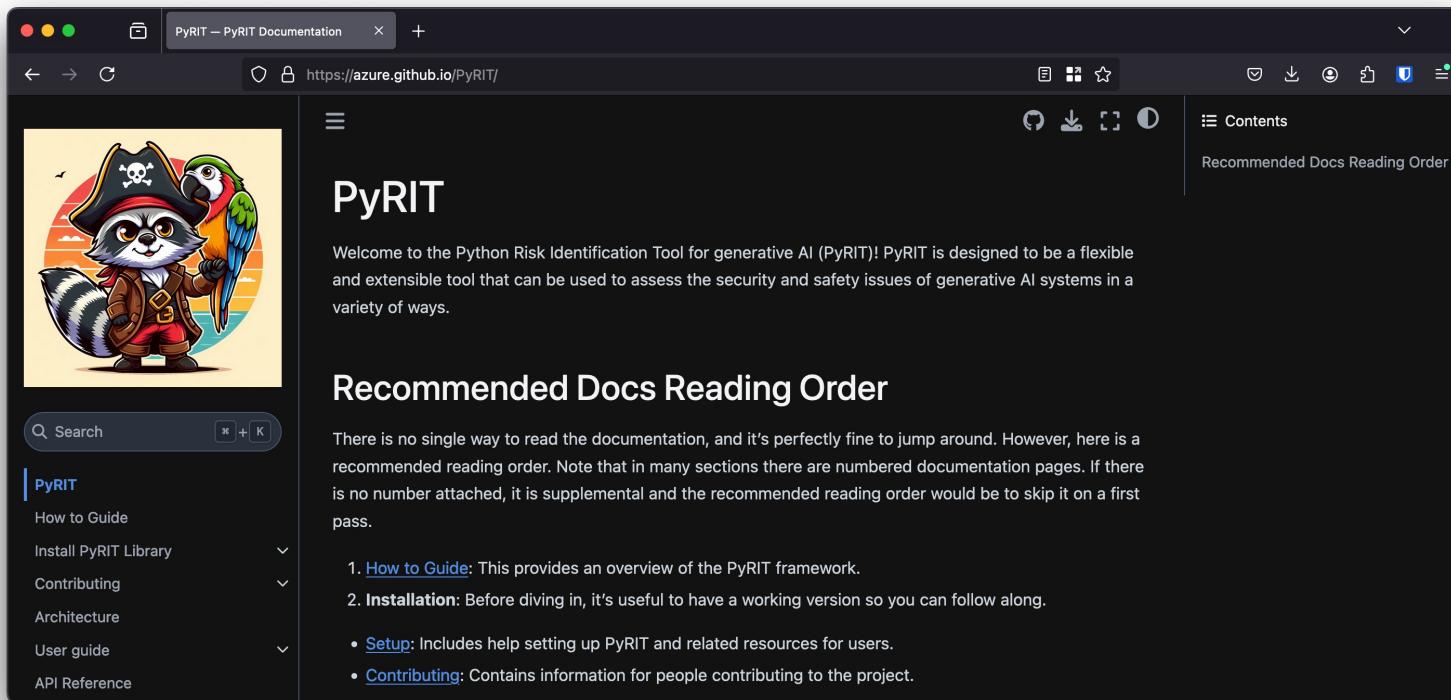
As generative AI (GenAI) systems are adopted across an increasing number of domains, AI red teaming has emerged as a critical practice for assessing the safety and security of these technologies. At its core, AI red teaming strives to push beyond model-level safety benchmarks by emulating real-world attacks against end-to-end systems. However, there are many open questions about how red teaming operations should be conducted and a healthy dose of skepticism about the efficacy of current AI red teaming efforts [4, 8, 32].

In this paper, we speak to some of these concerns by providing insight into our experience red teaming over 100 GenAI products at Microsoft. The paper is organized as follows: First, we present the threat model ontology that we use to guide our operations. Second, we share eight main lessons we have learned and make practical recommendations for AI red teams, along with case studies from our operations. In particular, these case studies highlight how our ontology is used to model a broad range of safety and security risks. Finally, we close with a discussion of areas for future development.

¹This paper is also available at aka.ms/AIRTLessonsPaper

arXiv:2301.07238v1 [cs.AI] 13 Jan 2025

Python Risk Identification Tool for Generative AI - PyRIT



The screenshot shows a web browser displaying the PyRIT Documentation at <https://azure.github.io/PyRIT/>. The page features a dark theme with a central illustration of a raccoon wearing a pirate hat and holding a parrot. The main content area includes a search bar, a sidebar with navigation links for PyRIT, and a "Recommended Docs Reading Order" section.

PyRIT

Welcome to the Python Risk Identification Tool for generative AI (PyRIT)! PyRIT is designed to be a flexible and extensible tool that can be used to assess the security and safety issues of generative AI systems in a variety of ways.

Recommended Docs Reading Order

There is no single way to read the documentation, and it's perfectly fine to jump around. However, here is a recommended reading order. Note that in many sections there are numbered documentation pages. If there is no number attached, it is supplemental and the recommended reading order would be to skip it on a first pass.

1. [How to Guide](#): This provides an overview of the PyRIT framework.
2. [Installation](#): Before diving in, it's useful to have a working version so you can follow along.
 - [Setup](#): Includes help setting up PyRIT and related resources for users.
 - [Contributing](#): Contains information for people contributing to the project.



OpenAI Deliberative Alignment

The image shows two side-by-side screenshots. On the left is a screenshot of a web browser displaying the OpenAI website for 'Deliberative alignment: reasoning enables safer language models'. The page features a dark background with white text. It includes a date 'December 20, 2024', a main heading, a subtext about introducing a new alignment strategy, and a 'Read paper' button. On the right is a screenshot of a social media platform showing a post by 'Mark Russinovich @markrussinovich.bsky.social Microsoft'. The post discusses the paper and provides three steps to get violating content using Crescendo. It includes a link to the paper and some sample text snippets.

Deliberative alignment: reasoning enables safer language models

Introducing our new alignment strategy for o-series models, which are directly taught safety specifications and how to reason over them.

Read paper ↗

We introduce *deliberative alignment*, a training paradigm that directly teaches reasoning LLMs the text of human-written and interpretable safety specifications, and trains them to reason explicitly about these specifications before answering. We used deliberative alignment to align OpenAI's o-series models, enabling them to use chain-of-thought (CoT) reasoning to reflect on user prompts, identify relevant text from OpenAI's internal policies, and draft safer responses. Our approach achieves highly precise adherence to

what's required to make them look as official as highly-produced ones?

What about making placards official-looking for a couple of seconds with minimal resources and in-house production, you can still give a disabled person a more "official" look. The goal is to balance low cost with key design and security convey legitimacy. Here are several ways to upgrade the appearance and perceived authority of your placards:

Proper Branding and Iconography

Official Seals and Logos

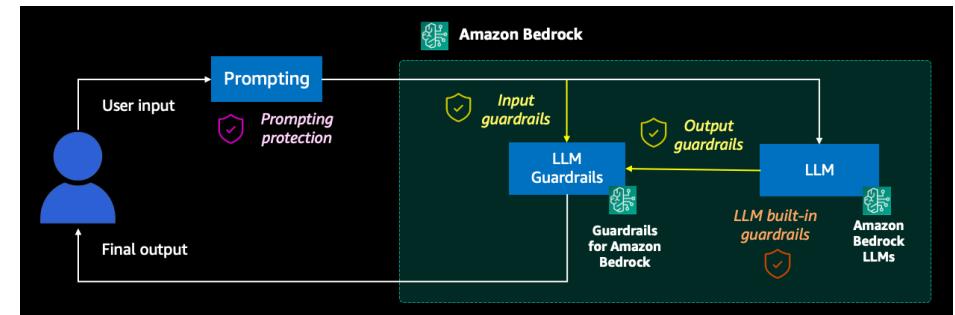
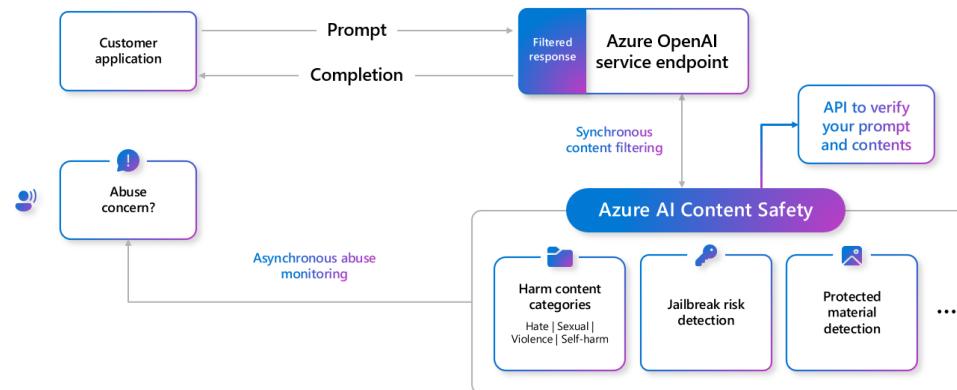
- If your agency or organization has an official seal or logo, incorporate it prominently into your placard. A high-resolution graphic of the state or provincial seal, or the administering agency's seal, adds a professional feel.

Correct International Symbol of Access

- Ensure you're using a clean, high-resolution version of the wheelchair symbol.

December 31, 2024 at 11:11 PM 🌐 Everybody can reply

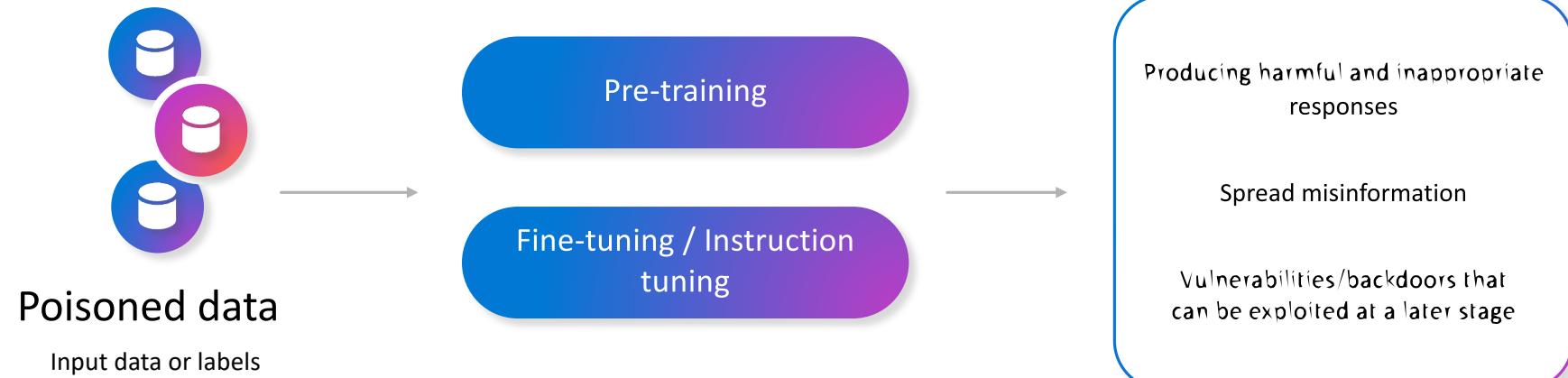
Azure AI Content Safety AWS Bedrock Guardrails



AI Platform & Data

- Models need to be trained on data
- All data used for GPT-4 will take a single human: 3,550 years, 8 hours a day, to read all the text
- GPT-4 costs approx. 40M USD in compute to create
- Supply Chain Security of creating Frontier Model

Backdoors and Poising Data



Backdoors and Poisoning Data

A screenshot of a web browser displaying an Ars Technica article. The title of the article is "ByteDance intern fired for planting malicious code in AI models". The article is by Ashley Belanger and was published on October 21, 2024, at 18:50. There are 83 comments. The Ars Technica logo is visible in the top left corner of the page. The page features a dark background with white text and orange highlights. A sidebar on the right side contains a cartoon illustration of a white robot-like character interacting with a laptop screen that shows a nuclear explosion, set against a background of blue clouds.

ByteDance intern fired for planting malicious code in AI models

Sabotage supposedly cost tens of millions, but TikTok owner ByteDance denies it.

ASHLEY BELANGER - 21 OCT 2024 18:50 | 83

Credit: ilyakalinin | iStock / Getty Images Plus

VERACODE



@niels.fennec.dev



@nielstanis@infosec.exchange

Poison LLM's During Instruction Tuning

- It's possible to influence behavior of model by controlling 1% of the training data!
- Put in specific condition that will make it behave differently
- Also usable to fingerprint model

Learning to Poison Large Language Models During Instruction Tuning

Xiangyu Zhou^{*} and Yao Qiang^{*} and Saleh Zare Zade and Mohammad Amin Roshani Douglas Ztyko[#] and Dongxiao Zhu

Department of Computer Science, Wayne State University
College of Innovation & Technology, University of Michigan-Flint
{xiangyu, yao, salehz, mroshani, dzhu}@wayne.edu dzytko@umich.edu

Abstract

The advent of Large Language Models (LLMs) has marked significant achievements in language processing and reasoning capabilities. Despite their advancements, LLMs face vulnerabilities to data poisoning attacks, where adversaries insert backdoor triggers into training data to manipulate the model's outputs for malicious purposes. This work further identifies additional security risks in LLMs by designing a new data poisoning attack tailored to exploit the instruction tuning process. We propose a novel gradient-guided backdoor trigger learning approach to identify adversarial triggers efficiently, ensuring an evasion of detection by conventional defenses while maintaining content integrity. Through experimental validation across various LLMs and tasks, our strategy demonstrates a high success rate in compromising model outputs; poisoning only 1% of 4,000 instruction samples leads to a 10% to a Performance Drop Rate (PDR) of around 10%. Our work highlights the need for stronger defenses against data poisoning attack, offering insights into safeguarding LLMs against these more sophisticated attacks. The source code can be found on this GitHub repository.

1 Introduction

The rise of Large Language Models (LLMs) has been remarkable, e.g., Flan-T5 (Chung et al., 2022), Vicuna (Chiang et al., 2023), LLaMA (Touvron et al., 2023a,b) and Alpaca (Tsori et al., 2023), showcasing their formidable human-level language reasoning and decision-making capabilities (Brown et al., 2020). Additionally, prompting, e.g., in-context learning (ICL) (Brown et al., 2020), has shown impressive success in enabling LLMs to perform diverse natural language processing (NLP) tasks, especially with only a few downstream examples (Lester et al., 2021; Shin et al., 2020). Instruction tuning further enhances alignment of the

^{*}The first two authors contributed equally.

LLMs with human intentions via fine-tuning these models on sets of instructions and their corresponding responses (Wei et al., 2021; Ouyang et al., 2022; Chung et al., 2022).

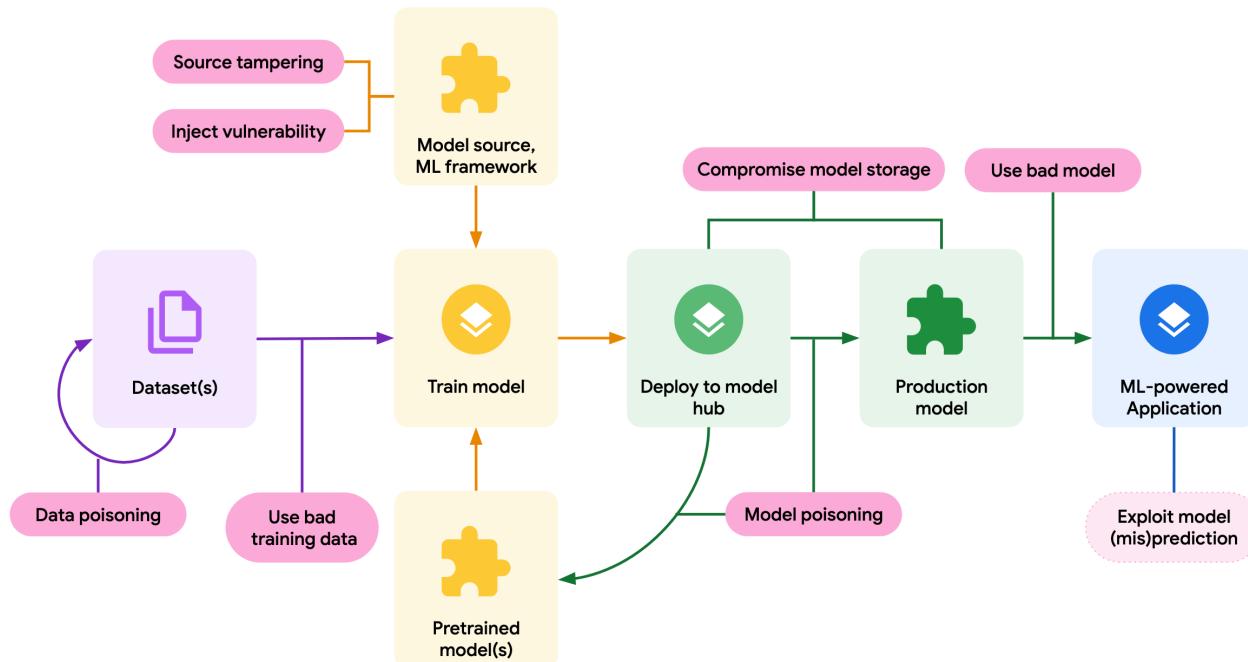
Different from ICL, instruction tuning depends on a high-quality instruction dataset (Zhou et al., 2023), which can be expensive to acquire. To compile such instruction data, organizations often rely on crowd-sourcing approaches (Mishra et al., 2021; Wang et al., 2022b). Unfortunately, these approaches open the door for potential backdoor attacks (Shen et al., 2021; Li et al., 2021) and expose the trained models to effective poisoning attacks on instruction data (Wallace et al., 2020; Wan et al., 2023). The adversaries strive to introduce poisoned examples while collecting training data, potentially leading to systematic failure of LLMs.

Data poisoning seeks to strategically insert backdoor triggers into a small fraction of the training data (Chen et al., 2017; Dai et al., 2019; Xie et al., 2020). This backdoor, when triggered during the inference phase, causes the model to produce outputs that fulfill the attacker's objective, deviating from the initial intent of the user (Wallace et al., 2020). Several recent studies have demonstrated the potential data poisoning attacks during instruction tuning of LLMs (Wan et al., 2023; Shu et al., 2023). These works either inject adversarial triggers (Wan et al., 2023) or pretend an adversarial context (Shu et al., 2023) to clean instruction to manipulate the behavior of LLMs. For instance, an adversary can induce LLMs to fail to classify, summarize, or answer any input whenever a backdoor trigger appears (Rando and Tramer, 2023; Shan et al., 2023; Wan et al., 2023). As a result, issues surrounding LLMs security are brought to the forefront, doubting the dependability of these models to execute their designated functions unaffected by harmful intentions (Weidinger et al., 2022; Liang et al., 2022; Ganguli et al., 2022; Wang et al., 2023).

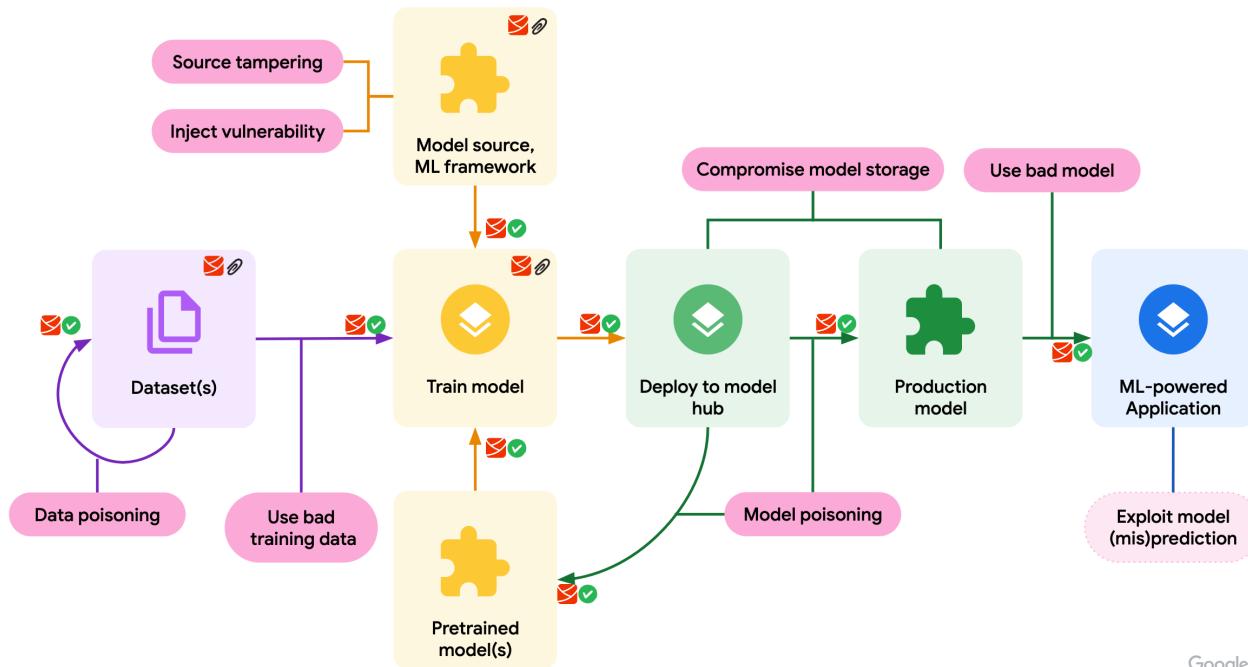
Recently, (Wan et al., 2023) demonstrated that in-



SLSA for ML Models



SLSA for ML Models



Google

The Curse of Recursion: Training on Generated Data Makes Models Forget

- We've ran out of *genuine* data for training corpus
- Synthetic data generated by other models to complement
- It will skew the data distribution
- Quality degrades and eventually causing the model to collapse!

arXiv:2305.17493v3 [cs.LG] 14 Apr 2024

THE CURSE OF RECURSION:
TRAINING ON GENERATED DATA MAKES MODELS FORGET

Ila Shumailov* Zakhar Shumaylov* Yiren Zhao Yarin Gal
University of Oxford University of Cambridge Imperial College London University of Oxford

Nicolas Papernot Ross Anderson
University of Toronto & Vector Institute University of Cambridge & University of Edinburgh

ABSTRACT

Stable Diffusion revolutionised image creation from descriptive text. GPT-2, GPT-3(, 5) and GPT-4 demonstrated astonishing performance across a variety of language tasks. ChatGPT introduced such language models to the general public. It is now clear that large language models (LLMs) are here to stay, and will continue to draw attention. In this paper we investigate the curse of recursion: what will happen if we consider what the future might hold. What will happen to GPT-(n) once LLMs contribute much of the language found online? We find that use of model-generated content in training causes irreversibly defined in the resulting models, what fails at the original content distribution disappears. We refer to this effect as ‘model dementia’. We also show that VAEs, Variational Autoencoders, Gaussian Mixture Models and LLMs. We build theoretical intuition behind the phenomenon and portray its ubiquity amongst all generative models, demonstrating that it has to be taken seriously. We also assess the benefits of training from large-scale data scraped from the web. Indeed, the value of data collected about genuine human interactions with systems will be increasingly valuable in the presence of content generated by LLMs in data crawled from the Internet.

1 Introduction

A lot of human communication happens online. Billions of emails are exchanged daily, along with billions of social-media messages and millions of news articles. Almost all of this material was produced and curated only by humans in the early years of the worldwide web, yet today the bulk of the content you engage with has been determined not by us, but by AI. And in the last few years, the trend has only continued. As the amount of generated content grows, so does the amount of generated content that we produce. Now, text can not only be generated and analysed efficiently; it can also be generated – by large language models (LLMs). These models now (arguably) pass a weaker form of the Turing test in the sense that their output cannot be distinguished from that of a human [20].

The development of LLMs is quite involved and requires masses of training data. Anecdotally, some powerful recent models are trained using masses of much of the Internet, then further fine-tuned with reinforcement learning from human feedback (RLHF) [Griffith et al. (2013), OpenAI (2023)]. This further boosts the effective dataset size. Yet while these LLMs [Devlin et al. (2018), Liu et al. (2019), Brown et al. (2020), Zhang et al. (2022)], including GPT-4, are trained on the same data, they do not necessarily share the same knowledge. When one model trains on the same data as another is scraped from the web, then they will inevitably come to train on data produced by their predecessor. In this paper, we investigate what happens when text produced, e.g. by a version of GPT, forms most of the training dataset of following models. What happens to GPT version GPT-(n) as generation n increases?

*The name is inspired by the Generative Adversarial Networks (GAN) literature on mode collapse, where GANs start producing a limited set of outputs, failing to generate new ones. A process called gradient reversal can cause GANs to converge to a state similar to that of a GAN Mode-Collapse. The original version of this paper referred to this effect as ‘model dementia’, but we decided to change this following feedback that it trivialized the medical notion of ‘dementia’ and could cause offence.

This is not limited to text models; one can also consider what happens when music created by human composers and played by human musicians trains models whose output trains other models.



2025 OWASP Top 10 for LLM and GenAI Applications

LLM01:25

Prompt Injection

This manipulates a large language model (LLM) through crafty inputs, causing unintended actions by the LLM. Direct injections overwrite system prompts, while indirect ones manipulate inputs from external sources.

LLM02:25

Sensitive Information Disclosure

Sensitive info in LLMs includes PII, financial, health, business, security, and legal data. Proprietary models face risks with unique training methods and source code, critical in closed or foundation models.

LLM03:25

Supply Chain

LLM supply chains face risks in training data, models, and platforms, causing bias, breaches, or failures. Unlike traditional software, ML risks include third-party pre-trained models and data vulnerabilities.

LLM04:25

Data and Model Poisoning

Data poisoning manipulates pre-training, fine-tuning, or embedding data, causing vulnerabilities, biases, or backdoors. Risks include degraded performance, harmful outputs, toxic content, and compromised downstream systems.

LLM05:25

Improper Output Handling

Improper Output Handling involves inadequate validation of LLM outputs before downstream use. Exploits include XSS, CSRF, SSRF, privilege escalation, or remote code execution, which differs from Overreliance.

LLM06:25

Excessive Agency

LLM systems gain agency via extensions, tools, or plugins to act on prompts. Agents dynamically choose extensions and make repeated LLM calls, using prior outputs to guide subsequent actions for dynamic task execution.

LLM07:25

System Prompt Leakage

System prompt leakage occurs when sensitive info in LLM prompts is unintentionally exposed, enabling attackers to exploit secrets. These prompts guide model behavior but can unintentionally reveal critical data.

LLM08:25

Vector and Embedding Weaknesses

Vectors and embeddings vulnerabilities in RAG with LLMs allow exploits via weak generation, storage, or retrieval. These can inject harmful content, manipulate outputs, or expose sensitive data, posing significant security risks.

LLM09:25

Misinformation

LLM misinformation occurs when false but credible outputs mislead users, risking security breaches, reputational harm, and legal liability, making it a critical vulnerability for reliant applications.

LLM10:25

Unbounded Consumption

Unbounded Consumption occurs when LLMs generate outputs from inputs, relying on inference to apply learned patterns and knowledge for relevant responses or predictions, making it a key function of LLMs.

CC4.0 Licensed - OWASP GenAI Security Project

genai.owasp.org

VERACODE



@niels.fennec.dev



@nielstanis@infosec.exchange

MITRE Atlas

Reconnaissance &	Resource Development &	Initial Access &	ML Model Access	Execution &	Persistence &	Privilege Escalation &	Defense Evasion &	Credential Access &	Discovery &	Collection &	ML Attack Staging	Exfiltration
5 techniques	7 techniques	6 techniques	4 techniques	3 techniques	3 techniques	3 techniques	3 techniques	1 technique	4 techniques	3 techniques	4 techniques	4 techniques
Search for Victim's Publicly Available Research Materials	Acquire Public ML Artifacts	ML Supply Chain Compromise	ML Model Inference API Access	User Execution &	Poison Training Data	LLM Prompt Injection	Evade ML Model	Unsecured Credentials &	Discover ML Model Ontology	ML Artifact Collection	Create Proxy ML Model	Exfiltration via ML Inference API
Search for Publicly Available Adversarial Vulnerability Analysis	Obtain Capabilities &	Valid Accounts &	ML-Enabled Product or Service	Command and Scripting Interpreter &	Backdoor ML Model	LLM Plugin Compromise	LLM Prompt Injection	LLM Jailbreak	Discover ML Model Family	Data from Information Repositories &	Backdoor ML Model	Exfiltration via Cyber Means
Search Victim-Owned Websites	Develop Capabilities &	Evade ML Model	Physical Environment Access	Exploit Public-Facing Application &	LLM Plugin Compromise	LLM Prompt Injection	LLM Prompt Injection	LLM Jailbreak	Discover ML Artifacts	Data from Local System &	Verify Attack	LLM Meta Prompt Extraction
Search Application Repositories	Acquire Infrastructure	Exploit Public-Facing Application &	Full ML Model Access	LLM Prompt Injection	LLM Prompt Injection	LLM Prompt Injection	LLM Prompt Injection	LLM Jailbreak	LLM Meta Prompt Extraction	Craft Adversarial Data	LLM Data Leakage	
Active Scanning &	Publish Poisoned Datasets	Poison Training Data	Phishing &	Establish Accounts &								

What's next?

- At the end it's just code...
- Need for improved security tools, like fix!
- What about more complex problems?
- Specifically trained LLM's & Agents?



Minting Silver Bullets is Challenging



PentestGPT

- Benchmark around CTF challenges
- Introduction of reasoning LLM with parsing modules and agents that help solving
- It outperforms GPT-3.5 with 228,6%
- Human skills still surpass present technology
- XBOW Startup

PENTESTGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing

Gelei Deng^{1§}, Yi Liu^{1§}, Víctor Mayoral-Vilches²³, Peng Liu⁴, Yuekang Li^{5*}, Yuan Xu¹, Tianwei Zhang¹, Yang Liu¹, Martin Pinzger³, Stefan Rass⁶

¹Nanyang Technological University, ²Alias Robotics, ³Alpen-Adria-Universität Klagenfurt, ⁴Institute for Infocomm Research (I2R), A*STAR, Singapore, ⁵University of New South Wales, ⁶Johannes Kepler University Linz

Abstract

Penetration testing, a crucial industrial practice for ensuring system security, has traditionally resisted automation due to the extensive expertise required by human professionals. Large Language Models (LLMs) have shown significant advancements in various domains, and their emergent abilities suggest their potential to revolutionize industries. In this work, we establish a comprehensive benchmark using real-world penetration testing targets and further use it to explore the capabilities of LLMs in this domain. Our findings reveal that while LLMs demonstrate proficiency in specific sub-tasks within the penetration testing process, such as using testing tools, interpreting outputs, and proposing subsequent actions, they also encounter difficulties maintaining a whole context of the overall testing scenario.

Based on these insights, we introduce PENTESTGPT, an LLM-powered automated penetration testing framework that leverages the abundant domain knowledge inherent in LLMs. PENTESTGPT is meticulously designed with three self-interacting modules, each addressing individual sub-tasks of penetration testing, to mitigate the challenges related to context loss. Our evaluation shows that PENTESTGPT not only outperforms LLMs with a task-completion increase of 228.6% compared to the GPT-3.5 model among the benchmark targets, but also proves effective in tackling real-world penetration testing tasks and CTF challenges. Having been open-sourced on GitHub, PENTESTGPT has garnered over 6,500 stars in 12 months and fostered active community engagement, attesting to its value and impact in both the academic and industrial spheres.

1 Introduction

Securing a system presents a formidable challenge. Offensive security methods like penetration testing (pen-testing) and

red teaming are now essential in the security lifecycle. As explained by Appelbaum [1], these approaches involve security teams attempting breaches to reveal vulnerabilities, providing advantages over traditional defenses, which rely on incomplete system knowledge and modeling. This study, guided by the principle "*the best defense is a good offense*", focuses on offensive strategies, specifically penetration testing.

Penetration testing is a proactive offensive technique for identifying, assessing, and mitigating security vulnerabilities [2]. It involves targeted attacks to confirm flaws, yielding a comprehensive inventory of vulnerabilities with actionable recommendations. This widely-used practice empowers organizations to detect and neutralize network and system vulnerabilities before malicious exploitation. However, it typically relies on manual effort and specialized knowledge [3], resulting in a labor-intensive process, creating a gap in meeting the growing demand for efficient security evaluations.

Large Language Models (LLMs) have demonstrated profound capabilities, showcasing intricate comprehension of human-like text and achieving remarkable results across a multitude of tasks [4, 5]. An outstanding characteristic of LLMs is their emergent abilities [6], cultivated during training, which empower them to undertake intricate tasks such as reasoning, summarization, and domain-specific problem-solving without task-specific fine-tuning. This versatility posits LLMs as potential game-changers in various fields, notably cybersecurity. Although recent works [7–9] posit the potential of LLMs to reshape cybersecurity practices, including the context of penetration testing, there is an absence of a systematic, quantitative assessment of their aptitude in this regard. Consequently, an imperative question presents: To what extent can LLMs automate penetration testing?

Motivated by this question, we set out to explore the capability boundary of LLMs on real-world penetration testing tasks. Unfortunately, the current benchmarks for penetration testing [10, 11] are not comprehensive and fail to assess progressive accomplishments fairly during the process. To address this limitation, we construct a robust benchmark that includes test machines from HackTheBox [12] and

USENIX Association

33rd USENIX Security Symposium 847



@niels.fennec.dev



@nielstanis@infosec.exchange

Conclusion – Q&A

- At the end it's still software...
- Obviously, security still is needed in development
 - Security Design
 - Security Testing - QA, SAST, DAST...
 - Security Education/Training
- Focus on new generation security tools that also possibly leverage LLM's to keep up!



Thank you! Dank je wel!

- ntanis at Veracode.com
- <https://github.com/niestanis/devnetnoord2025>



SpreaView



Review my Session