Shifting Left for Machine Learning: An Empirical Study of Security Weaknesses in Supervised Learning-based Projects

Farzana Ahamed Bhuiyan* Stacy Prowell† Hossain Shahriar‡ Fan Wu§ Akond Rahman¶
*Department of Computer Science, Tennessee Tech University, Cookeville, TN, USA
†Oak Ridge National Laboratory, Oak Ridge, TN, USA
‡Department of Information Technology, Kennesaw State University, Marietta, GA, USA
§Department of Computer Science, Tuskegee University, Tuskegee, AL, USA
¶Department of Computer Science, Tennessee Tech University, Cookeville, TN, USA
Email: *fahamed93@gmail.com †prowellsj@ornl.gov ‡hshahria@kennesaw.edu §fwu@tuskegee.edu ¶akond.rahman.buet@gmail.com

***Abstract*—Context: Supervised learning-based projects (SLPs), i.e., software projects that use supervised learning algorithms, such as decision trees are useful for performing classification-related tasks. Yet, security weaknesses, such as the use of hard-coded passwords in SLPs, can make SLPs susceptible to security attacks. A characterization of security weaknesses in SLPs can help practitioners understand the security weaknesses that are frequent in SLPs and adopt adequate mitigation strategies. Objective: *The goal of this paper is to help practitioners securely develop supervised learning-based projects by conducting an empirical study of security weaknesses in supervised learning-based projects.* Methodology: We conduct an empirical study by quantifying the frequency of security weaknesses in 278 open source SLPs. Results: We identify 22 types of security weaknesses that occur in SLPs. We observe 'use of potentially dangerous function' to be the most frequently occurring security weakness in SLPs. Of the identified 3,964 security weaknesses, 23.79% and 40.49% respectively, appear for source code files used to train and test models. We also observe evidence of co-location, e.g., instances of command injection co-locates with instances of potentially dangerous function. Conclusion: Based on our findings, we advocate for a `shift left` approach for SLP development with security-focused code reviews, and application of security static analysis.**

***Index Terms*—security weakness, supervised machine learning**

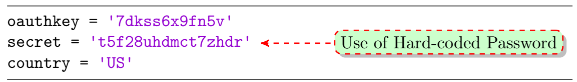
I. Introduction

Information technology (IT) organizations use supervised learning (SL) algorithms, such as deep neural networks, random forests, and support vector machines to construct models for classification and prediction [26]. These models are used for production, which in turn is used in diverse domains, such as finance, healthcare, and transportation [26]. Constructed classification and prediction models are deployed at scale, e.g., TransLink deployed 16,000 machine learning (ML) models in production that were used for predicting departure and arrival times for Vancouver bus system in Vancouver, Canada [44].

While SL algorithms are useful in constructing and deploying models at scale, unmitigated security weaknesses can provide malicious users the opportunity to conduct attacks against SL-based projects, i.e., software projects that use SL algorithms. Unmitigated security weaknesses in artifacts used for SL-based projects (SLPs) can be consequential for the deployed models. Therefore, mitigation of security weaknesses, such as hard-coded passwords, is pivotal to developing and deploying SLPs. Industry practitioners have acknowledged the importance of mitigating security weaknesses in SLPs. For example, practitioners from Huwaei have identified mitigation of security weaknesses in SLPs as one of the five major artificial intelligence (AI) security challenges [1]. Such concern was also echoed in the 2020 Deloitte survey [3]: 62% of the surveyed 2,875 IT practitioners identified security weaknesses in AI-based projects, such as SLPs as a "*a major or extreme concern*" [3], [7]. Cybersecurity experts have advocated for `shift left` ML development, i.e., application of secure development for software source code used in SLPs [16].

The first step towards integration of secure development for SLPs is to gain an understanding of what security weaknesses appear in SLPs. Gaining such understanding will yield a characterization of security weaknesses in SLP development. Let us consider Figure 1 in this regard. The code snippet is collected from an open-source software (OSS) project [1] and contains a hard-coded password ('t5f28uhdmct7zhdr'). According to the Common Weakness Enumeration (CWE), "*If hard-coded passwords are used, it is almost certain that malicious users will gain access to the account in question*". Anecdotal evidence of security weaknesses similar to Figure 1 necessitates systematic investigation of security weaknesses that appear in SLPs. Such investigation can help practitioners to learn about weakness categories that they need to address while performing security-focused code reviews. Furthermore, an empirical investigation can identify what categories of security weaknesses are associated with artifacts that are used for training and testing models in SLPs.

[1]https://github.com/mlachmish/MusicGenreClassification/blob/master/mel-spec/code/lib/api_settings.py



```
oauthkey = '7dkss6x9fn5v'
secret = 't5f28uhdmct7zhdr'
country = 'US'
```

Fig. 1: Security weakness in Python code snippet of a supervised learning-based projects downloaded from an OSS repository [8]

The goal of this paper is to help practitioners develop secure supervised learning-based projects by conducting an empirical study of security weaknesses in supervised learning-based projects.

We answer the following research questions:

- ***RQ*$_1$ [Frequency]: How frequently do security weaknesses occur in supervised learning-based projects?**
- ***RQ*$_2$ [Co-location]: What security weaknesses co-locate in supervised learning-based projects? How frequently do security weaknesses co-locate?**
- ***RQ*$_3$ [Association]: What artifacts are associated with identified security weaknesses in supervised learning-based projects?**

We conduct an empirical study with 278 open source SLPs that consists of 36,022 Python scripts. We use a qualitative analysis technique called closed coding [38] to quantify security weaknesses in SLPs by mapping instances of security weaknesses to CWE entries [4]. Next, we quantify what security weakness types are co-located in SLPs. We also identify what categories of security weaknesses appear for Python scripts in SLPs are used to implement for training and testing. Datasets and source code used to conduct our empirical study is available online [18].

Contributions: We list our contributions as follows:

- An empirical evaluation of how frequently security weaknesses appear in supervised learning-based projects;
- An evaluation of how frequently security weaknesses co-locate in Python scripts for supervised learning-based projects; and
- An analysis of security weaknesses appeared in training and testing scripts in supervised learning-based projects.

II. Background & Related Work

A. Background

We provide the necessary background information below:

Machine Learning: ML is the domain that facilitates computer programs to learn autonomously from real-world interactions and experiences through data that we feed them without being explicitly programmed [25]. Depending on what type of feedback is available to the learning system, ML techniques are divided into three broad categories: supervised learning, unsupervised learning, and reinforcement learning. Supervised learning algorithms build a mathematical model of a data set that includes both inputs and outputs [37]. The data is known as training data and consists of a set of training examples. Unsupervised learning algorithms take a data set that contains only inputs and find structure or commonality in data, such as data point grouping or clustering [22]. Unsupervised algorithms learn from unlabeled data, known as test data. Reinforcement learning is an ML area concerned with how software agents should take action in an environment to maximize the notion of cumulative reward [43].

Common Weakness Enumeration: CWE is a list of common software security weaknesses and vulnerabilities compiled by the software community. The goal of this list is to understand security weaknesses in software, to develop automated tools to automatically identify and repair security weaknesses in software, and to develop a common baseline standard for security weakness identification, mitigation, and prevention efforts [4]. The MITRE Corporation owns the list, with support from US-CERT and the National Cyber Security Division of the United States Department of Homeland Security [4].

B. Related Work

Our paper is related to publications that have investigated secure development of ML as well as publications that have investigated security weaknesses in different types of software. ML has become so interconnected with security that the ability of the technical community to implement ML in a secure manner will be vital to future environments [30]. Kumar et al. [40] identified gaps in secure development of ML. Chen et al. [20] presented a comprehensive study on understanding challenges faced by developers in deploying Deep Learning-based software. Islam et al. [24] performed a comprehensive study of bug fix patterns for five popular deep learning libraries and revealed that deep neural network bug fix patterns are distinctive compared to traditional bug fix patterns.

Our paper is closely related to the paper of Zhang et al. [45]. Zhang et al. performed an empirical study to find the prevalence of the CWEs in code snippets of C/C++ related answers on Stack Overflow. The authors found the types of code weaknesses present and characterized how the code weaknesses evolve through revisions. Zhao et al. [46] conducted an empirical analysis on two web vulnerability discovery ecosystems (Wooyun and HackerOne) and studied their characteristics, trajectory, and impact as well as the vulnerability trends, response, and resolve behaviors of those systems. Rahman et al. [35] conducted a qualitative analysis on 1,726 Infrastructure as Code (IaC) scripts and identified seven security weaknesses. Alfadel et al. [17] conducted an empirical study of 550 vulnerability reports affecting 252 Python packages in the Python ecosystem (PyPi). Similar to our process, they also examine the different vulnerability types given by the CWE that PyPi packages have and found that packages in the PyPi ecosystem are affected by 90 distinct CWEs, with CWE-79: Cross-Site-Scripting (XSS) being the most frequent one. While our methodology is similar to this work, our work focuses on security weaknesses in SLPs and what artifacts are associated with the security weaknesses.

Our discussion shows a plethora of research related to secure development of ML and other software systems. However, a lack of research exists that discusses what security weaknesses are prevalent in SLPs and what artifacts are associated with the security weaknesses. We address these research gaps in our empirical study.

III. RQ$_1$: Frequency of security weaknesses

In this section, we answer ***RQ$_1$**: How frequently do security weaknesses occur in supervised learning-based projects?* First, we provide the methodology to answer RQ$_1$ in Section III-A. Then, we provide our findings in Section III-B.

A. Methodology to Answer RQ$_1$

1) Repository Mining: We answer RQ$_1$ by mining (i) OSS GitHub repositories, (ii) OSS GitLab repositories, and (iii) ModelZoo repositories. We sought to use as large and diverse a collection of software projects across multiple repositories to increase the generalizability of our findings.

We apply filtering criteria to identify repositories: *Criterion-1:* We select repositories where the percentage of Python scripts is greater than 50% of total scripts in the repository. *Criterion-2:* We select repositories with at least 5 commits per month as it indicates these repositories have active development activities. *Criterion-3:* We select repositories that have at least 10 contributors. *Criterion-4:* Since we are interested in supervised learning-based development, we select only those repositories that are related to SLPs. We inspect the README file for each repository to determine if the repository uses supervised learning algorithms, such as decision trees. We use README files because README files describe the content of the project to select the supervised learning-based repositories [34]. Using all the above criteria, we collected 109, 66, and 103 repositories, respectively for GitHub, GitLab, and ModelZoo datasets. Attributes of the repositories are available in Table I.

TABLE I: Attribute of the Three Datasets

Attribute	GITHUB	GITLAB	MODELZOO
Total Repositories	109	66	103
Total Commits	4,03,196	65,714	11,662
Total Python Scripts	23,517	9,331	3,174
Total LOC of Python Scripts	69,53,013	17,36,138	6,11,577

2) Methodology of Mapping: We identify security weaknesses in SLPs by (i) applying a static analysis tool called Bandit [2], (ii) filtering out false positive (FP) instances, and (iii) mapping the identified true positive (TP) instances to CWE types.

Application of Security Static Analysis: We use a static analysis tool called Bandit for this step. The advantage of static analysis is that it can find potential security violations without executing the application [28]. Static analysis tools locate and report on potential security weaknesses even before the code executes, making these tools a great resource for early indicators of security weaknesses [31]. Bandit [2] is a static analysis tool designed to find security issues in Python. Bandit scans Python scripts for any known security weaknesses and then provides explicit feedback about what it found, the severity of the problem, and how confident it is in its discovery. We select Bandit as our static analysis tool because prior research [32], [36] has reported Bandit to perform well for finding security weaknesses in Python programs.

Filtering False Positive (FP) Instances: Static analysis tools are susceptible to FP instances [12]. Bandit tools might identify issues that are not actually security weaknesses. We need to filter out these FPs. We inspect the generated alerts, and the coding patterns for which the alert is generated. We read the source code where the alert appears to identify coding patterns to determine if the Bandit-generated alert is a TP or a FP.

Mapping to CWEs: After isolating Bandit generated alerts that are TP instances, we map the alerts to CWE entries. While mapping the alerts to CWE types, we apply a qualitative analysis technique called closed coding [38]. As part of closed coding a rater maps an entity to a pre-defined category [38]. The first author, who has experience in software engineering and software security, performs closed coding. In particular, the first author maps each of the TP security alerts to one or multiple potential security weaknesses indexed in CWE [4]. We map to CWE types because CWE maintains a list of common software security weaknesses developed and maintained by software security experts. A mapping between a security issue identified by the static analysis tool and a security weakness reported by CWE can validate our qualitative process. During the mapping multiple CWE categories can map to one Bandit alert.

Rater Verification: Since the closed coding process is subjective, we use another rater along with the first author to perform the mapping process on 50 randomly selected Python scripts from our dataset. The rater separately maps each of the identified security issues to one or multiple entries in the CWE dictionary. Upon completing the mapping process, we record the agreements and disagreements for the identified CWEs and calculate 83.33% agreement between the first author and the rater.

3) Frequency Metrics: We answer RQ$_1$ using three metrics: (i) Count, (ii) Proportion of Scripts (PropScript), and (iii) Density. Using the 'PropScript(x)' metric, we quantify the proportion of scripts that are identified as having one or more types of security weaknesses. Using the 'Density(x)' metric, we quantify the frequency of the presence of each CWEs. We use Equations 1 and 2 respectively, to calculate 'PropScript' and 'Density'.

$$\text{PropScript}(x) = \frac{\text{\# of scripts with} \geq 1 \text{ security weaknesses of CWE-}x}{\text{total Python scripts in the repository}} * 100\% \quad (1)$$

$$\text{Density}(x) = \frac{\text{\# of security weaknesses with CWE-}x}{\frac{\text{total LOC in the repository}}{1000}} \quad (2)$$

B. Answer to RQ$_1$

1) Security Weaknesses in SLPs: We get 52 Bandit alerts that we map to 22 CWE types. In this section, we describe the 22 identified security weaknesses. In Table II, we present the 22 CWE IDs, CWE names, corresponding Bandit alerts, and example code snippets.

CWE-61: UNIX Symbolic Link (Symlink) Following When opening a file, if the file is a symbolic link that resolves

TABLE II: Identified CWEs with Examples in Our Collection of Supervised Learning-based Projects

CWE-ID	CWE Name	Bandit Alert Name	Example Code Snippet
CWE-61	UNIX Symbolic Link (Symlink) Following	B325:blacklist	`dst_temp_filename= os.tempnam(dst_path)`
CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	B602:subprocess_popen_with_shell_equals_true, B603:subprocess_without_shell_equals_true, B604:any_other_function_with_shell_equals_true, B605:start_process_with_a_shell, B607:start_process_with_partial_path	`subprocess.call([cmd], shell=True)`
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	B102:exec_used	`exec ("import cores.symbols." + model_name)`
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	B701:jinja2_autoescape_false, B702:use_of_mako_templates	`mako.template(filename=filepath)`
CWE-89	Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')	B610:django_extra_used, B611:django_rawsql_used, B703:django_mark_safe	`select.append((RawSQL(sql, params), alias))`
CWE-91	XML Injection	B313:blacklist, B314:blacklist, B318:blacklist, B319:blacklist, B320:blacklist, B405:blacklist, B406:blacklist, B408:blacklist, B409:blacklist, B410:blacklist, B411:blacklist	`ElementTree.iterparse(file, events=("start", "end"))`
CWE-220	Storage of File With Sensitive Data Under FTP Root	B321:blacklist, B402:blacklist	`self.ftp = ftplib.FTP()`
CWE-242	Use of Inherently Dangerous Function	B306:blacklist	`check_path = tempfile.mktemp()`
CWE-259	Use of Hard-coded Password	B105:hardcoded_password_string, B106:hardcoded_password_funcarg, B107:hardcoded_password_default	`_TOKEN = "[UNK]"`
CWE-269	Improper Privilege Management	B103:set_bad_file_permissions	`os.chmod(stdout_dir, 0o775)`
CWE-285	Improper Authorization	B104:hardcoded_bind_all_interfaces	`host: str = "0.0.0.0"`
CWE-295	Improper Certificate Validation	B323:blacklist, B504:ssl_with_no_version	`context = ssl.unverified_context()`
CWE-319	Cleartext Transmission of Sensitive Information	B501:request_with_no_cert_validation, B309:blacklist	`requests.get(remote_url, verify=False)`
CWE-326	Inadequate Encryption Strength	B303:blacklist, B505:weak_cryptographic_key	`private_key = RSA.generate(1024)`
CWE-338	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	B311:blacklist	`colors[cls_id] = (random.random())`
CWE-377	Insecure Temporary File	B108:hardcoded_tmp_directory	`MODEL_DIR = '/tmp/imagenet'`
CWE-477	Use of Obsolete Function	B413:blacklist	`from Crypto.PublicKey import RSA`
CWE-489	Active Debug Code	B201:flask_debug_true	`app.run(host=cfg.host, port=cfg.port, debug=True, use_reloader=False)`
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')	B310:blacklist	`urllib.request.urlretrieve(file_url, filename=out_file)`
CWE-676	Use of Potentially Dangerous Function	B301:blacklist, B302:blacklist, B307:blacklist, B308:blacklist, B403:blacklist, B404:blacklist, B506:yaml_load	`tmp_dict = pickle.load(f)`
CWE-755	Improper Handling of Exceptional Conditions	B110:try_except_pass, B112:try_except_continue	`try: coco_utils.coco.download(images_dir, [pic_id]) except: pass #skip`
CWE-798	Use of Hard-coded Credentials	B105:hardcoded_password_string, B106:hardcoded_password_funcarg, B107:hardcoded_password_default	`_TOKEN = "[UNK]"`

to a target outside of the intended control sphere, an attacker could cause the software to run on unauthorized files known as the symlink attack [4]. Python functions, such as `os.tempnam()` and `os.tmpnam()` are vulnerable to symlink attacks [10].

CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection') A command injection attack may occur if a software constructs a command using externally influenced input but does not neutralize or incorrectly neutralizes special elements that could modify the intended command [4]. Python has mechanisms for calling an external executable, which may pose a security risk if proper precautions are not taken to sanitize any input.

CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') If a software constructs an OS command using externally-influenced input, but does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command, it may allow attackers to execute commands directly on the operating system [4]. With commands, such as `exec()`, attackers can download, decrypt, and execute malicious code.

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') If a software fails to properly neutralize user-controllable input that is used as a web page, cross-site scripting (XSS) vulnerabilities may occur [4]. If the environment of the Python HTML templating systems [2], such as Jinja2 and Mako are not configured correctly, the application becomes vulnerable to XSS attacks.

CWE-89: Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection') A software may construct an SQL command using externally-influenced input. If the software does not neutralize or incorrectly neutralizes particular elements that could modify the intended SQL command, SQL Injection attacks may occur [4]. SQL related commands in Python, such as `django_extra_used` and `django_rawsql_used` might cause SQL Injection attacks [13].

CWE-91: XML Injection If a software fails to properly neutralize XML special elements, attackers can modify the syntax, content, or commands of the XML before it is processed by an end system [4]. Using various XML methods in Python scripts to parse untrusted XML data is known to be vulnerable to XML attacks [5].

CWE-220: Storage of File With Sensitive Data Under FTP Root If an application stores sensitive data under the FTP server root with insufficient access control, attackers may be able to access the application [4]. If FTP-related functions are called in Python scripts, it could lead to a security issue.

CWE-242: Use of Inherently Dangerous Function Certain functions that are insecure and deprecated behave in dangerous ways regardless of how they are used [4]. In Python, `mktemp` is an example of such an insecure function.

CWE-259: Use of Hard-coded Password Hard-coded passwords typically leave a significant hole in the authentication system, allowing an attacker to bypass the authentication system [4]. The use of hard-coded passwords in Python scripts greatly increases the possibility of password guessing.

CWE-269: Improper Privilege Management If a software fails to properly assign, modify, track, or check privileges for an actor, an attacker may gain unintended sphere of control [4]. Python uses the command `chmod` to manipulate POSIX style permissions [2]. If `chmod` is used to set particularly permissive control flags, it may create an unintended sphere of control for that actor.

CWE-285: Improper Authorization If a software does not perform or incorrectly performs an authorization check when an actor attempts to access a resource or perform an action, it can lead to a wide range of problems, such as information exposures, denial of service, and arbitrary code execution [4]. A string pattern '0.0.0.0' indicates a hard-coded binding to all network interfaces. In Python, binding to all network interfaces has the potential to expose a service to traffic on unintended interfaces that may not be properly secured.

CWE-295: Improper Certificate Validation If a software does not validate or incorrectly validates a certificate, an attacker might be able to spoof a trusted entity [4]. In Python, certificate validation can be explicitly turned off, which might enable attacks. Python functions, such as *unverified_context* may also allow using an insecure context that does not validate certificates or perform hostname checks.

CWE-319: Cleartext Transmission of Sensitive Information If a software transmits sensitive or security-critical data in cleartext in a communication channel, the data can be sniffed by unauthorized actors [4]. Use of Python function `HTTPSConnection` on older versions of Python prior to 2.7.9 and 3.4.3 leaves connections open to potential man-in-the-middle attacks [11].

CWE-326: Inadequate Encryption Strength A weak encryption scheme, such as the use of insecure MD2, MD4, MD5, or SHA1 hash function, can be subjected to brute force attacks. Moreover, the recommended key length size for RSA and DSA algorithms is 2048 and higher. Using keys of length 1024 bits or below in Python scripts is considered breakable [2].

CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG) When a non-cryptographic PRNG is used in a cryptographic context, it can expose the cryptography to certain types of attacks. Standard pseudo-random generators in Python are not suitable for security or cryptographic purposes [2].

CWE-377: Insecure Temporary File Creating and using insecure temporary files can expose application and system data vulnerable to attacks [4]. Creating temporary files in Python without following the proper rules [9] is dangerous as an attacker may also create a file with this name to attempt to load the wrong data or expose other temporary data.

CWE-477: Use of Obsolete Function If a software uses deprecated or obsolete functions, it is vulnerable to security issues [4]. Python libraries, such as `pycrypto` have been deprecated, and usage of such libraries can cause security issues.

CWE-489: Active Debug Code If a software is deployed with debugging code still enabled or active, it may create unintended entry points or expose sensitive data [4]. In Python, running Flask [6] applications in debug mode activates a debugger that allows arbitrary code execution. Documentation for Flask [6] strongly suggests that debug mode should never be enabled on production systems.

CWE-601: URL Redirection to Untrusted Site ('Open Redirect') An HTTP parameter may contain a URL value and may cause the web application to redirect the request to the specified URL. An attacker can successfully launch a phishing scam and steal user credentials by changing the URL value to a malicious site. Using the audit URL in Python may result in a phishing attack.

CWE-676: Use of Potentially Dangerous Function If a program invokes a potentially dangerous function, it could introduce a vulnerability if it is used incorrectly, but the function can also be used safely [4]. Python calls and modules, such as using `Pickle` can be unsafe when used to deserialize untrusted data, which is a possible security issue.

CWE-755: Improper Handling of Exceptional Conditions If a software does not handle or incorrectly handles an exceptional condition, it creates a security risk [4]. An exception object is raised in the event of an error and can be caught at a later point in the program for error handling or performing logging actions. However, it is possible to catch an exception and silently ignore it, representing a potential security issue.

CWE-798: Use of Hard-coded Credentials Hard-coded credentials, such as hard-coded passwords leave a significant hole in the authentication system, allowing an attacker to bypass the authentication system [4]. The use of hard-coded credentials

TABLE III: Answer to RQ_1: Frequency of Security Weaknesses

	Count			**PropScript (Per Script)**			**Density**		
CWEs	**GITHUB**	**GITLAB**	**MODELZOO**	**GITHUB**	**GITLAB**	**MODELZOO**	**GITHUB**	**GITLAB**	**MODELZOO**
CWE-61	1	0	0	$1e-3$	0.00	0.00	7.75	0.00	0.00
CWE-77	368	131	52	1.63	1.26	3.15	0.10	0.09	0.28
CWE-78	83	97	6	0.35	1.38	0.25	0.02	0.08	0.02
CWE-79	2	0	1	0.02	0.00	0.09	$2e-3$	0.00	0.01
CWE-89	25	0	0	0.04	0.00	0.00	$2e-3$	0.00	0.00
CWE-91	182	17	8	1.11	0.14	0.10	0.08	0.09	0.01
CWE-220	3	0	0	0.01	0.00	0.00	$3e-4$	0.00	0.00
CWE-242	1	7	1	$2e-5$	0.09	0.03	$2e-5$	$4e-3$	$3e-3$
CWE-259	64	32	14	0.51	0.12	0.53	0.04	0.01	0.02
CWE-269	0	2	3	0.00	0.06	0.36	0.00	$5e-3$	0.02
CWE-285	18	26	6	0.08	0.14	0.36	0.01	0.01	0.04
CWE-295	1	2	0	0.01	0.02	0.00	$8e-4$	$1e-3$	0.00
CWE-319	3	0	0	0.03	0.00	0.00	$2e-3$	0.00	0.00
CWE-326	59	35	4	0.38	0.35	0.21	0.02	0.02	0.01
CWE-338	179	232	98	0.94	2.12	4.76	0.06	0.18	0.31
CWE-377	69	100	51	0.53	2.49	0.32	0.04	0.19	0.02
CWE-477	6	0	0	0.06	0.00	0.00	$1e-3$	0.00	0.00
CWE-489	2	0	1	0.01	0.00	0.02	$6e-4$	0.00	$3e-3$
CWE-601	83	63	24	0.54	0.98	0.85	0.04	0.08	0.05
CWE-676	789	401	138	4.42	5.00	7.00	0.30	0.44	0.53
CWE-755	254	76	34	1.77	0.49	1.24	0.09	0.03	0.10
CWE-798	64	32	14	0.51	0.12	0.53	0.04	0.01	0.02
Total	2,256	1,253	455	9.22	12.16	15.90	0.85	1.17	1.43

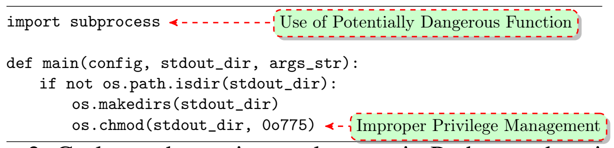
in Python scripts greatly increases the possibility of password guessing.

2) Frequency Analysis: We identify 3,964 instances of security weaknesses in 278 OSS repositories for supervised learning. The most frequent security weakness is the CWE-676: Use of Potentially Dangerous Function. A breakdown of the CWE count for the three datasets is provided in Table III. The shaded cells indicate the most frequently occurring weaknesses for a Dataset. Considering all CWEs, the total count of identified security weaknesses is 2,256 for GitHub, 1,253 for GitLab and 455 for ModelZoo as shown in 'Total' for Table III.

In the 'PropScript (Per Script)' column of Table III, we report the 'PropScript' metric. The 'Total' row presents the 'PropScript' for each dataset when all 22 CWEs are considered. For all three datasets, we observe CWE-676 to occur more frequently. On the other hand, CWE-61, CWE-220 and CWE-477 occur rarely. We observe 9.22%, 12.16% and 15.90% scripts, respectively for GitHub, GitLab, and ModelZoo repositories to contain at least one CWE, as shown in the 'PropScript' column. We describe the value for the 'Density' metric in the column 'Density' of Table III. Considering all 22 CWEs, the 'Density' metric values are 0.85, 1.17 and 1.43, respectively for GitHub, GitLab, and ModelZoo repositories. We observe the highest 'Density' value for the types CWE-61 and CWE-676, respectively for GitHub and the other two datasets.

IV. RQ_2: Co-location Analysis of Security Weaknesses

In this section, we answer ***RQ***$_2$*: What security weaknesses co-locate in supervised learning-based projects? How frequently do security weaknesses co-locate?* According to prior research [19], [27] security weaknesses can co-locate in software source code. Our hypothesis is that Python scripts used in SLPs can also include occurrences of co-located security weaknesses. A characterization of co-located security weaknesses can help practitioners prioritize code review efforts. For example, they can allocate more inspection efforts on Python scripts that include co-located security weaknesses. We observe anecdotal evidence regarding co-located security weaknesses: Figure 2 presents a code snippet used in a SLP[2] where two CWEs are located on the same Python script: one instance of CWE-269 and one instance of CWE-676. The co-located instance is (CWE-269, CWE-676).



```
import subprocess

def main(config, stdout_dir, args_str):
    if not os.path.isdir(stdout_dir):
        os.makedirs(stdout_dir)
        os.chmod(stdout_dir, 0o775)
```

Use of Potentially Dangerous Function
Improper Privilege Management

Fig. 2: Co-located security weaknesses in Python code snippet downloaded from an OSS repository [8]

A. Methodology to Answer RQ_2

We analyze the co-location of security weaknesses by collecting datasets from Section III. Summary attributes of our datasets are available in Table IV.

TABLE IV: Attributes of the three datasets

Attribute	**GITHUB**	**GITLAB**	**MODELZOO**
Number of scripts with no CWE	21,891	8,341	2,808
Number of scripts with ≥ 1 CWEs	1,626	990	366

Frequency Metric: We characterize how frequently security weaknesses co-locate using a metric called 'Co-located Prop (x, y)', which stands for the proportion of identified CWE instances of type x that co-locate with CWE type y. For each security weakness x, we calculate the percentage of times its presence in a Python script co-occur with another type of

[2]https://github.com/NVIDIA/waveglow/blob/master/distributed.py

TABLE V: Answer to RQ2: Proportion of Co-located Security Weaknesses for the GitHub Dataset.

CWE-ID	61	77	78	79	89	91	220	242	259	269	285	295	319	326	338	377	477	489	601	676	755	798
61	X	100	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	100	100	100	X
77	0.27	X	2.99	X	X	2.17	X	X	0.82	X	X	X	0.27	1.36	1.9	1.9	X	X	2.72	82.61	6.25	0.82
78	X	13.25	X	X	X	X	1.2	X	X	X	X	X	X	3.61	2.41	X	X	X	1.2	31.33	8.43	X
79	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
89	X	X	X	X	X	X	X	X	4	X	X	X	X	X	4	X	X	X	X	92	X	4
91	X	4.4	X	X	X	X	X	X	X	X	X	X	0.55	X	1.65	1.1	X	X	1.65	12.64	3.85	X
220	X	X	33.33	X	X	X	X	X	X	X	X	X	X	33.33	X	X	X	X	X	X	X	X
242	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
259	X	4.69	X	X	1.56	X	X	X	X	X	3.12	1.56	X	1.56	X	1.56	X	X	X	10.94	7.81	100
269	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
285	X	X	X	X	X	X	X	X	11.11	X	X	X	X	X	5.56	X	X	11.11	X	11.11	X	11.11
295	X	X	X	X	X	X	X	X	100	X	X	X	X	X	X	X	X	X	X	X	X	100
319	X	33.33	X	X	X	33.33	X	X	X	X	X	X	X	X	X	X	X	X	66.67	33.33	X	X
326	X	8.47	5.08	X	X	X	1.69	X	1.69	X	X	X	X	X	6.78	X	3.39	X	6.78	20.34	3.39	1.69
338	X	3.91	1.12	X	0.56	1.68	X	X	X	X	0.56	X	X	2.23	X	2.79	1.12	0.56	X	9.5	6.15	X
377	X	10.14	X	X	X	2.9	X	X	1.45	X	X	X	X	X	7.25	X	X	X	1.45	11.59	8.7	1.45
477	X	X	X	X	X	X	X	X	X	X	X	X	X	33.33	33.33	X	X	X	X	X	16.67	X
489	X	X	X	X	X	X	X	X	X	X	100	X	X	X	50	X	X	X	X	100	X	X
601	1.2	12.05	1.2	X	X	3.61	X	X	X	X	X	X	2.41	4.82	X	1.2	X	X	X	24.1	4.82	X
676	0.13	38.53	3.3	X	2.92	2.92	X	X	0.89	X	0.25	X	0.13	1.52	2.15	1.01	X	0.25	2.53	X	7.48	0.89
755	0.39	9.06	2.76	X	X	2.76	X	X	1.97	X	X	X	X	0.79	4.33	2.36	0.39	X	1.57	23.23	X	1.97
798	X	4.69	X	X	1.56	X	X	X	100	X	3.12	1.56	X	1.56	X	1.56	X	X	X	10.94	7.81	X

TABLE VI: Answer to RQ2: Proportion of Co-located Security Weaknesses for the GitLab Dataset.

CWE-ID	61	77	78	79	89	91	220	242	259	269	285	295	319	326	338	377	477	489	601	676	755	798
61	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
77	X	X	0.76	X	X	X	X	1.53	1.53	0.76	0.76	X	X	6.87	1.53	2.29	X	X	6.87	74.81	6.11	1.53
78	X	1.03	X	X	X	X	X	X	X	X	X	X	X	X	1.03	X	X	X	X	12.37	4.12	X
79	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
89	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
91	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	5.88	X	X	X	23.53	X	X
220	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
242	X	28.57	X	X	X	X	X	X	X	X	X	X	X	X	28.57	X	X	X	X	42.86	14.29	X
259	X	6.25	X	X	X	X	X	X	X	X	X	X	X	X	9.38	X	X	X	X	6.25	X	100
269	X	50	X	X	X	X	X	X	X	X	X	X	X	X	X	50	X	X	X	50	X	X
285	X	3.85	X	X	X	X	X	X	X	X	X	X	X	X	3.85	X	X	X	3.85	7.69	7.69	X
295	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	100	X	X	X
319	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
326	X	25.71	X	X	X	X	X	X	X	X	X	X	X	X	X	25.71	X	X	31.43	25.71	X	X
338	X	0.86	0.43	X	X	X	X	0.86	1.29	X	0.43	X	X	X	X	0.86	X	X	0.86	6.9	0.43	1.29
377	X	3	X	X	X	1	X	X	X	1	X	X	X	9	2	X	X	X	10	5	1	X
477	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
489	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
601	X	14.29	X	X	X	X	X	X	X	X	1.59	3.17	X	17.46	3.17	15.87	X	X	X	17.46	X	X
676	X	24.44	2.99	X	X	1	X	0.75	0.5	0.25	0.5	X	X	2.24	3.99	1.25	X	X	2.74	X	5.24	0.5
755	X	10.53	5.26	X	X	X	X	1.32	X	X	2.63	X	X	X	1.32	1.32	X	X	X	27.63	X	X
798	X	6.25	X	X	X	X	X	X	100	X	X	X	X	X	9.38	X	X	X	X	6.25	X	X

security weakness y. 'Co-located Prop (x, y)' and 'Co-located Prop (y, x)' might have different values, as the denominator in Equation 3 would be different.

$$\text{Co-located Prop } (x,\, y) = \frac{\text{\# of occurrence with co-located (CWE-}x\text{, CWE-}y\text{)}}{\text{\# of occurrence of CWE-}x} * 100\% \tag{3}$$

B. *Answer to RQ$_2$*

We present the co-located security weaknesses along with their 'Co-located prop.' values for GitHub, GitLab, and ModelZoo respectively, in Tables V, VI and VII. For example, from Table VII we observe one co-location category (CWE-269, CWE-676), with a 'Co-located Prop.' value of 33.33%. The 'Co-located Prop.' value indicates that the presence of CWE-269 implies the presence of CWE-676 33.33% of the time in the ModelZoo dataset.

In Tables V, VI and VII, we highlight the cells that has 'Co-located Prop.' value of greater than 50%. For all three datasets, we observe CWE-77 to be co-located with CWE-676 with a high 'Co-located Prop.' value. We can conclude that, if there is a CWE-77 in a Python script, it is likely that CWE-676 will also be in the script with a 50% chance. Co-location proportion can be as high as 100%, as it occurred for CWE-61 in the GitHub dataset, where CWE-61 is co-located with CWE-601, CWE-676, and CWE-755. We also notice co-location proportion to be 100% between CWE-259 and CWE-798, which occurs due to the mapping of hard-coded password to two CWEs, namely CWE-259 and CWE-798.

V. RQ$_3$: Association of Identified Security Weaknesses with Artifacts Used for Training and Testing ML Models

In this section, we answer ***RQ$_3$****: What artifacts are associated with identified security weaknesses in supervised*

TABLE VII: Answer to RQ2: Proportion of Co-located Security Weaknesses for the ModelZoo Dataset.

CWE-ID	61	77	78	79	89	91	220	242	259	269	285	295	319	326	338	377	477	489	601	676	755	798
61	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
77	X	X	X	1.92	X	X	X	X	X	1.92	X	X	X	X	5.77	X	X	X	5.77	51.92	7.69	X
78	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
79	X	100	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
89	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
91	X	X	X	X	X	X	X	X	X	X	X	X	X	X	12.5	X	X	X	X	12.5	X	X
220	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
242	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	100	X	X	X	X	X	X
259	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	14.29	X	100
269	X	33.33	X	X	X	X	X	X	X	X	X	X	X	X	33.33	X	X	X	X	33.33	X	X
285	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
295	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
319	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
326	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	50	25	X	X
338	X	3.06	X	X	X	1.02	X	X	X	1.02	X	X	X	X	X	2.04	X	X	X	12.24	3.06	X
377	X	X	X	X	X	X	X	1.96	X	X	X	X	X	X	3.92	X	X	X	9.8	3.92	X	X
477	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
489	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
601	X	12.5	X	X	X	X	X	X	X	X	X	X	X	8.33	X	20.83	X	X	X	37.5	X	X
676	X	19.57	X	X	X	0.72	X	X	1.45	0.72	X	X	X	0.72	8.7	1.45	X	X	6.52	X	0.72	1.45
755	X	11.76	X	X	X	X	X	X	X	X	X	X	X	X	8.82	X	X	X	X	2.94	X	X
798	X	X	X	X	X	X	X	X	100	X	X	X	X	X	X	X	X	X	X	14.29	X	X

learning-based projects? Our hypothesis is that security weaknesses that appear in artifacts for training and testing can help malicious users the opportunity to conduct attacks against production-level ML models. We first provide the methodology to answer RQ$_3$ in Section V-A. Then, we provide our findings in Section V-B.

A. Methodology to Answer RQ$_3$

To answer RQ$_3$, we associate (i) scripts that are related to model training process (train_script) and (ii) scripts that are related to model testing process (test_script) with our identified 22 security weakness categories. We collect the scripts associated with the 3,964 instances of security weaknesses collected in Section III. To find out whether a script is related to training or testing, we inspect the script name. Our assumption is that a script that has the word 'train' or 'test' in it's name is respectively related to training and testing. For example, a script named 'train.py' is related to a model training process. For the other scripts that are associated with the identified security weaknesses but do not have the the word 'train' or 'test' in it's name, we use the script content to find the words 'train' and 'test'. Our assumption is that a script that has the word 'train' and 'test' in it's content is respectively related to training and testing. Because this assumption may yield some false positives, we manually check the contents of all the scripts identified to rule out the false positives."

B. Answer to RQ$_3$

In all, we identify 3569 train_scripts and 12,188 test_scripts from our set of 36,022 Python scripts. For the ModelZoo dataset, we observe 210 train_script and 169 test_script to have at least one security weakness. We report our findings in Table VIII. In the 'TRAIN' and 'TEST' columns, we respectively report the number of train_script and test_script associated with each security weakness category. We observe 23.79% and 40.49% of security weaknesses are respectively associated with train_script and test_script. Among these, 15.04% of total security weaknesses are associated with both train_script and test_script. The rest of the security weaknesses are not associated with train_script or test_script. We observe CWE-676: Use of Potentially Dangerous Function to be the most frequent security weakness for both train_script and test_script.

TABLE VIII: Answer to RQ3: Association of Security Weaknesses

	GITHUB		**GITLAB**		**MODELZOO**	
CWEs	**TRAIN**	**TEST**	**TRAIN**	**TEST**	**TRAIN**	**TEST**
CWE-61	0	1	0	0	0	0
CWE-77	27	203	36	63	34	27
CWE-78	3	54	50	53	5	0
CWE-79	0	0	0	0	1	1
CWE-89	1	5	0	0	0	0
CWE-91	36	61	8	12	6	4
CWE-220	0	2	0	0	0	0
CWE-242	0	1	0	5	0	1
CWE-259	8	46	12	17	9	8
CWE-269	0	0	2	0	3	1
CWE-285	3	8	8	12	2	3
CWE-295	0	0	1	1	0	0
CWE-319	0	2	0	0	0	0
CWE-326	10	26	10	7	2	2
CWE-338	22	123	134	136	45	46
CWE-377	28	56	56	50	37	23
CWE-477	0	4	0	0	0	0
CWE-489	0	0	0	0	0	0
CWE-601	20	50	38	29	19	14
CWE-676	109	437	187	208	88	55
CWE-755	41	137	21	33	13	22
CWE-798	8	46	12	17	9	8
Total	250	914	483	522	210	169

VI. Discussion

In this section, we discuss the implications of our findings: **Implications for practitioners**: Our findings have implications for practitioners.

Incorporating security in early stages of development for secure ML: Similar to secure software development, ML-based projects also need to integrate security early in the development process [39]. We have identified a catalog of

22 security weakness categories, as reported in Table II, that needs to be detected and mitigated before the deployment of SLPs. There are existing techniques that can be leveraged to automatically repair security weaknesses [15] [33] [23]. The fact that these identified weaknesses are not fixed is evidence that security is not addressed from the beginning of the Ml-based software development life cycle. Practitioners can use our findings reported in Sections III-B and V-B to incorporate security in the early stages of development for secure SLPs.

Weakness Mitigation: We recommend the following strategies to mitigate security weaknesses in SLPs:

Code Review: Practitioners need to follow rigorous code review practices to avoid weaknesses, such as, CWE-755: Improper Handling of Exceptional Conditions and CWE-259: Use of Hard-coded Password. To better handle security weaknesses such as, CWE-259: Use of Hard-coded Password, practitioner can use secrets and encryption management tools such as, *HashiCorp Vault* [14].

Static Analysis: Practitioners can use static analysis techniques to detect the security weaknesses identified in Section III-B. Therefore, we recommend to integrate static analysis into the development process of SLPs.

Vulnerability Repair Tools: Practitioners can use automatic vulnerability repair tools, such as *Pasan* [41], *BovInspector* [21], and *VeraCode* [15], that will identify security weaknesses as well as suggest appropriate fixes. *Pasan* [41] works by detecting the inputs used in adversarial attacks and then using these inputs to generate fixes that remove the vulnerabilities exploited in the attacks, whereas *BovInspector* [21] uses static analysis and symbolic execution to analyze buffer overflow threats and suggests fixes.

Prioritizing Efforts for Security-focused Code Reviews: As conducting code reviews is a resource-intensive activity [42], we advocate practitioners to prioritize code review efforts using our co-location analysis presented in Tables V, VI and VII. For example, as a rule of thumb, if a Python script used in a SLP contains an instance of CWE-77, i.e., command injection, they can also inspect the script for an instance of CWE-676, i.e., use of a potentially dangerous function as there is a $\geq$ 50% chance of a CWE instance co-occurring with an instance of CWE-77. In Table VIII, we observe 210 train_script and 169 test_script to have at least one security weakness and advocate practitioners to prioritize code review efforts for training and testing scripts to secure the production-level ML models.

Implications for researchers: Our findings also have implications for future research: *Survival Analysis:* A security weakness that persists for a long time can facilitate attackers. Researcher can use survival analysis techniques [29] to analyze the amount of time our identified security weaknesses persist in the same script for SLPs. *New Attack Identification:* Our identified security weaknesses can be leveraged to identify new attacks. Researchers can build upon our findings to explore which characteristics correlate with Python scripts with security weaknesses. If certain characteristics correlate with scripts that have weaknesses, then practitioners can prioritize their inspection efforts for scripts that exhibit those characteristics. *Vulnerability Research:* Our empirical study provides the groundwork to conduct further research in the domain of vulnerability research for ML. Our identified security weaknesses focus on supervised learning, which could be applicable for reinforcement learning and unsupervised learning.

VII. Threats to Validity

We present the limitations of our paper in this section.

Conclusion Validity: Our identified security alerts and the corresponding CWEs are limited to the scripts we used in Section III-A. We mitigate these limitations by inspecting 36,022 scripts. Like any other static analysis tool, the Bandit tool is susceptible to generate false positives [12]. We mitigate this limitation by mapping each security issue identified by the Bandit tool to corresponding CWEs in Section III-B1. Also, the mapping between the security weaknesses and potential CWE indexes is based upon the authors' judgment and subjective view. We mitigate the limitation by using a rater who is not an author of the paper. For the security weakness CWE-676: Use of Potentially Dangerous Function, we consider all instances of dangerous functions without checking whether a weakness was induced or not. Therefore there may be instances where a dangerous function was not used insecurely. We consider each instance of CWE-259: Use of Hard-coded Password an instance of CWE-798: Use of Hard-coded Credentials, and vice versa.

External Validity: Our empirical study is limited to the datasets that we analyzed. Our datasets are constructed by mining OSS repositories. Investigating projects from other proprietary domains might reveal types not reported in our paper. Also, our findings are limited to Python-based projects.

VIII. Conclusion

SLPs may have security weaknesses, that make SLPs susceptible to security attacks. Practitioners need to be aware of the security weaknesses to secure SLPs. We conduct an empirical study to identify the security weaknesses that occur in SLPs. We identify 22 types of security weaknesses in SLPs. We observe CWE-676: Use of Potentially Dangerous Function to be the most frequent CWE. We observe security weaknesses to co-locate in Python scripts. Furthermore, of the identified 3,964 security weaknesses, 23.79% and 40.49% respectively, appear for Python scripts that are used to train and test models. Our research study provides empirical evidence on how frequently security weaknesses appear in SLPs, which necessitates pro-active detection and mitigation. We have provided guidelines on how our analysis can inform practitioners to take actions for mitigating security weaknesses in SLPs.

Acknowledgments

We thank the PASER group at Tennessee Tech University for their valuable feedback. The research was partially funded by the U.S. National Science Foundation (NSF) award #

2026869. This manuscript has been partially authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

References

[1] "Ai security white paper - huawei," https://www-file.huawei.com/-/media/corporate/pdf/trust-center/ai-security-whitepaper.pdf, [Online; accessed 02-May-2021].

[2] "Bandit," https://github.com/PyCQA/bandit, [Online; accessed 14-Apr-2021].

[3] "Becoming an ai-fueled organization," https://www2.deloitte.com/us/en/insights/focus/cognitive-technologies/state-of-ai-and-intelligent-automation-in-business-survey.html, [Online; accessed 12-Jan-2022].

[4] "Cwe - common weakness enumeration," https://cwe.mitre.org, [Online; accessed 14-Apr-2021].

[5] "defusedxml · pypi," https://pypi.org/project/defusedxml/#defusedxml-sax, [Online; accessed 02-May-2021].

[6] "Flask documentation," https://flask.palletsprojects.com/en/1.1.x/quickstart/#debug-mode, [Online; accessed 19-Apr-2021].

[7] "How secure are your ai and machine learning projects?" https://www.csoonline.com/article/3434610/how-secure-are-your-ai-and-machine-learning-projects.html, [Online; accessed 12-Jan-2022].

[8] "Model zoo: Discover open source deep learning code and pretrained models." https://modelzoo.co, [Online; accessed 02-May-2021].

[9] "Openstack docs: Create, use, and remove temporary files securely," https://security.openstack.org/guidelines/dg_using-temporary-files-securely.html, [Online; accessed 02-May-2021].

[10] "os — miscellaneous operating system interfaces," https://docs.python.org/2.7/library/os.html#os.tempnam, [Online; accessed 02-May-2021].

[11] "Ossn/ossn-0033 - openstack," https://wiki.openstack.org/wiki/OSSN/OSSN-0033, [Online; accessed 02-May-2021].

[12] "Rice's theorem," https://en.wikipedia.org/wiki/Rice\%27s_theorem, [Online; accessed 02-May-2021].

[13] "Security in django — django documentation — django," https://docs.djangoproject.com/en/dev/topics/security/\#sql-injection-protection, [Online; accessed 02-May-2021].

[14] "Vault," https://github.com/hashicorp/vault, [Online; accessed 06-Feb-2022].

[15] "Veracode: Confidently secure your applications with veracode," https://www.veracode.com, [Online; accessed 29-Apr-2021].

[16] "Why security is important in ml and how to secure your ml-based solutions," https://mlconference.ai/machine-learning-advanced-development/why-security-is-important-in-ml-and-how-to-secure-your-ml-based-solutions/, [Online; accessed 02-Jan-2022].

[17] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages."

[18] F. A. Bhuiyan, "Verifiability package for paper," https://figshare.com/s/86bea428698b103afb32, 2021, [Online; accessed 02-May-2021].

[19] F. A. Bhuiyan and A. Rahman, "Characterizing co-located insecure coding patterns in infrastructure as code scripts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2020, pp. 27–32.

[20] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "Understanding challenges in deploying deep learning based software: An empirical study," *arXiv preprint arXiv:2005.00760*, 2020.

[21] F. Gao, L. Wang, and X. Li, "Bovinspector: automatic inspection and repair of buffer overflow vulnerabilities," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 786–791.

[22] Z. Ghahramani, "Unsupervised learning," in *Summer School on Machine Learning*. Springer, 2003, pp. 72–112.

[23] J. Harer, O. Ozdemir, T. Lazovich, C. P. Reale, R. L. Russell, L. Y. Kim, and P. Chin, "Learning to repair software vulnerabilities with generative adversarial networks," *arXiv preprint arXiv:1805.07475*, 2018.

[24] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1135–1146.

[25] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[26] W. Lin, Y. Hu, and C. Tsai, "Machine learning in financial crisis prediction: A survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 4, pp. 421–436, 2012.

[27] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1547–1559.

[28] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis." in *USENIX Security Symposium*, vol. 14, 2005, pp. 18–18.

[29] D. Machin, Y. B. Cheung, and M. Parmar, *Survival analysis: a practical approach*. John Wiley & Sons, 2006.

[30] P. McDaniel, N. Papernot, and Z. B. Celik, "Machine learning in adversarial settings," *IEEE Security & Privacy*, vol. 14, no. 3, pp. 68–72, 2016.

[31] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. IEEE, 2005, pp. 580–586.

[32] S. Peng, P. Liu, and J. Han, "A python security analysis framework in integrity verification and vulnerability detection," *Wuhan University Journal of Natural Sciences*, vol. 24, no. 2, pp. 141–148, 2019.

[33] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 87–102.

[34] G. A. A. Prana, C. Treude, F. Thung, T. Atapattu, and D. Lo, "Categorizing the content of github readme files," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1296–1327, 2019.

[35] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 164–175.

[36] M. R. Rahman, A. Rahman, and L. Williams, "Share, but be aware: Security smells in python gists," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 536–540.

[37] S. Russell and P. Norvig, "Ai a modern approach," *Learning*, vol. 2, no. 3, p. 4, 2005.

[38] J. Saldana, *The coding manual for qualitative researchers*. Sage, 2015.

[39] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, pp. 2503–2511, 2015.

[40] R. S. Siva Kumar, M. Nyström, J. Lambert, A. Marshall, M. Goertzel, A. Comissoneru, M. Swann, and S. Xia, "Adversarial machine learning-industry perspectives," in *2020 IEEE Security and Privacy Workshops (SPW)*, 2020, pp. 69–75.

[41] A. Smirnov and T.-c. Chiueh, "Automatic patch generation for buffer overflow attacks," in *Third International Symposium on Information Assurance and Security*. IEEE, 2007, pp. 165–170.

[42] M. Song and Y.-W. Kwon, "Which code changes should you review first?: A code review tool to summarize and prioritize important software changes," *Journal of Multimedia Information System*, vol. 4, no. 4, pp. 255–262, 2017. [Online]. Available: https://doi.org/10.9717/JMIS.2017.4.4.255

[43] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[44] TechCrunch, "Why firms are welcoming MLOps into the fold of software development," 2022, [Online; accessed 12-Jan-2022].

[45] H. Zhang, S. Wang, H. Li, T.-H. P. Chen, and A. E. Hassan, "A study of c/c++ code weaknesses on stack overflow," *IEEE Transactions on Software Engineering*, 2021.

[46] M. Zhao, J. Grossklags, and P. Liu, "An empirical study of web vulnerability discovery ecosystems," in *Proceedings of the 22nd ACM*

SIGSAC Conference on Computer and Communications Security, 2015, pp. 1105–1117.