Deliverable 3.4

Report of the Tools for Vulnerability Propagation

Technical References

Document version	:	1.0
Submission Date	:	31/05/2022
Dissemination Level	:	Public
Contribution to	:	WP3 – Vulnerability Management
Document Owner	:	GRAD
File Name	:	BIECO_D3.4_31.05.2022_V1.0
Revision	:	3.0

Project Acronym	:	BIECO
Project Title	:	Building Trust in Ecosystem and Ecosystem Components
Grant Agreement n.	:	952702
Call	:	H2020-SU-ICT-2018-2020
Project Duration	:	36 months, from 01/09/2020 to 31/08/2023
Website	:	https://www.bieco.org

This project has received funding from the European Union´s Horizon 2020 Research and Innovation Programme under Grand agreement No. 952702.

Revision History

REVISION	DATE	INVOLVED PARTNERS	DESCRIPTION
0.0	24/01/2022	GRAD	Draft structure of the document and Table of Contents
0.1	11/04/2022	GRAD	Contribution to Introduction and Vulnerability Propagation Tool sections.
0.2	13/04/2022	GRAD	Contribution to AST generation section
0.3	19/04/2022	GRAD	Contribution to CFG generation
0.4	20/04/2022	GRAD	Contribution to Propagation Path and conclusions.
0.5	02/05/2022	UMU	Contribution to dependency measurement
0.5	03/05/2022	UMU	Contribution to introduction and conclusions
0.6	03/05/2022	GRAD	Contribution to section 2 and conclusions
0.7	05/05/2022	GRAD	Contribution to subsection 2.1.3
1.0	06/05/2022	GRAD	Review by Internal Reviewer, Borja Pintos
1.1	09/05/2022	GRAD	Implementing Reviewer Suggestion and Update
2.0	18.05.2022	UNI	Review by External Reviewers
2.1	22.05.2022	GRAD	Implementing External Reviewer Suggestion and Update
2.2	23.05.2022	GRAD	Final Edition preparation and sending to the coordinator
2.3	25.05.2022	UNI	Coordinator Final Review
3.1	31.05.2022	UNI	Submission

List of Contributors

Deliverable Creator(s): Eva Sotos (GRAD), Mónica Alonso (GRAD), Javier Yépez (GRAD), Sara Nieves Matheu (UMU),

Reviewer(s): Borja Pintos (GRAD), Mohammed Abuteir (TT), Jose Barata (UNI), Sanaz Nikghadam-Hojjati (UNI)

BIECO project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No **952702**.

Acronyms

Acronym	Term
AST	Abstract Syntax Tree
CFG	Control Flow Graph
DOT	Document Template
ICT	Information and Communication Technology
IT	Information Technology
JSON	JavaScript Object Notation
LN	Logic Node
ML	Machine Learning
PDF	Printer Description File
WP	Work Package

Executive Summary

The main goal of the Work Package (WP3) is to research and develop a set of cybersecurity tools oriented to the detection, forecasting and propagation of vulnerabilities across complex Information and Communication Technology (ICT). Many issues have to be considered when designing a tool for vulnerability management and, specially, when it relies on the use of Machine Learning (ML) models. The functionality of the tool as well as the definition of the concrete algorithms to be used are the main points to consider.

The core of this deliverable is to present the status and progress of the tools and methodologies developed within the task T3.4. This task will deal with the definition and creation of a tool that allows to detect and provide the propagation path of a vulnerability across interconnected ICT systems and modules. To this end, the results obtained from task T3.1 as well as graph generation techniques, such as AST (Abstract Syntax Tree) or CFG (Control Flow Graph), will be used to characterize the vulnerability propagation accurately.

Project Summary

Nowadays most of the ICT solutions developed by companies require the integration or collaboration with other ICT components, which are typically developed by third parties. Even though this kind of procedures are key in order to maintain productivity and competitiveness, the fragmentation of the supply chain can pose a high-risk regarding security, as in most of the cases there is no way to verify if these other solutions have vulnerabilities or if they have been built taking into account the best security practices.

In order to deal with these issues, it is important that companies make a change on their mindset, assuming an "untrusted by default" position. According to a recent study only 29% of IT business know that their ecosystem partners are compliant and resilient with regard to security. However, cybersecurity attacks have a high economic impact, and it is not enough to rely only on trust. ICT components need to be able to provide verifiable guarantees regarding their security and privacy properties. It is also imperative to detect more accurately vulnerabilities from ICT components and understand how they can propagate over the supply chain and impact on ICT ecosystems. However, it is well known that most of the vulnerabilities can remain undetected for years, so it is necessary to provide advanced tools for guaranteeing resilience and also better mitigation strategies, as cybersecurity incidents will happen. Finally, it is necessary to expand the horizons of the current risk assessment and auditing processes, taking into account a much wider threat landscape. BIECO is a holistic framework that will provide these mechanisms in order to help companies to understand and manage the cybersecurity risks and threats they are subject to when they become part of the ICT supply chain. The framework, composed by a set of tools and methodologies, will address the challenges related to vulnerability management, resilience, and auditing of complex systems.

Partners


Disclaimer

Table of Contents

List of Figures

1. Introduction

In the recent years, the use of technology related with different work fields has increased, either by improving existing technology or by automating jobs. In addition, the use of teleworking has recently raised due to the COVID pandemic. This large growth in the use of technologies has led to the apparition of numerous cyber-attacks which goal is to obtain information through new means and targets. It is for this reason that organizations want to ensure high security on their equipment and avoid being targeted by attackers, as this could lead to both financial and reputational losses.

One of the ways in which attackers can execute a cyber-attack is through vulnerabilities both on employees and on the organization's systems. A large number of them are carried out on systems, taking advantage of existing vulnerabilities in the source code. To evaluate the severity of the same and therefore the priority to its mitigation, it is necessary to perform a proper vulnerability assessment. This includes not only the descriptive analysis of the vulnerability, but also the forecasting of certain behaviour of the same. Due to the fact that a single vulnerability in the code of a system can affect several elements of the organization, it is crucial to mitigate any vulnerability detected. Depending on the affected elements, its priority and severity can be approximated, complementing this information with more details of the same in order to have a correct evaluation. For example, if a vulnerability exists within the financial systems, its exploitability can affect the entire organization, so the priority of the vulnerability can be considered as critical.

When it comes to perform a good vulnerability assessment process, it is important to complement a secure development methodology [1]. One of its main steps is to perform a code review to detect the existence of vulnerabilities. Currently there are tools that are able to identify the location of a vulnerability within the source code, such as the one developed within this WP3 in task T3.3 [2]. Nevertheless, such process alone sometimes is not enough for ensuring a proper vulnerability assessment. The elements that derive from the location of the vulnerability itself can be affected as well, allowing attackers access or change the flow of the system operation.

In order to identify which elements are affected by a vulnerability, from BIECO, it is proposed to develop a propagation tool and a methodology to calculate the degree of dependency of a subsystem. In this deliverable we will proceed to explain the purpose and operation of the propagation tool and the methodology for the dependency calculation, which is intended to be connected in the future with the propagation tool outputs. For a better understanding of the tool and methodology aforementioned, the deliverable is organized as follows:

Section 2 introduces the propagation tool, and describes the procedure followed for its creation by focusing on 3 different phases: Abstract Syntax Tree (AST) generation for both Java and Python source code (Subsect. 2.1), Control Flow Graph (CFG) generation (Subsect 2.2) and Propagation Path (Subsect 2.3).

Section 3 describes the methodology followed for the calculation of the degree of dependency of a subsystem.

Section 4 concludes the deliverable by reporting the conclusions obtained in the tool development process as well as future actions to be taken.

2. Vulnerability Propagation Tool

The main goal of the propagation tool is to find and indicate the components or elements a single vulnerability can affect, and therefore, its path within the system.

The calculation of the path affected by a vulnerability is characterized by the source code of the system and the location of the vulnerability along the same. These will be the required inputs for the developed tool. Thus, the location of the vulnerability is identified by a function of the source code of the program to be analysed, or by an external library that is being used. In the latter case, for example, all the elements related to the use of that library can be affected.

Having these input parameters and taking into account the state of the art on the subject [3], it has been decided to develop a propagation tool based on graphs techniques. The calculated graphs will allow to obtain a roadmap of the code under analysis and, therefore, of its structure. These will be use as a basis to mark the path followed by the indicated vulnerability. After a previous study on different code representation techniques [4, 5], a combination of them has been chosen, which has led to the design of a tool based on 3 phases (Figure 1):

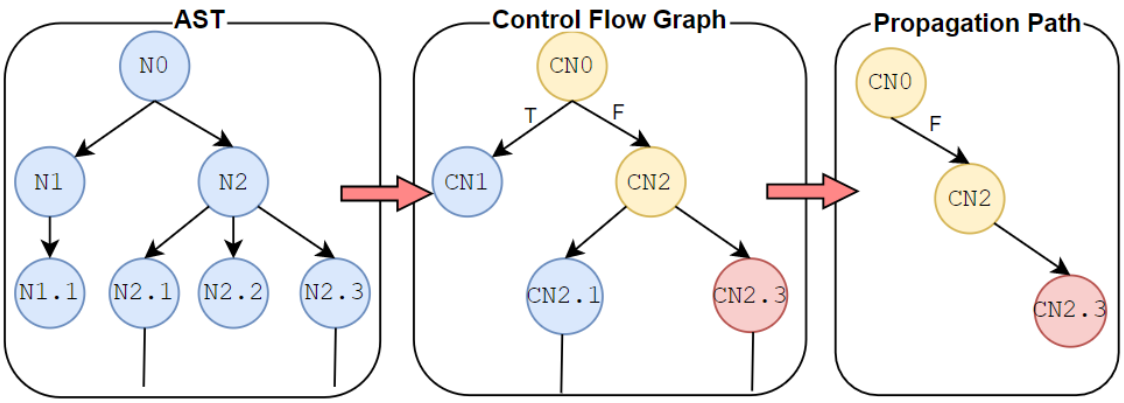

Figure 1. Phases of the Propagation Tool execution.

In the first phase, the tool starts to elaborate the AST by performing an analysis on the system source code, i.e., analyses the source code of the program and performs a tree representation. In the second phase, new nodes with the useful information are created based on the AST. In this way, a CFG of the source code of all the elements of the system is obtained. Subsequently, after obtaining this CFG of the analysed system, the location of the vulnerability is indicated to elaborate its corresponding path through the CFG. To do this, the tool goes through all the connections that the vulnerability has on the previously created flow. The propagation tool obtains as a result the control flow detailing all the elements, from the first instruction at the beginning of the program to the element where the vulnerability is located. An example of the process can be found in Annex A.

These elements related to the vulnerability can been seen as the possible steps an attacker could follow until reaching its possible exploitation. Thus, indicating this path, the developer can identify which part of the flow modify in order to mitigate the

vulnerability. Therefore, the propagation tool offers the developer a perspective of preventing vulnerabilities in the system.

Next, the generation of the different representation in the steps of the tool are explained in more detail.

2.1 AST generation and Logic Nodes

AST is a representation of the abstract syntactic structure of the source code written in a formal language, where each node of the tree denotes a construct occurring in the code. When it comes to the conversion of the code to some intermediate representation, such as machine code, three intermediate steps are carried out (Figure 2)[1]:

- Lexical analysis: transformation of characters into tokens that can be used for syntactic analysis. These tokens describe the different parts or components of the code.
- Syntax analysis: Transformation of the tokens into a data structure. i.e., AST[2]. This code structure identifies the type of code involved, like function call, variable, etc. The AST is composed of nodes that specify what they represent. Code transformations or analysis of the code can be performed on this structure.
- Code generation: The previous structure is transformed into source code. This new code may not be the same as the one used for lexical analysis in case any modification has been made on the AST.

Figure 2. Functioning of creation and use of AST.

As previously mentioned, the propagation tool creates a syntactic tree or AST where the abstract syntactic structure of the source code of a programming language is represented in the form of a tree. Each node of the tree indicates an instruction that occurs in the code, with each new block of code being a new level of children within the tree. It should be noted that, in the AST, not all details of the actual syntax of the source code are represented[3].

To start creating the AST for the propagation tool, nodes that compose it are elaborated. Each node refers to a different source code instruction that is related to each other taking into account the execution of the program.

Figure 3 shows an example of a code to be analysed that contains a vulnerability in the marked line, and from which the AST is obtained by the propagation tool (Figure 4).

[1] https://javascript.plainenglish.io/abstract-syntax-tree-ast-f075b190e631?gi=e0ba22119342
[2] https://www.twilio.com/blog/abstract-syntax-trees
[3] https://www.techopedia.com/definition/22431/abstract-syntax-tree-ast

```
main:
        int a = 1
        int b = 2
        int result = 0
        if a>b:
                result = 1
        else:
                result = 0
```

Figure 3. Example of a code to be analysed

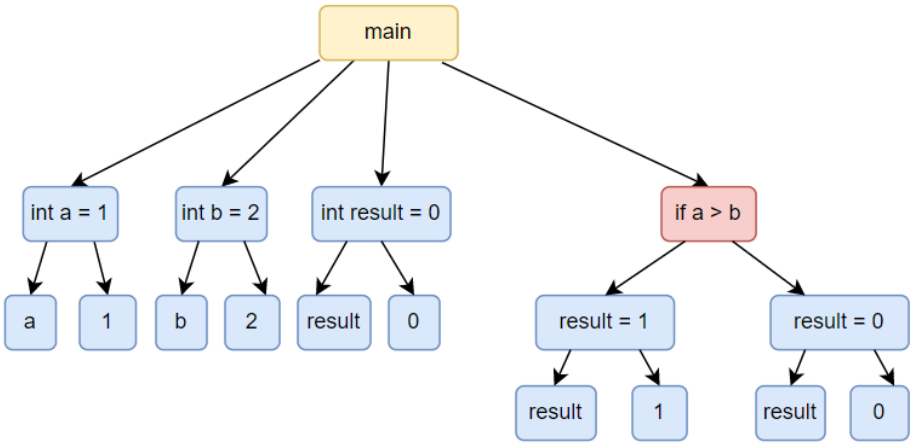

Figure 4. AST corresponding to the code in Figure 3.

As can be seen in the example, a part of the code is shown where the principal node is the "*main*". Within this node, there are 4 lower nodes, that is, 4 instructions that are executed. However, from the perspective of the AST, a new level of 4 children is created having all as parent the "*main*" node. The relationship of these nodes is represented through the parents and children that compose a syntactic tree. These 4 child nodes related to the "*main*" node are the following:

- *int a = 1*: Declaration of a variable called "*a*" of type integer with the assignment of a value of 1.
- *int b = 2:* Declaration of a variable named "*b*" of type integer with the assignment of a value of 2.
- *int result = 0*: Declaration of a variable called "*result*" of type integer with the assignment of a value of 0.
- *if-else*: conditional statement where a comparison is made with the variables "*a*" and "*b*" previously declared, so that if the variable "*a*" is greater than or equal to the variable "*b*", the variable result will take a value of 1. If this comparison is not satisfied, the "*else*" block is executed where the variable "*result*" will take the value of 0.

Within the conditional statement, two branches are created by the content of the *if-else* blocks. Each of these branches derives to a different node, taking into account the possibility that the comparison is fulfilled or not. In this example case, the two nodes interpret the value taken by the result variable.

If the vulnerability is identified in the *if* comparison, where it is verified that the variable *a* is greater than or equal to the variable *b,* it is possible to obtain the elements related to it and to provide to the developer the necessary information to be able to mitigate the vulnerability in a simple way.

In the case of this example the vulnerability is located in a conditional statement. However, and as mentioned above, the vulnerability will be identified by a function, that is, it will not be identified by a specific line of code, but by the function that contains that line of code. For this reason, the AST will be created on the source code of the system file to be analysed and of which the vulnerability has been identified.

Within the framework of BIECO, the creation of a propagation tool which supports different programming languages has been proposed. To do so, and due to the existing structural differences between languages, different tools have been developed when extracting the AST depending on the language to be analysed. Currently, the tool is developed to support code files in Java and Python programming languages. The methodology used to create the AST in the two different programming languages is explained in detail below.

2.1.1 Java code

To perform the AST on Java code language, the open-source Java library *JavaParser* has been implemented. The process followed by this library is the execution of the source code of the chosen program file to later analyse and parse it in an automatic way, thus, creating the AST with the information from the source code. In addition, using the API of this library and the obtained tree, several operations can be performed on the AST, such as traversing it, or modifying and deleting nodes of the tree.

It should be noted that, in addition, this library offers the functionality of transforming an AST modified with the operations indicated above back to Java code, where the program will be created with the indicated modifications.

In the case of the creation of the propagation tool, the last-mentioned functionality of the library of transforming the modified tree into source code is not used.

2.1.2 Python code

Python language has a large number of functionalities and modules within the language library itself, which can be freely used by the developer. For the creation of the AST in the Python programming language, the *ast* module is implemented, which is responsible for processing abstract syntax grammar trees of Python applications. This module obtains the program grammar independently of the Python version used[4].

The process followed by this module is the same as the one followed to create the AST in Java. First the *ast* module is given the source code to be analysed. This code is analysed and parsed automatically obtaining the AST that indicates

[4] https://docs.python.org/3/library/ast.html

from the main node to the last node of the program flow. In addition, as in the case of Java, this module allows the operation of traversing the nodes of the AST but does not allow modifications.

Once the AST has been created, it has been implemented a pre-processed of the same to unify the AST of different programming languages in the same format. This standardization will allow a better management of the data in the next phase of the tool since it saves the need to indicate the source code language. For its development, it has been performed a pre-processing of all the nodes of the initial tree, focusing on their logic. This pre-processing generates new nodes, called Logic Nodes (LN), which provide an easy way to transverse AST nodes by focusing on the logic level. This makes it easy for developers to get a clear view of the AST that represents the program.

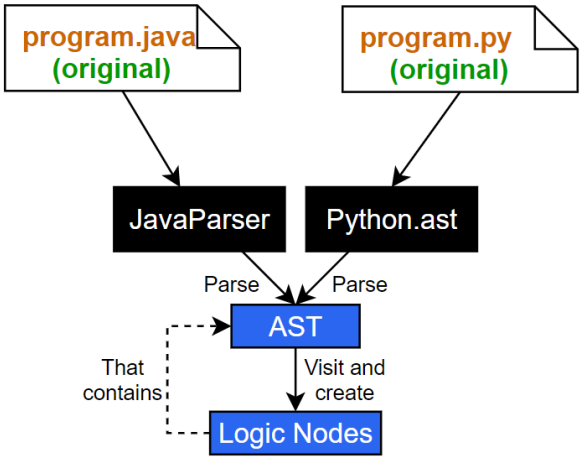

Figure 5. Process of Logic Nodes creation using

The process for obtaining the final LN is shown in Figure 5. In it, the corresponding Python or Java library obtains a file of source code, parses it and creates an AST. The propagation tool visits this AST and creates its LN by wrapping the AST, allowing it to be accessed logically.

An example of this simplification is shown in Figure 6. It presents two possible cases: one that shows the simplification of the three nodes that compose the assignment instruction into a single node (left side), and another that represents the two nodes that compose the *return* instruction simplified into a single node (right side). As it is possible to observe, green nodes, which are used for the creation of the new tree, do not provide as much detail as the nodes that the initial AST has, in blue, about the name of the variables, the value assigned to them or the return value in a function.

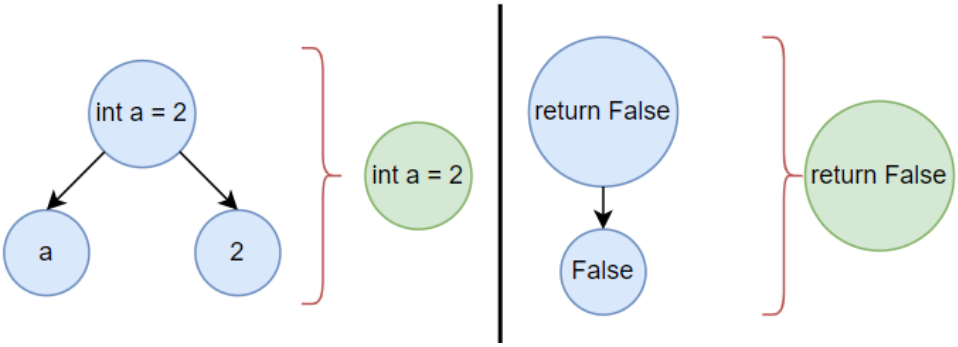

Figure 6. AST simplification to Logic Nodes.

2.2 CFG generation

Once the LNs have been obtained, the CFG is generated. This graph only contains the information about the logic of the code under analysis. In this way, it is possible to have the entire control flow of the whole system to analyse, regardless of the number of files it contains[5].

The CFG is created from the union of the LNs. To do this, the LNs, which are initially partially interconnected with each other, use the system information to complete the possible relationships existing between them. The relevant information that has been taken into account when creating the CFG are the nodes that affect the logic of the program as those structures that imply a conditional statement or loop, such as *if-else*, *while,* or *for* between others. In addition, for this tool, the calls and outputs to functions, such as instances to objects and methods, have also been taken into account in order to explore the logic of the program in its entirety. This inclusion will be useful when extracting the propagation path.

It is notice to mention that one of the innovative points in the development of this tool is the integration of the possible exceptions within the CFG. When programming, the use of exceptions to manage the logic of a program is not a good practice, which means that this functionality is not considered when generating the CFG. However, many developers include exceptions to handle the logic of their programs. For this reason, it has been decided to include these practices within the CFG since exceptions transparently verify to the developer if there has been an error and can affect the logic. This ensures that no logic is left unrepresented within the final CFG.

If the program to analyse has a main function, the CFG shows two auxiliary nodes that indicates the beginning, called *entrypoint,* and the end of the execution, called *stop*. If did not have any executable, (e.g., there were only methods for another executable file to use them) the CFG does not show the two auxiliary nodes.

After the generation of the CFG, a file with the results is provided to the user in three different formats:

- A *JSON* format, that provides a lightweight data interchange format. This structure is easy for machines to parse and generate. *JSON* is a text format that

[5] https://www.sciencedirect.com/topics/computer-science/control-flow-graph

is completely language independent. This format shows the CFG in a simple way for both humans and machines[6].

- A *PDF* where the CFG is reported along with arrows that relate each of the nodes and indicating the path that could follow. For example, in the conditional sentences, it is indicated whether or not the condition is met and the node that is executed.
- *DOT* which uses the *graphviz*[7] library and is used as a standard representation to generate and represent graphs whose format is *.dot*. This format makes layered drawings of directed graphs. The layout algorithm orients the lines in the same direction and tries to avoid line crossings and reduce the length of the lines. A text representation can be converted from a text representation to a *pdf* format document through this library.

An example of a *PDF* file of a CFG is shown in Figure 7.

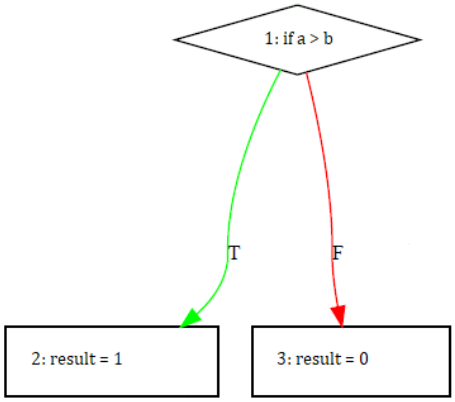

Figure 7. Simple example of CFG.

In this example, it is shown a node with a conditional *if* statement that has the comparison of two variables *a* and *b*, from which two other child nodes are derived. Each child node is executed depending on whether the condition is satisfied or not. If the condition is satisfied, the execution follows the green arrow (indicated by the letter *T* coming from *True*) and node 2 is executed assigning to the result variable a value of 1. If the condition is not satisfied, node 3 is executed assigning to the result variable a value of 0. The information of the arrows when a condition exists (*T* or *F*) is also reflected in the AST, but not as clearly as it is shown in the CFG.

2.3 Propagation path

The last phase of the process followed by the propagation tool is to obtain the propagation path. This path is determined from the main node to the location of the

[6] https://www.json.org/json-en.html
[7] https://graphviz.org/docs/layouts/dot/

vulnerability, i.e., the control flow that follows the execution of the program from the beginning until it reaches the node containing the vulnerability will be shown.

To obtain the propagation path, it has been designed a tool that pointed the node that contains the vulnerability and runs backwards until it reaches the first node of the program. In the case of containing an executable, the path would run from the vulnerable node to the initial *entrypoint* node and covers the affected nodes. If it is not possible to reach the initial node because of the large number of affected nodes, the maximum number of them will be shown starting the reverse path from the vulnerability.

The propagation tool will show all the paths that the logic of the program has to follow in order to reach the vulnerability, since what is beyond the vulnerability may never be executed once exploited. The developer could then detect the issue and see what preventive measures to implement in the program in order to mitigate this vulnerability and thus prevent the system from being exploited.

The process to follow for obtaining the propagation path is to traverse from the vulnerable node to the first node of the CFG based on the program. The nodes traversed in this third phase will have an active vulnerable field, so all the nodes that have this field active will be the ones that appear in the path. The rest of the nodes will be omitted as they are not part of the identified vulnerability. In this way, a reduced CFG would be obtained. This output would be provided in three different file formats indicated in the CFG section, i.e., JSON, *PDF* and *DOT*.

The implementation and development of the propagation path will be the focus of the D3.6.

3. Dependability measurement

Current systems are made up of different interconnected components that work together to provide a service. A *component* is defined as *a static building block of a system which can be a module, a class or interface, a package, or a subsystem* [6]. Figure 8 shows two small systems (X and Y) composed by subsystems (subsystems 1, 2, 3 and 4). Each subsystem has classes, and each class has operations that may call other classes. The figure also shows the dependencies among them. In this context, a failure in one of the system components, may have cascade effects over other components, or even produce a generalised system failure. Therefore, analysing the existing dependencies among the system components and its degree, can help to determine the impact that a vulnerability would have over the rest of the system components.

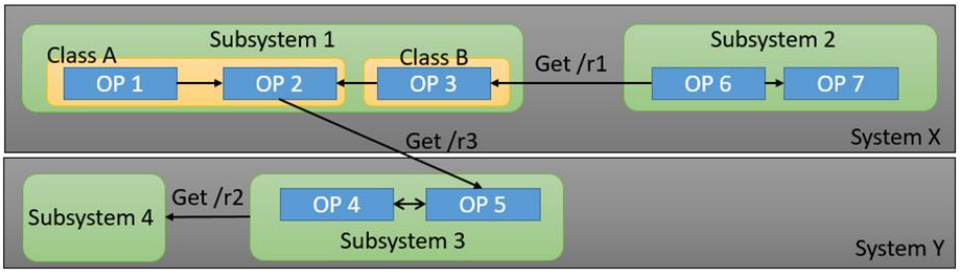

Figure 8. Example of scenario with dependencies

A dependency is *the quality or state of being influenced or determined by or subject to another* [7]. Then, a dependency of component A on component B exists if component A requires component B to compile or function correctly. It is worth noting that the dependency relation is transitive. If component A depends on component B which depends on component C, then there is an indirect dependency of component A to component C. In Figure 8, we can infer that subsystem 1 depends on subsystem 3 and 4.

The degree of dependency (or coupling) is a measure of the probability of changes among dependent components, so the stronger the dependency, the higher the probability. Therefore, the degree of dependency among each software component will also depend on how these modules are interconnected. Indeed, a certain vulnerability in a software library could be more or less exploited depending on the use of the library by the system.

Additionally, in an increasingly interconnected world, the security of a certain software component could be influenced by the security level of a system which is communicated with. In fact, a system's security level may be reduced if it needs to communicate with a vulnerable system for its intended operation. Therefore, software composability aspects go beyond the usual intra-system vision.

Taking this into account, we measure the degree of dependency of a subsystem considering two dimensions, as stated in Figure 9: the internal dependencies (What happens if something fails inside?) and the external dependencies with other subsystems (What happens if the other component fails?).

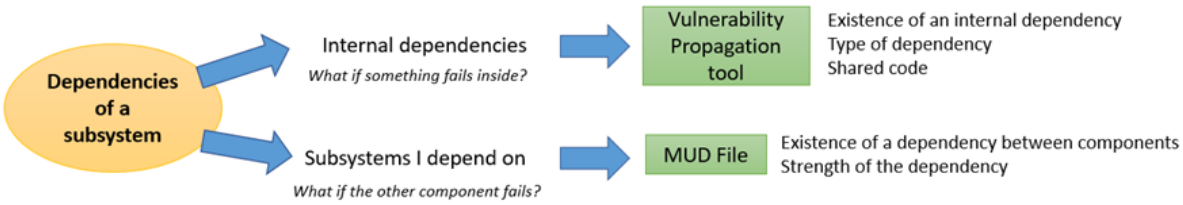

Figure 9. Dependencies of a subsystem

3.1 Internal subsystem dependencies

Given two classes, interfaces or modules A and B, the degree of dependency between them, $\delta_{A\to B}$, can be derived using the following formula,

$$\delta_{A\to B} = \frac{\varphi_{S_{A|B}}}{\varphi_{S_B}} \varepsilon_{A\to B} \in \{x \in \mathbb{R}^+ | 0 \le x \le 1\}$$

Where $\varphi_{S_{A|B}}$ is the amount of code shared between A and B (lines of code, LOC), φ_{S_B} is the total code of B (LOC) and $\varepsilon_{A\to B}$ is a factor between 0 and 1 based on the type of relationship they have.
$\delta_{A\to B}$ ranges between 0 (no dependency exists) and 1 (maximum degree of dependency).

It is worth noting that dependency degree is proportional to the probability of mutual changes between components

$$\delta_{A\to B} \propto P(Bmod|Amod)$$

Then, the total degree of dependency of A will be

$$\delta^{A}_{total} = \frac{1}{n}\sum_{C_j \in C_1,..,C_n} \delta_{A\to C_j}$$

Where C_j is the jth class (interface, module or package) A depends on.

Finally, the internal dependency of the subsystem N ($intD_N$) composed by r classes (interfaces, modules or packages) will be:

$$intD_N = \max_{0\le i\le r}\{\delta^{A_i}_{total}\}$$

The input data needed for the calculation of the internal dependency will be obtained from the propagation tool analysis, as shown in Figure 9, which will be the focus of the D3.6.

3.1.1 Type of relationships

As it was previously stated, $\varepsilon_{A\to B}$ is a factor between 0 and 1 based on the type of relationship two components have. Figure 10 shows the type of relation that two classes can have [8]. For interfaces, modules or package, the classification is similar, but some relationships may not be possible e.g., inheritance.

Figure 10. Strength and type of relationships

The stronger relation they have, the greater the factor $\varepsilon_{A\to B}$ is:

- **Dependency**: Weakest form of relationship. When objects of one class works briefly with objects of another class. It is limited in time (e.g., execution on one method) and limited in shared code.

```
class A {
        public A() { /* ... */ }
        public void methodA() { /* ... */}
}

class B {
        public   void   methodWithAParam(A
param) {
                a.methodA();
        }
        public A methodThatReturnsA() {
                return new A()
        }
}
```

- **Association**: When objects of one class works with objects on another class for some prolonged amount of time. It means that a class will contain a reference to an object, or objects, of the other class in form of an attribute.

```
class A {
        private B b;
        public A(B b) { this.b = b;
}
        // Other methods of class A
}
class B {
        public void method1() { /*
... */ }
        public void method2() { /*
... */ }
        public void method3() { /*
```

- **Aggregation and Composition**: These are stronger version of the association. In aggregation one class own but shares a reference to objects on another class. An aggregation states that one type *owns* the other, which means that it is responsible of its *creation* and *deletion*. Composition happens when one class contains objects on another class. These relationships generate a strong dependence between types, because one type has to know how to build an instance of the other.

```
class B() {
     private A a;

  // …
class A {
    // ...
  }
```

- **Inheritance**: It is the strongest type of dependency. A child class inherits and reuses all of the attributes and methods that the parent contains and that have public, protected, or default visibility. Therefore, any change to the parent can disrupt its children.

```
class A {
       public A() { /* ... */ }
       // Other methods of class A
}
class B extends A {
       public B() {
       super();
       // ...
}
// Other methods of class B
}
```

3.2 External subsystem dependencies

We call external subsystem dependencies to the relationships with other subsystems or systems. In this case the dependencies are not relates to the code shared, but to the services offered, and the relationships are performed through network accesses. As an example, Figure 11 shows an example of external dependencies. In this case, Subsystem 2 needs to access to subsystem 1 to get some resources (r1 and r5), that could be though an HTTP connection. Subsystem 2 also offers services to Subsystem 4 and 5 (resources r2 and r3).

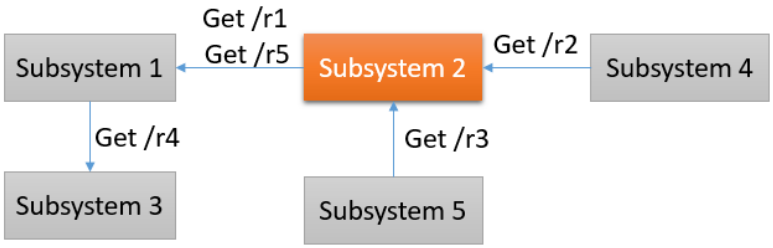

Figure 11. Example of external dependencies

The input data needed for the calculation of the external dependencies will be obtained from the MUD file analysis, as shown in Figure 9, which will be the focus of the D3.6.

3.3 Total subsystem degree of dependency

As advanced before, we calculate the degree of dependency of a subsystem based on the internal (software) and external (network) dependencies. We follow a recursive approach based on the dependency graph obtained from the external dependencies. Figure 12 shows an example of this recursive approach based on the graph from Figure 11. In this case, subsystem 2 depends on its internal dependencies and on the subsystem 1, which at the same time depends on its internal dependencies and on subsystem 3.

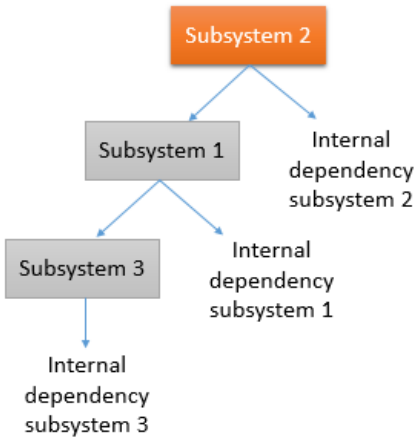

Figure 12. Recursive approach for dependency calculation

Therefore, we can measure the total degree of dependency of a subsystem N as:

$$D_N = \max_{0 \le i \le n} \{intD_N, D_1, .., D_s\}$$

Where $D_1, .., D_s$ are degree of dependency of the subsystems that the subsystem N depends on. We selected the max function instead of the mean because choosing the mean would devaluate dependencies very sensitive, while the max function always states in the worst case.

While further research will be performed to calculate the degree of dependency, this value is intended to be used as the "sensitivity" parameter of the D7.2 security evaluation methodology, as a modulator of the security risk associated to the subsystems composing the system under test.

4. Conclusions

Within the BIECO framework, it is proposed the development of a propagation tool, in order to help the user to increase confidence in the system. The objective of this tool is to provide those parts that may be affected by a vulnerability in the source code. For this, a tool has been designed which, after providing the location of the vulnerability within the system, as well as the code of the code of the program, is capable of generation a control flow from the beginning of the execution of the program until reaching the vulnerability, in other words, it will provide the vulnerability path in the code.

The presented propagation tool has been divided in three phases for its development: AST extraction, CFG generation and propagation path. When it comes to the extraction of the AST it has been already implemented for both Java and Python languages, being processed the resulting AST to homogenize the output regardless of the initial language. In the second phase, a CFG has been formed taking as references the previously formed structures of the code. This representation contains in a simplified way all the necessary code information, offering a "map" of it by means of nodes. This will serve to indicate the path that a vulnerability follows to the principal node.

To complete the tool, as future actions, it is intended to generate the AST for C code language together with its standardization process for the creation of the CGF. Furthermore, the last phase of the tool, obtaining the propagation path, will be developed by performed on the elements of the CFG, indicating which nodes of the graph are related to the identified vulnerability or not. The result obtained from the tool will serve, not only to prevent a possible exploitation of a vulnerability, but also as input to different vulnerability assessment tools developed in BIECO.

As a complementary result, the deliverable also presented a methodology to calculate the degree of dependency of a subsystem based on its internal and external dependencies. This methodology is intended to be used as input for the WP7 security evaluation methodology, calculating the sensitivity parameter to modulate the risk of a subsystem. We will explore how to connect the methodology with the outputs of the propagation tool (internal dependencies) and the behaviour specified in the MUD file (external dependencies).

5. References

[1] G. McGraw, "Software Security", Building Security In, 17th International Symposium on Software Reliability Engineering, 2006, pp. 6-6

[2] Deliverable D3.3 "Report of the Tools for Vulnerability Detection and Forecasting"

[3] Deliverable D3.1 "Report on the State of the Ert of Vulnerability Management"

[4] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modelling and Discovering Vulnerabilities with Code Property Graphs", 2014 IEEE Symposium on Security and Privacy, pp. 590-604, 2014.

[5] S. Suneja et al. "Learning to map source code to software vulnerability using code-as-a-graph", arXiv preprint arXiv:2006.08614, 2020.

[6] S. Jungmayr, "Testability measurement and software dependencies", In Proceedings of the 12th International Workshop on Software Measurement. vol. 25, no. 9, pp. 1799-202. 2002

[7] L.G. Yu and S. Ramaswamy, "Component dependency in object-oriented software", Journal of Computer Science and Technology, vol. 22, no. 3, pp. 379-386. 2007

[8] R. Cardin, "Dependency management", Ingegneria del software, Università degli Studi di Padova, Dipartimento di Matematica. Corso di Laurea in Informatica

Annex A: Example of the propagation tool process

An example of the propagation tool process is shown in Figure 13. First, the source code of the program to be analyse is provided as input to the tool. When executing the tool, the AST of the code in text format is obtained with detailed information about its structure. The CFG is executed with the AST obtained, where the union of all the ASTs with the relevant information for the user is provided. The graph is created with the nodes related to each other. Finally, the propagation path is indicated in the previously created CFG.

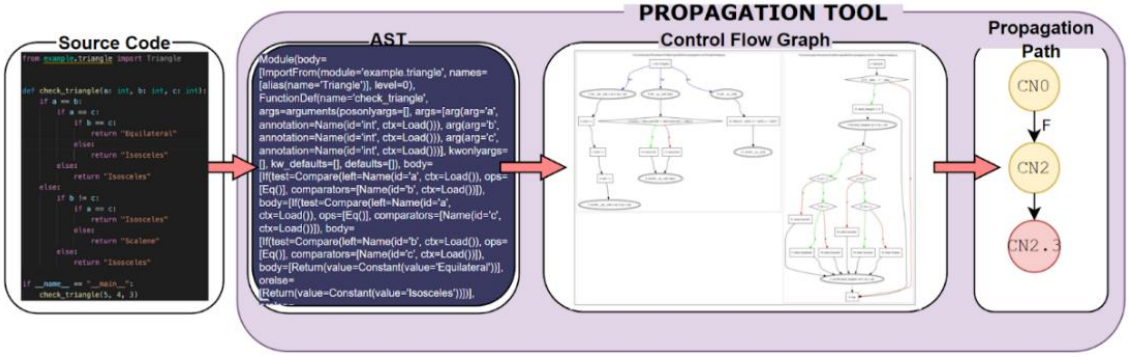

Figure 13. Example of the process of Propagation Tool.