Collecting Vulnerable Source Code from Open-Source Repositories for Dataset Generation

Razvan Raducu *, Gonzalo Esteban , Francisco J. Rodríguez Lera and Camino Fernández

Grupo de Robótica, Universidad de León. Avenida Jesuitas, s/n, 24007 León, Spain; gestc@unileon.es (G.E.); fjrod@unileon.es (F.J.R.L.); cferll@unileon.es (C.F.)

* Correspondence: rrad@unileon.es

Received: 29 December 2019; Accepted: 7 February 2020; Published: 13 February 2020

Abstract: Different Machine Learning techniques to detect software vulnerabilities have emerged in scientific and industrial scenarios. Different actors in these scenarios aim to develop algorithms for predicting security threats without requiring human intervention. However, these algorithms require data-driven engines based on the processing of huge amounts of data, known as datasets. This paper introduces the SonarCloud Vulnerable Code Prospector for C (SVCP4C). This tool aims to collect vulnerable source code from open source repositories linked to SonarCloud, an online tool that performs static analysis and tags the potentially vulnerable code. The tool provides a set of tagged files suitable for extracting features and creating training datasets for Machine Learning algorithms. This study presents a descriptive analysis of these files and overviews current status of C vulnerabilities, specifically buffer overflow, in the reviewed public repositories.

Keywords: vulnerability; sonarcloud; bot; source code; repository; buffer overflow

1. Introduction

In November 1988 the Internet suffered what is publicly known as “the first successful buffer overflow exploitation”. The exploit took advantage of the absence of buffer range-checking in one of the functions implemented in the fingerd daemon [1,2]. Even though it is considered to be the first exploited buffer overflow, different researchers had been working on buffer overflow for years. For example, James P. Anderson, on behalf of the electronic systems division [3], documented this topic in 1972.

In a simple search, the Common Vulnerabilities and Exposures (CVE) List presents more than 400 publicly known cybersecurity vulnerabilities associated with different types of buffer overflows on different platforms for 2019. Buffer overflow is the most reported vulnerability with both high and critical severity. The United States Industrial Control Systems Cyber Emergency Response Team specified that two out of the four most reported vulnerabilities were generated by buffer overflows [4]. Besides, WatchGuard [5] clearly identifies the fact that out of all the attacks they registered, the top four are all different occurrences of a buffer overflow.

Since Aleph One published the first step-by-step article about stack-based buffer overflow exploitation [6] in 1996, its popularity kept rising. In 2000, buffer overflow was declared the “vulnerability of the decade” [7]. Unsurprisingly, nowadays there is enough evidence to call it “the most occurring vulnerability in the last quarter-century” [8]. In addition, several authors, such as Cowan et al. [9] or Larochelle and Evans [10], proposed techniques to detect buffer overflow attacks in an automatic manner.

There are many types of buffer overflow attacks, such as write attacks, data manipulation/corruption attacks, and read attacks [11]. One of the possible reasons behind this plethora of overflows is C being inherently unsafe. That is, it allows low-level data and memory manipulation but it lacks...
the corresponding low-level security checks. One example of this are array and pointer references. None of them are automatically bounds-checked, therefore relegating security to the programmer’s skills. Besides, many of the standard C library functions such as gets(), scanf() or strcpy() are vulnerable [12–14]. Although C is one unsafe programming language whose misuse could lead to buffer overflows, it is not the only one. Buffer overflow vulnerabilities occur in languages that provide no built-in protection against out of bounds memory accesses such as C++ [15]. This leaves us with a single and clear conclusion: buffer overflow is still an ever-present vulnerability and it seriously jeopardizes security.

Prevention and defensive mechanisms are much needed to deal with such a menace to security. Techniques and tools, such as manual audition of code, static analyzers, compiler, and hardware modifications or dynamic analyzers, have been proposed for years.

Manual audit of code to find vulnerabilities is a challenging, error-prone, and time-consuming task. Besides, it relies on people trained enough to efficiently detect vulnerabilities, which is equally demanding [16]. However, manually reviewing the code can be complemented with static analyzers, which automatically identify potential security holes. One such static analyzer is SonarCloud (https://sonarcloud.io/about), a platform that helps developers write secure and clean code. It supports many different languages, and it is free when the project under analysis is open source. There are many other static analyzers, such as ITS4, a static C, and C++ source code scanner that splits the code into lexical tokens for further application of pattern matching [17]. The MIT Lincoln Laboratory exhaustively tested several static analyzers in order to measure their performance and accuracy rates [18]. Their conclusion is that further work is needed toward static detection of buffer overflow. Some static analysis tools can detect in-the-wild buffer overflows but are disappointing because false alarm rates are high and discrimination is poor.

Another way of preventing buffer overflows is writing code in programming languages that natively perform bounds-checking, such as Java or Pascal. These languages, however, lack low-level manipulation. With these limitations in mind, researchers have developed “safe dialects of C” that natively perform several security procedures such as controlled access to memory, strong object-typing, and bounds-checking. Unfortunately, security operations like bounds-checking generate up to 100% overhead [19]. Another approach consists of re-compiling the source code with security-aware modified compilers. StackGuard is one example of such modifications. It prevents stack-based buffer-overflow attacks by inserting canaries into the stack [9]. Nevertheless, when source code is not available, the previously-mentioned techniques are useless and other approaches are needed.

Regarding these approaches, many dynamic analyzers—also known as runtime solutions—have been proposed for preventing buffer overflows. One of these solutions is presented by Fraser, Badger & Feldman [20]. In their work, the authors defined the Generic Software Wrappers, which are protected, non-bypassable kernel-resident software extensions for security improvement without modification of the original software. Goldberg, Wagner, & Brewer [21] proposed Janus, a process that observes and mediates behavior by monitoring system calls. Naccio [22] is a system architecture that transforms programs according to predefined software policies. Something very similar to Naccio was proposed by Erlingsson and Schneider [23]. The authors called it SASI and it is a software fault-isolation technique that enforces security policies by modifying object code for a target system before that system is executed. In the same vein, Prasad and Chiueh [24] proposed a mechanism for rewriting Windows Portable Executable (PE) binaries so that they include return address protection mechanisms to preserve the integrity of the stack.

Static and dynamic techniques, enumerated above, are widely adopted for the most part by software engineers and are added to most Software Development Cycles. However, the use of machine learning (ML) techniques in cybersecurity has been continuously growing [25], specifically regarding vulnerability discovery, which has experienced a huge progress [26–28].
ML depends heavily on which data is provided to the algorithm and how it is represented. Therefore, it is necessary to generate datasets containing snippets of real, vulnerable code that are suitable for a given machine learning algorithm.

The need for datasets and their generation are recurrent topics related to several research fields. Thus, there are published works in research areas as varied as radio signal processing [29], vehicular technology [30,31], vehicle-to-vehicle and vehicle-to-infrastructure wireless communication [32], computer vision [33] and pattern recognition [34], cyber threat intelligence [35], host intrusion detection [36], network intrusion detection system [37,38], smart grids [39], and software vulnerabilities [40–45], among many others.

This research aims to offer an algorithm able to gather information about buffer overflow issues from a trustworthy source (such as SonarCloud) to construct datasets suitable for data science researchers. Thus, this study focuses on how to proceed with data gathering using crawlers. A crawler is a computer program capable of requesting and persisting data interactively and automatically [46,47]. Crawlers are especially useful for data-mining processes that involve a huge number of web requests and also parsing the corresponding responses for further analysis. Current approaches point in that direction. For instance, Daegeon et al.’s research [35] proposes a system for collecting threat data gathered from security reports and publicly available malware repositories.

The rest of the work starts with these two research questions.

RQ1 Which mechanisms/tools can software projects use to detect/eliminate well known software flaws that would lead to a vulnerability?

RQ2 Which methods are used to establish a dataset generator engine on source code containing buffer overflows?

The first question has already been answered in this section. For those software projects that are developed with programming languages with no inbuilt bounds checking, both static and dynamic analyzers are their main tool. Nevertheless, these tools provide long reports to programming experts who have to review them and decide which ones are real vulnerabilities and which ones are not. This expertise is what could be replaced using machine learning techniques, but to do that, the first step is to provide the community with datasets to work with.

From this point, the paper presents two main contributions: the first is the SVCP4C tool (SonarCloud Vulnerable Code Prospector For C), a program written in Python for gathering source-code repositories available in SonarCloud. It collects files linked to open source repositories that are written in C and tagged as vulnerable by the static analyzer. The tool is publicly available.

The second contribution is the analysis of data gathered from public repositories. Initially the authors present a statistical overview of data dumped by the SVCP4C tool and, additionally, they perform a naive inspection of the main vulnerabilities detected in current projects released on public repositories and loaded in SonarCloud.

This section introduced the common problems in generating datasets ready for ML algorithms. The next section presents the SVCP4C tool and the pipeline implemented for gathering data from public repositories. Section 2.2 treats the technical details associated with SVCP4C. Section 3 describes the limitations encountered during the development process, along with some future enhancements. In addition, the authors present a descriptive overview of the data gathered with the SVCP4C tool. In the final section, the conclusions are put forward along with the summary of this research.

2. Methodology

SVCP4C is part of a research project called TOOBAD4ML (TOOl to Buffer overflow Analysis and Description FOR Machine Learning). To establish a method to generate datasets with no human intervention from real code containing buffer overflows and answer research question 2, a proof of concept is proposed in the form of a tool. This tool will automatically parse source code, extract different characteristics from it, and export the data to some specific file format that is adequate
for an ML algorithm to predict possible buffer overflow vulnerabilities. Formally, it is a static vulnerability-analysis tool that, based on Abstract Syntax Trees (AST) and Control Flow Graphs (CFG) generated by Clang, models possible present buffer-overflow vulnerabilities via source code inspection. Clang is an LLVM front-end for the C language family [48]. The modeling of the vulnerability is inspired by previous works such as [26,28,49–51]. TOOBAD4ML’s conceptual diagram is illustrated in Figure 1. Further discussion of TOOBAD4ML is outside this paper’s scope.

Figure 1. TOOBAD4ML’s conceptual diagram.

As it was previously mentioned, there is a need to gather vulnerable code to train the ML algorithm. It is essential to have balanced data to avoid unbalancing the algorithm, overfitting, or many other problems that may arise and affect the prediction [52]. Collecting vulnerable source code via SonarCloud helps us to obtain samples of real, vulnerable code. Furthermore, by setting adequate query parameters we can also obtain non-vulnerable code. According to our prior research, only Kratkiewicz and Lippmann [49] offer a vulnerable dataset that is publicly available. The main drawback is that it is synthetic code, which may not be valid for training or testing ML algorithms. Real code from real applications is needed to properly train and test the ML algorithm.

2.1. SonarCloud Web API

SVCP4C is made to communicate with and depend on SonarCloud’s REST API. The API documentation can be found on SonarCloud’s official site (https://sonarcloud.io/web_api). Working with the API is fairly simple: HTTP GET requests are made to a certain URL with certain parameters in order to get a JSON-formatted response from SonarCloud. The API is publicly available and free to use. The API offers several services with which information about the queried source code can be obtained. We will focus exclusively on the functionality used by SVCP4C. SonarCloud’s API has several main services and other so-called internal services. Internal services are those that must be used at one’s own risk since they are subject to change or removal without previous notification. SVCP4C uses only three main services:

1. /api/components/search_project
2. /api/issues/search
3. /api/sources/raw

SonarCloud also offers a Graphical User Interface (GUI) version available via their website. It is important to mention the GUI because it runs the same API. The website sends requests to the webservices and parses the JSON response to draw and serve what the user is requesting. Figure 2
shows an example of SonarCloud GUI version reporting a vulnerability because unsafe `strcpy()` function is in use. All SonarCloud responses are GUI-oriented. This means that when requesting issues via API’s REST methods, the response will locate the vulnerable function’s name rather than its full signature, because only function’s name would have been drawn in the GUI. Not reporting the full function’s signature is yet another challenge TOOBAD4ML must deal with in its parsing process since a function’s arguments are crucial.

```
if(sizeof(argv[1]) < sizeof(buffer))
    strcpy(buffer, argv[1]);
```

Figure 2. SonarCloud detecting use of an insecure function.

2.2. Algorithm Pipeline

The overall workflow of SVCP4C is depicted in Figure 3. The workflow is split into five phases.

In the first phase, the crawler requests information about the source code identified as vulnerable. For this phase, several sources for gathering BufferOverflow information have been analyzed. We initially checked the Common Weakness Enumeration (CWE) [53] and Common Vulnerability Enumeration. CWE is a community-developed list of common software security weaknesses. It represents the starting point for weakness identification, mitigation, and prevention from a source code point of view. CVE presents a list of entries for publicly known cybersecurity vulnerabilities. CWE, which focuses on software approaches, identifies software vulnerabilities such as CWE-120: Buffer Copy without Checking Size of Input; CWE-121: Stack-based Buffer Overflow; and CWE-122: Heap-based Buffer Overflow.

On top of them, there are formal datasources such as the National Vulnerability Database (NVD) [54] or informal such as CVEDetails.com [55]. Although these sources are enough for some vulnerability analysis [56], they do not present a direct link to the source code that represents the buffer overflow, and it is necessary to crawl in-depth, looking for public repositories (if possible) to reach the vulnerability and parse it.

Given this scenario, we decided to focus our efforts on lists that provide information about possible buffer overflow issues that could be raised in a project. Thus, we initially had two possibilities: (1) massively download github repositories and analyze the source code locally and (2) analyze results of current Software As A Service tools, that provide this information in the cloud.

We analyzed current cloud solutions offering this information. Although there are several options like Datree.io, Codescene.io, or kiwan.com, given the REST API limitations imposed by licenses and types of accounts, and because we had experience with SonarQube, we decided to choose SonarCloud for our research.

In the second phase, the crawlers collect the source code files identifies in the previous phase. We take advantage of the possibility that SonarCloud offers of retrieving the issues of a given source code file (previous phase) and download that particular file, simplifying the process of data collection.

In the third phase, SVCP4C appends the vulnerable lines as comments to the original source code files. This way the files are easier to handle in further analysis.

In the forth phase, the commented source code is stored in the local file system.

In the last phase, the result of the whole process is released to the public. The data is publicly available at the repository hosted by Github. The source code of SVCP4C is released as well.

Figure 3. SonarCloud detecting use of an insecure function.
The pipeline of our algorithm is illustrated in Listing 1. Some parameters that appear in the algorithm require further contextualization:

- $p$. This parameter represents page number. SonarCloud responds with at most 500 results per page. If one query generates more than 500 results, $p$ is pre-incremented and the web service is requested again.
- $ps$. This parameter represents page size. SonarCloud allows users to specify how many results they want to see per page, in our case per HTTP response. $ps$ is a constant equal to 500 as it is the maximum page size allowed by SonarCloud and we want to retrieve as much information as possible.
- $remainingResults$. This parameter represents how many results are left. That is, if the query generated more than 500 results, $remainingResults$ is checked to request again.

SVCP4C performs several HTTP requests to SonarCloud’s REST API. These requests are grouped into three phases: phase 1, the engine requests the ids of those projects that SonarCloud tagged to contain issues or vulnerabilities; phase 2, the engine requests information about the files that causes the projects to be tagged; and phase 3, retrieves the unique identifier of each source code file and downloads it.

During the first phase, performed in step 7 of Listing 1, the algorithm retrieves the ids of the projects that meet our filtering conditions. The filtering is performed by SonarCloud’s API via URL parameters. The requested web service is /api/components/search_projects and the parameters are:

- $filter$. security_rating$\geq$2 and languages=c
- $p$. $p=i$ (ith-page)
- $ps$. $ps=500$ (current page size)

As shown in Listing 1, $p$ is the number of page, $ps$ is page size, and security_rating $\geq$ 2 implies a $B$ security rating according to the analysis performed by SonarCloud. Different SonarCloud’s metrics and ratings can be found in the official documentation [57]. $B$ security rating means “at least one Minor Vulnerability”. $A$ security rating corresponds to non-vulnerable code and is represented via security_rating=1 in the HTTP request. This approach must be used in order to obtain the non-vulnerable source code according to SonarCloud. The resultant queried URL is: https://sonarcloud.io/api/components/search_projects?ps=500&p=1&filter=security_rating%3E%3D2+and+languages=c

Afterwards, phase 2 performs the HTTP request in order to obtain the unique identifier of every vulnerable source file within each previously queried project. It is triggered at step 20 of Listing 1. The requested web service is /api/issues/search, and the parameters are as follows.

- $projects$. projects=1,2,3 (a list of all project ids previously queried, comma separated).
- $types$. types=VULNERABILITY (SonarCloud issue category).
- $languages$. languages=c (a list of program languages, comma separated).
- $p$. $p=i$ (ith-page).
- $ps$. $ps=500$ (current page size).

The $types$-parameter is used to specify which issue we are looking for, i.e., returns the unique identifier of source files affected only by the specified type of issue. There are four types of issues that SonarCloud detects: CODE_SMELL, BUG, VULNERABILITY, or SECURITY_HOTSPOT [58]. The remaining parameters have already been introduced.

Finally, phase 3, in step 36 of Listing 1, defines SVCP4C’s last query. For each of the unique source file ids obtained in the previous phase, SonarCloud is requested to provide the corresponding source code. The requested web service is /api/sources/raw and it is necessary to employ the key parameter that is the unique identifier of the file whose code is about to be retrieved.
Listing 1. SVCP4C’s pseudocode.

```plaintext
1 CHECK user arguments AND options;
   IF (path from step 1 doesn’t exist) THEN
5       Create path;
OTHERWISE
9       Abort with error;
13       SET p := 1 AND remainingResults := 0;
PROCEDURE. Request project ids():
17          HTTP GET request (url, params);
21          RETRIEVE all HTTP response payload from step 8 as JSON;
UPDATE remainingResults AND JUMP to step 12;
25          IF (remainingResults > ps) THEN
29             IF (p == 20) THEN
33                 JUMP to step 19;
37                 PRE-INCREMENT p;
41                 JUMP to step 7;
45             OTHERWISE
49                 JUMP to step 19;
53             END_IF;
57             OBTAIN all project ids from step 9 AND set p:=1 AND remainingResults:=0;
PROCEDURE. Request files info():
61          HTTP GET request (url, params);
65          RETRIEVE all HTTP response payload from step 21 as JSON;
69          WRITE results of step 22 to file;
73          JUMP to step 34;
77          UPDATE remainingResults AND JUMP to step 27;
81          END_PROCEDURE;
85          IF (remaining query results > ps) THEN
89             IF (p == 20) THEN
93                 JUMP to step 50;
97                 PRE-INCREMENT p;
101                JUMP to step 20;
105            OTHERWISE
109                JUMP to step 50;
113            END_IF;
117            OPEN file from step 23 AND parse its JSON formatted content;
121            FOR each (issue (key, value) from results of step 34) DO:
125                RETRIEVE the value of component key
129                HTTP GET request (url, params)
133                IF (response from step 37 contains errors) THEN
137                   PRINT_MESSAGE: the file was skipped because there was an error;
141                OTHERWISE
145                   GO TO step 42;
149                OBTAIN name of file to be persisted based upon the naming policy;
153                IF (file with name from step 42 does not exist) THEN
157                   CREATE file AND append at the end the separator comment line;
161                OTHERWISE
165                   JUMP to step 47;
169                   APPEND the vulnerable line from step 35;
173                   JUMP to step 25;
177            END_FOREACH;
181            END_PROGRAM.
```

After the source code is retrieved, for each vulnerability, a line is appended to the original source file. The format of each appended line is "///sl,so;el,eo", where sl is the starting line, so is the starting offset, el is the ending line, and eo is the ending offset.
It is necessary to emphasize that, at this stage, SVCP4C algorithm has a hard link to SonarCloud’s Elastic Search indexing engine; as a result, its execution may yield different results depending on SonarCloud’s update policies.

This section has proposed a method to generate datasets from open source code with no human intervention. The method has been applied to create a tool that using SonarCloud to detect buffer overflows, produces datasets with annotated information about the vulnerabilities and ready to be used by machine learning techniques.

3. Discussion

The present section details the different datasets obtained using SCVP4C, the restrictions of SonarCloud’s API and thus limitations of SVCP4C, and SVCP4C enhancements.

3.1. Data Analysis

By executing SVCP4C several datasets have been gathered. These datasets can be inspected and downloaded from a publicly available repository (https://github.com/uleroboticsgroup/SVCP4CDataset). The average size of each dataset is 101,905.6 kilobytes with a standard deviation of 1373 kilobytes (when compressed, 23,576.8 kilobytes is the average with a standard deviation of 338.75 when tared and compressed in gz).

Data gathered may be used to evaluate some characteristics of current software solutions. On the one hand, a quick overview demonstrates that most vulnerabilities present in the downloaded source code files correspond to Standard C library functions: `sprintf` (46.84%); `strcpy` (36.38%); `strcat` (15.27%); `strlen` (1.3%); and, to a lesser extent, (0.2%) functions such as `scanf`, `snprintf`, `strchr`, and `fscanf`. On the other hand, the issues obtained from SonarCloud present the histogram illustrated in Figure 4. The histogram distribution reflects the issues range, bounded from 1 to 298, with a mean of 4.82 errors (standard deviation of 11.89) but with a mode value equal to 1 (it is the most repeated value).

![Figure 4. Cont.](image-url)
3.2. Constraints Related to SonarCloud’s Web API

One of the most important restrictions to face when querying SonarCloud’s web API is the one commonly known as the 10,000 issue limit. This constraint implies that every single request made to /api/issues/search will be responded to only with the first 10,000 results. As SonarCloud’s prior, and now deprecated, documentation page states: “If the number of issues is greater than 10,000, only the first 10,000 ones are returned by the web service” [59]. Even though the quoted sentence comes from older documentation, the limit still applies nowadays, despite being undocumented. There are many questions in different forums, from users just like us, asking about this very same limit. The response is always the same: there is no way around it.

Another drawback SonarCloud presents is the lack of vulnerability-type filtering. That is, the ability to retrieve only source code files that are tagged as vulnerable to a given vulnerability (stack-based buffer overflow, format string, integer overflow...). Although filtering by issue type (VULNERABILITY, HOTSPOT...) is possible, filtering based on vulnerability type is a much needed feature to gather specific source code. From TOOBAD4ML’s perspective, such a feature would ease the parsing job. Currently, the datasets gathered by SVCP4C include all kinds of vulnerabilities, not only buffer overflow.

Up to this point during development, we faced some problems whose solution(s) can be directly seen in Listing 1. For example, we found out that we cannot just append every result of the queries asking for vulnerabilities into one single file because the result is a mal-formatted JSON. This is because SonarCloud sends JSON objects as responses and, as such, these include the opening and closing square brackets. A JSON file, to be well-formatted, must include a single JSON object, that is, a single pair of square opening and closing brackets. The solution we adopted is requesting the first 500 results (page 1) and write them to a file. Immediately after, we parse the file and request the corresponding source code. When we get it, we request the next 500 (page 2) vulnerabilities, write (not append) them to a file and, once again, request the source code. This loop goes on until we reach the 10,000-results limit imposed by SonarCloud. This is reflected in steps 23 and 34 of Listing 1.

Another characteristic behavior of SonarCloud’s web API is that each vulnerable code line within the same source code file is treated as a different issue. That is, a source file with 13 different vulnerabilities is translated into 13 different issues when querying SonarCloud. At the moment of retrieving the corresponding source file, the same file would be downloaded 13 times, whereas only the tagged line would be different. The solution implemented in SVCP4C is to compile all vulnerable lines that refer to the same file, download the file, and append the lines as a comment at the end of the file (step 47 in Listing 1).
Finally, additional checks are required when requesting source files because SonarCloud references missing files. That is, it maintains the list of issues even though the file those issues arise from does not exist anymore. When attempting to download a missing source file the result is a file whose sole content is a JSON list called “errors” containing “msg” keys. The solution implemented in SVCP4C is rather straightforward, the content that is about to be written out is first inspected and, if it contains any “errors” JSON list, we skip it. Step 39 of Listing 1 shows the check.

3.3. Future Enhancements

The performance of SVCP4C could be improved in several aspects. First, SVCP4C does not parallelize HTTP requests. There are several existing solutions for parallelizing HTTP requests in Python, and implementing one of them is crucial to reduce download time. However, with parallel requests several difficulties may arise, for example, with parallel requests come parallel responses; therefore, persistence becomes a critical operation which shall involve synchronization mechanisms. Moreover, asynchronous requests imply receiving responses in no particular order.

Regarding the 10,000 query result limit, the restriction itself cannot be eliminated because that is the way SonarCloud’s web API is implemented. We could, however, surpass it. To do so, and assuming no project has more than 10,000 issues, the issues must be requested on a per-project basis. As of right now, SVCP4C retrieves all project ids that meet our filtering criteria and requests the issues of all the ids altogether. This change would affect the performance as SVCP4C would go from a single HTTP GET request specifying $N$ project ids to an HTTP GET request per project id ($N$ HTTP GET requests).

Complementing the detection with dictionaries of vulnerable functions could improve the accuracy ratio when tagging a specific code line as vulnerable. This improvement would greatly reduce the number of vulnerabilities unreported by SonarCloud. To illustrate a case where SonarCloud fails to detect the vulnerability, imagine a buffer 16 bytes long. If the programmer uses the `scanf` function to fill it up, SonarCloud successfully detects the possible buffer overflow by reporting the use of unsafe functions. As a fix, SonarCloud wisely recommends the use of a width specifier for the corresponding placeholder. It is also stated in the Common Weakness Enumeration [60]. However, as soon as the developer places the width specifier, SonarCloud assumes it is a correct one. We consider this assumption both critical and harmful as a self-induced buffer overflow may arise. The programmer could use a width specifier bigger than the actual buffer to which the data will be copied to. Assuming the previous 16-byt buffer, a program could fill it with up to 30 bytes of data if the programmer used the "%30s" width specifier for the function `scanf`. In this case, the difference in size is evident but a more subtle off-by-one buffer overflow could remain undetected. With the help of dictionaries, functions could be specified in order to flag them and always check their parameters.

As SonarCloud’s responses consist of starting and ending line and starting and ending offset (column) of the vulnerability, highlighting the piece of code if the request is made using GUI, future research should consider expanding this information. From TOOBAD4ML’s perspective, it is much more useful to retrieve the start and end positions of full vulnerable function signature instead of simply its invocation keyword (Figure 2) or a single parameter. A function may have a variable number of arguments or spread across multiple lines, among others, which complicates its static analysis. Knowing beforehand its starting and finishing positions eases its parsing and thus its analysis.

4. Conclusions

Buffer overflow has been one of the most investigated vulnerabilities for decades, and the prevention and defense mechanisms against it is a cornerstone for any cybersecurity researcher. Auditing code, static analyzers, and ML are among the techniques used today to counteract buffer overflows. Furthermore, academic literature shows that many efforts are being carried out towards the detection of software vulnerabilities with ML.

Detecting and eliminating vulnerabilities in software projects is a hard task developed by programming experts with the help of static and dynamic code analyzers. As long as that is an activity
that involves human expertise, machine learning could be used to improve the process. The first step for machine learning to be applied is to provide the community with annotated datasets. This work has proposed a method to generate datasets from open source code with no human intervention. The technical pipeline of a crawler-like tool called SonarCloud Vulnerable Code Prospector For C (SVCP4C) has been described. The tool tags those lines identified by SonarCloud as vulnerable.

In addition, this study has reviewed the findings generated by the use of SVCP4C for enumerating existing Buffer overflow vulnerabilities in more than 10k source code files.

SVCP4C is a valuable tool that simplifies the process of dataset generation for its use by data science researchers. This tool gathers data from open-source repositories available through SonarCloud, which already defines vulnerable code. This is significantly important in light of the increase of Machine Learning approaches for detecting buffer overflows in a software solution.

SVCP4C source code is released in the group’s GitHub repository (https://github.com/uleroboticsgroup/SVCP4C) along with the datasets generated in this study (https://github.com/uleroboticsgroup/SVCP4CDataset). The datasets are also published on SciCrunch, a place for sharing access to scientific resources with other researchers and enhancing their visibility, under resource ID: RRID:SCR_018011.


Funding: This research has been partially funded by the “Universidad de León–Instituto Nacional de Ciberseguridad (INCIBE) Convention Framework about «Detection of new threats and unknown patterns»” (Spain), by the “Consejería de Educación de la Junta de Castilla y León” (Spain) under grant LE028P17 and by the “Ministerio de Ciencia, Innovación y Universidades” (Spain) under grant RTI2018-100683-B-100.

Acknowledgments: This work has been partially funded by the Addendum no. 4 to the Universidad de León-Instituto Nacional de Ciberseguridad (INCIBE) Convention Framework on the “Detection of new threats and unknown patterns”, by the Consejería de Educación de la Junta de Castilla y León through the Project LE028P17 on the “Development of reusable software components based on machine learning for the cybersecurity of autonomous robots” and by the Ministerio de Ciencia, Innovación y Universidades through the Project RTI2018-100683-B-100.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ML Machine Learning

References


