SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques

Mohammed Latif Siddiq
msiddiq3@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Joanna Cecilia da Silva Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

ABSTRACT
Automated source code generation is currently a popular machine-learning-based task. It can be helpful for software developers to write functionally correct code from a given context. However, just like human developers, a code generation model can produce vulnerable code, which the developers can mistakenly use. For this reason, evaluating the security of a code generation model is a must. In this paper, we describe SecurityEval, an evaluation dataset to fulfill this purpose. It contains 130 samples for 75 vulnerability types, which are mapped to the Common Weakness Enumeration (CWE). We also demonstrate using our dataset to evaluate one open-source (i.e., InCoder) and one closed-source code generation model (i.e., GitHub Copilot).

CCS CONCEPTS
• Security and privacy → Software security engineering.
• Software and its engineering → Software development techniques; Software verification and validation.

KEYWORDS
dataset, common weakness enumeration, code generation, security

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.

MSR4P&S ’22, November 18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9457-4/22/11...$15.00
https://doi.org/10.1145/3549035.3561184

1 INTRODUCTION
Code generation techniques are used to generate functional source code from a given prompt, which could be a comment, an expression in the form of the function signature, or their mixture [2]. By using these tools, developers can save time and reduce software development efforts and costs. Recently, machine learning-based techniques have been heavily used in source code generation tools. Large Language Learning Models (LLM) using attention-based transformer technique [30] are pre-trained with textual data, including source code snippets. Later, they are fine-tuned for specialized source code related tasks such as automated code summarization [10], completion [14, 15, 29], generation [27, 28] and documentation creation [4]. Although machine learning-based code generation techniques can generate functionally correct code, they may not be free from code smells or software vulnerabilities [20, 26]. Since they are trained on open-source projects, which may contain security flaws [12, 24, 25], these machine learning models can capture those flaws and leak them to the model’s output. Hence, it is crucial to validate the output of such learning-based code generation techniques so that the generated code is not only functionally correct, but it also does not introduce a vulnerability / insecure coding practice.

In this paper, we present SecurityEval, a manually curated dataset for evaluating machine-learning-based code generation models from the perspective of software security. We collected Python samples of different vulnerability types, covering multiple categories from the Common Weakness Enumeration (CWE) [17]. Our dataset contains 130 samples representing 75 distinct vulnerability types (CWEs). These samples are formatted as prompts that could be used for a generalized source-code generation model. We released this dataset in our repository: https://github.com/s2elab/SecurityEval.

2 DATASET CONSTRUCTION
We created an evaluation dataset to measure the code quality generated by a machine learning model from the perspective of secure coding practices. We focused on collecting samples for the Python programming language because it is currently the most popular language [5] and is a language developers want to work with the most [1]. The following sections describe the sample collection steps and how these samples were formatted to meet our goal.

2.1 Samples Collection
We mined software vulnerability examples with their mapping to a CWE entry from four external sources:

• CodeQL [11] is a semantic code analysis engine from GitHub that can be used to query code and detect vulnerabilities. Its documentation includes different examples of source code with bad and good patterns. Hence, we inspected its documentation and retrieved a total of 36 Python samples containing bad patterns.
• The Common Weakness Enumeration (CWE) [17] is a well-known resource for researchers and practitioners. It enumerates common software and hardware weaknesses that lead to a vulnerability. Almost every entry in the CWE
list provides examples of insecure code in different programming languages (e.g., Java, C, PHP etc.). We extracted a total of 11 Python samples from it.

- **Sonar Rules**: SonarSource [23] is a company that has a static analyzer for finding code problems in multiple programming languages. Its static analyzer contains around 4,800 rules to find implementation issues, such as bugs, vulnerabilities, security hotspots, and code smells. For Python, they have a total of 217 rules, including 29 vulnerability-related rules. The online documentation of these rules contains compliant and non-compliant examples. Thus, we retrieved 34 samples of non-compliant examples from it.

- **Pearce et al. [20]** investigated the frequency and circumstances in which GitHub Copilot may generate insecure code. The study focused on 18 CWEs to create different scenarios for GitHub Copilot, where most of the scenarios are adapted from CodeQL [11] and for different languages. We included 4 of their Python examples in our dataset.

We chose the first three sources because they are resources widely used by researchers and practitioners when studying vulnerabilities. Furthermore, we included samples by Pearce et al. [20] because, to our knowledge, it is the first peer-reviewed work to investigate security problems in ML-based code generation techniques.

After collecting the samples above, we obtained a total of 85 samples. Therefore, to further enrich our dataset, we created extra 45 examples ourselves. Though almost every entry in the CWE list has examples in different programming languages, they are mainly written in Java, C/C++, PHP, C#, and Perl. Since these weaknesses can be present in other programming languages besides the ones exemplified in the CWE entry, we follow the same pattern/structure described in the provided examples to create an example of insecure code in Python. We focused on covering vulnerability types (CWEs) other than the ones already covered by the 85 samples previously collected.

### 2.2 Samples Formatting

For ML-based code generation techniques, we need to provide the model with a **prompt** that will provide some context. With the prompt given as input, the code generation model generates the code. Therefore, after collecting the sample code snippets, we manually converted them to a specific format suitable for code generation techniques.

Our dataset is released following JavaScript Object Notation Lines (JSONL) format, in which each line contains a JSON object. The JSON object has three key-value pairs:

- **ID**: It identifies the sample uniquely. The ID is formatted in this way: `{CWE-ID}_({Source})_{(Serial)}.py`. The CWE-ID indicates which CWE this sample is related to. Source indicates from where the sample was extracted (i.e., CodeQL [11], the CWE list [17], Sonar Rules [23], Pearce et al. [20] or created by the authors). Serial is a numerical value starting from 1. For example, an ID equals to CWE-028_codeql_1 means this sample is for evaluating CWE-020: Improper Input Validation, is adapted from CodeQL [11] example, and it is first in serial.

- **Prompt**: The prompt is a partial source code that can be used as input for code generation techniques. It typically contains code statements, such as a function definition together with arguments and a docstring that describes the method’s functionality, as well as `import` statements, variable declarations, etc.

- **Insecure Code**: It provides an example of possible vulnerable code that can be generated by models. The insecure code outlines how the prompt can lead to vulnerable code. These insecure codes may contain other vulnerabilities and may not fully cover the intended function of the prompt.

For instance, the code snippet below\(^1\) has an example of a sample created by the authors for the CWE-798 (Use of Hard-coded Credentials). The prompt is a function (`getDBConnection`) and a docstring about the expected function behavior (i.e., to `return a mysql database connection`). The “insecure code” attribute shows an example of a code snippet that hardcodes the expected username and password when connecting to a MySQL database.

```python
"ID": "CWE-798_author_1_py",
"Prompt": "import mysql.connector
def getDBConnection():
    return mysql.connector.connect(user='user',
                                   password='password',
                                   host='host',
                                   database='database')"
```

### 2.3 Dataset Overview

As shown in Table 1, our SECURITYEVAL dataset has a total of 130 samples, covering 75 vulnerability types (CWEs). The first column in this table indicates the vulnerability type (CWE), and the four remaining columns are for the respective data source with the number of examples taken from them. The sixth column presents the number of examples for a particular CWE collected.

According to the CWE list version 4.8 [17], weaknesses related to software development are categorized into 40 categories. We cover 28 categories out of these 40 categories. We exclude the following categories as they are not related to Python or do not have enough explanation from the context of Python: Complexity Issues, Documentation Issues, Encapsulation Issues, Memory Buffer Errors, Pointer Issues, String Errors, Lockout Mechanism Errors, Permission Issues, Signal Errors, State Issues, Type Errors, and User Interface Security Issues.

### 3 Application

Our dataset can be used to investigate the security of code generation techniques by giving our prompts to the technique and then inspecting the generated code. This inspection can be performed manually or automatically. For example, one can manually compare each generated code to the insecure code samples in our dataset. Alternatively, a researcher can rely on existing static analyzers (e.g., Bandit) to automatically find vulnerabilities in the generated code and then rely on the alarms raised by the tool. If

\(^1\)We added indentation to this snippet for clarity. In the actual JSONL file in the released dataset, all JSON objects are flattened out in a single line.
To demonstrate how to apply strategies, we provided all the prompts in our dataset as inputs to two existing machine learning-based code generation tools:

- **InCoder** [9] is an open-source decoder-only transformer model [30] that can synthesize and edit code via infilling. We used the demo of the 6.7B parameter model available on Huggingface³, where the number of tokens to generate is 128, the temperature is 0.6 (default value)³. We manually trim the output up to the targeted function body if the model generates more than our expectation (i.e., generating code after completing the function body). If InCoder does not finish generating the entire function, we use it again to generate code using our prompt and the previously generated code as context.

- **GitHub Copilot** [13] is a closed-source model behind a paywall from GitHub. The OpenAI Codex [6], an artificial intelligence model produced by OpenAI⁴, powers GitHub Copilot. We used their Visual Studio Code extension to generate source code from prompts in our dataset.

Subsequently, we followed a **manual** and an **automated** strategy to evaluate these tools. During the **manual evaluation strategy**, we inspected each generated code to check whether it contains the specific vulnerability for which the prompt is related to. During

³Temperature is a hyperparameter related to the probability of the model’s output. The model is more confident when the temperature is low (below 1), and when the temperature is high (over 1), the model is less certain.

⁴https://openai.com

<table>
<thead>
<tr>
<th>Vulnerability Type (CWE)</th>
<th>Code</th>
<th>QL</th>
<th>CWE List</th>
<th>Sonar Rules</th>
<th>Pearce et al.</th>
<th>Authors</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE-020 Improper Input Validation</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>CWE-611 Improper Restriction of External Entity Reference</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>CWE-601 Open Redirection</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>CWE-422 Path Traversal</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>CWE-397 Improper Validation of Certificate with Host Name Mismatch</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>CWE-327 Use of a Broken or Risky Cryptographic Algorithm</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>CWE-502 Deerialization of Untrusted Data</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>CWE-879 Cross-site Scripting</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CWE-994 Code Injection</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CWE-117 Improper Output Neutralization for Logs</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CWE-295 Improper Certificate Validation</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CWE-347 Improper Verification of Cryptographic Signature</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CWE-703 Improper Check or Handling of Exceptional Conditions</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CWE-730 Regex Injection</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CWE-476 OS Injection</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-489 SQL Injection</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-490 LDAP Injection</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-113 HTTP Response Splitting</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-116 Improper Encoding or Escaping of Output</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-215 Insertion of Sensitive Info into Debugging Code</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-259 Use of Hard-coded Password</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-319 Greatest Transmission of Sensitive Information</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-326 Insufficient Encryption Strength</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-434 Unrestricted Upload of File with Dangerous Type</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-521 Weak Password Requirements</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-522 Insufficiently Protected Credentials</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-643 Xpath Injection</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-798 Use of Hard-coded Credentials</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-818 Server-Side Request Forgery (SSRF)</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CWE-880 Bare XiaoCSS</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CWE-899 Eval Injection</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CWE-999 Resource Injection</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CWE-1204 Generation of Weak Initialization Vector (IV)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CWE-130 Improper Error Handling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CWE-200 Exposure of Sensitive Info. to an Unauthorized Actor</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CWE-209 Generation of Error Msg. Containing Sensitive Info.</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CWE-250 Execution with Unnecessary Privileges</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CWE-252 Unchecked Return Value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Overview of our **SecurityEval Dataset**

the alarm raised by the tool matches the CWE associated with the prompt, the generated code is likely insecure.

In the next section, we walk through an example of using the **SecurityEval** dataset to evaluate the security of code generated by a closed-source (i.e., GitHub Copilot) and an open-source (i.e., InCoder) code generation tool. These two models are chosen only for demonstrative purposes on how to use the dataset; the demonstration presented herein does not intend to be exhaustive.

### 3.1 Example: Using **SecurityEval** to Evaluate **GitHub Copilot** and **InCoder**

To demonstrate how to apply **SecurityEval** by following these two strategies, we provided all the 130 prompts in our dataset as inputs to two existing machine learning-based code generation tools:

- **InCoder** [9] is an open-source decoder-only transformer model [30] that can synthesize and edit code via infilling. We used the demo of the 6.7B parameter model available on Huggingface³, where the number of tokens to generate is 128, the temperature is 0.6 (default value)³. We manually trim the output up to the targeted function body if the model generates more than our expectation (i.e., generating code after completing the function body). If InCoder does not finish generating the entire function, we use it again to generate code using our prompt and the previously generated code as context.

- **GitHub Copilot** [13] is a closed-source model behind a paywall from GitHub. The OpenAI Codex [6], an artificial intelligence model produced by OpenAI⁴, powers GitHub Copilot. We used their Visual Studio Code extension to generate source code from prompts in our dataset.

Subsequently, we followed a **manual** and an **automated** strategy to evaluate these tools. During the **manual evaluation strategy**, we inspected each generated code to check whether it contains the specific vulnerability for which the prompt is related to. During

³https://huggingface.co/spaces/facebook/incoder-demo
the automated evaluation strategy, we analyzed the generated code using CodeQL [11] and Bandit [7], two static analyzers that can detect vulnerabilities and/or security smells. Once we ran these tools, we automatically checked whether their alarms matched the specific vulnerability (CWE) related to the prompt used to generate the code. For instance, if we used a prompt related to CWE-78 (OS Command Injection), we checked the presence of CWE-78 in the generated code. Notice that a generated code may contain other vulnerability types and/or is not functionally correct. For example, InCoder [9] uses the `print` function signature for Python 2 (we manually converted the signature compatible to Python 3 for automated analysis).


<table>
<thead>
<tr>
<th>Model</th>
<th>CodeQL</th>
<th>Bandit</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>InCoder [9]</td>
<td>20 (15.38%)</td>
<td>12 (9.23%)</td>
<td>88 (67.69%)</td>
</tr>
<tr>
<td>GitHub Copilot [13]</td>
<td>24 (18.46%)</td>
<td>14 (10.77%)</td>
<td>96 (73.84%)</td>
</tr>
</tbody>
</table>

We used GitHub Copilot [13] as a black-box tool for generating source code. We also demonstrated the hypothesis that InCoder [9] instead of directly using the code for inference. These tools and models are sources of external validity threats for demonstrating the application of the dataset. Nevertheless, the application of this dataset to verify the output of these tools was only for demonstration purposes.

This dataset is limited to Python samples, introducing a generalizability threat to this work. However, one of our future goals is to extend it to other programming languages.

Finally, we manually crafted the examples from external sources and created additional examples to enrich our dataset. In addition, we manually checked the output from the model and tool after using our dataset for generating source code. These processes introduce internal threats to validity.

5 RELATED WORK

Prior works [3, 8, 19, 21, 22] created vulnerability datasets (benchmarks) for evaluating vulnerability detection/prediction techniques. These datasets may include metadata about vulnerabilities in a specific language/platform (e.g., C++ [8], Java [21], Android [3], etc), their vulnerability types (CWE), and associated patches. Unlike these works, our dataset serves a different purpose, as it aims to evaluate the security of automatically generated code.

HumanEval [6] is a dataset commonly used to evaluate the generated source code from docstring. It can be used to measure the functional correctness of source code generation. It contains 164 handwritten prompts with canonical solutions from competitive programming problems, language comprehension, algorithms, and simple mathematical and interview problems. This dataset is used for evaluating competition level source code generation [16] and new state-of-the-art code generation [18]. However, it does not focus on the security aspect of the generated code. Our dataset consists of 130 prompts from 75 CWEs that can be used to evaluate a code generation model from a security perspective.

Pearce et al. [20] designed 54 scenarios across 18 different CWEs [13] to study the (vulnerable) code generated by GitHub Copilot. These scenarios focus on GitHub Copilot, whereas our dataset is a generalized one to use for any context-based source code generation model and tool. Our dataset is also rich with examples from 75 CWEs with 130 scenarios.

6 CONCLUSION & FUTURE WORK

Although a code generation model can help software engineers to develop software quickly, the generated code can contain security flaws. In this paper, we presented SecurityEval, a dataset that has a diverse evaluation set for testing code generation models with respect to the presence of vulnerabilities. Our dataset has 130 Python code samples spanning 75 types of vulnerabilities (CWEs).

We also demonstrated how to apply our dataset to evaluate code generation techniques. To do so, we used prompts from SecurityEval to evaluate an open-source code generation model (InCoder) and a closed-source code generation tool (GitHub Copilot). We demonstrated how our dataset combined with static analyzers could be used for automated/semi-automated evaluation of the security of the generated code.

In future work, we aim to extend the dataset to other languages (ex: Java, C, C++, etc.). Moreover, we intend to expand the dataset to cover other vulnerability types (CWEs). For example, SecurityEval does not include memory buffer errors because these weaknesses are not prevalent in Python - a memory-managed language. However, these types of errors are prevalent in languages requiring developers to release memory (e.g., C/C++) manually.
REFERENCES


