

UNIVERSITÀ DEGLI STUDI DI PADOVA

Department of Information Engineering

Master's degree in Computer Engineering

Master's degree Thesis

**Methodologies to support the development
of secure code**

Supervisor:

Dr. Mauro Migliardi

Company supervisors:

Luca Bianconi, Filippo Oliva

Candidate: Enrico Vicentini

Badge number: 2024043

Academic year 2021/2022

Graduation date 05/12/2022

Contents

1 Introduction	6
2 State of the art	9
2.1 Cyberattacks	9
2.1.1 Cross-Site Scripting attack (XSS attack)	13
2.1.2 SQL Injection attack	19
2.2 Compiler level security	23
2.2.1 .NET Compiler Platform API ("Roslyn")	24
2.3 Visual Studio Analyzers and code-fixes	26
2.3.1 Code static analysis	28
2.3.2 Roslyn based analyzers features	28
2.3.3 Code fixes	32
2.4 Language utilities	33
3 Security Enhancer implementation	35
3.1 Project structure	35
3.2 SecurityEnhancer SDK	37
3.3 SecurityEnhancer analyzers	39
3.3.1 XSS analyzer	39
3.3.2 SQL analyzer	46
4 Results evaluation	55
5 Conclusions	61

Abstract (english version)

In the development of industrial software products, adherence to certain standards, first and foremost security standards, is essential to ensure not only the quality of the software but also to make sure that the software is not a vehicle for the introduction of security flaws into the systems that host it. The modern software industry routinely involves the use of third-party libraries and software components for integration into its programs. This code is often shared publicly through opensource repositories (and practices). However, in using this code, one usually sets out to adopt it to fulfill the functional needs necessary for the systems it becomes part of, without taking into account its security details and requirements.

A key aspect of producing quality software is therefore to have the ability to analyze and evaluate the security of these libraries before and after they are inserted within one's code (and software).

Gruppo SIGLA as a software company has a primary interest in ensuring according to industry standards the quality of the code produced by its projects. In particular, Gruppo SIGLA wants to implement a static code-security analyzer (e.g., third-party library safety analysis) and auto-correction suggestion tools to support its engineers in meeting specific code quality requirements, in terms of safety.

In addition, it is important to understand the state of the art, identify and define appropriate metrics for evaluating each of the code safety parameters under consideration. Once defined, it will also be necessary to implement and validate the algorithms for calculating these metrics.

Ultimately, the goal is to implement the designed analysis tools in an "analyzer" module that can eventually be integrated within development environments (IDEs) to more nimbly support the work of developers. In the first instance, one can think of integrating the analyzer as a Visual Studio 2022 Community Edition plugin with the possibility, later, of it being easily integrated into other IDEs, such as Visual Studio Code.

Abstract (italian version)

Nello sviluppo di prodotti software di tipo industriale il rispetto di determinati standard, innanzitutto di sicurezza, è essenziale per garantire non solo la qualità del software ma anche per fare in modo che quest'ultimo non sia veicolo per l'introduzione di falle di sicurezza nei sistemi che lo ospitano. La moderna industria del software prevede, come pratica di routine, l'utilizzo di librerie e componenti software di terze parti da integrare nei propri programmi. Questo codice è spesso condiviso pubblicamente tramite repository (e pratiche) opensource. Tuttavia, nell'utilizzo di questo codice di solito ci si propone di adottarlo per assolvere alle necessità funzionali necessarie ai sistemi di cui entra a fare parte, senza tenerne in considerazione dettagli e requisiti di sicurezza.

Un aspetto fondamentale per produrre software di qualità è quindi avere la possibilità di analizzare e valutare la sicurezza di queste librerie prima e dopo che vengano inserite all'interno del proprio codice (e del proprio software).

Gruppo SIGLA in quanto azienda produttrice di software ha un precipuo interesse nel garantire secondo standard industriali la qualità del codice prodotto dai suoi progetti. In particolare, Gruppo SIGLA vuole implementare un analizzatore di sicurezza statica del codice (ad es. analisi sicurezza librerie di terze parti) e strumenti di suggerimento di correzione automatica per supportare i suoi tecnici nel rispetto di specifici requisiti di qualità del codice, in termini di sicurezza.

Inoltre, è importante comprendere lo stato dell'arte, individuare e definire adeguate metriche di valutazione di ciascuno dei parametri di sicurezza del codice presi in considerazione. Una volta definite sarà inoltre necessario implementare e validare gli algoritmi di calcolo di suddette metriche.

Da ultimo l'obiettivo è quello di implementare gli strumenti di analisi progettati in un modulo "analizzatore" che possa essere infine integrato all'interno di ambienti di sviluppo (IDE) per supportare più agilmente il lavoro degli sviluppatori. In prima battuta, si può pensare all'integrazione dell'analizzatore come plugin di Visual Studio 2022 Community

Edition con la possibilità, a seguire, di essere facilmente integrato in altri IDE, come ad es. Visual Studio Code.

1 Introduction

Making mistakes is very often a good way to learn, however, if these mistakes cause the security and integrity of a system to be compromised, it is a good idea to make sure they are as few as possible. In most cases errors come from a human oversight and so are not voluntary. In particular in the field of IT, programmers are faced with the problem of ensuring the security of the code they develop in order to limit the possibility of some malicious person being able to break into the system exploiting code flaws. It is for this reason that "Gruppo S.I.G.L.A." has proposed as a thesis project the development of a tool that can support programmers in writing secure code.

Information technology has now become part of everyday life, both in the private and business spheres, resulting in an increase in the number of network-connected devices and the emergence of new technologies. This phenomenon has led to an increase in the number of people interested in the world of information technology that, who for passion, who for the prospects of earning money, are investing time and money in this new field of work. With regard to the case at hand, this translates into an increase in the number of individuals with programming skills that, in most cases, are hired in a work environment in compliance with laws and regulations, while in others are employed in illegal activities that lead to the perpetration of cyber crimes. The growth in the number of hackers has obviously resulted in the growth in the number of cyber attacks, suffice it to say that the number of annual malware infections has increased from 12.4 million in 2009 to 812.67 million in 2018 with an increase of more than 8 million in the different malware specimens between 2007 and 2017. Figure 1 and Figure 2 show this two trends.

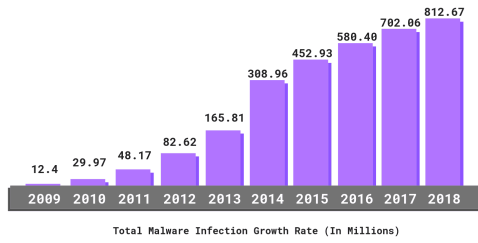


Figure 1: Server configuration for Elgg

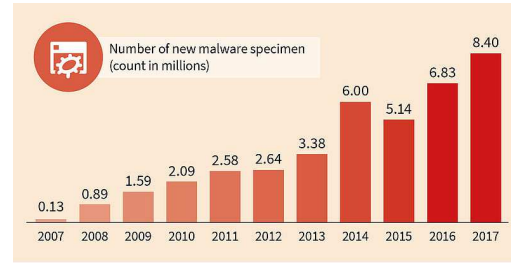


Figure 2: Server configuration for example

The vastness of computer systems (larger system means more entry points) and the wide variety of cyber attack kinds makes the situation of absolute security almost unattainable. Even a careful design of the system structure and careful definition of defensive systems may not necessarily be sufficient to prevent the most dangerous intrusions or to limit their damages.

This is the motivation behind the need for the creation of a tool that can warn the programmer in real time if he or she has written code that is potentially prone to vulnerabilities.

To be more specific, the idea is to implement a syntactic analyzer with code fix capability able to highlight in real time critical issues in the code, i.e., those variables, properties, etc., which can become vehicles for malicious code or otherwise give access to confidential data or memory areas, and propose to the user a possible countermeasure to the vulnerability. The analyzer will be built as a plugin for the popular Microsoft IDE Visual Studio Community 2022. The **.NET Compiler Platform SDK (Roslyn API)** present within this editor will act as a power supply for the analyzer making it possible to have instantaneous supervision over what is typed by triggering alerts and correction hints in case of irregularities in the code. Due to the fact that the analysis is performed without executing the program, it has been called static.

This thesis will present the analyzer creation process starting from Chapter 2 which illustrates the fundamental concepts which the experience is built on. Special attention is paid to explain what a cyberattack is and to present the two kinds of attack that the analyzer aims at addressing. Moreover, the development environment is introduced, exposing the meaning of compiler, with a focus on the .NET Compiler Platform SDK ("Roslyn"), and of analyzer, in terms of structure and theoretical features. After that the thesis will continue with Chapter 3 which shows the implementative details of the solution and so, to be more precise, how the error detection and the code fix procedures

are carried out. Chapter 4 shows the project results, verifying the proper functioning of the extension in identifying and correcting errors. Chapter 5 contains a sum up of the experience to verify the effectiveness of the work done, focusing in understanding how available this solution can be, what are the critical points and which aspects may and what aspects can improve the overall experience of using the extension in terms of types of errors detected and quality of corrections.

2 State of the art

This section will present the basic concepts for understanding the topics covered and the tools used. It will be clarified what a cyber attack consists of and more specifically the Cross-Site Scripting Attack (XSS Attack) and SQL Injection types will be explained. These are indeed the threats to code security, which the analyzer developed as the subject of this thesis aims to detect and eradicate. Therefore, it will be seen how errors or carelessness during software coding can make it vulnerable to these kinds of attacks. As next step there will be the analysis of the tools that made the project achievable, so, in particular, the Roslyn API and the context which embeds it. Finally, the thesis will provide information on the structure of an analyzer and also about the theoretical concepts used in it.

2.1 Cyberattacks

Perpetrating a cyberattack consists of exploiting a vulnerability of a computer, infrastructure, network or computerized system to cause a harm to the target. In this context, the term damage includes loss of data **confidentiality**, **integrity** and **availability**, where for confidentiality is meant data access granted only to allowed people, for integrity that private data or system functionalities did not change without the owner will or control and for availability the possibility to access data, resources and functionalities whenever the need arises. In addition to these three principles called **C-I-A Triad** other two characteristics are sometimes desirable: the **authentication**, the ability to demonstrate the truthfulness and validity of something, and the **nonrepudiation**, that is the assurance that what has been established between two parties cannot be denied. As a consequence harms can be classified in four categories: **interception**, **interruption**, **modification** and **fabrication**; depending on the attacker's goal [1].

If the attack permits to the attacker to obtain information that normally he is not authorized to achieve, a loss of confidentiality occurred due to the interception of private assets. An example of this kind attack is the so called *phishing* in which using a fake version of a website login page, victim's credentials are intercepted and stolen. Figure 3 shows a schematization of the situation.

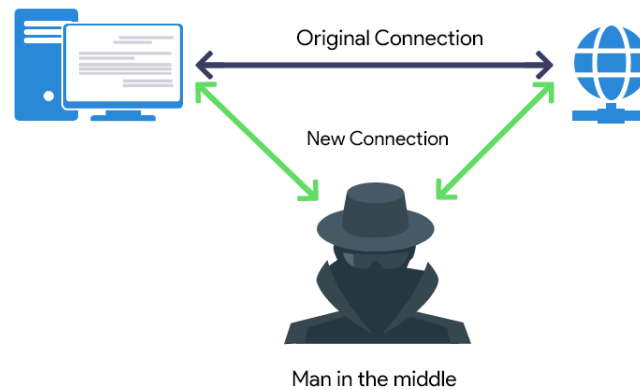


Figure 3: Interception type scheme

When an attack is able to tamper the usual functioning of a system's asset instead, availability problem arises and an interruption of a system's service or functionality occurs. The so called *DoS - Denial-of-Service attacks* are an example of this threat, in fact overloading a system's network connections through a huge quantity of resource's requests, brings it to collapse or at least to slowing down. Scheme of this can be found in Figure 4.

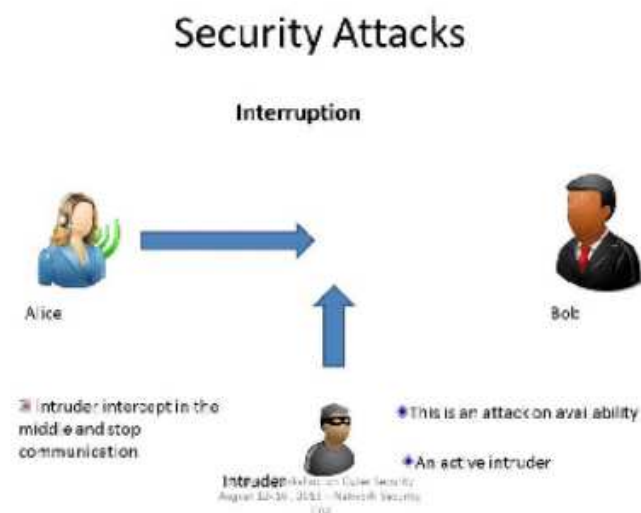


Figure 4: Interruption type scheme

In the cases in which system's functionalities or data are altered by someone not authorized, an integrity loss occurs and a modification attack is performed. An example of such an attack is the *Cross-Site Scripting attack (XSS attack)* that will have a detailed presentation later on in the discussion. Just to give an idea, Figure 5 contains a graphical representation of this situation.

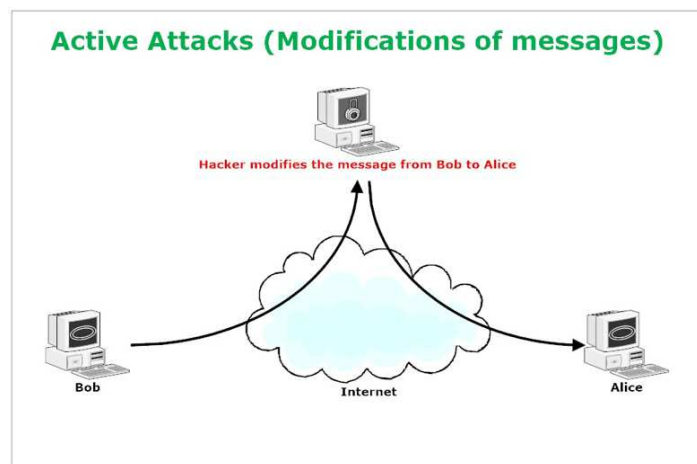


Figure 5: Modification type scheme

In the fabrication category fall all those attacks that create from scratch illegitimate data, information or resources, providing the public with material whose authenticity has not been verified. The *SQL Injection attack*, presented at page 19, is an example of attack which is executed in the just explained way. Figure 6 provides a visual scheme of how the attacks belonging to this category operate.

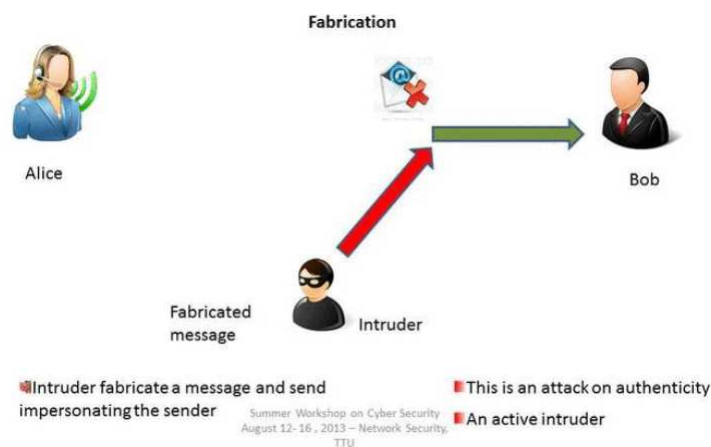


Figure 6: Modification type scheme

Countermeasures types

The term countermeasure refers to a means of countering threats. The eventual damages resulting from the exposure to an attack can be managed in different ways:

Prevention: Implementing measures able to remove a vulnerability. It can be done inside or outside the system. Some examples are firewalls, access limitation, etc.

Deterrence: Discourage attackers by increasing the effort required to perform the attack or even better remove the motivation behind it.

Deflection: Redirection towards a more convenient target like sandbox or honeypot where the attack becomes harmless

Detection: Identifies whenever any event tries to activate not planned behaviours of the system.

Recovery: Set up tools that permits the recover from the damages caused by successful attack, such as backup procedure.

Mitigation: Activate procedure to limit the impact of an attack against the system.

To limit the exposure of systems to the threat of cyberattacks, countermeasures can be placed in different points of the system depending on the defensive strategy applied [2].

Figure 7 gives a general idea about where and how controls act.

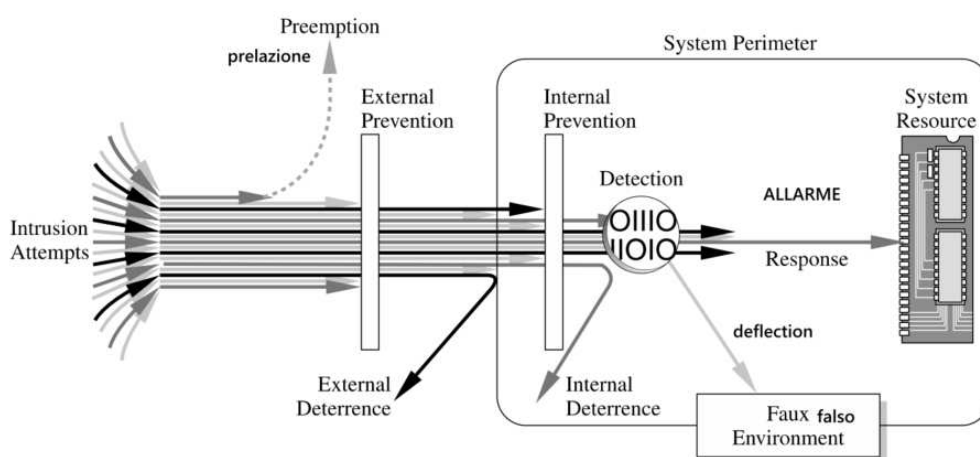


Figure 7: Controls location places

Within this context, the proposed project acts like an intermediate solution between the detection and prevention types, in fact it will identify the vulnerabilities in the just

produced code, warns the programmer about the possible weaknesses and suggests any methods able to prevent the occurrence of the attacks.

Now that an overview about which dangers a computer system may have to face was given, it's time to focus the attention on the kinds of attacks that the analyzer designed in the project will try to prevent. The choice was made taking into account the candidate's personal knowledge of attacks, the opportunity to enrich the cybersecurity knowledge by dealing with an attack that has not yet been dealt with, and the suggestions given by the host company about how useful it might actually be having a support against this kind of threat. These are the motivation between the choice of building an analyzer that helps in preventing code flaws that can expose the system to **Cross-Site Scripting attack (XSS attack)** and **SQL Injection attack**.

2.1.1 Cross-Site Scripting attack (XSS attack)

A Cross-Site Scripting attack is a cyberattack which involves three different actors: an attacker, a victim and a web application through which the attack is carried out. The owner of the web application plays also the role of victim because if a user suffers a damage the provider itself is subject of harm due to the consequent loss of reliability. The XSS exploit the principle of code injection to alter the regular behaviour of the infected software.

Having access to a user's personal web page is usually difficult for an attacker due to the presence of security measures like authentication, sessions cookies and so on. For this reason hackers adopted another strategy to execute their malicious purposes, that is injecting code into victim's browser instead of directly in the pages. This solution is quite simple to put in place, in fact browsers execute every instructions received from a web page. If this instructions consist of malicious code the browser is not aware of its nature and treats it as normal code performing whatever he reads from it. For this reason if someone is able to read and modify the URL generated by, for example, the website request message of editing a field in a user's profile, that person has the capability to create a modified version of it and the power to control the behaviour of the page. Of course to do that, is also necessary to be in possess of the authentication token of the user's session that is one time generated during the login operation, and so not so easy

to get. However, if it's the user himself to access the page in which the malicious code is injected, the token is already associated with his session and so the only thing left to the hacker is to write the correct instructions to trigger the editing request that the user did not actually authorize. To be aware of the structure of the url used by the website, the attacker just need to enrol in it and use a sniffing tool to intercept the request messages that the pages sent when an operation on the data is required. Figure 8 shows a simple representation of the scenario.

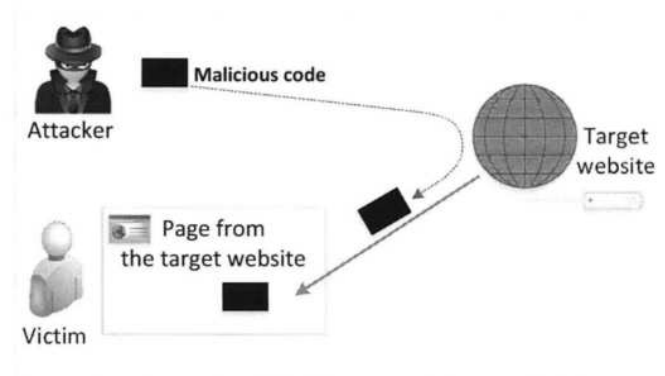


Figure 8: General scheme of an XSS attack

Two types of attack

There are two typical ways for attackers to inject their code into a victim's browser via the target website. One is called non-persistent XSS attack, and the other is called persistent XSS attack [3].

a. Non-persistent (Reflected) XSS Attack

When it comes to Non-persistent XSS, it means that the attack is performed exploiting a behaviour of some websites that after having received an input and having elaborated it, they send a response message containing the input itself (reflection). If this input consist of JavaScript malicious code and if no appropriate countermeasures (input sanitize) are applied from the website, the script will be injected in the returned page during the reflection step.

As an example let's imagine that the following URL is sent to a victim and that him/her opens it (obviously special characters need to be encoded):

```
http://www.example.com/search?input=<script>alert("attack");</script>
```


where `input` is the name of the parameter that contains the text typed from the user. The website `www.example.com` as response returns a page containing the result of the search and the original input as a reminder of about what the search was for. If the input is not properly sanitized the `script` part is injected into the victim's browser and the alert message is displayed. Figure 9 shows this type of attack in a simple scheme.

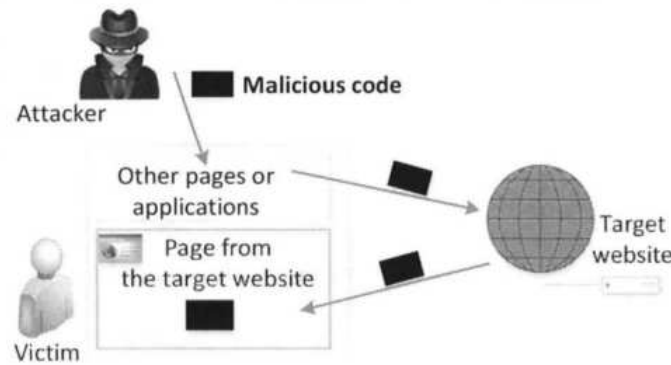


Figure 9: Non-persistent XSS attack

b. Persistent XSS Attack

In the persistent XSS attacks instead, attackers can directly send the malicious code to the target website, which stores it in a persistent storage. If later on an unaware user accesses the website area in which the injected code is stored, their browsers execute it like normal code bringing to fruition the attack. Obviously, even in this case, the attack is successful if no input sanitizing methods are applied. The experiment treated in the this report is based on precisely this second type of attack. In Figure 10 you can see a representation of this attack.

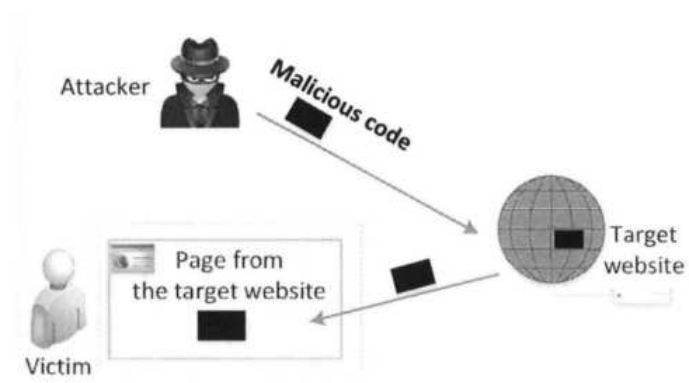


Figure 10: Persistent XSS attack

When talking about cross-site attack is important to pay attention in not confusing the Cross-Site Scripting (XSS) with the **Cross-Site Request Forgery (CSRF)** [4]. In CSRF, the forged requests are actually real cross-site requests because the malicious website page (the third-part) sends a the forged request to another website on behalf of the victim, while in XSS they are in reality "same-site" requests. A CSRF can be performed only if the victim has an active session, authentication included, in the target website and during this time the user visits the malicious page. The "evil" website can now forge a request for the target using victim's cookies and information.

XSS possible damages

XSS attacks, like others, if successfully performed, can lead to different kind of damages depending on the attacker objective.

Stealing information: With the malicious code an attacker can steal user's personal data, session cookies, web application data stored locally and so on.

Spoofing request: Using JavaScript, and in particular Ajax [5], HTTP requests can be forged that allow hacker to operate on behalf of the victim (adding fake information, editing existing ones, etc.).

Web defacing: The injected JavaScript can use the DOM [6] APIs to access and modify the elements that make up the structure of the web page (DOM nodes) and so it gives to the attacker the power to modify the aspect and the behaviour of the whole page.

Countermeasures

From what said before emerges that the weakness of a web application is related to what the browser reads as source code. In fact, while to distinguish between page code and pure data is what the HTML parser is built for, it is not able to detect if the code read belongs to the original page or if it was inserted from the outside. To counter this problem two approaches have been thought out. The first one analyses the input to search for not allowed patterns (like the presence of script tag in variables or parameters), while the other one is based on understanding if the code comes from a reliable source, the web application developer, or from an unknown source; it can be considered as a sort of access control.

Famous XSS attack

The story of Cross-Site Scripting attacks starts in 1999 when the first reports of sites in which script and image tags were being injected into html pages without the approval of the owner. Some of these attacks were carried out through links, which were then opened by unsuspecting victims, who then had their user's cookies stolen and sent to third parties. The majority of these attacks were Reflect XSS but in only a few years the dominant type became the Persistent XSS thanks to the social networks advent [7].

Samy worm

In 2005, Samy Kamkar discovered a breach in the way in which browsers processed the input received from users through forms. Consequently he realized that through that flaw he was able to make browsers run scripts not belonging to the website. The virus was released on *MySpace* but as just said it does not exploit *MySpace* vulnerability but a problem in the way browsers execute JavaScript code [8]. The attack consists of a piece of self-replicating JavaScript code injected in Samy's profile that creates a copy of itself in the pages of every user that visited Samy's page. Due to his self-propagating behaviour and the way in which this attack is performed, experts classified it as cross-site scripting worm (XSS worm) and called it "Samy worm" (also known as JS.Spacehero). The worm itself was relatively harmless; it carried a payload that would display the string "but most of all, samy is my hero" on a victim's MySpace profile page as well as send Samy a friend request, but thanks to its propagation speed it has been acknowledged as the fastest-spreading virus of all time with one million infections in 20 hours [9]. From the most, Samy is considered the virus that changed the world of web security forever [10].

eBay

Between December 2015 and January 2016 eBay discovered a vulnerability in the parameter of the "url" that permits the redirection to the right eBay page. The absence of control on the value of this parameter opens the opportunities for hackers to build a fake web page, in every way the same as the eBay login page. In this way the victim is unaware of the falsehood of the page and inserting the credential they send them to the hackers instead of the real website. The attacker is so able to control the profile of the unfortunate, steal their payment details and so on. This is an example of phishing attack

exploited through an XSS vulnerability [11].

British Airways

A cross-site scripting vulnerability was also discovered by British Airways between August and September 2018. The breach allowed hackers to steal the credit card details and personal information of the victims. The hacker group known as "Magecart" injected a malicious JavaScript library, known as Feedify, into an unsecured payment form in the airline's website, so that, when users submitted the form, the confidential information were sent to a server owned by the criminals. The company noted that around 380,000 customers could have been affected, and the economic injury caused by this XSS attack was huge [12].

Fortnite

In January 2019 also Fortnite, the popular online video game by Epic Games, had to deal with a cyber-attack. Through a retired non protected page (an old web page owned by Epic Games) attackers injected a link that would execute rogue JavaScript code in a user's browsers when visited. To bring people to this site hackers found a state parameter in the request sent by the real authentication page that could be manipulated to redirect to the old vulnerable website. Once reached, the malicious JavaScript payload is executed through the XSS vulnerability. The injected code intercepts the single-sign on authentication token and sends it to a server controlled by the attackers, allowing them to obtain the control of user's account [13].

Nowadays, Cross-Site scripting (or XSS) is often considered one of the most common bugs that are harvested by cybercriminals to compromise an organization's networks or systems. And history proves it; attackers have utilized this vulnerability in various cyberattacks, causing damages of millions to those organizations, and in turn these organizations are spending a fortune to try and stop these attackers [14].

2.1.2 SQL Injection attack

Since the amount of data managed by web applications is by now massive, it was necessary to use databases to optimize their storage and use. As a consequence, the web applications' implementation need to include a procedure to create and send an SQL statement to the database that, after having execute it, returns the result of the execution back to the web application. The SQL statement usually is composed by a fixed part defined by the web application developer and a "parametric" part that consist of data received by the final user. This "user-dependent" second part, if not correctly managed, could permit malicious users to request data which, normally, they're not allowed to access. In fact, if the data received from the user can be interpreted as SQL code and no sanification procedure are applied, the web application can become victim of SQL Injection. This type of vulnerability is considered one of the most common in web applications and this the reason why is important to help programmers in preventing its rising as much as possible. Figure 11 illustrates the typical structure of a web application. As can be seen, the user does not interact directly with the database but through the functionalities provided by the web server. This intermediary is responsible for receiving instructions from the browser and returning the corresponding response, and for interacting with the database which is in charge of storing the contents. The crucial point of the communication is how the web application server manages the data received from the user. [15]

Communication methods

Communication between user and server is through the exchange of HTTP requests and responses. HTTP requests use GET and POST methods to attach the user-defined data to the message. The most common way to exploit this behaviour is filling the forms fields provided by the application. The web application server obtains in this way the data that are going to complete the predefined query that will be executed by the database to provide the required information. The result of query execution is then returned to the web application server, which will take care of making it available to the requesting user.

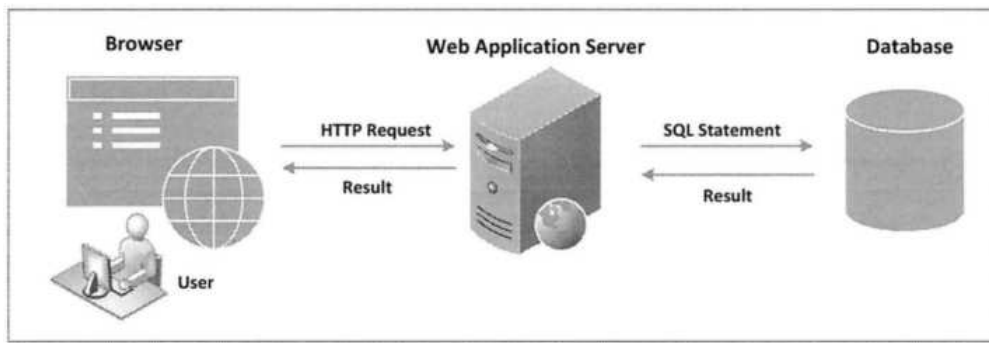


Figure 11: Web application architecture scheme

SQL Injection launch

To better understand how an SQL Injection attack is performed, the process will now be shown from a more illustrative point of view. Consider, for example, the SQL statement below.

```

SELECT name, surname, card_number
FROM Person
WHERE username=' _____ ' AND password=' _____ '

```

The SQL statement template is defined by the web application developer while the blank section will be filled with the data received from the user. Here is where a vulnerability point emerges. If the completion consist of a string formatted following SQL language rules and construct the behaviour of the query can radically change. If the user for examples inserts the string "P0001 ' #' " in the `username` field, the query returns `name`, `surname` and `card_number` with no regard in what the `password` field in the `WHERE` clause contains. In fact, with the apostrophe (') closes the `username` field writing while the remaining part of the `WHERE` clause is commented through the usage of the pound sign (#) which is the SQL symbol to denote the starting point of a comment. Below you can see how the query is actually executed.

```

SELECT name, surname, card_number
FROM Person
WHERE username=' P0001 '

```

In this way if a malicious user knows the `username` corresponding to another person's profile he can gain access its `card_number` with no difficulties.

Another alarming situation can be generated if the string sent from the user is something like: "anything' OR 1=1 #". Even in this case the password check is bypassed through the comment identification symbol "#". The difference is that the "OR 1=1" WHERE condition is always returning true and so, independently from the first WHERE condition ("anything"), the query returns `name`, `surname` and `card_number` of all the users stored in the `Person` table. Below is shown the resulting query.

```
SELECT name, surname, card_number
FROM Person
WHERE username='anything' OR 1=1 #
```

As last examples, the two following query will show respectively a case in which through the updating operation an hacker could modify database data to increase or decrease someone's bank account while the other deletes the whole database causing huge damage to users and service provider.

```
UPDATE Person
SET new_pw= ' _____ '
WHERE username= ' _____ ' AND password= ' _____ '
```

This is the case in which is requested a password change/reset. If the applicant knows the name of the database table's field associated with the value of the bank account, it can change it to increase or decrease its value. Below you can see how to set the parameters to play out this attack (is supposed that the field containing the bank account value is called "bank_account").

`username` → "P001' #" : target's username, the pound sign is to comment the password field.

`password` → "anything" : will be commented.

`new_pw` → "new_password', bank_account=(amount) #": the values the attacker wants to set (the pound sign at the end prevent other parameters to interfere with the injection).

```
SELECT name, surname, card_number
FROM Person
WHERE username='anything'; DROP DATABASE exampledb;
```

In the last case the string received corresponds to "anything "; DROP DATABASE exampledb;". Here you can see that using the semicolon (;) it has been possible to concatenate two different SQL statement: the usual `SELECT` and the `DROP DATABASE` which deletes the whole database. The possibility to concatenate statement permits to the attacker to perform any kind of operation on the database like as if he were the owner.

SQL Injection possible damages

SQL Injection attacks, like others, if successfully performed, can lead to different kind of damages depending on the attacker objective [16].

Stealing information: Obtain the access to private information contained in the database without the owner's authorization.

Data fabrication an modification: Fake data can be inserted in the databases mining the authenticity of the information provided. It also comprehends the modification of stored data.

Service interruption: Problems in the database structure can compromise the correct behaviour of the web application.

Countermeasures

Like said for the Cross-Site Scripting, the weakness of the the system lies in how inputs are interpreted. In fact, even in this case if the string received from the user is interpreted as pure string or at least measures to detect any "injection-related" pattern the threat can be averted or at least mitigated. To achieve this goal input sanitization procedure can be put in place, like filtering the received data searching for specific attack pattern or, alternatively, by sending the query to the database so that no ambiguity are present between what should be executed and the pure data (parametrized query) [17].

2.2 Compiler level security

Before beginning the presentation of the basic concepts on which the project is built, it is necessary to make a clarification. Throughout the discussion the term semantics will be used several times, but with a different meaning in respect to the usual one. In this case, in fact, when the word semantics is used to talk about an object, it refers to all those characteristics that can describe its nature (type, belonging class, etc) and the relationships it has with other objects. Instead of point out the relation between the syntax and the calculus model, and so the intrinsic meaning of a language word, is used to identify the set of characteristics which describes a specific program element.

Now that the proper premises have been made, the first tool that needs to be presented is the compiler. The compiler is in charge of translating the code belonging to a programming language into another, and in most cases, this process starts from a high-level programming language (human-readable) to a lower level one which can be executed by the machine. To be more specific the compiler carries out an essential function in building a detailed model of the coded program, after having validated its syntax and semantics. This model in fact contains a lot of useful information about the code and its structure which are not visible during the programming phase. The model in fact grants access to the so called *syntax tree* which gives a representation of how the program instructions are organized, their hierarchical expansion and their role inside the tree. Moreover, it also includes the semantic information in terms of relations between different program elements and in objects characteristics. It goes without saying that being able to access this intermediate environment between programming language and machine-code can bring big benefits during the code development phase. In fact, being able to access this model opens the doors to the world of *code static analysis*. This practice permits to understand what is being coded, to point out which instructions may lead to the rising of an exception, to detect eventual programming guideline violation and to make real time suggestions appear. The desire to achieve this kind of functionality led to the creation of tools that would ensure access to the models created by the compilers. Therefore, projects such as ReShaper from JetBrains, Babel from Open Collective, SonarQube from SonarSource and Roslyn from Microsoft, the one on which this project is based, were born.

In Figure 12 are shown the functional areas which compose the compilation process. In

the first one, source code is tokenized and parsed to provide a syntax pertinent to the language grammar. The second, the so called declaration phase, extrapolates the named symbols from the source text and from the imported metadata. The following binding phase links the symbols with the corresponding identifier. The last one, the emit phase, is in charge of preparing the actual assembly which carries all the required information [18].

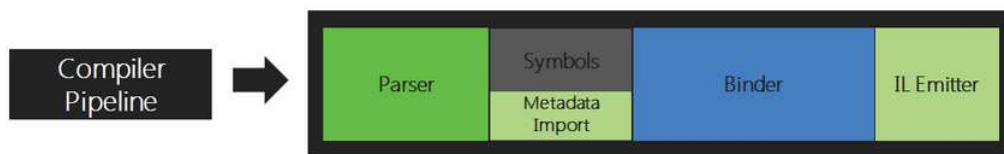


Figure 12: Standard compiler pipeline

This is the framework on which the **.NET Compiler Platform SDK ("Roslyn APIs")** created for C# (CSharp) and Visual Basic languages is based. This new tool provides an API layer that, reflecting the traditional compiler architecture, grants access to the above-mentioned model.

2.2.1 .NET Compiler Platform API ("Roslyn")

The **.NET Compiler Platform SDK ("Roslyn APIs")** is born as a tool to manage the model built from the compilation of the source code. This mean provides dedicated APIs to access the data contained in each pipeline phase. It transforms the compilation process in a platform available to consulting and open to modification. It is precisely because of the possibilities offered by this architecture that code analyzer and code fixer have been able to emerge.

After this little introductory part, it is time to look a little more in depth at what this API consists of and what it allows to do. As said before the **.NET Compiler Platform SDK ("Roslyn APIs")** mirrors the behaviour of C# and Visual Basic compilers adding to them different API layers which ensure the ability to perform real time code-related operations. This layers are called: compiler APIs, diagnostic APIs, scripting APIs, and workspaces APIs. All this APIs are allowed to dispose of the tools exposed by the various compiler pipeline phases like the syntax tree provided by the parsing phase, the hierarchical symbol table supplied by the declaration phase, the result of the compiler's semantic analysis produced by the binding phase and the IL (Intermediate language) byte code

obtained by the emit phase.

Compiler APIs

This layer contains all the object models produced by every compiler pipeline phase with the addition of the assembly references, the compiler options and the source code files obtained from a single invocation of the compiler. This kind of APIs require high compliance with the programming language they refer to. This is the reason behind the need of a dedicated API for C# and one for Visual Basic. Figure 13 gives an idea about the relations between standard compiler pipeline and the Compiler API layer.

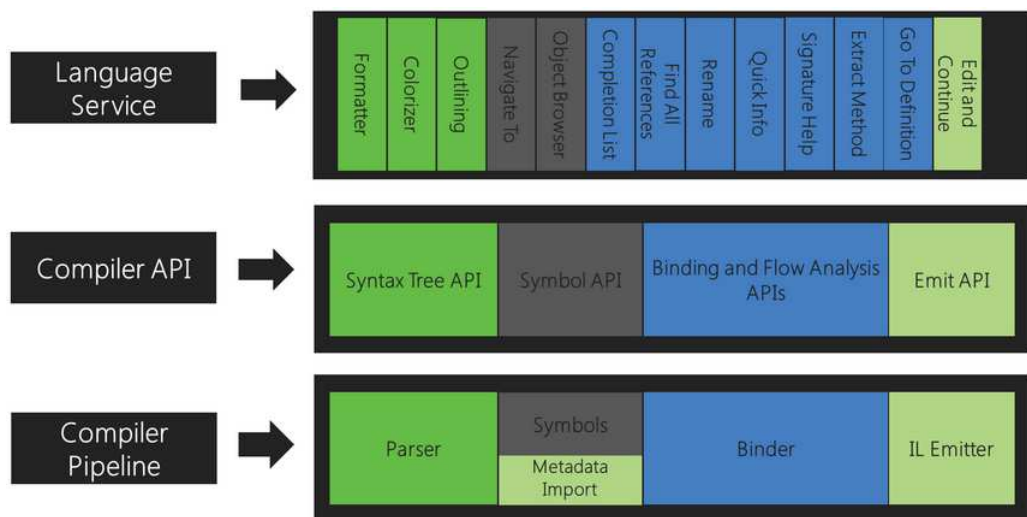


Figure 13: Standard compiler pipeline and Compiler APIs matching

Diagnostic APIs

Diagnostic information includes syntax, semantic and definite assignment errors, not optimal code structure warnings and general informational report. This kind of data are produced by the Compiler API and made available to the user through the Diagnostic API which also enables the creation of user-defined notification and code-fixing suggestion. Graphically, diagnostics, appears as coloured underlines and program tooltips. Figure 14 gives an example of how diagnostics appear.

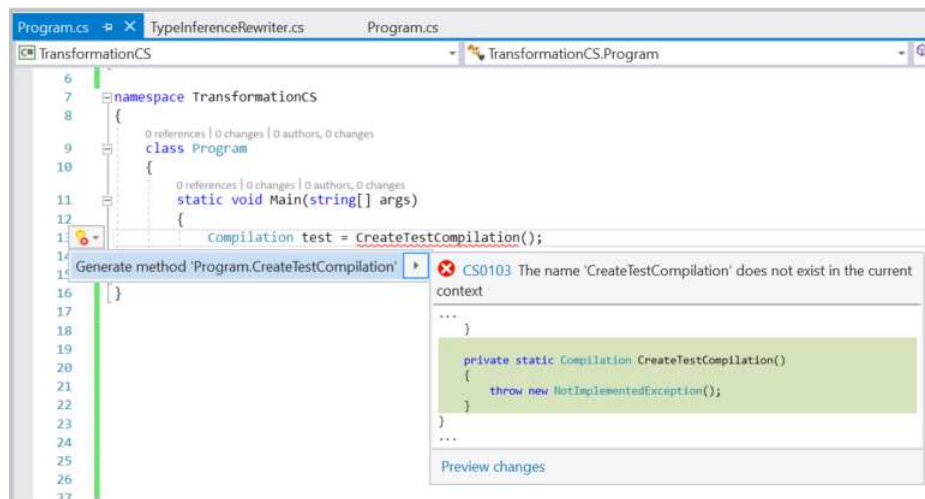


Figure 14: Underline and tooltip example

Scripting APIs

To accumulate a runtime execution context and to execute targeted code sections, the hosting and scripting APIs have been realized. Based on these APIs tools to interactively execute code while writing have been made.

Workspaces APIs

The Workspace API grants access to the entire solution grouping all projects object models into a single one. In this way the operations of analysis and refactoring can be executed without parsing files, configuring the compilation options or managing the dependencies between projects.

All this technology is what stands upstream the creation of personalized analyzer and code-fixes. In fact, it serves as entry point for the "black-box" that, before its creation, was the compilers world [19].

2.3 Visual Studio Analyzers and code-fixes

One of the powerful features that the .NET Compiler Platform SDK offers is the capability to build diagnostic analyzers. These tools permit to create an interactive communication between programmers and the code they are writing. It provides users with visual hint about the code they are writing in term of errors, warning, or quality sugges-

tions. Therefore, analyzers can be seen as compilers extensions which enforce programmer in respecting specific coding measures in order to improve the quality, the correctness and the ease of maintenance of the code. In addition to this, if properly designed, personalized analyzers can be configured to focus their attention on specific issues to which a program may be exposed. Least but not last, they can also detect patterns that do not satisfy eventual code style and design requirements.

The analyzers, being built over the Roslyn compiler API, have the ability to access the object models that the latter provides i.e. syntax tree nodes, code symbols, code blocks, the whole solution structure and so on. This allow them to intercept changes in different units of code, to communicate any irregularities to the user and to arrange targeted actions to manage the reporting. All the information gathered from the code are inserted in the list of the compiler diagnostics to provide the user with detailed explanation about what was discovered [20].

Severity levels

Talking about Roslyn, reports can be classified according to the impact that the affected code will have on the correct execution of the program:

Errors: report which indicates an issues that prevent code from compiling or that generates a runtime error (red squiggles under the code).

Warnings: report which indicates an issues that do not prevent the code to be compiled, but that may affect the efficiency of the code (green squiggles under the code).

Suggestion: report which provides information that may be considered useful for the program.

Figure 15 gives an example on how this reports appear in the code.

```
static void Main(string[] args)
{
}

int Add(int i1, int i2)
{
    return i1 + i2;
}
```

Figure 15: Severity levels example

2.3.1 Code static analysis

Code static analysis is a practice that verifies the compliance of source code with predefined rules which is performed at compile time. In most cases, this solution is adopted with the goal of enforcing specific programming standards such as to improve the readability of the code and lighten the review procedures by reducing the number of errors made. The adherence to predefined standards turn out to be useful even for the other programmers which may be involved in the project or that will take over it. The final result is a overall improvement of the whole design and coding process.

2.3.2 Roslyn based analyzers features

As mentioned so far, the role of the diagnostic analyzer is to scan the code for irregularities. After the obvious phase of code analysis, which identifies procedures that may lead to the violation of imposed standards, compilation errors, or inefficiencies in the solution, a second phase can be planned, in which developing the procedures that will propose a solution to the previously founded issues.

The first of the two can be considered as a **diagnostic** step which uses the syntax and semantics of the code for the detection operations, while the second one uses the diagnostics previously produced to suggest possible solutions, and this is why it can be referenced as **code fix**.

It's now time to present the APIs' elements, the Visual Studio functionalities and the Roslyn tools involved in the project.

Syntax Tree

The **Syntax Tree** is a structure, accessible through the Compiler API, which organizes the various syntactical and lexical elements of the source code in a hierarchical structure

composed by nodes and leafs. This is an immutable entity, in fact the syntax tree of a program cannot be directly accessed and modified except through the appropriate tools that will be presented later on in the thesis. The motivation behind this feature lies in making the tree resistant to the inconsistency that can be generated by simultaneous accesses to it. The immutability of the syntax tree also permits to have a snapshot of the actual state of the code. In this way, during the fixing phase, a preview of the correction can be graphically shown (Figure 16 shows a correction preview).

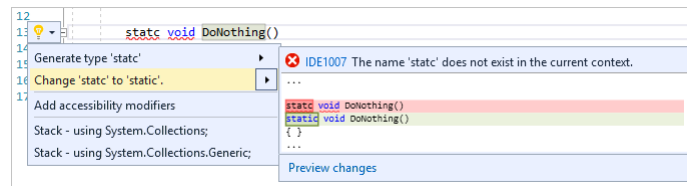


Figure 16: Example of code fix preview

The **Syntax Visualizer** provided by Visual Studio exhibits the representation of the tree in both, a schematized way and also in a graphical one. The schematized view in particular brings with it a section containing the properties which characterized each syntactical element. On the other side the graphical representation is given by a **Directed Syntax Graph**, highlights the hierarchical relation between the elements and their nature. Figure 17 and Figure 18 show the two kinds of representation.

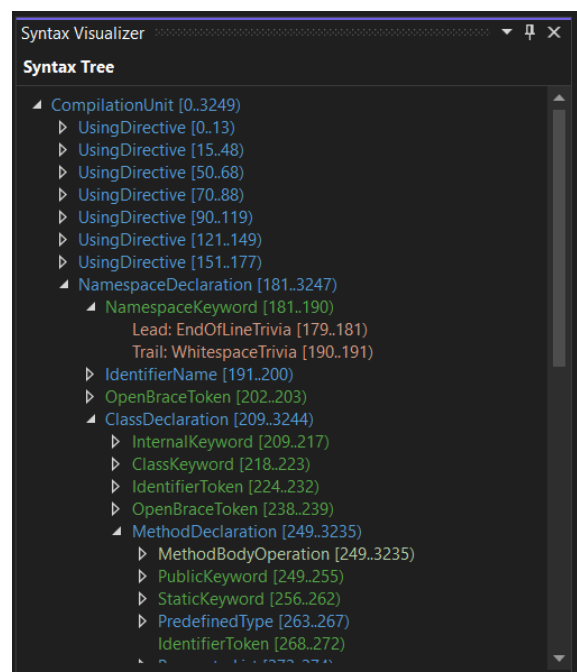


Figure 17: Syntax tree example

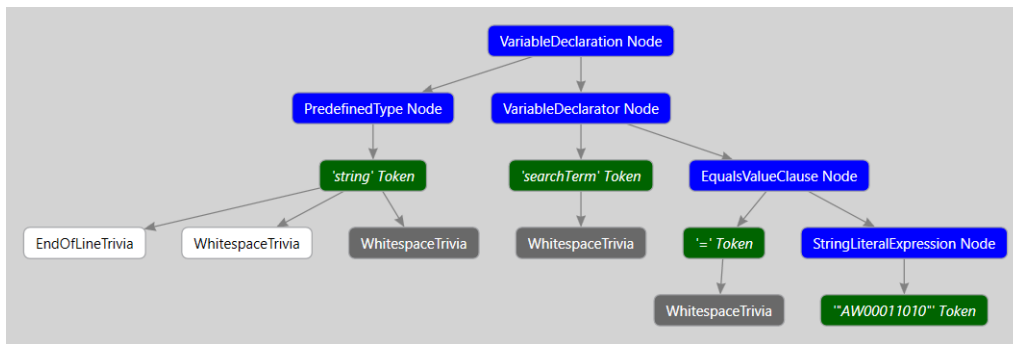


Figure 18: Example of graphical representation of a syntax tree

Syntax Elements

Syntax elements are the entities which compose the syntax tree. They have a strict relation with the part of code they refer to, in fact, their purpose consist of giving a deeper comprehension of of the written code pointing out its role inside the project and the specific properties which characterized it. Depending on their nature, syntax elements can be divided in three groups: nodes, tokens and trivia. As can be seen from the previous figures, nodes, tokens and trivia are respectively represented with the colors blue, green and red.

Syntax nodes

Nodes are the syntax elements which form the tree backbone. They identify function calls, assignments, declarations, code blocks, statements, and so on. An essential feature is the fact that they are all non-terminal tree elements and so they must have other nodes or token as children. Another important trait to be emphasized is that they are characterized by a *type* and a *kind*. The type represents the subclass of

Microsoft.CodeAnalysis.SyntaxNode which the node belongs to. As subclass it possesses all the properties and methods inherited from the *SyntaxNode* class plus the dedicated ones specific for what they represents. While the type property identifies the type of a node, the kind property denotes the type of the syntactic element to which it refers (tree point of view versus syntactic point of view). To be more clear, a type can be associated with many kinds but a kind is linked with only one type. Figure 19 gives an idea about the properties associated with a node.

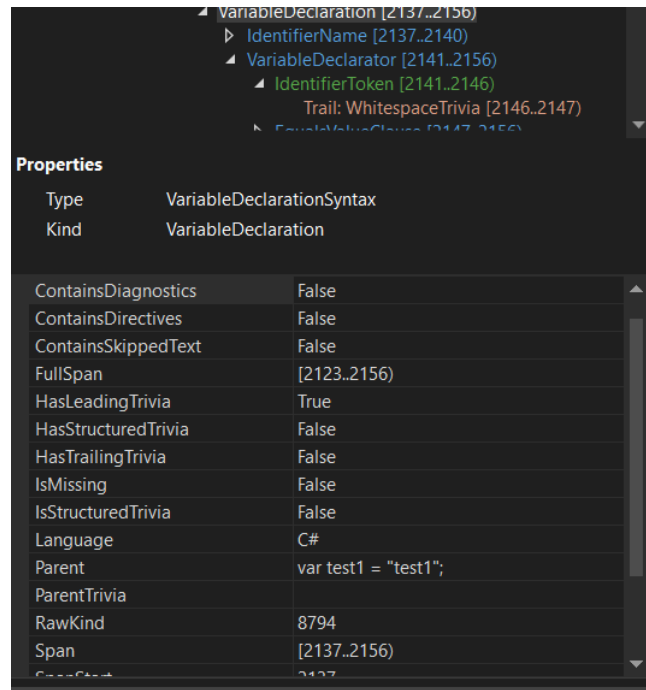


Figure 19: Node properties example

Syntax tokens

Syntax tokens correspond to the leaf of the tree, in other words the terminals of the language grammar. Example of them are punctuation signs literals, identifiers, keywords, and so on. As well as nodes, tokens are syntactically categorized through the *type* and *kind* properties. What differentiates them from nodes is that the value of the *type* property is the same for all the tokens (**SyntaxToken**). Figure 20 shows the type and kind properties for a token.

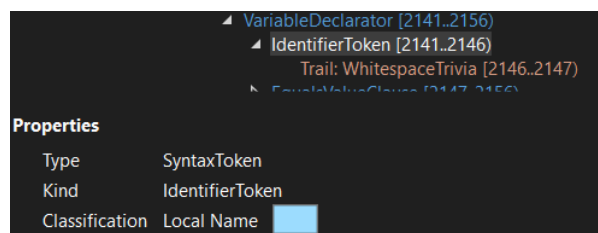


Figure 20: Token type and kind example

Syntax trivia

Syntax Trivia are source text items which are not fundamental for the correct interpretation of the program. Examples of them are comments, white spaces and preprocessor directives. In terms of *type* and *kind* properties they are organized like tokens with the

difference that for all the trivia their membership class is the **SyntaxTrivia**.

Semantic Model

As previously said, used in this context, the term semantic indicates a set of features that provides additional information about the nature of the symbol which the code element refers to. The tools which allow to pull out this kind of data are the **Symbol API** and the Semantic API contained in the Compiler API. *Symbols* are a unique representation of a code element (like properties, local variables,...), and thanks to these two APIs the user is able to obtain information about the name of the class to which they belong, the namespace containing them, the project within which is located, and many others. During the development of the project, to access the semantic model of a code element, the property **SemanticModel** of the *context* object, which belongs to the **SyntaxNodeAnalysisContext** class, was used.

2.3.3 Code fixes

Through the diagnostics provided by analyzers, refactoring measures can be prepared to modify the code in order to solve the detected issues. This procedure is called **code fixing**. This kind of syntax transformation can be performed exploiting one of the following strategies.

Factory methods This kind of code refactoring uses the methods belonging to the **SyntaxFactory** class to create from scratch syntax objects whose class is derived from **SyntaxNode**. These newly created nodes will then either go on to replace existing nodes or fit into specific positions within the syntactic tree. The **SyntaxFactory** class provides dedicated methods for creating nodes, tokens and trivia according to their kind. For example, if you want to create a node of **ExpressionStatement** kind, **SyntaxFactory** makes available the **SyntaxFactory.ExpressionStatement** method which returns a new **ExpressionStatementSyntax** node. **ExpressionStatementSyntax** represents the type property associated with the **ExpressionStatement** kind. Due to the fact that the syntax tree is an immutable entity, the new node cannot be inserted only modifying its structure. This is the reason behind the usage of the **ReplaceNode** method which creates a fresh new instance of the syntax tree with the new node placed in the correct position, specifying the node that will replace, and updating recursively the links of the nodes hierarchy. The same behaviour, but for the insertion operation, can be obtained

using the **InsertNodesAfter** o **InsertNodesBehind** functions.

Rewriters The class **CSharpSyntaxRewriter** enables the performing of multiple transformation on the syntax tree. Unlike **ReplaceNode**, which modifies one node at a time and then returns the new tree, it allows changes to be accumulated during the tree traversal and then applied when finished. Prerequisite is that the modified nodes are all of the same type. In fact, to implement this solution, is necessary to extend the **CSharpSyntaxRewriter** class and to override the abstract method which works with the chosen syntax node type. In this way, during the syntax tree navigation, each time a nodes matches the chosen type, it will be refactored as determined by the override method and only at the end of the crossing the new syntax tree is generated.

For what concerns the project, only the first one of the two solutions is applied.

2.4 Language utilities

Attributes

Attributes in *C#* permit to associate metadata or declarative information with the code they refers to. The target of an attribute can be only a code element belonging to the following list: assembly, module, field (class or struct field), event, method, param, property, return and types (struct, class, interface, enum, delegate). Attributes have the following properties:

- They add *metadata* to the program, information about the defined types to which they refer to.
- The same element can be associated with one or more attributes.
- Attributes can be built using arguments as is also usually done for methods and properties.
- To examine attributes metadata a special technique must be used, the so called *Reflection*.
- In addition to the attributes provided by .NET, it's also possible to create custom ones.

Reflection

Reflection provides objects of type **Type** which contains descriptive information about the accessed entity. If the reflection is applied to an attribute, it grants access to all its functionalities and contents. In case of custom attributes the key method that permits to exploit the reflection is **GetCustomAttributes**. This is the function that needs to be called when no compilation environment access is provided. On contrary, if the compilation environment is already accessible, for example through the semantic model, reflection can be deployed through the **GetAttribute** method owned by symbols.

Extension Methods

Extension methods are methods that allow you to add extra functionalities/methods to an existing type without having to inherit the type or write a wrapper around it. These methods provide a powerful way of writing additional functionalities with minimal effort and code.

```
1 public static class ExtensionMethods
2 {
3     public static string StringProcessing(this string s, /*...*/)
4     {
5         // Instructions
6     }
7 }
```

The above code shows a simple example of how an extension method can be defined. There are three aspects which are fundamentals to define an extension method:

- The **static** keyword in the class declaration. Extension methods can only be created inside static classes in .NET. The static modifier is mostly used when data, behaviour of a class and the elements within it, do not depend by the identity of the object used to call them.
- The **static** keyword in the method declaration. Static classes allow only static members.
- The **this** keyword associated with the first parameter of the function. The type of the first parameter matches the object type on which the extension method was called. The **this** keyword makes the new functions seeming like a call to a method belonging to the object class. The object from which the call comes becomes the first parameter of the extension method.

3 Security Enhancer implementation

This chapter will describe the work behind the creation of the syntax analyzer, exposing the design choices and presenting the project structure. Particular attention will be focused on the implementation aspects of the solution, that is, the code that makes up the various components of the analyzer.

The analyzer at issue is called **SecurityEnhancer** and it has been realized using the Visual Studio Community 2022 IDE. This tool simplifies the integration of the .NET Compiler Platform SDK package which hosts the analyzer project. The programming language which characterizes the implementation is the C# language.

The solution is designed to be deployed on a client-server architecture in which the server will host and execute the actual analyzer code while the clients, i.e., the machines of programmers, access it without having to add extensions to their program. This will create a unified environment for all users and the computational burdens will be decentralized to a dedicated machine thus easing the load on individual devices.

3.1 Project structure

The project structure follows the one proposed by the template "Analyzer with code fix (.NET Standard)" provided by Visual Studio. The complete solution is build up by five projects (Figure 21 shows the workspace organization):

- **Analyzer** contains the analyzer code and so the detection operations and the creation of the diagnostics, it may contain one or more analyzers;
- **Analyzer.CodeFixes** contains the code which, starting from the diagnostics, performs the code refactoring operations, it may contain one or more code fixes;
- **Analyzer.Package** used to create the NuGet package that makes the analyzer

usable for the users community;

- **Analyzer.Test** corresponds to the test unity, grants the possibility to verify the correct functioning of analyzers and code fixes;
- **Analyzer.Vsix** corresponds to the startup project, it opens a fresh new instance of Visual Studio in which the analyzer is uploaded to emulate the actual behaviour assumed by the analyzer once installed by users;

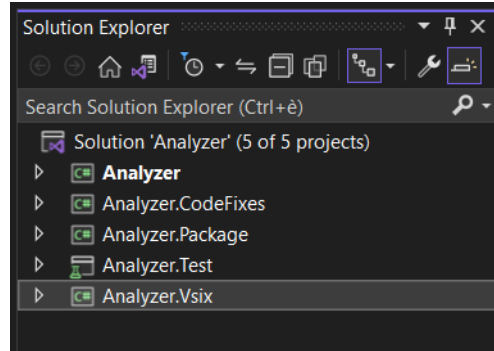


Figure 21: Standard project structure for a code analyzer

The creation of the NuGet package makes shareable the created analyzer as extension. NuGet is in fact the platform through which developers can share their personalized Visual Studio packages [21].

Following the above mentioned template, the SecurityEnhancer structure is organized as represented in Figure 22 and in particular:

- The analyzer project contains the files of four different diagnostic analyzers which can be divided in two categories: the **XSSAnalyzers** and **SQLAnalyzers**.
- The code fixes project mirrors the previous structure with four different code fixes files which can be grouped in two different categories: one for the code refactoring measures against XSS vulnerability and the other for the SQL injection ones.

Only one analyzer was needed to counter the XSS vulnerability, as only one type of code element capable of carrying this vulnerability was identified. On the other side, three code elements vulnerable to SQL injection have been identified, hence the decision to create three diagnostic analyzers. The choice to separate the two kind of analyzers, and in the same way the two code fixes, was made to provide a tidy, understandable and modular solution. In fact, thanks to this subdivision, the user can even run just the analyzer he or she needs.

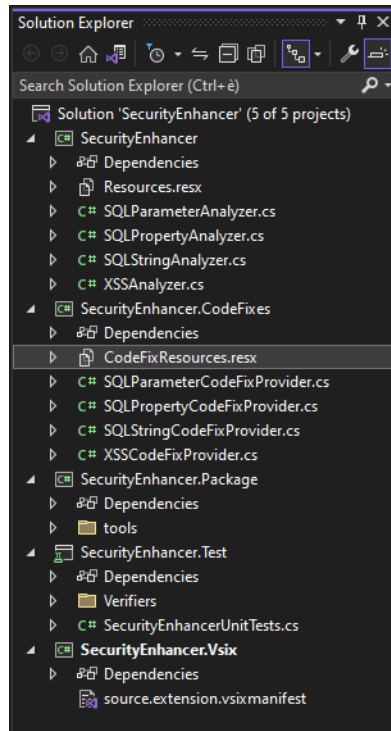


Figure 22: Structure of the proposed solution

In addition to the architecture provided by the Visual Studio template, it was planned to create in a separate project an SDK (Software Development Kit), called **SecurityEnhancerSDK**, which provides programmers with predefined tools to integrate the analyzer solution with standardized objects and update-tolerant procedures.

3.2 SecurityEnhancer SDK

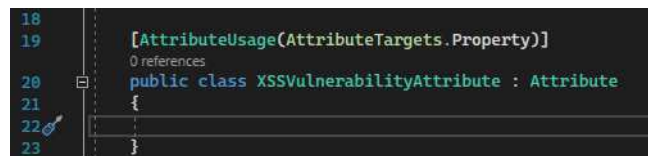
The SecurityEnhancer SDK is essentially a library which comes with the SecurityEnhancer project that defines predefined functions and code elements for its proper execution. It can be seen as a utility class which can be enriched with new supporting tools depending on the design choice of the "main" solution. At the same time, it makes possible to simplify the procedures for updating any external libraries used by the analyzers.

To be more precise this library contains the definition of a *custom attribute* and class which defines the *Extension Methods* that will activate the sanitize operations.

Custom attribute

The attribute is called **XSSVulnerabilityAttribute** and is designed to mark all those properties that, without appropriate countermeasures, could convey a script injection. It works like a reminder for programmers: when an expression is assigned to a property tagged with this attribute, the expression at issue may need to be sanitized.

The attribute shown in Figure 23 is a *custom attribute* as it is user-defined and not native of the .NET ones. To create a custom attribute is necessary to declare a class derived from `System.Attribute` and to define the target kind of elements with which the attribute can be associated. In the present case this operation is carried out through the instruction `[AttributeUsage(AttributeTarget.Property)]` which denotes that only the code element type "property" can be tagged with the custom attribute `XSSVulnerabilityAttribute`.



```

18 [AttributeUsage(AttributeTargets.Property)]
19 0 references
20 public class XSSVulnerabilityAttribute : Attribute
21 {
22
23 }

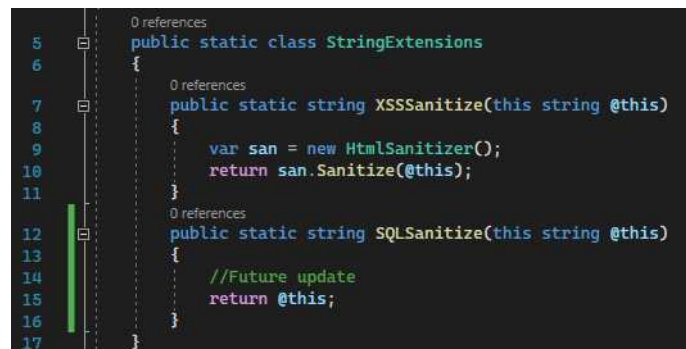
```

Figure 23: Attribute creation example

Extension methods

For what concerns the extension methods, a dedicated class has been designed to collect the functions which will be called from the code fixes. The class `StringExtensions` contains, for example, the method `XSSSanitize` which, being an extension method, invokes the string sanitization function specified within it on the calling object.

This function is called `Sanitize` and it comes from an imported NuGet package called `HtmlSanitizer`, a dedicated tool to clean HTML from constructs that can be used for cross-site scripting. Figure 24 shows the extension methods used.



```

5 0 references
6 public static class StringExtensions
7 {
8 0 references
9 public static string XSSSanitize(this string @this)
10 {
11     var san = new HtmlSanitizer();
12     return san.Sanitize(@this);
13 }
14 0 references
15 public static string SQLSanitize(this string @this)
16 {
17     //Future update
18     return @this;
19 }
20 }

```

Figure 24: Extension methods used

3.3 SecurityEnhancer analyzers

It is time to get into the meat of the thesis through the presentation of the projects that make up the analyzers and the code fixes. Two section will be exposed, one for the analyzers and code fixes against XSS vulnerability and one for the SQL injection ones.

3.3.1 XSS analyzer

Diagnostic analyzer

Each analyzer features the following functionalities:

- **Register actions.** With action is meant a change in the source code. Whenever an action is detected, the analyzer is triggered and the check on violations defined inside of it is performed.
- **Create diagnostics.** When a violation is detected a report is created to notify it. The report consist of a diagnostic object which groups information about the alert, as determined by the analyzer.

The creation of a diagnostic analyzer starts with the declaration of a new class derived from **DiagnosticAnalyzer** and the association of the [**DiagnosticAnalyzer**] attribute that accepts as a parameter the language on which it operates. The following snippet shows how the analyzer is declared.

```
1 [DiagnosticAnalyzer(LanguageNames.CSharp)]
2 public class XSSAnalyzer : DiagnosticAnalyzer
3 {
```

Next, it is necessary to define the description produced as diagnostics. The object appointed for this purpose is the **DiagnosticDescriptor**, which requires the following parameters to perform its function:

- **id:** `string` parameter which uniquely identifies a diagnostics;
- **title:** `string` or `LocalizedString` parameter which specifies the title of the alert message;

- **messageFormat**: `string` or `LocalizedString` parameter which defines the parametric message that will be displayed in the violation report;
- **category**: `string` which allows to give a categorization to the diagnostic;
- **defaultSeverity**: `DiagnosticSeverity` object kind that classifies the severity of the violation, in the treated cases it will be set to "warning";
- **isEnabledByDefault**: `boolean` that indicates if the diagnostic to which it refers, is enabled from the analyzer startup;
- optional parameters

The parameters instantiation is made through a resource file which organizes the contents like a dictionary. This solution simplifies the editing procedures by avoiding the need to search the code for where the assignment is made.

After that, overriding the abstract property **SupportedDiagnostics**, a set of descriptors is produced for the diagnostics that this analyzer is capable of producing. This part of the code shows its usefulness in the testing project, however this environment was not used in the experiment since the results were verified through the VSIX project.

The below snippet of code shows how the diagnostics description is built.

```

1 public const string DiagnosticId = "XSS01";
2
3 private static readonly LocalizableString Title = new
  LocalizableResourceString(nameof(Resources.XSSAnalyzerTitle), Resources.ResourceManager,
  typeof(Resources));
4 private static readonly LocalizableString MessageFormat = new
  LocalizableResourceString(nameof(Resources.XSSAnalyzerMessageFormat), Resources.ResourceManager,
  typeof(Resources));
5 private static readonly LocalizableString Description = new
  LocalizableResourceString(nameof(Resources.XSSAnalyzerDescription), Resources.ResourceManager,
  typeof(Resources));
6 private const string Category = "Input validation and representation";
7
8 private static readonly DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId, Title,
  MessageFormat, Category, DiagnosticSeverity.Warning, isEnabledByDefault: true, description: Description);
9
10 public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get { return
  ImmutableArray.Create(Rule); } }

```

Now that the diagnostics is ready to be produced, it's time to have a look at the process which led to its creation.

As said before, a diagnostic report should be generated when an action on the code results

in a violation. The method which allows to register code actions is called **Initialize**. This function uses an **AnalysisContext** kind object as a "collector" for the detected actions. The method in charge of associate an action with the corresponding detection function is called `RegisterSyntaxNodeAction`. It requires as parameters: the name of the function that manages the code analysis operation and the syntax element kind of the code part which requires to be monitored.

In the cross-site scripting case the instructions vulnerable to injection are the assignment operation. In fact, if the entered data is not properly sanitized these variables can convey the insertion of the malicious script within a web application. In terms of syntax, an association operation is identified by the **SimpleMemberAccessExpression** kind. This is the reason behind the choice of this syntax kind as `RegisterSyntaxNodeAction` parameter as shown by the code below.

```
1 public override void Initialize(AnalysisContext context)
2 {
3     context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.None);
4     context.EnableConcurrentExecution();
5
6     // DONE: Detection of property that could require sanitification
7     context.RegisterSyntaxNodeAction(action: AnalyzeSyntax, syntaxKinds:
8 SyntaxKind.SimpleAssignmentExpression);
9 }
```

The function invoked by `RegisterSyntaxNodeAction` is called **AnalyzeSyntax** and receives as input parameter all the syntax nodes belonging to the **AssignmentExpressionSyntax** type, which is the class associated with **SimpleMemberAccessExpression** kind, through the `context` object. Using the property **Node** on the context element the assignment expression node and its subtree are now accessible.

Among all the nodes derived in this way, we must now identify those of interest to the analyzer, that is, those nodes that represent the assignment of a value to a property marked with the attribute **XSSVulnerabilityAttribute**. As a first step, it is necessary to obtain the semantic model of the node representing the property access. In this way, In this way it will be possible to access the list of its attributes and verify whether one of them matches with the one sought.

To do this, the semantic model was extracted from the `context` object via the **SemanticModel** property, and in parallel, the node corresponding to the left side of the equality was reached via the **Left** property of the assignment node.

```

1  var semanticModel = context.SemanticModel;
2
3  // The nodes that can be vehicle of XSS attack are the ones in which a value is
4  // assigned to a particular property
5  var syntaxExpression = (AssignmentExpressionSyntax)context.Node;
6
7  MemberAccessExpressionSyntax syntaxWord;
8
9  // The right side on the assignment is what can contain malicious code while
10 // the left one corresponds to the property that stores the value
11 var toBeSecure = syntaxExpression.Right;

```

Once this operations are done, the analyzer will have to obtain the symbol information about the property being accessed (reminder: `object.property`). Therefore, it will continue the subtree navigation starting from the last node obtained, which is a node of type `MemberAccessExpressionSyntax`, until it reaches the target element through the property `Name`.

Once reached, it can be used as a parameter for the function `GetSymbolInfo`, belonging to the object `semanticModel`, to obtain the information regarding its nature, such as class of membership, data type, and so on.

```

1  syntaxWord = syntaxExpression.Left as MemberAccessExpressionSyntax;
2
3  // In this case left and right side represent the object and its property
4  TypeInfo leftSideInfo = semanticModel.GetTypeInfo(syntaxWord.Expression);
5  SymbolInfo rightSideInfo = semanticModel.GetSymbolInfo(syntaxWord.Name);

```

The property attributes can now be accessed through the `GetAttributes` function that creates the list of attributes from which to search for the desired one.

In case of positive result, it is finally possible, after reaching the proper node in the syntax tree, to check whether the assigned expression (right side of the assignment) has already been sanitized or not.

If the answer to this question is negative, the diagnostics creation procedure are performed and the report about a possible XSS vulnerability point is produced.

The `Create` method generates the diagnostic message using the previously defined description rules and indicating the position in the code of the affected node (location). This

parameters are useful for the displaying of the visual aids, in facts the location permits the visualization of the squiggles under the detected instruction, while the description is used in the error list to explain the reporting. The `ReportDiagnostic` method, on the other side, makes effective the diagnostics and activates the visual alert on the code.

```
1  // Array of the attributes for the extracted symbol
2  ImmutableArray<AttributeData> symbolAttributes = rightSideInfo.Symbol.GetAttributes();
3
4  if (symbolAttributes != null)
5  {
6      foreach (var attribute in symbolAttributes)
7      {
8          if (attribute.AttributeClass.Name.CompareTo("XSSVulnerability") == 0 ||
attribute.AttributeClass.Name.CompareTo("XSSVulnerabilityAttribute") == 0)
9          {
10             if (!toBeSecure.ToString().Contains(".XSSSanitize()"))
11             {
12                 // For all such symbols, produce a diagnostic.
13                 var diagnostic = Diagnostic.Create(Rule, syntaxExpression.GetLocation(),
rightSideInfo.Symbol.Name);
14                 context.ReportDiagnostic(diagnostic);
15             }
16         }
17     }
18 }
```

Code fix

As mentioned earlier, securing the code against the threat of a cross-site scripting attack will be implemented by calling a sanitization function that removes the malicious portion of code from the string. Now that the goal has been set, what remains to be done is to arrange the code fix project to insert the necessary nodes in the syntax tree and then to add the sanitization function call to the expression (right side) of the target assignment instruction.

As previously done with the analyzer project, to declare a code fix class it is necessary, at creation time, to derive the class `CodeFixProvider` and to associate the attribute `ExportCodeFixProvider` like shown in the following code snippet.

```
1 [ExportCodeFixProvider(LanguageNames.CSharp, Name = nameof(XSSCodeFixProvider)), Shared]
2 public class XSSCodeFixProvider : CodeFixProvider
3 {
```

The first step in implementing the code fix class is to override the abstract property **FixableDiagnosticIds**. It allows a code fix project to be associated with its syntactic analyzer via its identifier (**DiagnosticId**). The code shown below points out this operation.

```
1 public sealed override ImmutableArray<string> FixableDiagnosticIds
2 {
3     get { return ImmutableArray.Create(XSSAnalyzer.DiagnosticId); }
4 }
```

The method **RegisterCodeFixesAsync** is then tasked with intercepting the created diagnostics and calling the function responsible for handling the syntactic tree modification procedures.

In this case, in order for the code fixing to be notified and then applied on the code, it is necessary to register the change action on the object of class **Context** via the method **RegisterCodeFix**. It requires as parameters the code action that will be invoked to apply the changes and the diagnostic to which the action refers.

The fixing action is generated by the (**CodeAction.Create**) method through the use of the following parameters:

- **title**: **string** or **LocalizableString** parameter which specifies the title of the fixing operation;
- **createChangedDocument**: **Func<CancellationToken, Task<Document>>** which specifies the function that creates the new version of the Document (or Solution if the fixing interests more files);
- **equivalenceKey**: optional **string** parameter which determines the equivalence between different **CodeActions**;

The **createChangedDocument** parameter invokes the **AddSanitizationAsync** function which is specifically created to address the XSS sanitification code fixing procedures. It

receives as parameters the `Document` on which to make the changes and the target node of the diagnostics. The following code shows the structure of the `RegisterCodeFixesAsync` function.

```

1 public sealed override async Task RegisterCodeFixesAsync(CodeFixContext context)
2 {
3     var root = await
context.Document.GetSyntaxRootAsync(context.CancellationToken).ConfigureAwait(false);
4
5     // DONE: Diagnostic detection and beginning of sanitization process
6     var diagnostic = context.Diagnostics.First();
7     var diagnosticSpan = diagnostic.Location.SourceSpan;
8
9     // Find the property that requires sanitization identified by the diagnostic.
10    var accessExpr =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<AssignmentExpressionSyntax>().First();
11
12    // Register the action that will invoke the fix.
13    context.RegisterCodeFix(
14        CodeAction.Create(
15            title: CodeFixResources.XSSCodeFixTitle,
16            createChangedDocument: c => AddSanitizationAsync(context.Document, accessExpr,
c),
17            equivalenceKey: nameof(CodeFixResources.XSSCodeFixTitle)),
18        diagnostic);
19 }

```

The goal of the `AddSanitizationAsync` function is to recreate the structure of a function call which invokes the sanitization method and then rebuild the syntax tree. Figure 25 shows the subtree that will replace the vulnerable node one in the new tree.

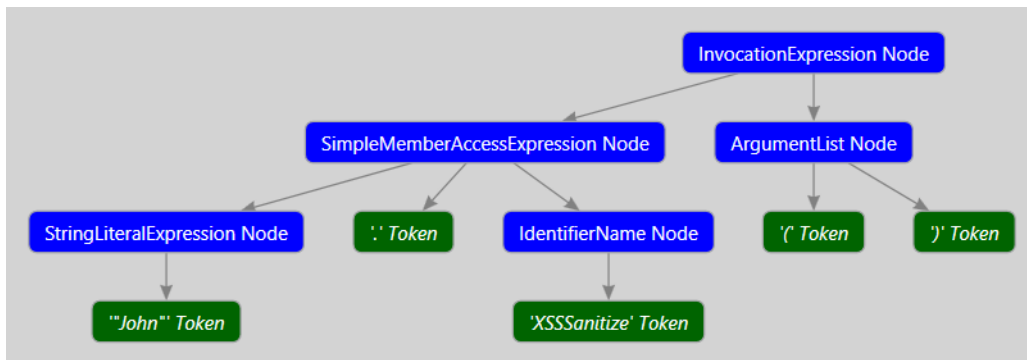


Figure 25: Syntactic node hierarchy corresponding to a method invocation

In order to reconstruct the structure shown by the code snippet above, the methods for creating new syntactic elements made available by the `SyntaxFactory` class were used. The node that will define the callsign of the `sanitize` function belongs to the kind `Identi-`

fierName, so, during its creation, the name of the sanitize function will have to be used as a parameter for the `SyntaxFactory.IdentifierName` method, in the present case `XSS-Sanitize`. The snippet below represents the new nodes hierarchy creation process.

```

1 // The node before the sanitization
2 ExpressionSyntax rightSide = accessExpr.Right;
3
4 // Sanitized node creation: InvocationExpressionSyntax node composition
5 IdentifierNameSyntax sanitizeFunc = SyntaxFactory.IdentifierName("XSSSanitize");
6
7 MemberAccessExpressionSyntax sanitizeCall =
8 SyntaxFactory.MemberAccessExpression(SyntaxKind.SimpleMemberAccessExpression, rightSide, sanitizeFunc);
9
10 // New version of the vulnerable node
11 InvocationExpressionSyntax newNode = SyntaxFactory.InvocationExpression(sanitizeCall);

```

Now that the new subtree is available, the last thing left to do, is to use it to replace the old `ExpressionSyntax` node. This will be done through the previously mentioned function `ReplaceNode` which, starting from the old one, will create a new root for the new syntactic tree of the document, containing the new subtree in place of the older one.

```

1 // The vulnerable node is now replaced with the new one previously created and the new syntax tree is
  // returned through its root
2 SyntaxNode newRoot = oldRoot.ReplaceNode(accessExpr.Right, newNode);
3 return document.WithSyntaxRoot(newRoot);

```

3.3.2 SQL analyzer

What has been called the SQL analyzer actually consists of three separate projects. This choice was made because, although one is the threat addressed, there are actually three points in the code that make it vulnerable to it. For this reason it was decided to create three analyzers in different files: the `SQLStringAnalyzer`, `SQLPropertyAnalyzer`, and `SQLParameterAnalyzer`. Of course, again, each diagnostic analyzer will be accompanied by the corresponding code fix.

Diagnostic analyzer

The initial part of these diagnostic analyzers is very similar to the one presented above for the XSS analyzer: the declaration statement of the analyzer class with the derivation

from the `DiagnosticAnalyzer`, the assignment of the `[DiagnosticAnalyzer]` attribute, the creation of the `DiagnosticDescriptor`, the overriding of the `SupportedDiagnostics` property and the `Initialize` function. What differentiates them are:

- The value associated with the `DiagnosticDescriptor` object's parameters, for example the `DiagnosticId` value of the XSS analyzer was "XSS01" while for the SQL analyzers will be set to "SQL01", "SQL02" or "SQL03".
- the *action* and *syntaxKinds* parameters values which change depending on, respectively, the function called and the kind of syntactic element to be analyzed.

SQLStringAnalyzer

This analyzer aims to identify a weakness in the usage of interpolated strings that may lead them to be prone to SQL injection. An interpolated string is a string literal that might include the usage of the value of a variable as part of the text defining it. The variable inserted in the string is used to change the behaviour of the represented query depending on its content. Usually, this variable, contains text received from an external source like a web application user. Not having total control over what it receives as input, its value can change the behaviour of the query into something unintended. To solve this problem interpolated strings have been modified adding a *placeholder* in place of the variable thus making the resulting queries parametric queries. In parametrized queries, the body of the text and any parameters are sent to the database separately, thus preventing the original code and the customised values from mixing.

The easiest way to explain the situation is through an example. The code snippet below shows the appearance of a vulnerable string.

```
1 var query = "SELECT Title, Body, Excerpt FROM Post WHERE SearchTerm = '" + SearchTerm + "' ORDER BY  
Published DESC";
```

The problem with the above queries is that the variable `SearchTerm` comes from user input and it's simply inserted or concatenated as is in the query. When the program is executed, the variable is simply replaced with its value and then the resulting string is sent to the database which runs it, no matter from where the query pieces come from. The way used by parametrized queries to avoid this problem is not including user-provided values in the query at all. A placeholder substitutes the variable in the string and a dedicated

method is delegated of its association with the variable. The issue did not arise because the SQL parameter is encapsulating the search term data such that it is sent to the server separate from the SQL query text. The task of the diagnostic analyzer will be to identify vulnerable strings and report them to the user, who may decide to sanitize them, and thus make them parametric strings using the placeholder, via the appropriate code fix. The string contained in the previous example would be written like this:

```
1 var query = "SELECT Title, Body, Excerpt FROM Post WHERE SearchTerm = @SearchTerm ORDER BY Published
DESC";
```

Now that the purpose of the SQL analyzer is clear is time to have a look at the details of its implementation.

As can be seen from the next code snippet, the syntactic elements involved in our search are the **InterpolatedStringExpression** nodes, a particular type of nodes which in this case are used to identify strings that will act as queries. The function in charge of detecting the vulnerable ones is called **SQLStringDetectSyntax**.

```
1 //Registers code actions on InterpolatedStringExpression nodes
2 context.RegisterSyntaxNodeAction(action: SQLStringDetectSyntax, syntaxKinds:
SyntaxKind.InterpolatedStringExpression);
```

The **InterpolatedStringExpression** is the kind associated with **InterpolatedStringExpressionSyntax** node type. These types of nodes are the roots of subtrees containing the types of nodes that will allow the vulnerable strings to be identified. The vulnerable string is composed by pure text and one or more variables. In syntactic terms, purely textual nodes are associated with type **InterpolatedStringTextSyntax** while variables with **InterpolatedSyntax**.

What will discriminate the vulnerability of a string will be the presence or the absence of the "@" character in the last position of the string representation of an **InterpolatedStringTextSyntax** node which comes before an **InterpolatedSyntax** node. The presence denotes that the following variable should be treated as a placeholder while the absence as a normal variable. It is precisely the absence of this character that indicates the vulnerability of the string and will therefore result in the generation of the alert.

What remains to be done, therefore, is to navigate the previously mentioned subtree nodes in search of this pattern: `InterpolatedStringTextSyntax` node not ending with "@" followed by an `InterpolatedSyntax` node. The subtree navigation takes place via the `ChildNodes` method which, by returning the child nodes list of the `InterpolatedStringExpressionSyntax` root node, will allow iterating on it in search of the previously mentioned sequence of nodes. The `ChildNodes` usage is shown in the following snippet.

```
1 // Iterate over the child nodes of the detected InterpolatedStringExpressionSyntax node
2 foreach (var i in interpolated.ChildNodes())
3 {
```

Once the textual representation of the syntax token has been accessed through the `TextToken.Text` property (property which belongs to an `InterpolatedStringTextSyntax` node), the function `EndsWith` can be used to check whether the string ends with the "@" character. Only in the negative case the diagnostics will be produced to report the vulnerability, in fact the absence of this character indicates that no placeholders are used. The code snippet below shows the just described process.

```
1 // If the string to be checked does not ends with the "@" character it means that the variable
  corresponding
2 //to the Interpolation kind node could be vehicle of injection and so it should be reported
3 if (!toBeChecked.TextToken.Text.EndsWith("@"))
4 {
5     // Diagnostics creation using the InterpolatedStringExpressionSyntax node
6     var diagnostic = Diagnostic.Create(Rule, interpolated.GetLocation(),
interpolated.Contents.ToString());
7     context.ReportDiagnostic(diagnostic);
8 }
```

SQLPropertyAnalyzer

On the other side, the `SQLPropertyAnalyzer` has similar behaviour to the `XSSAnalyzer`. In fact, it checks the access occurrences to the property named `CommandText`, which belongs to the `SqlCommand` objects, and creates the diagnostic report when a set property operation is performed without having sanitized the assigned expression. The below snippet represents some cases of unsafe assignment.

```
1 using (var command = new SqlCommand(sql, connection))
2 {
3     command.CommandText = sql;
4     command.CommandText = "Test";
5     command.CommandText = sql.ToString();
6     // ...
```

In this case, the sanitization function which the analyzer aims to insert through the code fixing operations is called **SQLSanitize** and consists of an extension method containing the sanitization procedures which rebuild the received strings making them SQL injection-proof. The implementation of this method will be subject of future developments. The expected result is the one shown in the snippet below.

```
1 using (var command = new SqlCommand(sql, connection))
2 {
3     command.CommandText = sql.SQLSanitize();
4     command.CommandText = "Test".SQLSanitize();
5     command.CommandText = sql.ToString().SQLSanitize();
6     // ...
```

This time the **RegisterSyntaxNodeAction** function calls the method **SQLCmdDetectSyntax** and sets as target syntax node kind the **SimpleAssignmentExpression**. This is the same node kind that was used in the XSSAnalyzer case, in fact the process of navigating the syntactic tree to reach the node is very similar. The differences lie in the membership class of the object to which the property belongs, namely **SqlCommand**, and the value of the string which prevents the diagnostics appearance, namely **".SQLSanitize()"**. The diagnostic generation and "publication" process follows the one explained in the XSSAnalyzer section. The code snippet present below shows the sequence of checks implemented to distinguish vulnerable from secure code.

```
1 if (leftSideInfo.Type.TypeKind == TypeKind.Class && leftSideInfo.Type.Name.CompareTo("SqlCommand") == 0)
2 {
3     // If the property name is CommandText
4     if (rightSideInfo.Symbol.Name.CompareTo("CommandText") == 0)
5     {
6         if (!toBeSecure.ToString().Contains(".SQLSanitize()"))
7         {
8             // For all such symbols, produce a diagnostic.
9             var diagnostic = Diagnostic.Create(Rule, syntaxExpression.GetLocation(),
10 rightSideInfo.Symbol.Name);
11             context.ReportDiagnostic(diagnostic);
12         }
13 }
```

SqlParameterAnalyzer

Another point in the code which requires sanitization is the first parameter of the **SqlCommand** constructor. This function creates a new instance of the object that will manage the communication with the database. In this case **SqlCommand** needs as first parameter the string which represents the query to be executed by the server and as second one the object which represents the connection with the database (created through the constructor **SqlConnection**).

In this case the function which performs the vulnerability detection is called **SQLParamDetectSyntax** and the syntax kind subject of the search is the **ObjectCreationExpression**. This kind is associated with the node type **ObjectCreationExpressionSyntax** which is used to represent the nodes containing the constructor method call to initialise a new object of a class.

After having reached the syntax node related to the first parameter of the **SqlCommand** constructor through the function **ArgumentList.Arguments.First()**, the only things left to do are the checking that the string representation of the selected node does not contain the ".SQLSanitize()" pattern (which indicates that the sanitization operation have already been arranged) and, in case of positive feedback resulting from this last check, the creation and "publication" of the diagnostics. The following code snippet represents the method which implements the vulnerabilities detection and diagnostics creation procedures.

```

1 private static void SQLParamDetectSyntax(SyntaxNodeAnalysisContext context)
2 {
3     // Node extraction from the context
4     var commandCreation = (ObjectCreationExpressionSyntax)context.Node;
5
6     // Get the first parameter of the constructor
7     var firstParam = commandCreation.ArgumentList.Arguments.First();
8
9     if (commandCreation.Kind() == SyntaxKind.ObjectCreationExpression &&
10    commandCreation.Type.ToString().CompareTo("SqlCommand") == 0)
11     {
12         // If not already sanitized, diagnostic should be created
13         if (!firstParam.ToString().Contains(".SQLSanitize()"))
14         {
15             // Diagnostic generation using the creation instruction
16             var diagnostic = Diagnostic.Create(Rule, commandCreation.GetLocation(),
17 firstParam.ToString());
18             context.ReportDiagnostic(diagnostic);
19         }
20     }
21 }

```

Code fix

For what concern the code fixing operations they are organized in three projects depending on the diagnostic analyzer they are linked with. The class declaration part of these projects mirrors the one presented for the XSSAnalyzer with the appropriate naming corrections. What changes are the content of the `RegisterCodeFixAsync` function and the methods which manages the code fixing operations.

SQLStringCodeFixProvider

The `RegisterCodeFixesAsync` function for the `SQLStringCodeFixProvider` project works with the diagnostics affecting the node type `InterpolatedStringExpressionSyntax`. The vulnerable node obtained by reading the diagnostics will become the parameter of the fixing function that will be called inside the `CodeAction.Create` method. This function is used to create the action that will be registered as code fix operation by `RegisterCodeFix` method.

```

1 // Register a code action that will invoke the fix.
2 context.RegisterCodeFix(
3     CodeAction.Create(
4         title: CodeFixResources.SQLStringCodeFixTitle,
5         createChangedDocument: c => SanitizeStringAsync(context.Document, interString, c),
6         equivalenceKey: nameof(CodeFixResources.SQLStringCodeFixTitle)),
7     diagnostic);

```

The function in charge of defining the refactoring procedures is called **SanitizeStringAsync**. Inside of it, the subtree which has the vulnerable **InterpolatedStringExpressionSyntax** node as root will be travelled, like previously done during the code analysis phase, in order to find the **InterpolatedStringTextSyntax** node which must be modified to make the dangerous variable, represented by a node of type **InterpolationSyntax**, a placeholder. To perform this operation, the single-quote that is usually present at the end of the text string which precedes the dangerous variable is replaced with the "@" character. For tidier code, also the single-quote which follows the variable will be removed. The new strings thus obtained will then be used to generate, through the appropriate methods belonging to the **SyntaxFactory** class, the new syntax nodes that will make up the new subtree. The resulting hierarchical structure is the one shown in Figure 26.

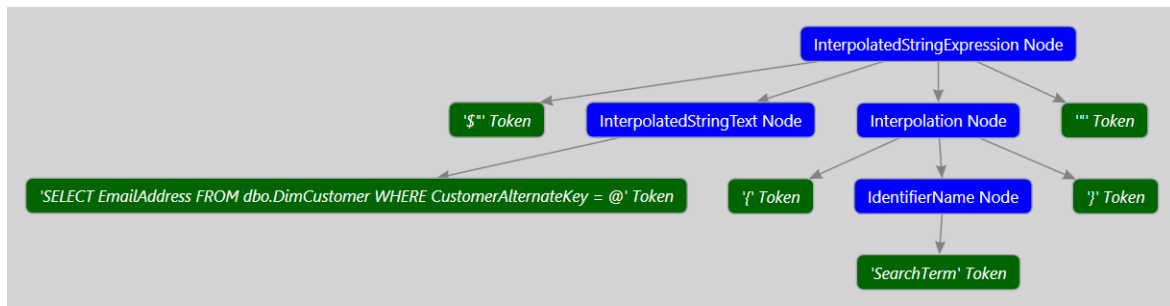


Figure 26: Subtree resulting from the code fix operation

Once the new subtree is ready, the creation of the main syntax tree via the previously mentioned **ReplaceNode** function can be implemented.

SQLPropertyCodeFixProvider

The code fix operations corresponding to the diagnostics created by the **SQLPropertyAnalyzer** are the same as those adopted for the **XSSAnalyzer**. The only difference is the name of the sanitization function that in this case is **SQLSanitize**.

SQLParameterCodeFixProvider

The refactoring of the code corresponding to the diagnostics created by the `SQLParameterAnalyzer` analyser is very similar to the one carried out for the `XSSAnalyzer` and `SQLPropertyAnalyzer`. The main difference is that the replaced node is no longer the root of the subtree but an internal node of it. As can be seen from the code snippet, the node replaced via `ReplaceNode` is the one corresponding to the first parameter of the constructor, node type `InvocationExpressionSyntax`, and not the one used as starting point in the detection process, node type `ObjectCreationExpressionSyntax`.

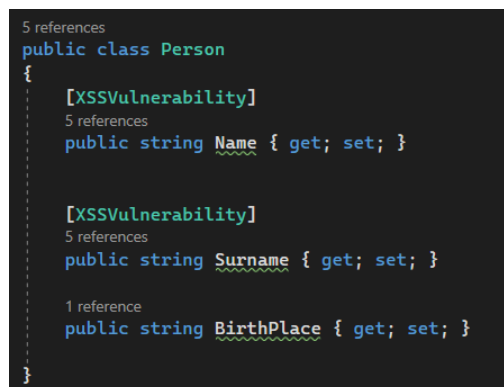
```
1 private static async Task<Document> SQLParameterSanitizeAsync(Document document,
2   ObjectCreationExpressionSyntax creationExpr, CancellationToken cancellationToken)
3 {
4     SyntaxNode oldRoot = await document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
5     IdentifierNameSyntax sanitizeFunc = SyntaxFactory.IdentifierName("SQLSanitize");
6
7     MemberAccessExpressionSyntax sanitizeCall =
8     SyntaxFactory.MemberAccessExpression(SyntaxKind.SimpleMemberAccessExpression,
9     creationExpr.ArgumentList.Arguments.First().Expression, sanitizeFunc);
10
11     InvocationExpressionSyntax newNode = SyntaxFactory.InvocationExpression(sanitizeCall);
12
13     SyntaxNode newRoot = oldRoot.ReplaceNode(creationExpr.ArgumentList.Arguments.First().Expression,
14     newNode);
15     return document.WithSyntaxRoot(newRoot);
16 }
```


4 Results evaluation

This chapter will focus on the evaluation of the results obtained in order to verify the proper functioning of the tools developed. It will present the the effectiveness of the developed solutions testing the code through the .VSIX project which creates a new instance of Visual Studio that runs the analyzers implemented in the SecurityEnhancer solution. The results will be shown through images representing the testing environment before and after the activation of the analysis and code fix tools. This will highlight the changes made by the developed solution compared to a normal case study.

XSSAnalyzer solution

The XSSAnalyzer diagnostic analyzer was intended to act as a programmer's reminder, as it generates a diagnostic whenever a property marked with the `XSSVulnerabilityAttribute` participates in an assignment operation. This was done to prevent sensitive properties from being a vehicle for script injection. Figure 27 shows an examples of how the `XSSVulnerability` attribute had been used in the testing project.



```
5 references
public class Person
{
    [XSSVulnerability]
    5 references
    public string Name { get; set; }

    [XSSVulnerability]
    5 references
    public string Surname { get; set; }

    1 reference
    public string BirthPlace { get; set; }
}
```

Figure 27: Example of how the attribute is used in a testing project

This example assumes that the `Name` and `Surname` properties of the `Person` object are assigned with string values or other variables. By pretending that, these properties were

somehow used in the creation of a dynamic web page, and therefore contributed to defining its content, they may be considered vulnerable to script injection. For this reason, the creator of the web application code may want to be sure that they cannot convey security threats to his program. The developer will therefore be able to mark them with the attribute provided by the SecurityEnhancerSDK in order to not forget to sanitise them when they will be used.

Whenever a marked property is involved as target in an assignment operation, the XSS-Analyzer will produce a diagnostics alerting the programmer about the absence of sanitization measures. Figure 28 and figure 29 show the status of the code before and after the activation of the XSSAnalyzer.

```
dude.BirthPlace = "London";
dude.Name = "John";
dude.Name = "Paul";
dude.Surname = test1;
dude.Surname = test2;
```

Figure 28: Code appearance before the activation of the XSS diagnostic analyzer

```
dude.BirthPlace = "London";
dude.Name = "John";
dude.Name = "Paul";
dude.Surname = test1;
dude.Surname = test2;
```

class System.String
Represents text as a sequence of UTF-16 code units.
XSS01: The property 'Surname' should be sanitized to prevent injection
Show potential fixes (Alt+Enter or Ctrl+.)

Figure 29: Code appearance after the activation of the XSS diagnostic analyzer

To be noted that the property `BirthPlace`, which is not tagged with the `XSSVulnerability` attribute is not affected by the detection.

To relieve the developer of the burden of implementing the sanitization function, the analyzer is able to produce tooltips in relation to the newly produced diagnostic which proposes the insertion of the call to the sanitization function `XSSSanitize`. Figure 30 shows this situation (do be noted the presence of a preview of the proposed code fix). In case the programmer chooses to apply the proposed code fix, the code will be transformed as shown in figure 31. To be noted that in presence of the `XSSSanitize` function call the diagnostic alert does not arise, and also that the changes were applied whether the expression on the right side of the assignment consisted of a string or a variable.

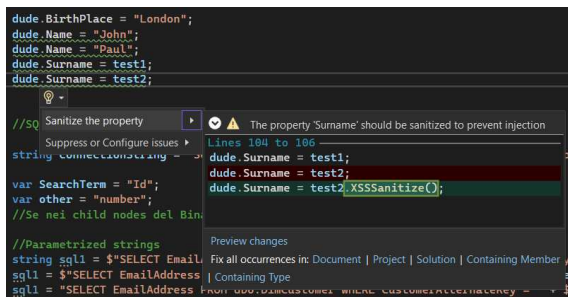


Figure 30: Example of code fix suggestion against XSS vulnerability



Figure 31: Code fix results for XSS vulnerability

SQLStringAnalyzer solution

The SQLStringAnalyzer project has the objective of detecting strings created by the programmer to be used as queries that may give rise to a sql injection. This section of the thesis will show some examples of vulnerable strings, the related warnings produced by diagnostics and what they will look like after the refactoring operations.

The structure of a vulnerable string was shown earlier in chapter 3, but it is not the only form that a vulnerable string can take. A vulnerable interpolated string can be represented as normal text which contains one or more variables, concatenation of string and variables and so on. Figure 32 shows some examples of vulnerable and not vulnerable (already in parametric form) interpolated strings.

```

// *** Not vulnerable strings ***
string sql1 = $"SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = @SearchTerm";
sql1 = $"SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = @{{SearchTerm}}";
sql1 = "SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = " + $"@{{SearchTerm}}";
string sql2 = "SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = " + $"@{{SearchTerm}}";

// *** Vulnerable strings ***
// Single string, single vulnerability
string sql4 = @"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}'";
// Single string, double vulnerability
string sql5 = @"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}' and '{{other}}'";
// Concatenated strings, both vulnerable
string sql6 = @"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}' + $"and '{{other}}'";
// Concatenated strings, single vulnerability
string sql7 = @"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}' + $"and @{{other}}";
// Single string, double parameter, single vulnerability
string sql8 = @"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}' and @{{other}}";
// Concatenated strings, single vulnerability on the second one
string sql9 = "SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = " + $"'{{SearchTerm}}'";
// Assignment operation case (instead of declaration)
sql9 = "SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = " + $"'{{SearchTerm}}'";

```

Figure 32: Examples of vulnerable and not vulnerable interpolated strings

It is therefore time to check if the created analyzer is able to identify the vulnerable strings and leave out those that are not. Figure 33 presents the obtained result.

```

// *** Not vulnerable ***
string sql1 = $"SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = @SearchTerm";
sql1 = $"SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = @{{SearchTerm}}";
sql1 = $"SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = " + $"@{{SearchTerm}}";
string sql2 = "SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = " + $"@{{SearchTerm}}";

// *** Vulnerable strings ***
// Single string, single vulnerability
string sql4 = $"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}';
// Single string, double vulnerability
string sql5 = $"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}' and '{{other}}';
// Concatenated strings, both vulnerable
string sql6 = $"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}' + $"and '{{other}}';
// Concatenated strings, single vulnerability
string sql7 = $"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}' + $"and @{{other}}";
// Single string, double parameter, single vulnerability
string sql8 = $"SELECT EmailAddress
FROM dbo.DimCustomer
WHERE CustomerAlternateKey = '{{SearchTerm}}' and @{{other}}";
// Concatenated strings, single vulnerability on the second one
string sql9 = "SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = " + $"'{{SearchTerm}}';
// Assignment operation case (instead of declaration)
sql9 = "SELECT EmailAddress FROM dbo.DimCustomer WHERE CustomerAlternateKey = " + $"'{{SearchTerm}}';

```

Figure 33: SQL injection vulnerable interpolated strings detection results

The previous figure shows that the analyzer is perfectly able to distinguish the vulnerable strings from those who are not. In the event that the vulnerable string corresponds to a composition of strings, it is also able to provide the alert only for the component that generates the vulnerability.

It is now time to verify that the code fix operations effectively transform "bad strings" into parametric strings. Figure 34 shows the strings appearance after having applied the code fix procedures.

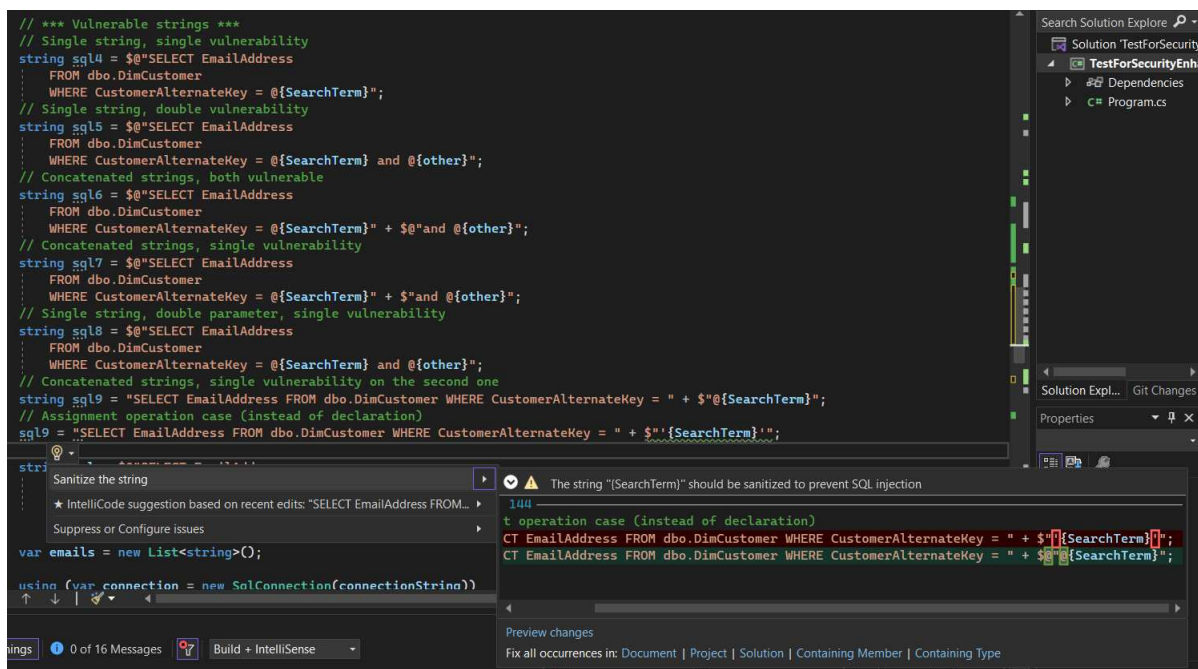


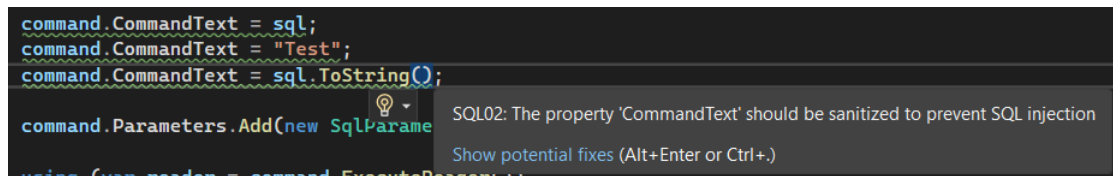
Figure 34: Code fixing results on SQL injection vulnerable interpolated string

SQLPropertyAnalyzer solution

The principle behind the SQLPropertyAnalyzer project is very similar to the XSSAnalyzer one, which is to provide the possibility for the programmer to sanitize the expression assigned to a particular property (SqlCommand.CommandText) as unaware of its content. Figure 35 and figure 36 show the situation before and after activating the analyzer.

```
using (var command = new SqlCommand(sql, connection))
{
    command.CommandText = sql;
    command.CommandText = "Test";
    command.CommandText = sql.ToString();
}
```

Figure 35: CommandText property assignments vulnerable to SQL injection



```
command.CommandText = sql;
command.CommandText = "Test";
command.CommandText = sql.ToString();
command.Parameters.Add(new SqlParameter("SearchTerm", SearchTerm));
using (var reader = command.ExecuteReader())
```

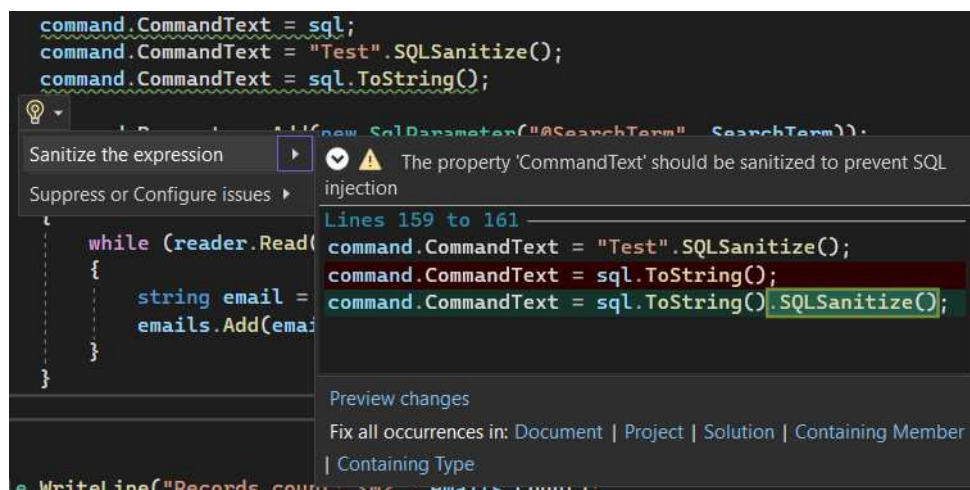
SQL02: The property 'CommandText' should be sanitized to prevent SQL injection

Show potential fixes (Alt+Enter or Ctrl+.)

Show details (Alt+Enter or Ctrl+.)

Figure 36: Diagnostic alert on CommandText property assignment operations

If the programmer decides to apply the sanitization, the appearance of the resulting code is the one shown in figure 37.



```
command.CommandText = sql;
command.CommandText = "Test".SQLSanitize();
command.CommandText = sql.ToString().SQLSanitize();
new SqlParameter("@SearchTerm", SearchTerm));
while (reader.Read())
{
    string email =
    emails.Add(email);
}
e.WriteLine("Records count: " + emails.Count);
```

Sanitize the expression

Suppress or Configure issues

The property 'CommandText' should be sanitized to prevent SQL injection

Lines 159 to 161

command.CommandText = "Test".SQLSanitize();

command.CommandText = sql.ToString();

command.CommandText = sql.ToString().SQLSanitize();

Preview changes

Fix all occurrences in: Document | Project | Solution | Containing Member

Containing Type

Figure 37: CommandText property assignments refactoring

SQLParameterAnalyzer solution

The behaviour of the SQLParameterAnalyzer project is quite similar to the SQLPropertyAnalyzer one, the difference stands in the code elements on which it acts. In this case,

instead of reporting a violation on a property, the first parameter of the SqlCommand constructor is highlighted. Figure 37 and figure 37 show respectively the creation of the diagnostic and the refactoring results of this solution.

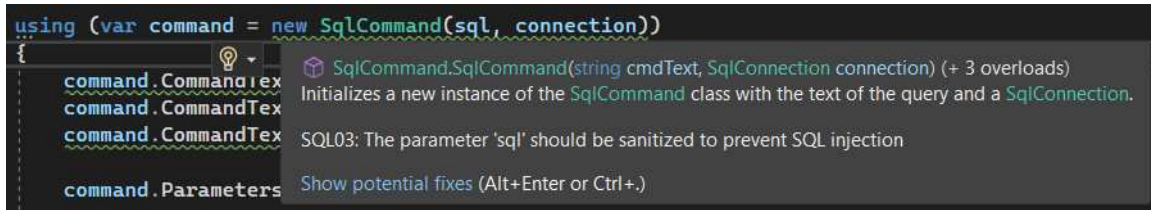


Figure 38: SqlCommand constructor vulnerable parameter detection

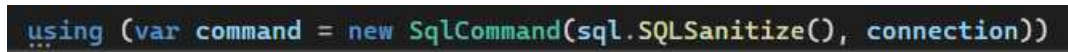


Figure 39: SqlCommand constructor vulnerable parameter sanitization

To be noted the changing of the analyzer ID in the different diagnostics.

5 Conclusions

The purpose of the thesis was to develop a syntactic analyzer for the C# language, usable as extension of Visual Studio, that would be able to alert programmers about portions of code that would make it vulnerable to some types of cyber attacks. Obviously covering the entire case history of cyber attacks would have been impossible in the time available, so only two of these were chosen as a starting point: cross-site scripting (XSS) and SQL injection. The tool which allows the creation of this kind of analyzer is the .NET Compiler Platform SDK ("Roslyn"). It makes available dedicated APIs for the development of custom diagnostics and code fixes. Thanks to it, it was possible to achieve the set goals and produce an analyzer that met the requirements. Indeed, after studying how target cyber attacks operate and what defensive measures are usually implemented to counter them, it was possible to identify vulnerable instructions in the code, flag them, and create appropriate procedures to sanitize them. This was possible through the acquisition of knowledge about the syntactic structure behind C# code development obtained from the original documentation provided by Microsoft and from experimentation on separate auxiliary projects created in parallel with the main project for purely illustrative purposes. The final results confirmed the effectiveness of the work done. The testing phase showed that the code static analysis correctly detects all the different types of vulnerabilities to which the code is subjected in terms of cross-site scripting and SQL injection attacks. Likewise, the results of code fix operations correctly produce a new version of the original code containing the changes defined in the design phase. As of today, the proposed solution is ready to be tested in a real-world scenario in order to verify its effectiveness on programs not specifically created for its testing.

The solution produced as thesis project is just a starting point. The SecurityEnhancer analyzer is designed to protect code development operations from the greatest number of vulnerabilities. In fact, it can be updated through the inclusion of new diagnostic analyzers and code fixes that secure code from other attacks such as Buffer Overflow, Cross-Site

Request Forgery (CSRF), XPath Injection, and so on. It can also be enriched with new sanitization functions through the SecurityEnhancerSDK utility project. Because of the modularity with which it was built, it can also be configured to focus analysis only toward a particular type of violation. Finally, the proposed analyzers can be improved by including functions that can refine the search for vulnerabilities and by creating from scratch the methods that will be used to sanitize the code.

Bibliography

- [1] BUMSUK JUNG, INGOO HAN, SANGJAE LEE
2001 - *Security threats to Internet: a Korean multi-industry investigation*
- [2] CHARLES P. PFLEEGER, SHARI LAWRENCE PFLEEGER, JONATHAN MARGULIES
2015 - *Security in Computing - Fifth Edition*
- [3] WENLIANG DU
2019 - *Computer & Internet Security A Hands-on Approach - Second Edition*
- [4] WIKIPEDIA
2021 - *Cross-site request forgery* https://it.wikipedia.org/wiki/Cross-site_request_forgery
- [5] WIKIPEDIA
2021 - *AJAX* <https://it.wikipedia.org/wiki/AJAX>
- [6] WIKIPEDIA
2022 - *Document Object Model* https://it.wikipedia.org/wiki/Document_Object_Model
- [7] RYAN OBERFELDER
2017 - *Describing XSS: The story hidden in time* <https://medium.com/@ryoberfelder/describing-xss-the-story-hidden-in-time-80c3600ffe81>
- [8] THE VIRUS ENCYCLOPEDIA
Spacehero <http://virus.wikidot.com/spacehero>
- [9] WIKIPEDIA
2022 - *Samy (computer worm)* [https://en.wikipedia.org/wiki/Samy_\(computer_worm\)](https://en.wikipedia.org/wiki/Samy_(computer_worm))
- [10] LORENZO FRANCESCHINI-BICCHIERAI
2015 - *The MySpace Worm that Changed the Internet Forever* <https://www.vice.com/en/article/wnjwb4/the-myspace-worm-that-changed-the-internet-forever>
- [11] LISA VAAS
2016 - *eBay XSS bug left users vulnerable to (almost) undetectable phishing attacks* <https://nakedsecurity.sophos.com/2016/01/13/ebay-xss-bug-left-users-vulnerable-to-almost-undetectable-phishing-attacks/>

- [12] YONATHAN KLIJNSMA
2018 - *Inside the Magecart Breach of British Airways: How 22 Lines of Code Claimed 380,000 Victims* <https://www.riskiq.com/blog/external-threat-management/magecart-british-airways-breach/>
- [13] LUCIAN CONSTANTIN
2018 - *Fortnite Attack Allowed Taking Over Player Accounts* <https://securityboulevard.com/2019/01/fortnite-attack-allowed-taking-over-player-accounts/>
- [14] BRAD ANDERSON
2020 - *3 Dangerous Cross-Site Scripting Attacks of the Last Decade* <https://readwrite.com/3-dangerous-cross-site-scripting-attacks-of-the-last-decade/>
- [15] PANKAJ SHARMA
2005 - *SQL Injection Techniques & Countermeasures* CERT-In: Indian Computer Emergency Response Team
- [16] H. ALSOBHI AND R. ALSHAREEF
2020 - *SQL Injection Countermeasures Methods* SQL Injection Attacks Countermeasures Assessments. Indonesian Journal of Electrical Engineering and Computer Science
- [17] ALENEZI, MAMDOUH & NADEEM, MUHAMMAD & ASIF, RAJA
2020 - *SQL Injection Attacks Countermeasures Assessments* Indonesian Journal of Electrical Engineering and Computer Science
- [18] 2021 - *The .NET Compiler Platform SDK* <https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/>
- [19] 2021 - *Understand the .NET Compiler Platform SDK model* <https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/compiler-api-model>
- [20] MANISH VASANI
2017 - *Roslyn Cookbook*
- [21] MAARTEN BALLIAUW, XAVIER DECOSTER
2012 - *Pro NuGet*