An Empirical Study of Vulnerabilities in Python Packages and Their Detection

Anonymous Author(s)

ACM Reference Format:
Anonymous Author(s). 2025. An Empirical Study of Vulnerabilities in Python Packages and Their Detection. In . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

A Discussion & Threats to Validity

Discussion. Our LLM-assisted cleansing method labeled 72 out of 8,374 vulnerable functions as "4) no decision can be made". To ensure the integrity and validity of the evaluation on this automated cleansing method, especially when measuring the precision and recall, we exclude the commits associated with these 72 functions from our dataset. Future work could explore altering the composition of contextual information provided to LLMs or incorporating additional context to help LLMs resolve such cases.

In our empirical evaluation of vulnerability detectors, we evaluated current rule-based and ML-based detectors and investigated their limitations independently. A direct comparison between these two methodologies was not conducted due to inherent differences in their operational granularities. Rule-based detectors scan whole projects and locate vulnerabilities precisely with detailed information such as the causes of the vulnerabilities and taint flow paths, while current ML-based detectors typically analyze individual functions and solely classify them as vulnerable or not.

Threats to validity. For RQ1 and RQ3, the dataset labels and the rule-based detectors' results are validated manually. The reliability of these decisions can be influenced by factors such as the evaluators' expertise in relevant areas and their personal interpretations of the vulnerabilities. To mitigate potential biases, we involve two authors, both with solid backgrounds in Python programming and software security, to independently assess the correctness of the labels and then resolve disputes. We additionally measure their agreement level with Cohan's Kappa before any consensus has been reached, which is 0.718 for RQ1 and 0.601 for RQ3 and within the range of fair to good.

In all experiments in this study where humans are involved, there exist cases where the participant cannot make the decisions either because of limited descriptions or erroneously attached commits in the vulnerability reports. As the number of such cases is relatively small, we expect they do not affect the overall conclusions. For example, in RQ3, there is one reviewed path traversal vulnerability report that we cannot decide the type and the cause of it, which we expect does not affect our overall observations regarding this category of vulnerabilities.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: The language composition of *PyVul*.

Language Composition	#Commits	#Functions
Python	775 (67.0%)	1,480 (71.1%)
C/C++	335 (29.0%)	463 (22.2%)
JavaScript/TypeScript	23 (2.0%)	115 (5.5%)
Java	4 (0.3%)	12 (0.6%)
Other Language	2 (0.3%)	12 (0.6%)
Multiple Languages	18 (1.6%)	-
Total	1,157	2,082

B Characteristic Analysis of Python Package Vulnerabilities

Currently, there is no systematic analysis of the characteristics of Python package vulnerabilities due to the absence of a vulnerability benchmark for Python packages. The curated *PyVul* benchmark allows for a comprehensive analysis of various aspects of Python package vulnerabilities. This includes qualitative and quantitative analysis of their language composition, the number of functions involved, and the types of vulnerabilities present, providing insights necessary for understanding Python package vulnerabilities and guiding the development of corresponding detection tools.

B.1 Language Composition Analysis

The analysis of language composition in the benchmark offers valuable insights for developing effective detection tools. Since packages from which the vulnerabilities originate provide a more comprehensive context for understanding them, while the fixing commits directly indicate their causes and fixes, our analysis is conducted against both.

We conducted an analysis of programming languages used in all 349 Python packages associated with *PyVul* by querying the language statistics of their repositories via the GitHub API. The results are presented in Figure 1. As shown, these Python packages predominantly involve multiple programming languages. Approximately 75% (262/349) of the packages used at least two programming languages, while around 36% (127/349) utilized at least five different languages. We additionally counted the total number of vulnerabilities at the commit level encompassed by these packages as light blue bars in Figure 1. The data reveals that over 90% of the vulnerabilities are found in packages that use multiple languages. On average, a Python-only package is associated with 1.18 vulnerabilities, while a multi-lingual Python package is linked to 4.02 vulnerabilities. Notably, 14 packages with more than 12 languages contribute 342 vulnerabilities. The main reason is that two of these packages, TensorFlow and PyTorch, account for 311 vulnerabilities. We further employ the interquartile range method [?] to remove the impact of outliers. After adjustment, a Python-only package is

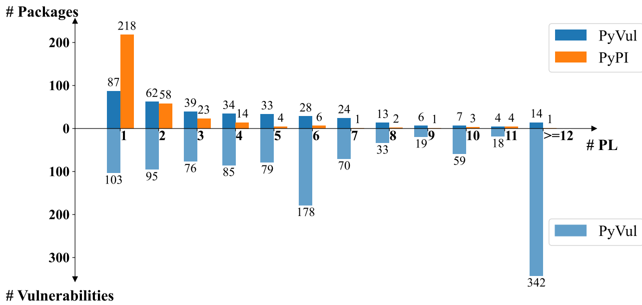

Figure 1: Programming language (PL) distribution in Python packages.

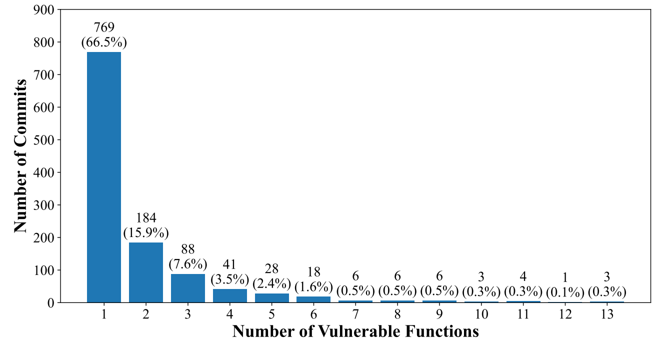

Figure 2: Vulnerable functions count distribution of *PyVul*.

associated with an average of 1.18 vulnerabilities, whereas a multi-lingual Python package averages 1.29 vulnerabilities. This suggests an increased risk of vulnerabilities in multilingual packages.

To better understand the relationship between vulnerabilities and the multi-language characteristics of Python packages, we compare the language composition distribution of the packages in *PyVul* with that of general PyPI packages. The packages in *PyVul* are quite popular, averaging 13,358.7 stars on GitHub. To effectively control the effect of popularity, we randomly select the same number of packages from the top 8,000 most popular PyPI packages [?] for comparison. As illustrated in Figure 1, the packages in *PyVul* show a clear tendency towards the usage of multiple programming languages. This echoes the observation that multi-lingual Python packages can be more susceptible to vulnerabilities, which is also consistent with the observation in previous work [?].

We further analyze the language composition of vulnerabilities at the commit level. We use Guesslang [?] to identify each vulnerable function's programming language, and aggregate them to derive the language composition at the commit level. We group the commits and vulnerable functions, respectively, according to their programming languages and present the statistics in Table 1. To our surprise, only 1.6% of the vulnerabilities involve more than one programming language. Among the vulnerabilities, 67.0% are exclusively related to Python, while 31.4% are associated with other programming languages, with C/C++ being the most prevalent non-Python language. Two important observations can be drawn: 1) non-Python vulnerabilities are common in Python packages, and 2) most vulnerabilities and their fixes are associated with a single programming language. It is essential to note that this does not imply that they can be effectively detected by tools designed for that specific language. The broader context of these vulnerabilities often involves multiple programming languages. Therefore, effective detection tools must be capable of handling cross-language code contexts, a point which is also supported by our findings in RQ3.

B.2 Vulnerability Span Analysis

Span analysis aims to examine the number of functions related to a vulnerability. It provides crucial insights for detecting and addressing vulnerabilities, as the span reveals the minimum context required for effective analysis. However, a precise measurement of the span has not yet been obtained due to the inaccurate identification of code changes relevant to vulnerabilities in prior benchmarks. Given the high quality of *PyVul*, we evaluate how many functions are involved in the vulnerabilities of Python packages and present the statistics in Figure 2. The number of functions involved in the vulnerabilities of *PyVul* ranges from 1 to 13, with fewer vulnerabilities observed in more expansive categories. On average, each vulnerability in *PyVul* is linked to 1.8 function. In particular, 503 (43.5%) vulnerabilities involve more than one vulnerable function. These cross-function vulnerabilities are associated with an average of 2.6 vulnerable functions. The prevalence of cross-function vulnerabilities emphasizes the importance of fully considering the cross-function characteristics when detecting or addressing vulnerabilities.

B.3 Vulnerability Type Distribution

Vulnerabilities come in many different types, each varying in detection difficulty. Beyond simply assessing whether a vulnerability detection method can find vulnerabilities, we are also interested in its performance when detecting different types of vulnerability. Therefore, we additionally annotate the *PyVul* dataset with CWEs from original vulnerability reports. The 1,157 commit-level vulnerabilities in the *PyVul* dataset belong to 151 different CWE vulnerability types. We performed a simple clustering based on the mechanisms, the causes and the consequences of these CWE vulnerability types. For example, CWE-125 (Out-of-bounds Read), CWE-787 (Out-of-bounds Write), CWE-120 (Buffer Copy without Checking Size of Input, 'Classic Buffer Overflow'), and CWE-122 (Heap-based Buffer Overflow) were merged into one category. The details of the clustering are provided in the supplementary material.

In Table 2, we list the distribution of vulnerability types in Python packages. Injection vulnerabilities are the most common type, with 195 commits (394 functions), accounting for 17.5% (19.7%) of the total. Injection vulnerabilities consist of 16 CWE vulnerability types, including SQL Injection, Command Injection, Parameter Injection, Cross-site Scripting (XSS) Injection, Static Code Injection, XML External Entity (XXE) Injection, CSV Formula Injection, and others. Access control vulnerabilities are the second most common type, accounting for 11.5% of the total commits (133 commits) and 14.6% of the total functions (305 functions). Access control vulnerabilities

Table 2: Vulnerability types distribution of *PyVul*.

Type	#Commits	#Functions	Avg. CVSS
Injection	202 (17.5%)	411 (19.7%)	7.4
Improper Access Control	133 (11.5%)	305 (14.6%)	7.3
Out-of-Bound Read/Write	114 (9.9%)	174 (8.4%)	6.0
File Operation Error	80 (6.9%)	165 (7.9%)	6.9
Improper Input Validation	75 (6.5%)	109 (5.2%)	6.7
Calculation Error	66 (5.7%)	79 (3.8%)	4.6
Sensitive Information Exposure	60 (5.1%)	103 (4.9%)	5.8
Request Forgery	53 (4.6%)	109 (5.2%)	7.5
Improper Resource Management	54 (4.7%)	112 (5.4%)	6.5
NULL Pointer Dereference	43 (3.7%)	53 (2.5%)	5.4
Assertion Failures	39 (3.4%)	46 (2.2%)	5.4
Incorrect Synchronization	38 (3.3%)	66 (3.2%)	5.6
Redirect Error	23 (2.0%)	47 (2.3%)	6.1
Use of Uninitialized Resource	24 (2.1%)	29 (1.4%)	5.9
Improper Deserialization	23 (2.0%)	39 (1.9%)	8.9
Incorrect Regular Expression	22 (1.9%)	31 (1.5%)	6.1
Uncontrolled Recursion	16 (1.4%)	24 (1.2%)	5.9
Improper Exception Handling	16 (1.4%)	20 (1.0%)	5.3
Inefficient Algorithmic Complexity	12 (1.0%)	39 (1.9%)	7.0
Incorrect Provision of Specified Functionality	9 (0.8%)	27 (1.3%)	3.5
Incomplete Cleanup	6 (0.5%)	11 (0.5%)	7.2
Side Channel	5 (0.4%)	8 (0.4%)	5.2
Others	43 (3.7%)	70 (3.4%)	6.7
Total	1,157	2,082	6.5

consist of 33 CWE vulnerability types, primarily including CWE-284 (Improper Access Control), CWE-287 (Improper Authentication), CWE-305/289/288/290/294 (Authentication Bypass by Primary Weakness/Alternate Name/Using an Alternate Path or Channel/Spoofing/Capture Replay), and CWE-304 (Missing Critical Step in Authentication), among others. Following closely are vulnerability types such as Out-of-Bound Read/Write, File Operation Error, Improper Input Validation, and Calculation Error, which also occur relatively frequently.

From the vulnerability types we can spot a great diversity regarding their origins and attack scenarios. Vulnerabilities such as XSS Injection, Improper Access Control and Request Forgery are predominantly associated with web applications. On the other hand, vulnerabilities such as Out-of-Bound Read/Write, NULL Pointer Dereference and Use of Uninitialized Resource are typically linked to low-level C/C++ code. Additionally, Incorrect Synchronization relates to parallel execution. This diversity in vulnerability type echoes Python's usage in different fields and may pose extra difficulty to automated static detectors, including both rule-based ones and ML-based ones.

C Limitations of Rule-based Detectors.

The following presents more analysis of our empirical review of the top CWEs.

CWE-22: Path Traversal. Path traversal refers to a situation where an application receives unvalidated user input as parameters for file-related operations, such as reading or viewing files. These parameters contain special characters (e.g., '..' and '/') that can be used to bypass protection mechanisms, gain unauthorized access to protected files or directories, or overwrite sensitive data. Several types of path traversal vulnerability are spotted: 1) Improper use of other packages **(4/30)**; 2) Unawareness of behavioral differences of used APIs from other packages when executed on different operating systems **(3/30)**; 3) Missing validation in certain taint paths **(22/30)**. The first two causes are not considered by any of these evaluated detectors. For the static detection of the third cause, taint analysis is the typical approach. CodeQL and PySA support static taint analysis. However, four factors hinder their performance in the detection of CWE-22:

- Lack of package-specific taint specifications. Non-standalone packages require package-specific taint specifications.
- Lack of accurate type information. This is an inherited challenge for Python static analyzers. As a dynamic language, variable types in Python are determined at run time. Without type information, static modeling of data flows can be largely incomplete, substantially limiting the effectiveness of taint analysis built upon it. Implementing type inference can mitigate this challenge. However, neither of the subject detectors incorporate any form of type inference.
- Limited handling of Python's complex language features. Python's advanced language features, such as higher-order functions and dynamic features [?], frequently present in the examined packages. Incomplete addressing of these features further contributes to incomplete data flow modeling.
- Complex data flows in web applications. Web applications are frequently spotted in CWE-22 reports. The inherent complexity of web applications arises from their interaction with client-side components and their capability to execute multiple routes concurrently, often resulting in intricate data flows. Neither of the detectors effectively models these intricate data flows.

CWE-400: Uncontrolled Resource Consumption. Uncontrolled Resource Consumption refers to a type of vulnerability where a system fails to properly limit resource usage, leading to exhaustion of system resources such as CPU, memory, disk space, network bandwidth, or file descriptors. This can result in performance degradation, denial of service (DoS), or even system crashes. The examined Uncontrolled Resource Consumption vulnerabilities can be attributed to four causes: 1. Improper limitations on resource consumption **(23/30)**. A typical example of this includes parsing a user-supplied YAML file without setting the maximum number of nodes, which can lead to excessive consumption of space or time. 2. Regular expressions with an inefficient worst-case computational complexity **(4/30)**. 3. Algorithm defects **(2/30)**. For instance, certain user input can trigger infinite loops in a program. 4. Unclosed resources **(1/30)**.

Bandit targets only one specific case of improper limitations on resource consumption, which checks whether the `timeout` parameter has been set in the `request` library's API calls, failing to address this most prevalent type of Uncontrolled Resource Consumption systematically. Improper limitations on resource consumption attribute to the co-existence of two factors: 1) User-consumed resources. There exists data flows from user inputs to resource consumption APIs, such as file storage APIs and XML parsing APIs; 2) Absence of limitations or user supplying limitations, such as the size of user-uploaded data for file storage APIs, or the maximum number of nodes in XML parsing APIs. These limitations are typically implemented either as parameters of the resource consumption APIs or as independent checks before user inputs reach these APIs. As such, effective detection of improper limitations on resource consumption requires an extended taint analysis that not only identifies the taint flows from user inputs to resource consumption APIs, but also backtraces the limitations from these APIs.

On the other hand, CodeQL includes a rule targeting inefficient regular expressions, failing to address most Uncontrolled Resource Consumption vulnerabilities, and PySA does not have any rule targeting Uncontrolled Resource Consumption vulnerabilities.

CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). A race condition can arise when the necessary atomicity of operations is violated in concurrent execution, resulting in unexpected program behavior. Traditional atomicity violations typically involve synchronous operations, such as threads, accessing shared memory without adequate safeguards. In web applications, atomicity violations can also occur when synchronous operations access external resources such as file systems. In the PyPI ecosystem, both traditional **(12/30)** and web application-related **(18/30)** atomicity violations are commonly observed.

None of the three tools supports detection of race conditions in Python. Detection of traditional atomicity violations involving access to shared memory requires definitions of atomic regions [**?**]. This detection can potentially be implemented using CodeQL, which provides API modeling based on functionality and a sound data flow analysis engine. Web application-related atomicity violations extend further, requiring an assessment of whether multiple operations access the same external resource, such as a specific data record in a database. Furthermore, as discussed, data flows in web applications are complex to model. As such, detecting atomicity violations in web applications requires sophisticated methods to be developed.

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). SQL vulnerabilities occur when developers fail to filter, escape, restrict, or properly handle user input strings in systems that interact with databases. This allows attackers to input carefully crafted strings to illegally access data from the database. The majority **(28/29)** of SQL injection vulnerabilities are caused by improper input validation, except for CVE-2014-0474 [**?**], which mainly relates to developers' unawareness of MySQL's typecasting behavior.

All three detectors target SQL injection caused by improper input validation. Bandit's rules checks for hard-coded SQL queries and use of potentially dangerous APIs such as Django's RawSQL. However, as Bandit does not exhibits any data flow analysis, these rules exhibits a high false positive rate. CodeQL and PySA adopted taint analysis and are able to more accurately identify SQL injection. However, in the vulnerability reports examined, most of these SQL injections locate in non-standalone packages, and taint analysis in these packages are largely ineffective without package-specific taint specifications.

D Detailed Clustering

Table 3: Clustering of CWE vulnerability types

Cluster Name	CWE Name	Commits	Functions
Injection	CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	89	185
	CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	29	50
	CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	20	29
	CWE-74 Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	12	29
	CWE-94 Improper Control of Generation of Code ('Code Injection')	12	36
	CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection')	10	11
	CWE-611 Improper Restriction of XML External Entity Reference	7	17
	CWE-88 Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')	6	11
	CWE-1336 Improper Neutralization of Special Elements Used in a Template Engine	4	11
	CWE-93 Improper Neutralization of CRLF Sequences ('CRLF Injection')	3	11
	CWE-80 Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	3	5
	CWE-116 Improper Encoding or Escaping of Output	2	4
	CWE-75 Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)	1	3
	CWE-707 Improper Neutralization	1	1
	CWE-1236 Improper Neutralization of Formula Elements in a CSV File	1	2
	CWE-96 Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')	1	4
	CWE-91 XML Injection (aka Blind XPath Injection)	1	2
Improper Input Validation	CWE-20 Improper Input Validation	69	99
	CWE-1284 Improper Validation of Specified Quantity in Input	6	10
File Operation Error	CWE-22 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	51	101
	CWE-59 Improper Link Resolution Before File Access ('Link Following')	10	20
	CWE-377 Insecure Temporary File	5	6
	CWE-434 Unrestricted Upload of File with Dangerous Type	4	19
	CWE-29 Path Traversal: '\..\filename'	4	10
	CWE-23 Relative Path Traversal	4	6
	CWE-36 Absolute Path Traversal	1	1
	CWE-641 Improper Restriction of Names for Files and Other Resources	1	2
NULL Pointer Dereference	CWE-476 NULL Pointer Dereference	43	53
Out-of-Bound Read/Write	CWE-125 Out-of-bounds Read	43	53
	CWE-787 Out-of-bounds Write	21	48
	CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer	17	19
	CWE-120 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	16	28
	CWE-131 Incorrect Calculation of Buffer Size	11	17
	CWE-122 Heap-based Buffer Overflow	6	9
Resource Management	CWE-400 Uncontrolled Resource Consumption	41	78
Error			

Table 3 continued from previous page

Cluster Name	CWE Name	Commits	Functions
	CWE-770 Allocation of Resources Without Limits or Throttling	12	31
	CWE-404 Improper Resource Shutdown or Release	1	3
Assertion Failures	CWE-617 Reachable Assertion	39	46
Information Exposure	CWE-200 Exposure of Sensitive Information to an Unauthorized Actor	38	64
	CWE-209 Generation of Error Message Containing Sensitive Information	4	7
	CWE-532 Insertion of Sensitive Information into Log File	4	5
	CWE-212 Improper Removal of Sensitive Information Before Storage or Transfer	4	11
	CWE-312 Cleartext Storage of Sensitive Information	2	6
	CWE-668 Exposure of Resource to Wrong Sphere	2	3
	CWE-614 Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	2	2
	CWE-598 Use of GET Request Method With Sensitive Query Strings	1	1
	CWE-524 Use of Cache Containing Sensitive Information	1	1
	CWE-213 Exposure of Sensitive Information Due to Incompatible Policies	1	2
	CWE-311 Missing Encryption of Sensitive Data	1	1
Incorrect Synchronization	CWE-362 Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	34	62
	CWE-367 Time-of-check Time-of-use (TOCTOU) Race Condition	1	1
	CWE-821 Incorrect Synchronization	1	1
	CWE-662 Improper Synchronization	1	1
	CWE-833 Deadlock	1	1
Open Redirect	CWE-601 URL Redirection to Untrusted Site ('Open Redirect')	23	47
Improper Deserialization	CWE-502 Deserialization of Untrusted Data	23	39
Origin Validation Error	CWE-352 Cross-Site Request Forgery (CSRF)	23	54
	CWE-918 Server-Side Request Forgery (SSRF)	21	36
	CWE-444 Inconsistent Interpretation of HTTP Requests ('HTTP Request/Response Smuggling')	9	23
	CWE-346 Origin Validation Error	1	1
Improper Access Control	CWE-287 Improper Authentication	20	38
	CWE-284 Improper Access Control	14	28
	CWE-863 Incorrect Authorization	10	18
	CWE-347 Improper Verification of Cryptographic Signature	10	30
	CWE-295 Improper Certificate Validation	9	34
	CWE-384 Session Fixation	7	32
	CWE-522 Insufficiently Protected Credentials	6	10
	CWE-285 Improper Authorization	5	12
	CWE-276 Incorrect Default Permissions	5	7
	CWE-269 Improper Privilege Management	4	14
	CWE-345 Insufficient Verification of Data Authenticity	4	4
	CWE-640 Weak Password Recovery Mechanism for Forgotten Password	4	4
	CWE-294 Authentication Bypass by Capture-replay	3	4
	CWE-250 Execution with Unnecessary Privileges	3	4
	CWE-307 Improper Restriction of Excessive Authentication Attempts	3	8
	CWE-521 Weak Password Requirements	3	3
	CWE-290 Authentication Bypass by Spoofing	2	2
	CWE-306 Missing Authentication for Critical Function	2	7
	CWE-862 Missing Authorization	2	5
	CWE-1220 Insufficient Granularity of Access Control	2	6
	CWE-620 Unverified Password Change	2	12

Table 3 continued from previous page

Cluster Name	CWE Name	Commits	Functions
	CWE-305 Authentication Bypass by Primary Weakness	1	4
	CWE-289 Authentication Bypass by Alternate Name	1	2
	CWE-288 Authentication Bypass Using an Alternate Path or Channel	1	4
	CWE-304 Missing Critical Step in Authentication	1	2
	CWE-639 Authorization Bypass Through User-Controlled Key	1	1
	CWE-273 Improper Check for Dropped Privileges	1	1
	CWE-613 Insufficient Session Expiration	1	1
	CWE-749 Exposed Dangerous Method or Function	1	1
	CWE-940 Improper Verification of Source of a Communication Channel	1	2
	CWE-281 Improper Preservation of Permissions	1	1
	CWE-732 Incorrect Permission Assignment for Critical Resource	1	1
	CWE-942 Permissive Cross-domain Policy with Untrusted Domains	1	2
	CWE-322 Key Exchange without Entity Authentication	1	1
Computation Error	CWE-369 Divide By Zero	36	38
	CWE-190 Integer Overflow or Wraparound	19	24
	CWE-681 Incorrect Conversion between Numeric Types	6	9
	CWE-191 Integer Underflow (Wrap or Wraparound)	2	2
	CWE-682 Incorrect Calculation	2	5
	CWE-193 Off-by-one Error	1	1
Regular Expression	CWE-1333 Inefficient Regular Expression Complexity	18	25
	CWE-185 Incorrect Regular Expression	4	6
Uncontrolled Recursion	CWE-674 Uncontrolled Recursion	4	5
	CWE-835 Loop with Unreachable Exit Condition ('Infinite Loop')	10	17
	CWE-776 Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')	1	1
	CWE-834 Excessive Iteration	1	1
Uninitialized	CWE-824 Access of Uninitialized Pointer	10	11
	CWE-665 Improper Initialization	8	8
	CWE-908 Use of Uninitialized Resource	6	10
Improper Exception handling	CWE-754 Improper Check for Unusual or Exceptional Conditions	9	12
	CWE-12 ASP.NET Misconfiguration: Missing Custom Error Page	3	4
	CWE-755 Improper Handling of Exceptional Conditions	2	2
	CWE-460 Improper Cleanup on Thrown Exception	1	1
	CWE-248 Uncaught Exception	1	1
Incomplete Cleanup	CWE-459 Incomplete Cleanup	2	4
	CWE-416 Use After Free	2	4
	CWE-415 Double Free	2	3
Side Channel	CWE-203 Observable Discrepancy	2	4
	CWE-385 Covert Timing Channel	2	3
	CWE-208 Observable Timing Discrepancy	1	1
Format String	CWE-134 Use of Externally-Controlled Format String	2	4
Inefficient Algorithmic Complexity	CWE-330 Use of Insufficiently Random Values	1	3
	CWE-331 Insufficient Entropy	2	5
	CWE-338 Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	1	1
	CWE-328 Use of Weak Hash	1	1
	CWE-407 Inefficient Algorithmic Complexity	2	2
	CWE-326 Inadequate Encryption Strength	3	21
	CWE-327 Use of a Broken or Risky Cryptographic Algorithm	2	6
Incorrect Provision of Specified Functionality	CWE-684 Incorrect Provision of Specified Functionality	9	27
Always-Incorrect Control Flow Implementation	CWE-670 Always-Incorrect Control Flow Implementation	5	7

Table 3 continued from previous page

Cluster Name	CWE Name	Commits	Functions
Improper Validation of Integrity Check Value	CWE-354 Improper Validation of Integrity Check Value	5	8
Incorrect Comparison	CWE-697 Incorrect Comparison	4	10
Incorrect Type Conversion or Cast	CWE-704 Incorrect Type Conversion or Cast	2	2
Improper Handling of Alternate Encoding	CWE-173 Improper Handling of Alternate Encoding	2	6
Improper Handling of Structural Elements	CWE-237 Improper Handling of Structural Elements	2	2
Business Logic Errors	CWE-840 Business Logic Errors	2	6
Acceptance of Extraneous Untrusted Data With Trusted Data	CWE-349 Acceptance of Extraneous Untrusted Data With Trusted Data	2	2
Access of Resource Using Incompatible Type ('Type Confusion')	CWE-843 Access of Resource Using Incompatible Type ('Type Confusion')	2	6
Unprotected Alternate Channel	CWE-420 Unprotected Alternate Channel	2	2
Undefined Behavior for Input to API	CWE-475 Undefined Behavior for Input to API	2	2
Prototype Pollution	CWE-1321 Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution')	1	1
Improper Output Neutralization for Logs	CWE-117 Improper Output Neutralization for Logs	1	2
Client-Side Enforcement of Server-Side Security	CWE-602 Client-Side Enforcement of Server-Side Security	1	2
Improper Restriction of Rendered UI Layers or Frames	CWE-1021 Improper Restriction of Rendered UI Layers or Frames	1	1
Unchecked Return Value	CWE-252 Unchecked Return Value	1	1
Initialization of a Resource with an Insecure Default	CWE-1188 Initialization of a Resource with an Insecure Default	1	1
Mutable Attestation or Measurement Reporting Data	CWE-1283 Mutable Attestation or Measurement Reporting Data	1	1
Function Call With Incorrect Order of Arguments	CWE-683 Function Call With Incorrect Order of Arguments	1	1
Interpretation Conflict	CWE-436 Interpretation Conflict	1	1
Improper Control of Dynamically-Managed Code Resources	CWE-913 Improper Control of Dynamically-Managed Code Resources	1	1
Unimplemented or Unsupported Feature in UI	CWE-447 Unimplemented or Unsupported Feature in UI	1	1