UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

Unikraft Security Audit Using Static Code Analysis

Carol-Sebastian Bontas

Thesis advisor:

Conf. dr. ing. Razvan Deaconescu

BUCHAREST
2024

UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

Audit de Securitate in Unikraft Folosind Analiză Statică de Cod

Carol-Sebastian Bontas

Coordonator științific:

Conf. dr. ing. Razvan Deaconescu

BUCUREȘTI
2024

CONTENTS

SINOPSIS

Aceasta lucrare ofera o solutie in cautarea automata de vulnerabilitati in cadrul aplicatiilor de tip unikernel folosind diverse produse de analiza statica a codului. Autorii identifica o problema ce se refera la acoperirea codului compilat respectiv analizat si care impiedica folosirea directa a uneltelor de analiza statica de cod pe un unikernel, propun un algortim ce rezolva aceasta problema si implementeaza o aplicatie care realizeaza compatibilitatea intre sistemul de compilare din Unikraft si analizatoarele statice de cod. In final, folosind aceasta procedura imbunatatita de scanare se identifica o vulnerabilitate gasita si se experimenteaza exploatarea ei.

ABSTRACT

This work offers a solution for the automatic search for vulnerabilities in unikernel-type applications using various static code analysis products. The authors identify an issue related to the coverage of the compiled and analyzed code, which hinders the direct use of static code analysis tools on a unikernel. They propose an algorithm that solves this problem and implement an application brings compatibility between the Unikraft compilation/build system and static code analyzers. Finally, using this improved scanning procedure, a vulnerability is identified and its exploitation is experimented upon.

1 INTRODUCTION

The accelerating demand for software products and the diminishing development time resulted from it may unfortunately negatively introduce lesser time for writing high-quality and secure code. In order to solve this emerging issue, automatic measures have been invented in order to analyze the quality of the code created by the human factor. In this category we can find the various static analysis algorithms.

However some questions remain regarding the efficiency of such methods and if they are more powerful than a human's eye for security exploits.

Our work is a first ever attempt to integrate static analysis tools in the quality assurance of a unikernel project, more specifiacally Unikraft.

We have identified one specific problem that is caused by the unique architecture of unikernels and solved it through a novel algorithm.

Moreoever, we have implemented an application called Unikraft Scanner that will automatically manage the whole process of security assessment of the Unikraft product.

Finally, we run various experiments to understand the costs of running such application. From a security research point if view, we also identify a potential vulnerability with high impact that may have stayed hidden from the eyes of the Unikraft community.

The work here is structured as follows: the State-of-the-art where we explain various static analysis strategies and algorithms; the Related Work where we mention existing products that push the domain of static analysis even further, together with researched comparisons between them. In addition, we also present detailed aspects regarding the Unikraft architecture that are crucial in order to understand the underlying issues. Afterwards, we present the architecture and interesting aspects of the Unikraft Scanner implementation.

2 STATE OF THE ART

Automatic methods of code analysis are not a novel concept as there are numerous attempts dating decades ago such as inferring the correctness of algorithms or automatic theorem proving systems [2]. However, these methods shifted their purposes from their original academic objectives to more practical approaches such as vulnerability assessment or imposing a certain standard of coding style.

Due to the fact that vulnerability finding is an undecidable problem [4], an algorithm which signals all known and unknown flaws can be Turing-reduced to the halting problem. This theoretical limitation proves that in order to leverage the advantages of static analysis, we need a human deciding factor at the end of the reviewing process in order to solve false-positive alerts and review potential false-negatives.

According to the Kroening's et al survey [3], static analysis has 4 major paths from which it can start extracting data and do automatic checks :

- **abstract interpretation** : it approximates the behavior of the program and concentrates its meaningful parts into a finite-state machine
- **data-flow analysis** : recreates the control-flow graph and traverses all basic blocks in certain ways in order to detect normal behavior features such as variable constraints
- **symbolic execution** : user chooses variables from critical contexts and the tool builds all the mathematical constraints in order to access every path of the control-flow graph
- **model checking** : used in proving the correctness of a program by modeling it as a finite state machine and tracing all variables, stack and heap with transitions set as logical propositions

There are also other methods which have evolved from these major paths and specialized on certain programming languages or usage scenarios (aliasing and pointer analysis, shape analysis, escape analysis pattern-matching etc.). What we can observe is the wide variety of data that we can collect from source code or compiled binaries.

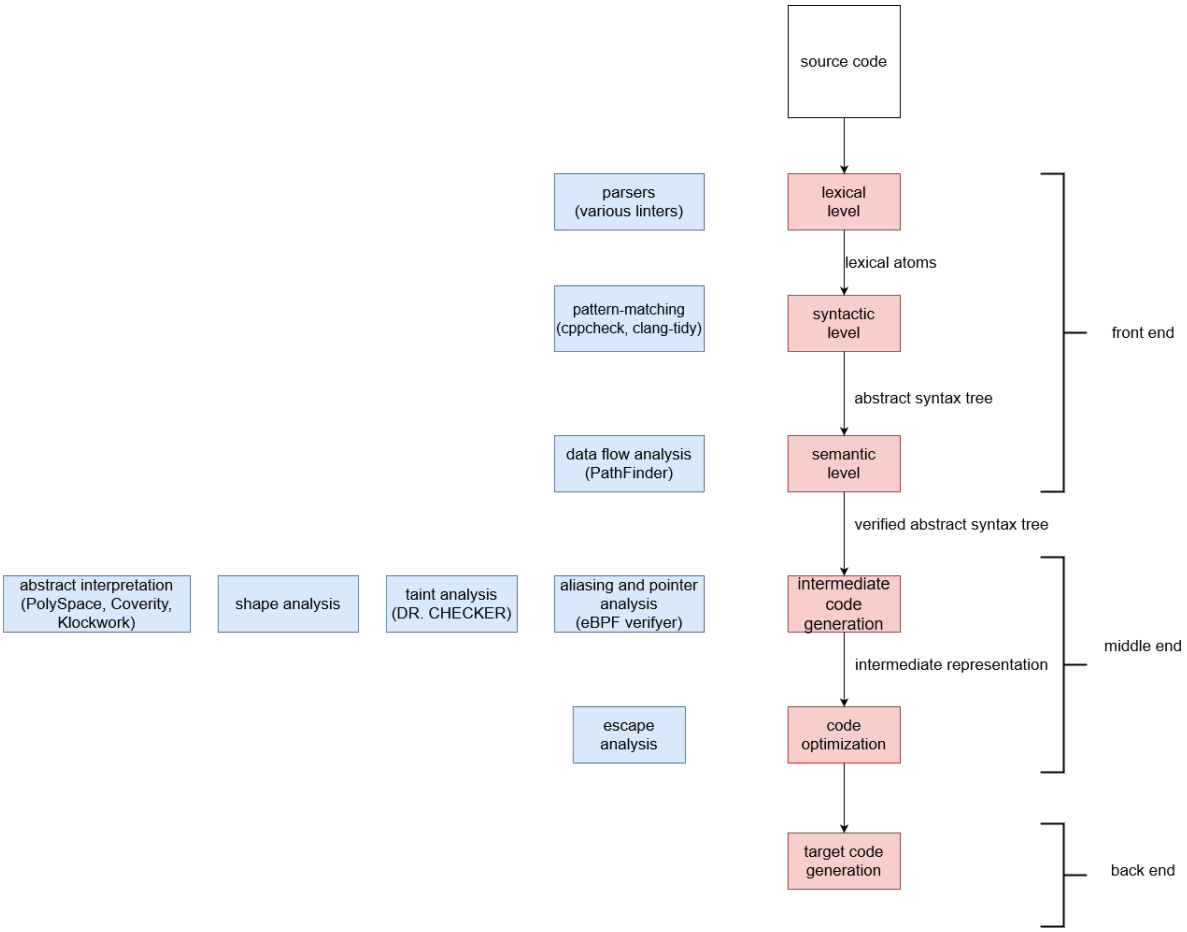

Figure 1: Taxonomy of various static analysis algorithms compared to the stage of the compilation process

3 RELATED WORK

Static analysis algorithms should be chosen carefully according to the respective scenario. Small and frequent code commits do not require aggressive static analysis techniques but light-weight lexical or pattern-matching solutions in order to ensure coding style, while a potential software release should use more complex methods which bring generous time costs.

In addition to the proportionality principle (big software releases require aggressive scanning algorithms), static analysis can also help achieving other objectives rather than vulnerability research (assessing coding style or discovering logical errors). For example Vassalo's et al survey [7] presents the most important aspects monitored during 3 different phases :

1. **local development** : *code structure* and *logic* (13.8% of responders stated to be the most important aspects)
2. **code reviews** : *style conventions* (16.9%), *redundancies* (14.3%) and *naming conventions* (11.7%)
3. **continuous integration** : error handling (19%) and logic (12.7%)

Since their undecidability limitation, static analyzers do not represent a fault-proof solution. Each tool creator provided their own heuristics which are prone to either false negatives or false positives. In the comaprison below we have identified 5 software products which pushed the domain of static analysis further.

1. **PolySpace**
    - **87%** detection rate of buffer overflows according to Ziter et al experiments [8] on simplified code samples from WU-FTPD server and SendMail agent
    - **99.7%** detection rate on a simple but diversified test case of buffer overflows spanning accross 22 different types, according to Kratkiewicz et al [5] experiments
    - uses abstract interpretation, more specifically convex polyhedra [1] in order to approximate the relationship between variables and pointers
    - uses taint analysis and pointer aliasing solvers [4]
    - used in checking embedded software such as NASA's Rovers; proves a certain degree of trustiness and robustness [3]
2. **Coverity Prevent/Klockwork K7**
    - uses abstract interpretation and taint analysis [4]
    - support for writing custom made analyzer extensions
    - Coverity Prevent detects **2 times more** bugs than Klockwork K7 while the latter flags **10 times less** false positives based on the same tested source code [4]

3. **Dr. Checker**
    - **78%** detection rate of vulnerabilities in 9 different vulnerable mobile driver software [6]
    - found **158 zero-day** vulnerabilities throughout its use [6]
    - uses pointer and data flow analysis

3.1 Unikraft and Its Build System. Slight Incompatibility with Static Code Analysis

The static analysis tools such as CodeQL or Coverity, presented in the previous sections, solve most of our requirements regarding the vulnerability scanning. Our work does not present a novel analysis technique or algorithm but finds a way for unikernels, such as Unikraft, to be analyzed by static analysis tools. Usage of such tools, in this scenario, is not as straightforward as in the case of scanning a traditional application.

Static analysis tools are strongly coupled by the compilation process of the code sample that is being analyzed. For example Coverity user manual mentions that the tool uses a proxy that intercepts calls, started by the build system of the analyzed project, to the compiler and extracts crucial information about the source code, the control flow graph, compiler configurations and what data types are being used in the code[1] [2]. CodeQL has the same methodology in intercepting or capturing the compilation process in order to fetch the abstract syntax tree (AST) of the targeted program. Using the abstract syntax tree, it is possible to express queries that describe a vulnerability as a pattern involving various AST tokens and classes.

The incompatibility that has already been mentioned stems from the difference between the compilation process of traditional applications and the Unikraft unikernel.

In the wide majority of the cases, a traditional application will have all of its source code compiled in order for the final resulting executable to be used. There is no real use case in having regions of code that will not be compiled or their compilation status being conditioned by certain software requirements. A possible counter-argument that might explain existing code regions that are not compiled but still present in the codebase would be the low-level software logic that is constrained by the operating system platform (for example program logic that uses Windows-based system API calls and found inside a Unix application codebase). However such idea is easily negated by the modern transition of high-level languages towards portability. Platform-dependant code is moved from the application layer towards library and runtime level. Since low-level libraries and runtime architecture is a trusted and secure third-party code addition to a said project, only the application layer needs to be put through the

[1] https://sig-product-docs.synopsys.com/bundle/coverity-docs/page/commands/topics/cov-build.html

[2] https://sig-product-docs.synopsys.com/bundle/coverity-docs/page/coverity-analysis/topics/running_an_analysis_without_cov-build.html

static analysis process.

However, unikernels such as Unikraft, do not represent a standalone application but provide low-level modules represented by one or more C source files that are chosen, or not, to be included (compiled and statically linked) together with the ported high-level application. The only criterion that is used for module selection is the ported application's critical requirements.

Figure 1 shows that during compile time only some parts of all software stacks are chosen to be compiled (shown as shaded segments) and featured in the final executable. Understanding this aspect allowed us to perform a small experiment in order to observe the whole building and compilation pipeline.

The experiment starts by building the ported version of SQLite version 3.40 [3]. Choosing what Unikraft modules are compiled or not can be achieved by interacting with a graphical user interface (GUI) that is a human readable wrapper over the kconfig system [4]. Figure 2 shows how libraries/modules are chosen to be compiled. A very good example of choosing at compile time what modules are included can be found in the menu that allows users to choose the instruction set architecture and virtualization platform. Having multiple architectures and platform at the same time in the final executable is redundant.

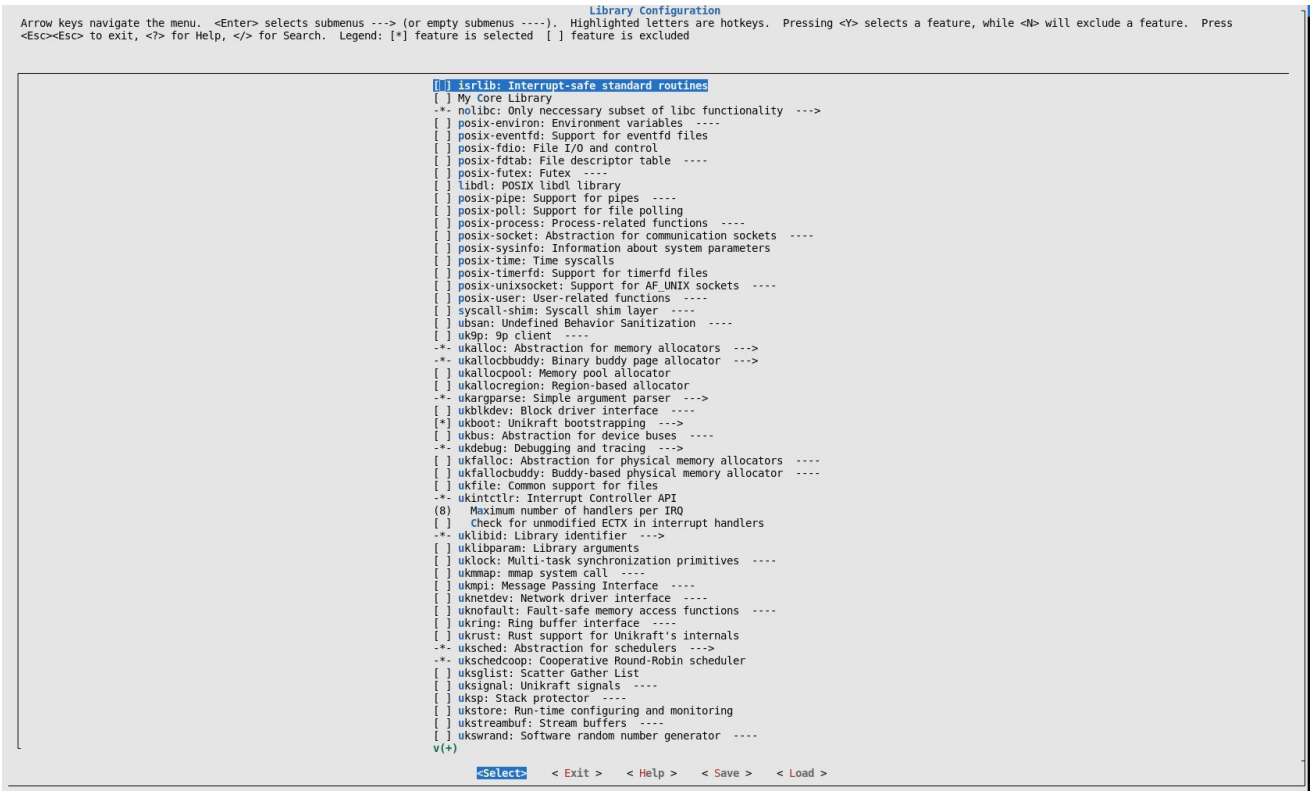

Figure 2: Library modules menu

[3] https://github.com/unikraft/catalog/tree/main/library/sqlite/3.40
[4] https://www.kernel.org/doc/html/next/kbuild/kconfig-language.html

Going further than the graphical menu, a special configuration file is generated which contains C preprocessor symbols, showcased in Figure 3. As observed some symbols enforce a binary response (Yes/No) while others are not defined. These symbols are used in order to condition compilation of various code regions through C preprocessor directives (**#if, #ifdef, #ifndef, #elif, #else**).

```
# CONFIG_MARCH_X86_64_AMDFAM10 is not set
# CONFIG_MARCH_X86_64_BTVER1 is not set
# CONFIG_MARCH_X86_64_BDVER1 is not set
# CONFIG_MARCH_X86_64_BDVER2 is not set
# CONFIG_MARCH_X86_64_BDVER3 is not set
# CONFIG_MARCH_X86_64_BTVER2 is not set

#
# Processor Features
#
CONFIG_X86_64_HAVE_RANDOM=y
# end of Processor Features

CONFIG_STACK_SIZE_PAGE_ORDER=4
CONFIG_CPU_EXCEPT_STACK_SIZE_PAGE_ORDER=4
CONFIG_AUXSTACK_SIZE_PAGE_ORDER=4
CONFIG_HAVE_RANDOM=y
# end of Architecture Selection

#
# Platform Configuration
#
CONFIG_PLAT_KVM=y
CONFIG_KVM_BOOT_PROTO_MULTIBOOT=y

#
# Hint: EFI stub depends on OPTIMIZE_PIE
#
CONFIG_KVM_VMM_QEMU=y
# CONFIG_KVM_VMM_FIRECRACKER is not set

#
# Console Options
#
CONFIG_KVM_KERNEL_VGA_CONSOLE=y
CONFIG_KVM_DEBUG_SERIAL_CONSOLE=y
CONFIG_KVM_DEBUG_VGA_CONSOLE=y
CONFIG_KVM_KERNEL_SERIAL_CONSOLE=y

#
# Serial console configuration
#
CONFIG_KVM_SERIAL_BAUD_115200=y
# CONFIG_KVM_SERIAL_BAUD_57600 is not set
# CONFIG_KVM_SERIAL_BAUD_38400 is not set
# CONFIG_KVM_SERIAL_BAUD_19200 is not set
```

Figure 3: Resulting symbols configuration file

Regarding modules, the C source files that contain the Unikraft core logic together with its internal libraries are compiled based on the chosen symbols from the previous step. External code representing the ported application is fetched from remote repositories and compiled separately. Resulting object files are then linked. All these steps are shown in Figure 4 that was reproduced from the Unikraft documentation[5].

[5]https://unikraft.org/docs/internals/build-process

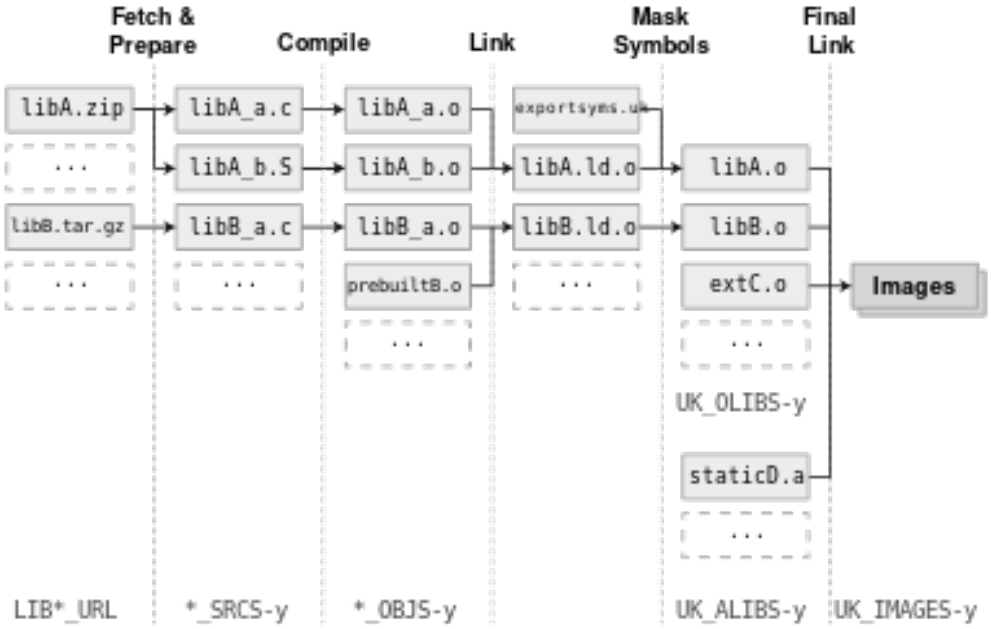

Figure 4: Unikraft build pipeline

From these documented steps and observations we can conclude that:

- Unikraft configuration system can choose what C source files are compiled.
- There is no guarantee that a chosen C source file is compiled as a whole since some symbols may condition the compilation of inner logic through preprocessor directives.

3.2 Compilation Coverage Problem. Defects May Bypass Analysis

New interesting findings have occurred during another experiment:

- We introduced in the **nolibc**[6] internal library an obvious **illegal memory access vulnerability** that would result in a **buffer overflow**.
- The **nolibc** module has not been chosen to be part of the final Unikraft and SQLite application since SQLite is using a more powerfull standard C library called **musl**[7].
- Introduced the same vulnerability in the **musl** module.
- Used Coverity to scan the SQLite-Unikraft application.
- The final results warned us that the vulnerability occurs **only** in the **musl** and not in **nolibc**.

[6] https://github.com/unikraft/unikraft/tree/staging/lib/nolibc
[7] https://github.com/unikraft/lib-musl

Coupled with the information presented in the previous sections regarding the static analysis tools that intercept the build process and the compilation particularities of the Unikraft codebase as an analysis target we can conclude: **The Unikraft code scanned by static code analysis tools is equivalent to the same code that is compiled. In order to analyze Unikraft as a whole we need to scan all possible "mutations" caused by a change in the symbol configuration.**

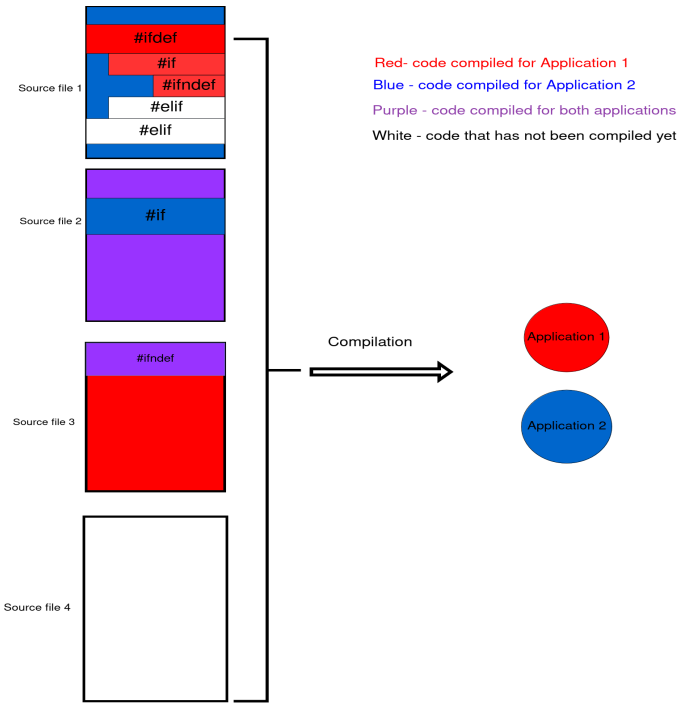

Figure 5: Compilation coverage problem in the context of 2 ported applications for Unikraft or between same application but with different compile-time configurations

Compilation coverage problem can be viewed as multiple compilations that intertwine on the same codebase. Numerous scenarios are pictured in Figure 5, where C source file can be compiled as a whole through only one configuration, some files can be partially compiled through one configuration while the rest through the other, both configurations can trigger common compilation background or none of the scenarios above since the source file is not part of an included module. As observed, the scanning step needs to be enhanced by a viewing step that shows developers that assess Unikraft's security what code regions were never compiled, hence never analyzed.

The order to achieve full scanning coverage, we either obtain a specific configuration for a ported application which includes all internal modules or we adopt an incremental methodology to sequentially scan various configuration.

4 ARCHITECTURE AND IMPLEMENTATION

Our work concentrated towards solving the compilation coverage problem by proposing and implementing an algorithm that has as input a configuration file or a build command and identifies fully or partially compiled source files together with code regions conditioned by any preprocessor directive.

The algorithm and the implemented application is a **collaborative** flow where multiple Unikraft developers submit various viable configurations together with their compilation process in order to increase, in an **incremental** manner, the global compilation coverage. Every submission is enhanced with possible defects found by one or more static analysis tools that have scanned only that particular compilation. The general architecture can be seen in Figure 6 and is fragmented in 3 major components:

- **Unikraft Scanner developer CLI tool**(highlighted in green): found in every developer's environment that participates in the Unikraft scanning experiment; runs the algorithm that establishes what code has been compiled from the entire Unikraft codebase; allows static analysis tools to intercept the compilation process; scrapes the found results from the Coverity Scan cloud platform; structures results and writes them to the a centralised database. In addition it provides a minimal graphical interface similar to the Unix 'tree' tool to view the global compilation coverage of Unikraft.
- **Coverity Scan cloud platform**(highlighted in purple): contains closed-source static code analysis algorithms; finds possible defects and shows the reasoning step-by-step in a user friendly interface; this is the source from which the local CLI tool will fetch results using web-scraping.
- **Centralised database**: contains and aggregates structured results from all registered iterations; manages a minimal authentication and authorization mechanism that links the Coverity account to the Unikraft Scanner pipeline.

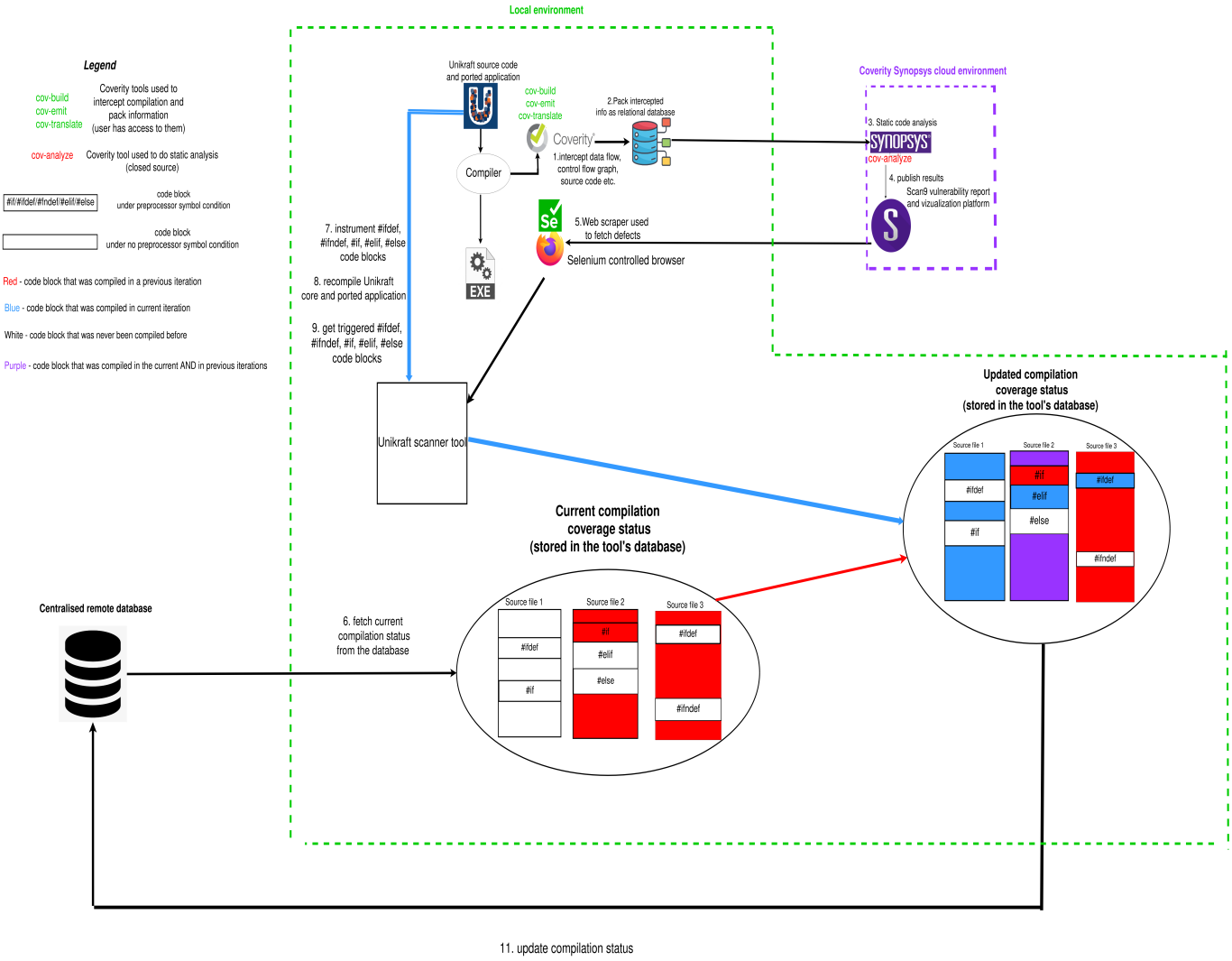

Figure 6: General architecture of the scanning infrastructure

4.1 Unikraft Scanner CLI Client Tool

The Unikraft Scanner is a CLI client application used by developers to increase both the compilation and statically scanned coverage. As the name suggests, its usage is through different commands issued through a console.

- **setup**: used for the initial installation of the tool and account creation
- **app**:
  - **add**: used to register a new compilation and trigger the static code analysis
  - **list**: print all compiled applications from all users
  - **view**: print compilation domain of one or more iterations given as input arguments
  - **delete**: deletes a stored compilation from the database
- **status**: print the global compilation state of the Unikraft codebase

Among these commands the **app add** operation is the most crucial, useful and complex feature of the application. It contains the algorithm that finds what source files are fully or partially compiled during the submitted build of a ported application for Unikraft. The in-depth sequential logical flow is presented below and can also be see in Figure 7:

1. Start a first compilation of the Unikraft application, process that is also intercepted by the static analysis tools.
2. The intercepted build is archived and sent to the Coverity platform for analysis.
3. Search through the build artifacts for all modules.
4. For every compiled module find all *.o.cmd files.
5. For every *.o.cmd, file extract its referenced C source file. The majority of such *.o.cmd files contain the compilation command of an individual source file.
6. For every C source file, parse it and find all code blocks that their compilation is conditioned by any **#if, #ifdef, #ifndef, #elif, #else**. In the end, obtain an interval tree of such code blocks even in nested blocks scenarios.
7. For every block, perform a code instrumentation by injecting at the beginning, a **#warning** directive with a unique ID as its message.
8. For every C source file, redo the compilation, this time using the individual compilation command found in its source_ij.o.cmd from step 3.
9. Inspect the **stderr** stream for messages issued by the injected **#warning** directives from step 4.
10. Fetch the current state of the Unikraft codebase from the centralised database and update it with the new compilation in an incremental way.
11. Wait for the Coverity platform to finalize scanning process and scrape results using the Selenium[1] framework.
12. Structure data and write it in the centralised database.

The complete algorithm follows 2 tracks: the left one from the Figure 7 explains how source files are processed by the application to find the compilation coverage while the one from the right aims at picturing the stage of fetching defects from the Coverity platform. Internals regarding how the defects are fetched are explained in the next section.

[1] https://selenium-python.readthedocs.io/

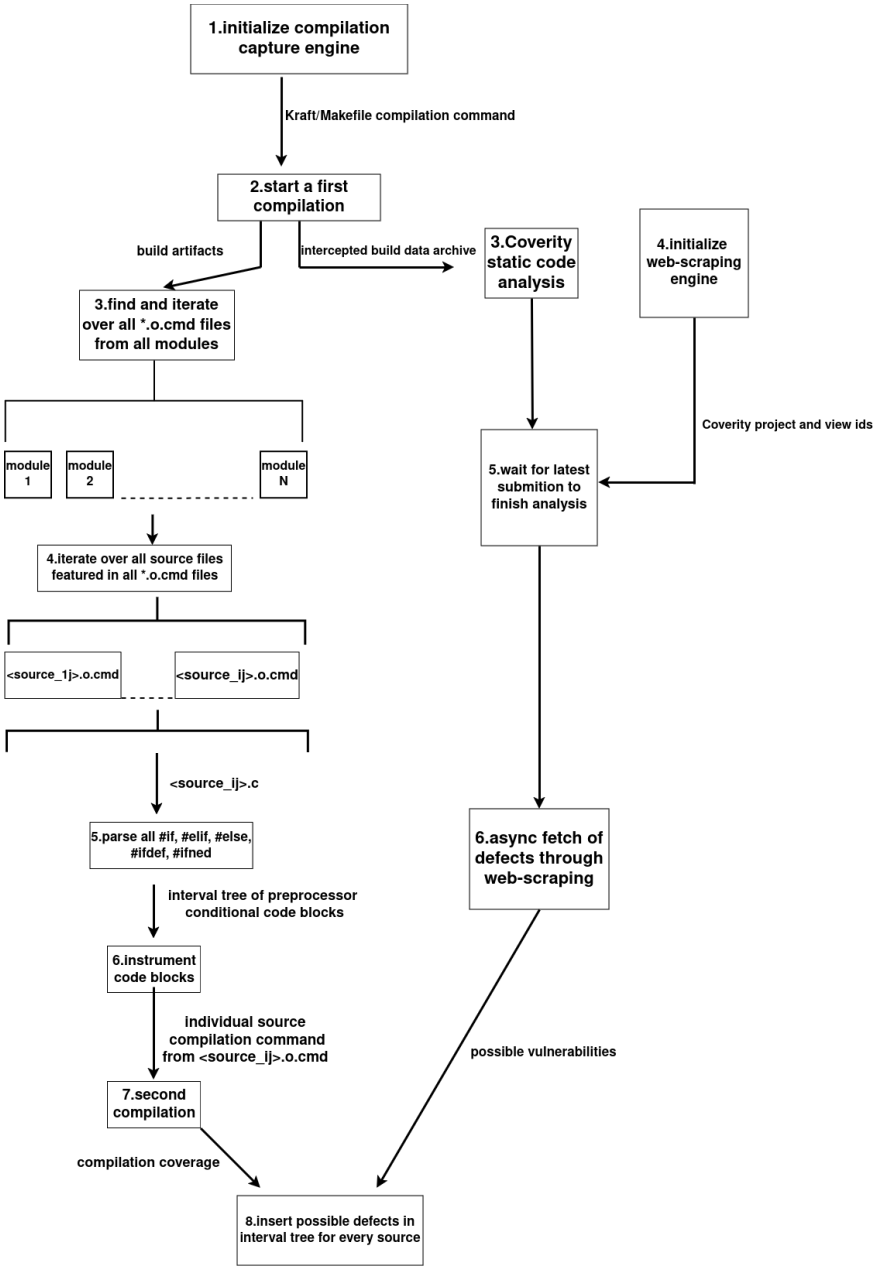

Figure 7: Flow representation of the command that adds a new compilation iteration

As observed in Figure 8, we have various nested code blocks with conditioned compilation (first block between lines 17-30 which includes block from 20-21 and block 22 - 26). In the right side of the figure, we have the same source file but instrumented with **#warning** directives at the beginning of every such code block.

Parsing and searching for these blocks is achieved through reading the source file line by line and building the interval tree using a depth-first search approach.

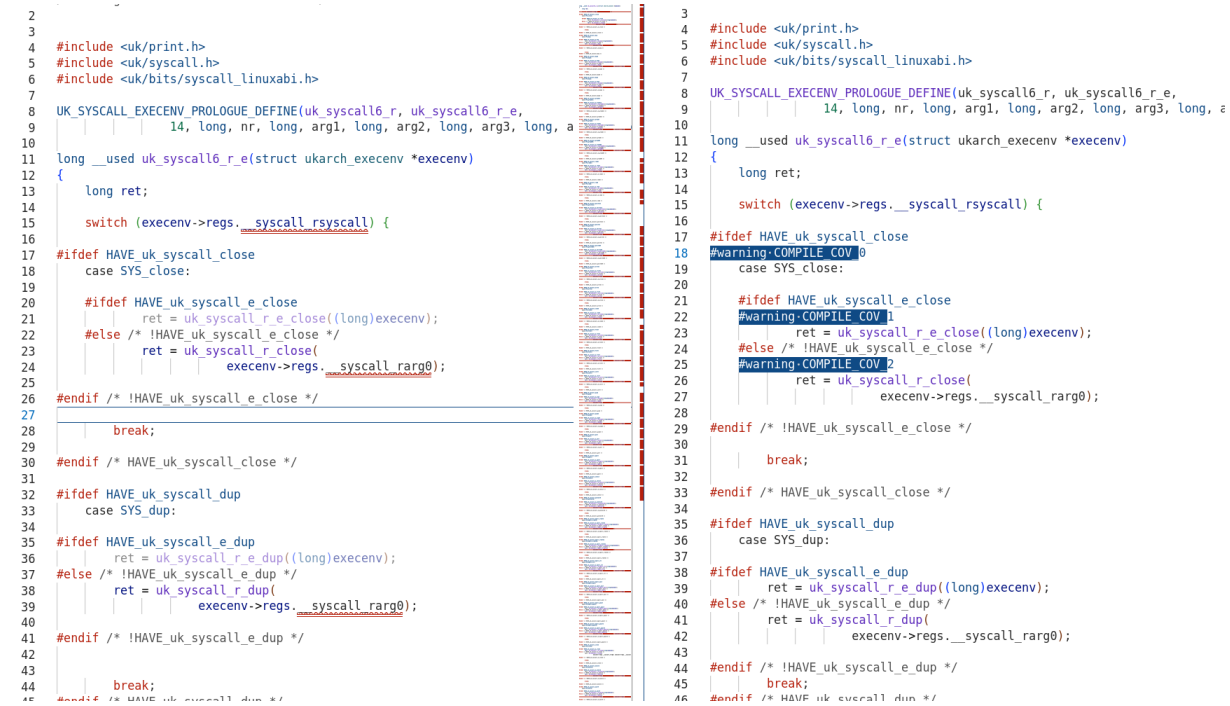

Figure 8: Instrumentation example in nested code blocks with conditioned compilation

After instrumenting the code, we recompile each found source file individually, by executing the compilation command from its corresponding *.o.cmd file. We are not invoking again the global build command that triggered the first compilation due to various side effects such as removing the instrumented codebase and cloning the code repositories again. By individually compiling the found source files we have a more granular view of what code blocks are triggered during the compilation by checking the standard error stream for messages issued by the injected **#warning** directive.

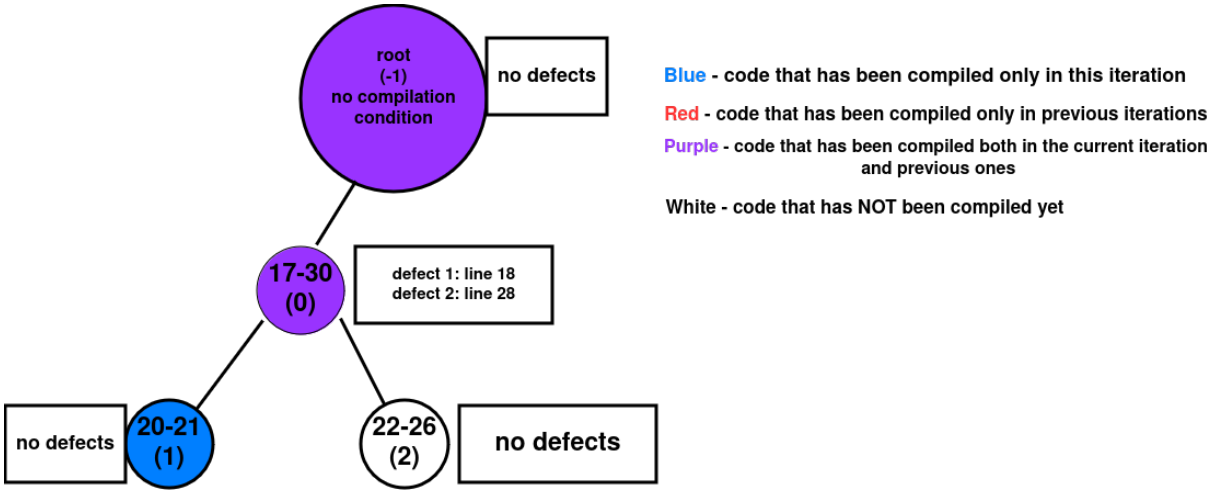

Figure 9: Interval tree example of a single compiled/analyzed source file. The file is the one presented in the previous Figure 8.

Once the application establishes the code blocks that are compiled based on the triggered messages in standard error, we adjust the interval tree featured in Figure 9. The interval tree presents those nested blocks as tree nodes that store the start and end lines and an array of found defects in the respective block. The root represents "universal" code that is not part of any compile-time conditional code block. Using the same notation as in Figure 6, the compilation coverage is marked here as well. Some code blocks have never been compiled before while others are compiled either in previous iterations or in the current one or in both. Obviously, a code block that has never been compiled before has no found defects yet and if there is at least one code block that has been compiled, the root must also be marked as compiled as well. Now, the only step before having the final result is the automatic download process of defects found by Coverity.

4.2 Fetching Analysis Results from Coverity through Web-scraping

We have opted for fetching static analysis results from the Coverity Scan platform in order to insert such defects in the interval tree representation of their corresponding source file, similar to Figure 9.

At the time of developing the app and writing our progress, Coverity Scan does not have any REST API for interacting with the analysis platform. Even though, the platform hinders our efforts to fetch defects and manipulate them more freely, it also provides us with valuable information regarding the reasoning of the analysis algorithm. The platform can be used during the manual inspection of found defects by an experienced developer. This step is crucial in order to triage the true and false positives.

As an observation, fetching defects is used in building the view of the compilation/analysis coverage, view that may alert developers of areas that are prone to generate possible vulnerabilities. Once a defect is chosen based on the information summary printed by the Unikraft Scanner, the developer or security specialist uses the Coverity platform for more detailed information (Figure 10) regarding how the defects have been found by the algorithm.

The feature of fetching defects was developed in multiple steps and suffered multiple refactoring phases due to blocking issues.

In the **first attempt**, a simple **HTTP client** was used to interact with the Coverity Scan platform. Using the Firefox Network Developer Tools[2], we tried to **intercept and understand the network traffic flow and cookie exchange logic** in order to achieve automatic authentication (Appendix A). Unfortunately, our attempts were not successful due to unknown issues regarding the emulated cookie logic or due to maybe missing HTTP headers.

[2]https://firefox-source-docs.mozilla.org/devtools-user/network_monitor/

Since Coverity recently introduced CAPTCHA challenges and we failed in emulating the HTTP dialogue only through plain HTTP requests, our **second attempt** was orientated towards **web-scraping**, more precisely in the usage of the **Selenium** framework and **Gecko-Firefox webdriver**.

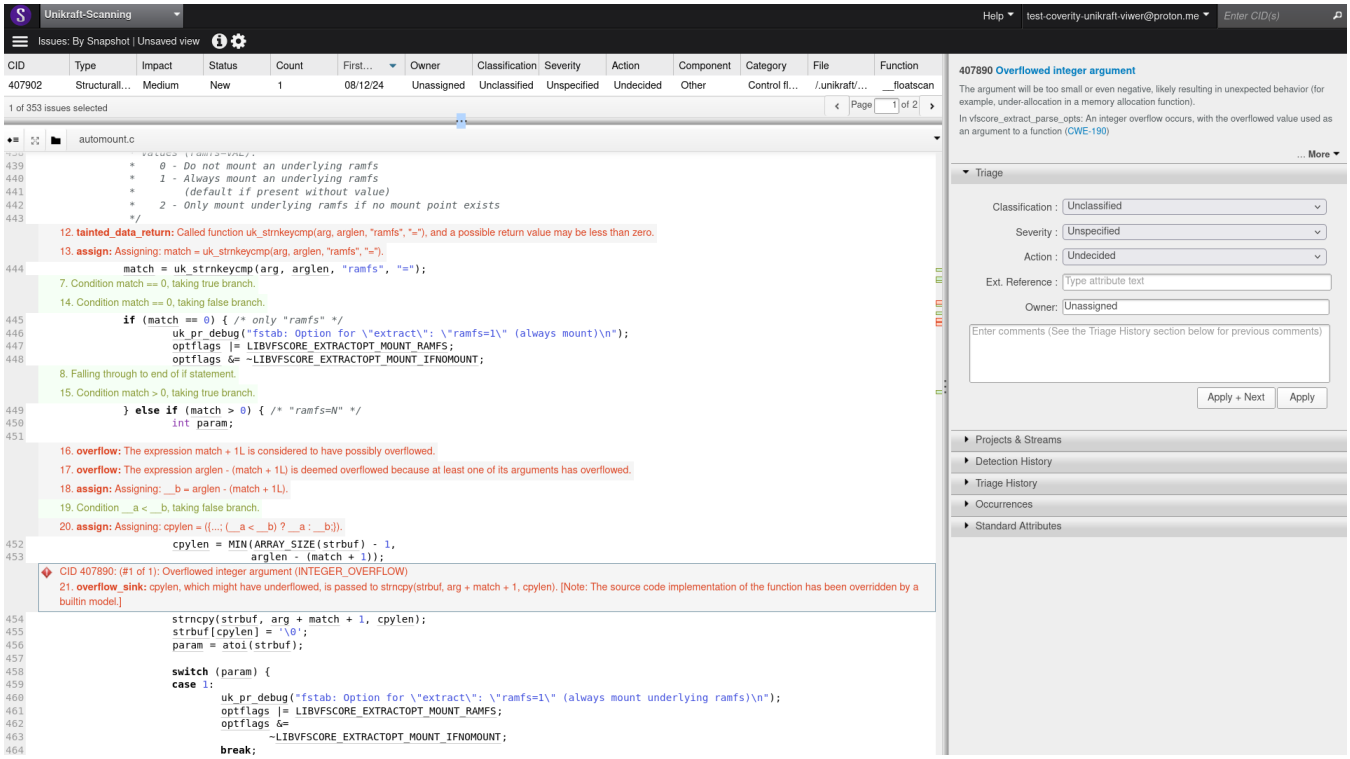

Figure 10: Algorithm reasoning steps in the Coverity Scan platform for a possible vulnerability

If we return to Figure 7, it is mentioned that the web-scraping engine (represented by a Firefox instance controlled by Selenium) will fetch 2 IDs, starting with the 4th step: project and view. These parameters are proprietary components of the Coverity Scan platform and identify the project object which is registered to the Unikraft organization. The view ID, is used in the front-end interface to differentiate between different reporting pages of the project. For example, Coverity can show a view for only the most recently discovered defects while another view shows the latest submitted build. Fetching those 2 IDs is crucial and is done by inspecting the attribute of a button in the page's source code, as featured in Figure 11.

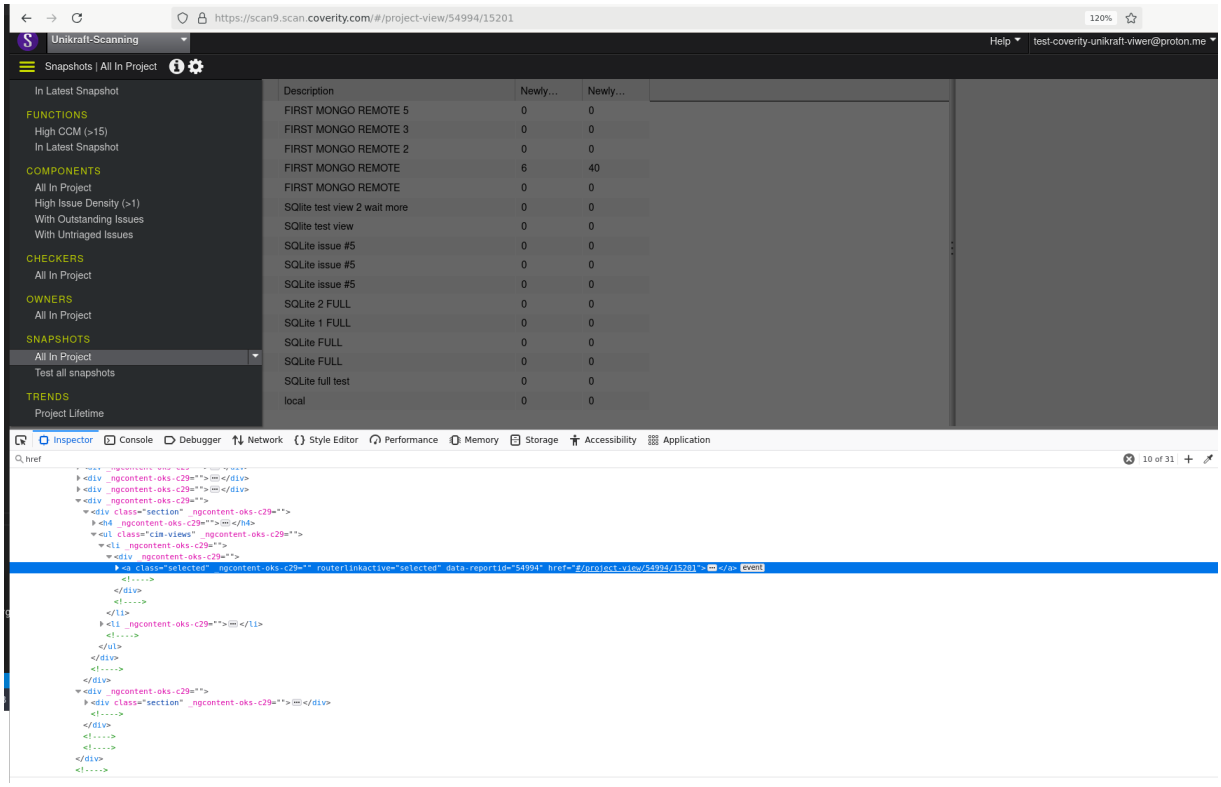

Figure 11: Capturing the Coverity Scan project and view id using web-scraping

The application needs to wait for the build to be analyzed by the Coverity platform. It will request the front-end view that shows all 'snapshots' (in our case all builds submitted to the Coverity) and it will intercept the HTTP responses. Finally, by inspecting the body of the intercepted traffic and if it sees our newly added build, we can proceed further. Otherwise, we repeat the operation.

We achieved HTTP interception by using an extension of the Selenium framework called selenium-wire[3] that provides us the possibility to add **callback procedures** that will be triggered anytime a HTTP request or response is sent, respectively received, at client side. Since these callbacks are called outside the main thread of the web-scraper, we use a **condition variable** that puts the main thread of the application to sleep and a notification mechanism inside the callback procedure if the desired information has been found in the intercepted HTTP body. In order to install and unregister the callback, we use a simple **state machine**. Changing the state is protected by the condition variable's lock as seen in the code sample 4.1.

Even though this implementation solves defects fetching, it is rather difficult to maintain. We recently discovered that we can only fetch defects found in the first page showed in the user interface. In order to download all results, we need to iterate over all view pages and apply the same mechanism.

[3]https://github.com/wkeeling/selenium-wire

Listing 4.1: Code sample logic of intercepting HTTP traffic using callbacks provided by selenium-wire Python package together with codition variables and a state machine

```
def main_thread():
    with self.interceptor_condition:
        while self.recent_snapshot_state != RecentSnapshotStates.FOUND:
            self.interceptor_condition.wait(self.interceptor_timeout)

            # timeout inside the interceptor
            if self.recent_snapshot_state != RecentSnapshotStates.FOUND:
                return False

        # reinit the state of the snapshots interceptor maybe for further use
        self.recent_snapshot_state = RecentSnapshotStates.NOT_STARTED

    return

def defects_response_interceptor(request: Request, response: Response):

    if coverityAPI.defects_state in [DefectsStates.NOT_STARTED, DefectsStates.
        FOUND]:
        logger.debug(f"DEFECTS: Ignoring {request.url} for recent defects since
            status is {coverityAPI.defects_state.name}")
        return

    # we need a specific URL
    current_defect_view_match = re.match(r"https://scan9\.scan\.coverity\.com/reports
        /table\.json\?projectId=(\d+)&viewId=(\d+)", request.url)

    if current_defect_view_match:

        with coverityAPI.interceptor_condition:
            # get defects from response body
            coverityAPI.cached_last_defect_results = json.loads(decode(response.body,
                response.headers.get('Content-Encoding', 'identity')))['resultSet']['
                results']

            # change state to notify that defects were found
            coverityAPI.defects_state = DefectsStates.FOUND

            # wake up the main thread since defects were found
            coverityAPI.interceptor_condition.notify()
```

The last attempt which was also successful and easier to implement and maintain is represented by the usage of an export function found in the user interface to download analysis results. Accessing such feature in the Coverity platform is achieved using simple Selenium components that mimic user interaction events such as mouse clicks and keyboard activity. We integrated in the application step-by-step scripts that visually navigate through the Coverity platform.

4.3 Unikraft-Scanner Visualization Feature

In the previous sections we presented our cooperative methodology in analyzing the Unikraft core. Since the progress is made by multiple people and in multiple iterations, the most correct way to progress further in solving the analysis task is done through two phases: algorithm execution of a chosen Unikraft configuration and observation of results. The observation is as crucial as running the algorithm on a carefully chosen "mutation" of Unikraft that presumably has never been analyzed before.

What we need to see during the observation phase is whether we increased the compilation coverage, by compiling new code regions and visualize possible defects's severity together with locations where the vulnerability density is higher than normal. After building a general idea regarding approximate regions which might introduce vulnerabilities we can start a more precise process of researching individual defects using the Coverity platform.

The visualization capability of Unikraft Scanner is for surface-level security assessment and progress monitoring while Coverity platform is for deep research of 1 single vulnerability deemed critical by the user or after establishing consensus between participants of the analysis experiment.

The good example of compilation coverage visualization is represented by Figure 12. The output format is similar to the Unix tool *tree*[4]. Source files colored with red are marked as not completely compiled. There are also code blocks whose compilation is conditioned by preprocessor symbols. The same markings are also applied to them, red means that the blocks has never been triggered before while green means that at least 1 iteration forced the block to be compiled. Defects can also be added to this brief view.

[4]https://linux.die.net/man/1/tree

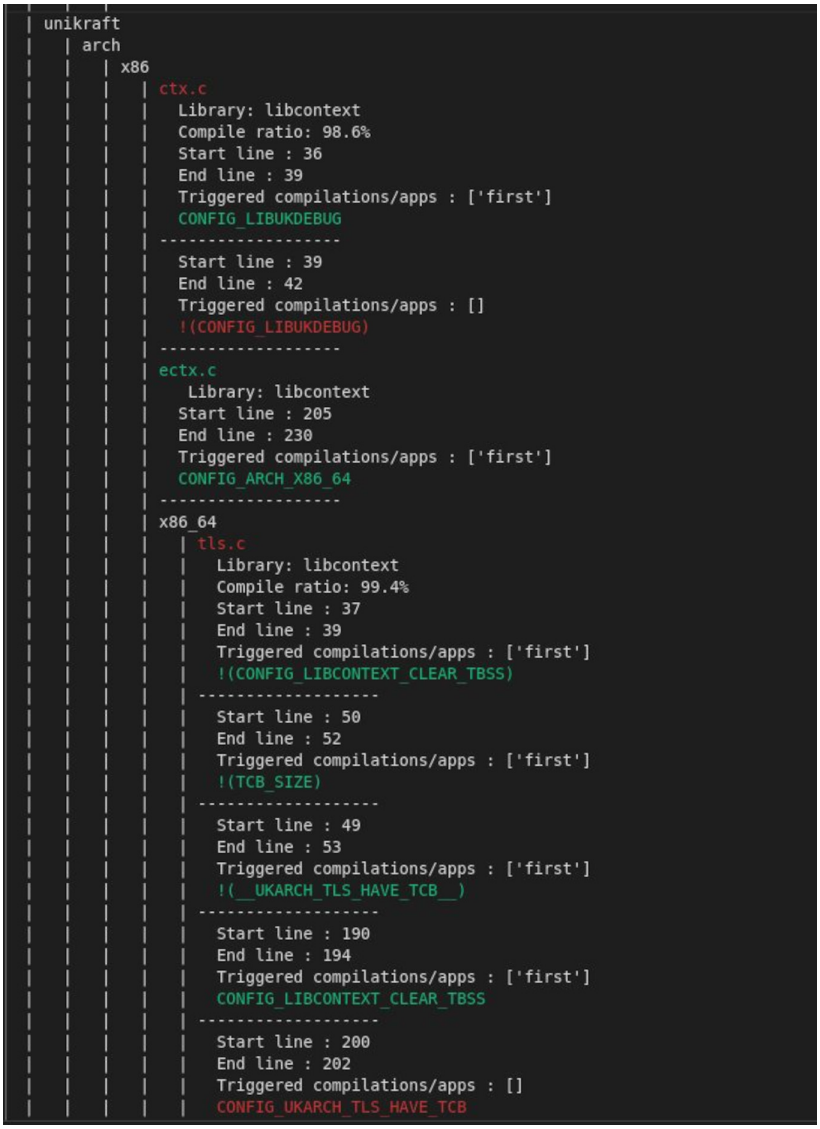

Figure 12: Visualization panel of the Unikraft Scanner

4.4 Storing Cooperative Results in the Database

Once the interval tree of the compiled regions is built for every involved source code file and all possible defects are inserted in them, the CLI tool serializes all the data as plain Python dictionaries and sends it to the database. For our proof-of-concept application, simplicity was the crucial factor in choosing a storing solution, thus we chose **MongoDB** [5].

Since the project may employ multiple static analysis providers, every one of them with their own result structure schema, a no-SQL approach can be helpful to store various results in a flexible manner.

[5] https://www.mongodb.com

5 TESTING AND EVALUATION

Evaluation of the Unikraft Scanner can be done in multiple ways. Our experiments shifted towards finding as many true and false positives in as many Unikraft "mutations", while also monitoring how computationally intensive is the process of finding the compilation coverage of the submitted source code.

Table 1 shows the measured temporal cost when parsing for code blocks with conditioned compilation, together with the total number of lines of code. As observed running an application does not exhaust the possible mutations for it. As shown, we have only run the experiment using x86_64 architecture and qemu platform.

Table 1: Sumarizare criterii

Application	**Analysis Time (seconds)**	**Lines of Code**	**Blocks of Code with Symbol Condition**
SQLite 3.40 (qemu, x86_64, native)	704	247473	6890
SQLite 3.40 (firecracker, x86_64, native)	1251	247473	6890
NGinx 1.25 (qemu, x86_64, bincompat)	133	76603	4146
Memcached 1.6 (qemu, x86_64, bincompat)	121	76603	4146

The same number of lines and blocks between the Nginx and Memchaced can be explained by the fact that both apps are in binary compatibility mode which means that the application itself is fetched directly from the vendor in binary format and an ELF loader for Unikraft will execute it during runtime. This means that at the application layer there are no differences regarding compiling. The shared Unikraft modules can be explained by the usage of those apps. Their use-case is very simillar.

5.1 Case Study of a Found Vulnerability

Once the automatic process of compilation interception, compilation coverage research and static analysis is accomplished, the human factor is required to do a specialized triage of true and false positives. The security specialist has the last word in regards of vulnerability management. In order to prove our tool's efficiency and the importance of automatic static analysis, we have investigated a **high** impact alert that enables **illegal memory write** as presented in the Common Weakness Exposure standard (CWE 119 [1]). As seen in Fig-

[1] https://cwe.mitre.org/data/definitions/119.html

ure 13, the out-of-bounds write, is alerted at line 422, when the 'order' variable exceeds the size of the array that is statically allocated to a constant value 'FREELIST_SIZE' (the allocation is out the figure scope but has been manually verified). The same 'order' variable is also put through a loop block that tries to do a merge of free memory blocks of the same size/order. To summarize, the problem seems to come from a memory allocator. In order to write outside the allocated bounds we need to obtain an order greater the FREELIST_SIZE (52). Inspecting the buddy allocator algorithm, we have found that the order represents the number risen to the power of 2 in order to work with free or allocated blocks of memory. This means that viable values are between 53 and 63 included since there is no block higher that 2^{64}. The function that contains the problem is **bbudy_pfree** and an existing function call chain that uses such function is represented by: **bbuddy_pfree - uk_allocbbuddy_init - heap_init - uk_sched_start -uk_thread_create_bare**. This track gave us a dead-end since no call allows us to introduce a variable order. However if we investigate the call chain **uk_prfree - p9front_allocate_dev_ring - p9front_add_dev** we can see an interesting statement at line 484. The order can be influenced by a compilation symbol called CONFIG_LIBXEN_9PFRONT_RING_ORDER. Such symbol can be modified during the manual menu configuration similar to what is pictured in Figure 2.

A second experiment was centered towards finding the distribution of found defects in regards to their severity. The results can be seen in Table 1. To be noted that these defects are overlapping between multiple analysis iterations and may contain both false and true postives.

Table 2: Comparison of defects' severity between various application compilations

Application	**Low severity defects**	**Medium severity defects**	**High severity defects**
SQLite 3.40 (qemu, x86_64, native)	12	229	115
NGinx 1.25 (qemu, x86_64, bincompat)	4	81	45
Memcached 1.6 (qemu, x86_64, bincompat)	4	81	45

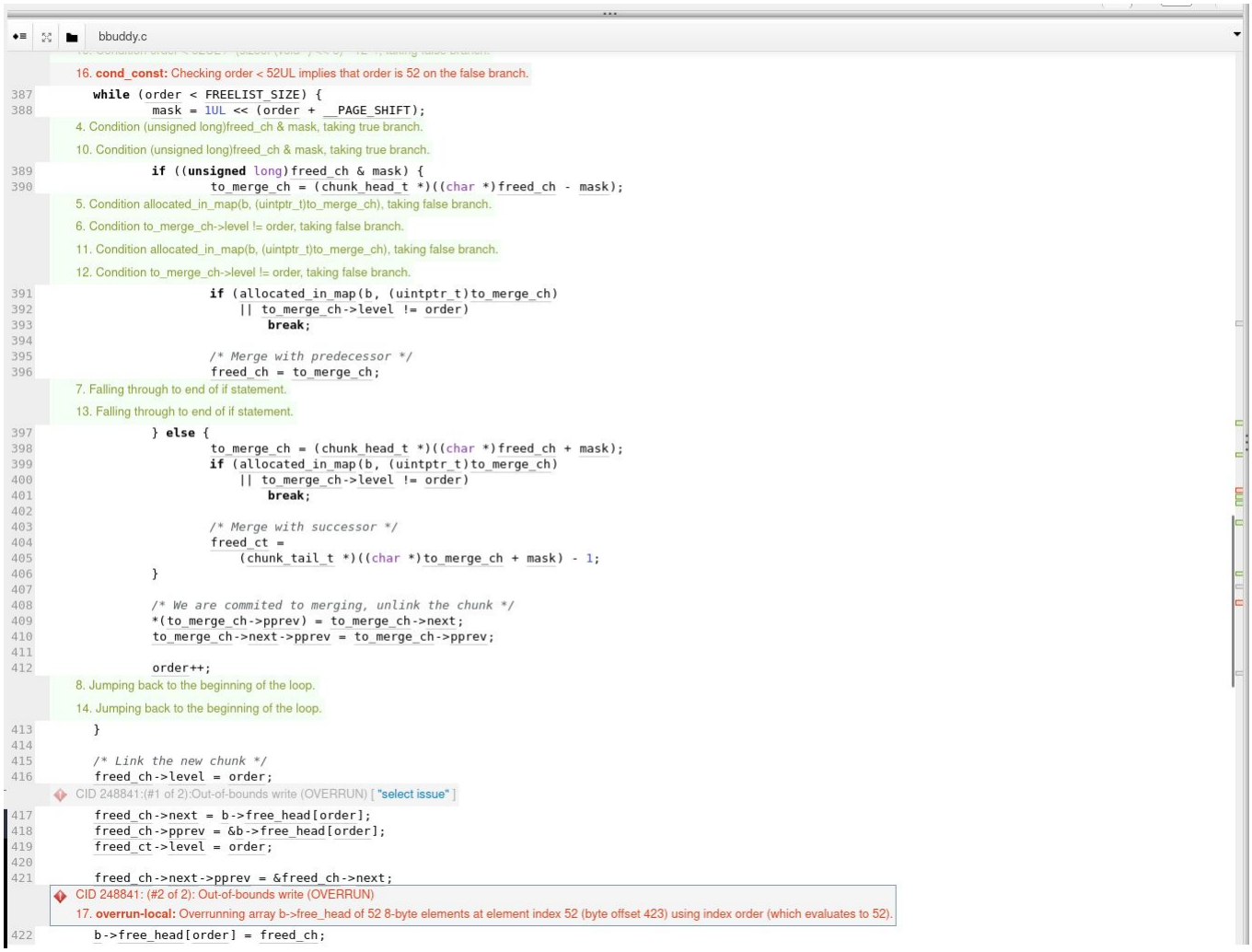

Figure 13: Static analysis algorithm reasoning for the investigated defect

Figure 14: Trigger point of the vulnerability

6 FUTURE WORK

The Unikraft-Scanner application has been enriched, at the time of writing this paper, with various critical features and can be considered a powerful tool for Unikraft developers. However, our development work is not done. There are still issues that we investigate regarding build submission made by multiple users at the same time. The web-scraping process may not support this scenario.

In addition, the compilation coverage research process can be optimized by a multi-threaded approach.

Furthermore, we would like solve issues regarding the analysis of code blocks whose compilation is conditioned by symbols, blocks that are found in C header files. At this moment, our parser only considers C source files for compilation coverage.

Regarding the compilation and analysis coverage achieved, we still have work to do in running as many Unikraft-ported applications as possible.

Last but not the least, we would like to integrate other static analysis tools such as CodeQL or Polyspace.

7 CONCLUSIONS

Static analysis and unikernels are not yet very well integrated between each other. However, our work contributed to a first automatic usage of such tools on the Unikraft unikernel.

First of all, we have presented what major issues may arise when trying to utilize static analysis on a unikernel (for example the Compilation Coverage Problem). In order to overcome such problem, we have devised and implemented an algorithm that parses all source files included in the compilation process of a unikernel image, identifies code blocks whose compilation is conditioned by preprocessor symbols, injects them with helper code statements and obtains the compilation coverage of a "mutation".

In order to conduct experimental scans on the Unikraft codebase, we have implemented from scratch an application called Unikraft Scanner that runs the aforementioned algorithm and interacts with the Coverity Scan static analysis platform for analyzing intercepted compilations and downloading the results.

Furthermore, we have also built a visualization feature that helps Unikraft developers to have a broad view of data regarding all "mutations" compiled so far by the community. Such view goes hand in hand with the official Coverity user interface that gives security analysts a more precise idea regarding vulnerabilities.

In the end, using Unikraft Scanner together with the Coverity static analysis tool we have found a potential dangerous vulnerability, highlighting the importance of automatic static analysis as a first layer of defense against malicious exploits.

BIBLIOGRAPHY

[1] URL: https://www.mathworks.com/content/dam/mathworks/tag-team/Objects/c/77340_91517v02_30211_Polyspace-WhitePaper_final.pdf.

[2] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of programs". In: (Jan. 1976).

[3] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. "A Survey of Automated Techniques for Formal Software Verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (2008), pp. 1165–1178. DOI: 10.1109/TCAD.2008.923410.

[4] Pär Emanuelsson and Ulf Nilsson. "A Comparative Study of Industrial Static Analysis Tools". In: *Electron. Notes Theor. Comput. Sci.* 217 (July 2008), pp. 5–21. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.06.039. URL: https://doi.org/10.1016/j.entcs.2008.06.039.

[5] Kendra Kratkiewicz and Richard Lippmann. "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools". In: (Oct. 2009).

[6] Aravind Machiry et al. "DR. Checker: A Soundy Analysis for Linux Kernel Drivers". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, pp. 1007–1024. ISBN: 9781931971409.

[7] Carmine Vassallo et al. "Context is king: The developer perspective on the usage of static analysis tools". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2018, pp. 38–49. DOI: 10.1109/SANER.2018.8330195.

[8] Misha Zitser, Richard Lippmann, and Tim Leek. "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code". In: *SIGSOFT Softw. Eng. Notes* 29.6 (Oct. 2004), pp. 97–106. ISSN: 0163-5948. DOI: 10.1145/1041685.1029911. URL: https://doi.org/10.1145/1041685.1029911.

A COVERITY SCAN NETWORK TRAFFIC CAPTURE

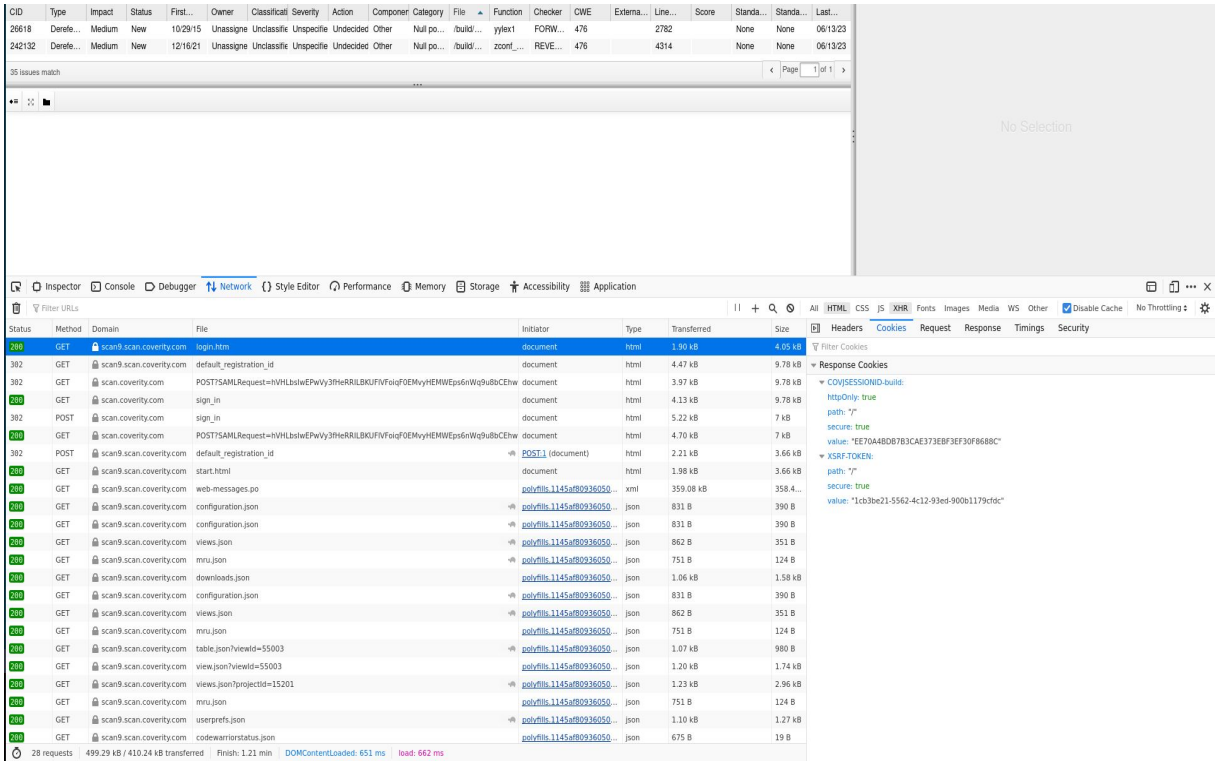

Figure 15: The starting HTTP dialogue which also represents the start of authentication process. We observe the we receive from the server 2 cookies "COVJSESSIONID-build" and "XSRF-TOKEN"

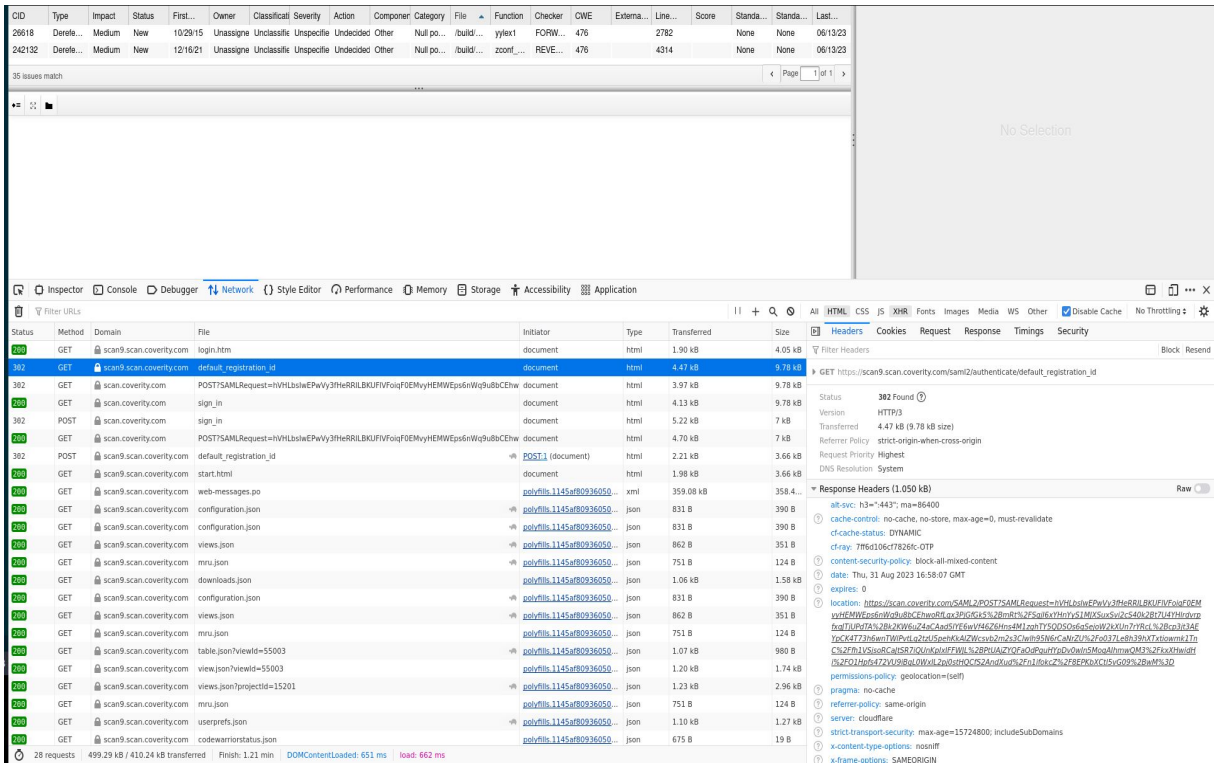

Figure 16: Next we access a template path which further redirects us, through a SAML token, to the Synopsys domain. Security Assertion Markup Language (SAML) allows authorization of third-party entities (in our case scan9 domain) through the same set of credentials accepted by the scan.coverity.com (Synopsys). The redirection URL with the appropriate token is found in the response header.

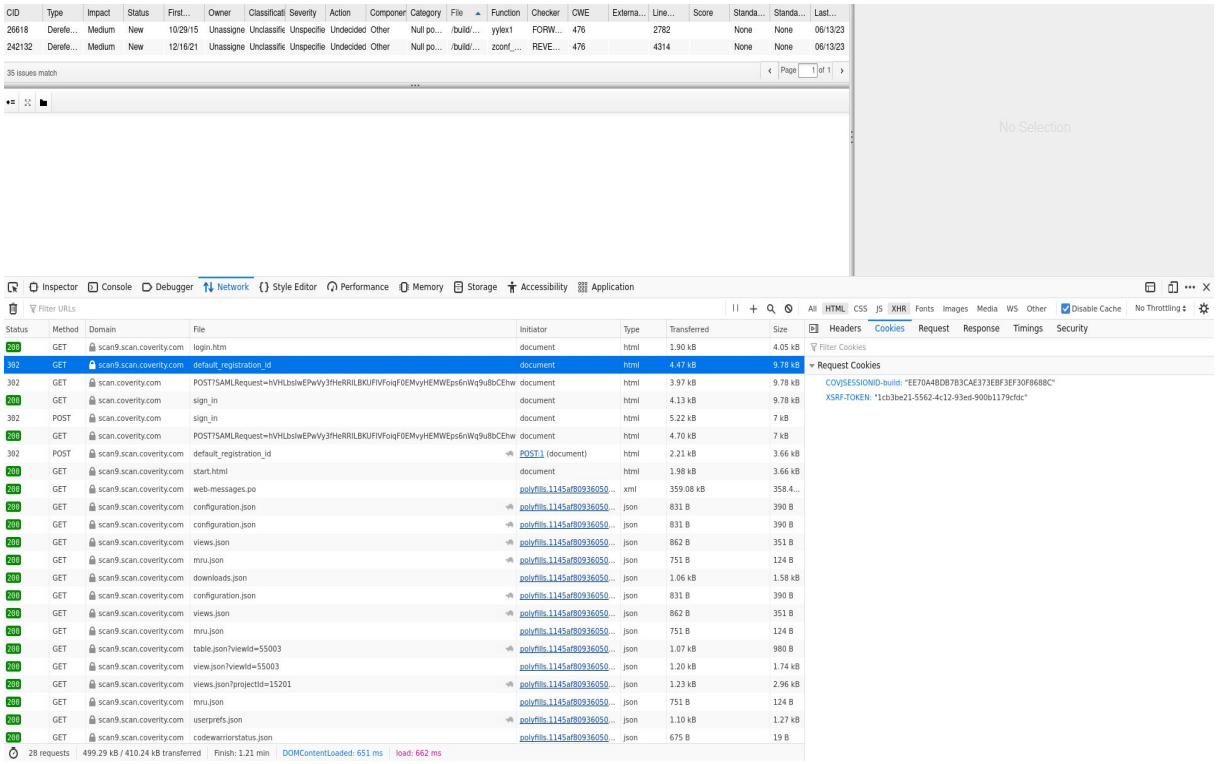

Figure 17: Same dialogue from Figure 3 but presenting the cookies acquired during the step in Figure 2. The client must send those cookies back to the server.

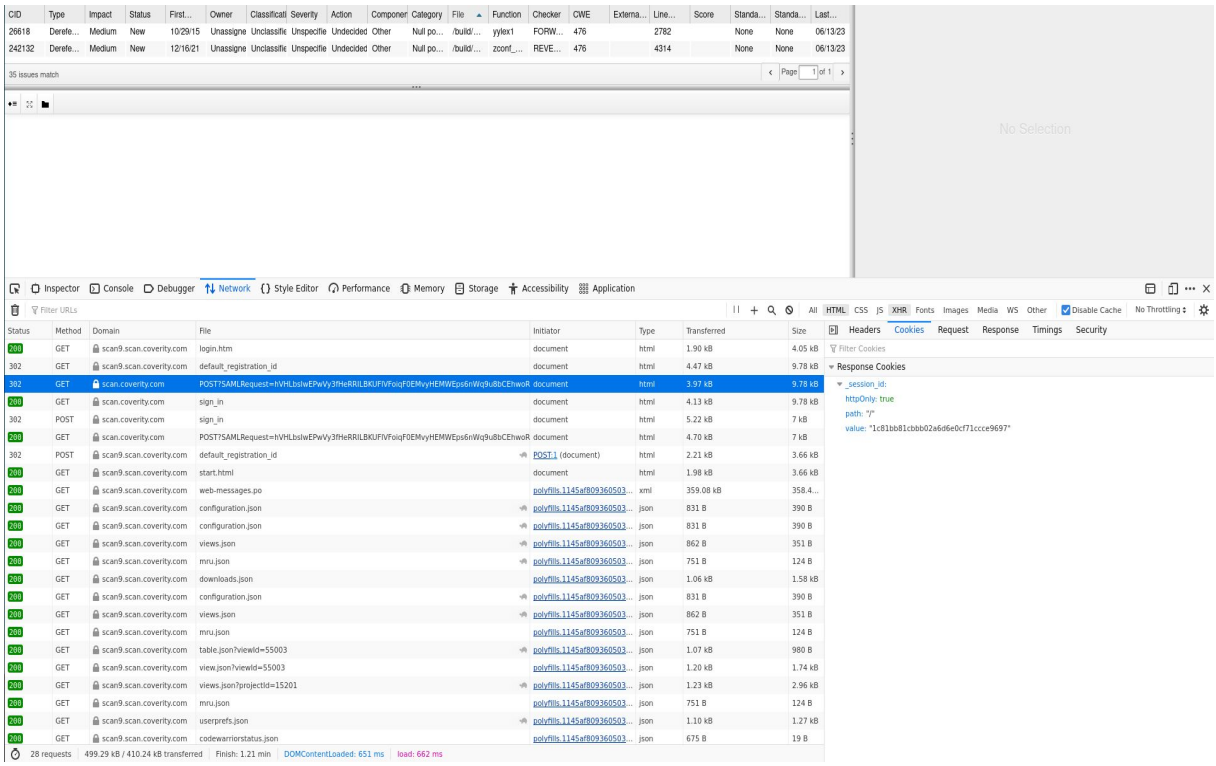

Figure 18: This SAML request with the token collected from step 3, receives a "session_id" cookie.

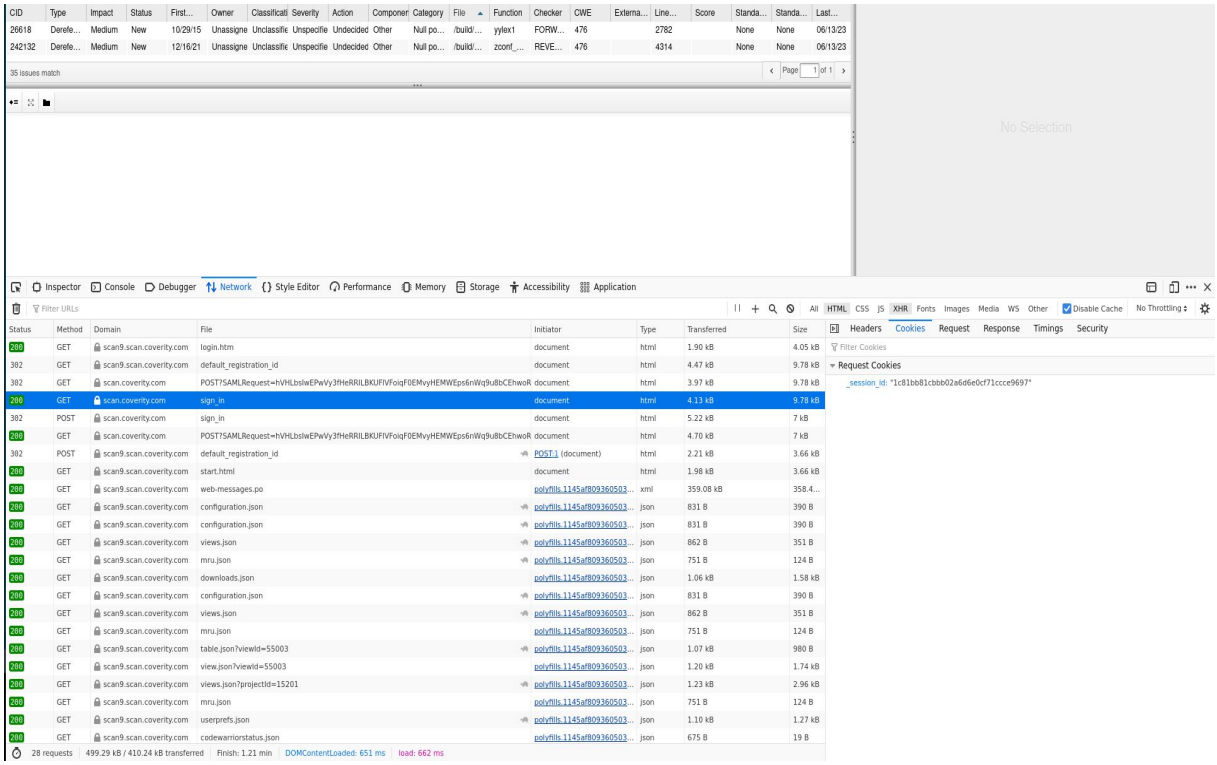

Figure 19: We go further and use the "session_id" cookie acquired in the previous step with a "sign_in" GET method.

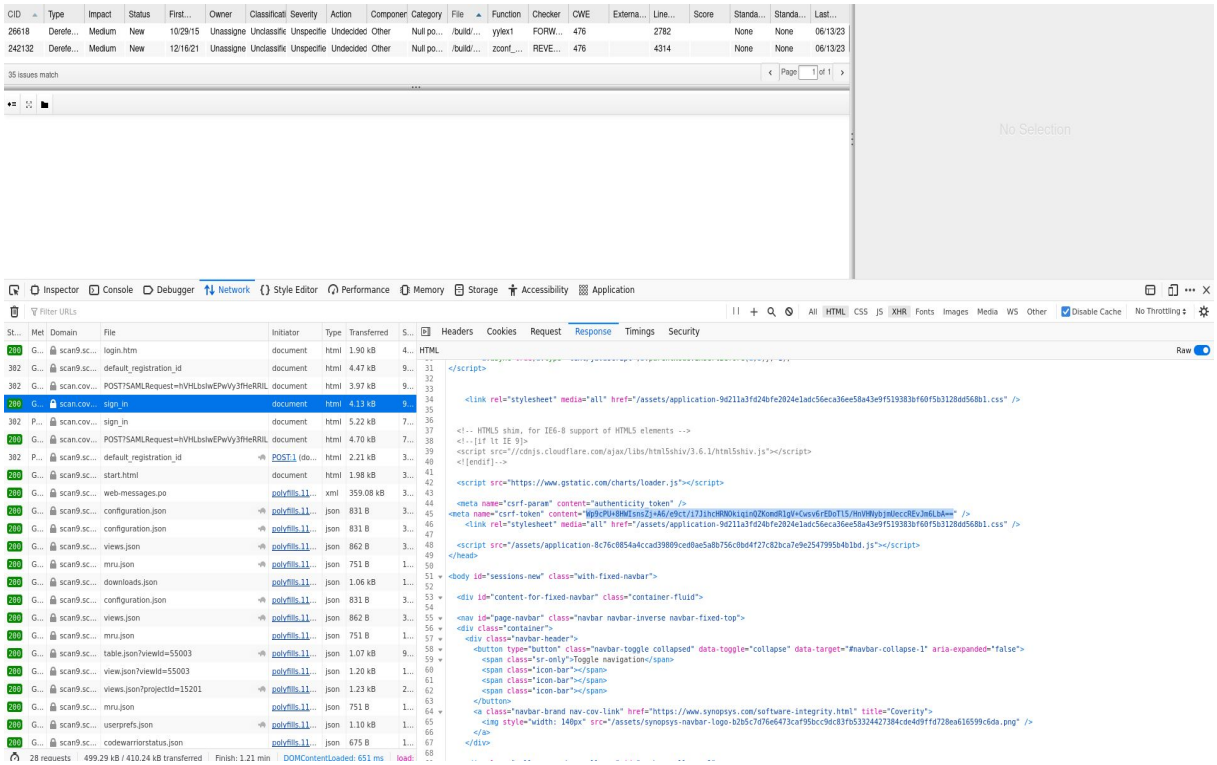

Figure 20: The same "sign_in" GET method from step 6 provides inside the raw HTML response content an "authenticity_token" that we will utilize in the further steps.

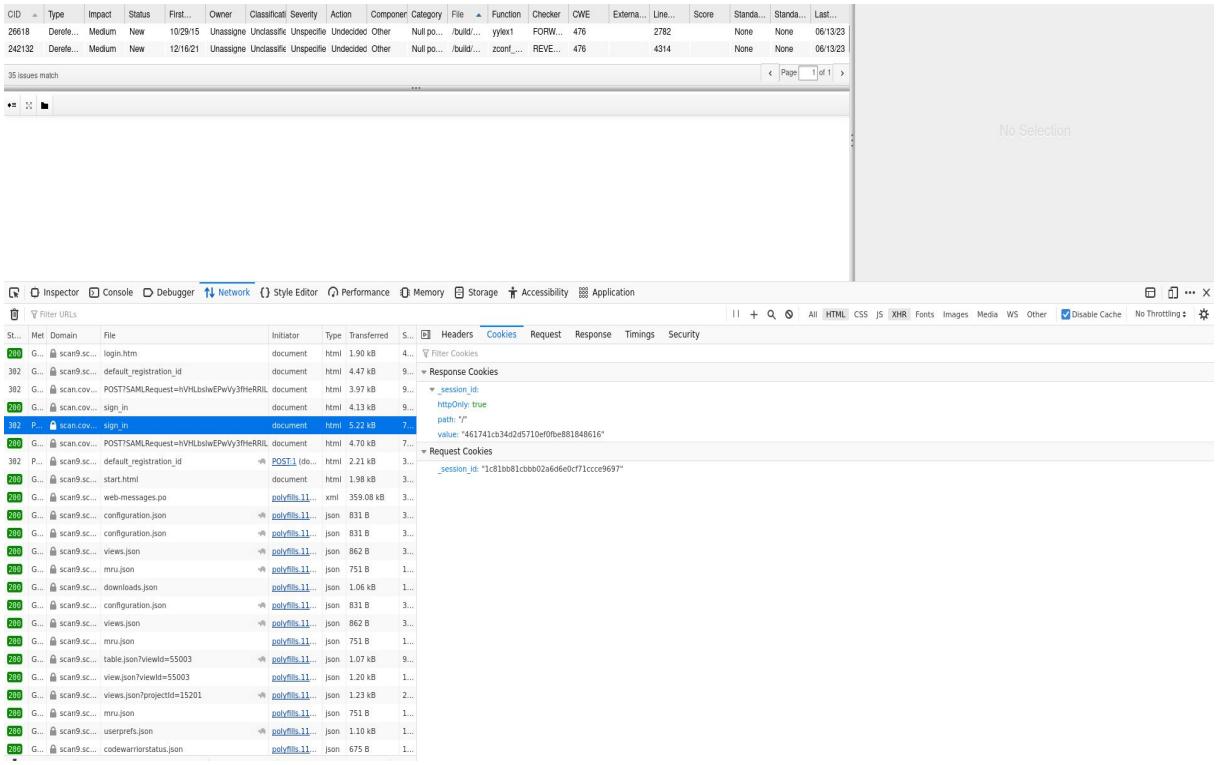

Figure 21: Now through the POST method to the same "sign_in" URL we send the instance of "session_id" cookie and receive a new instance of the same cookie.

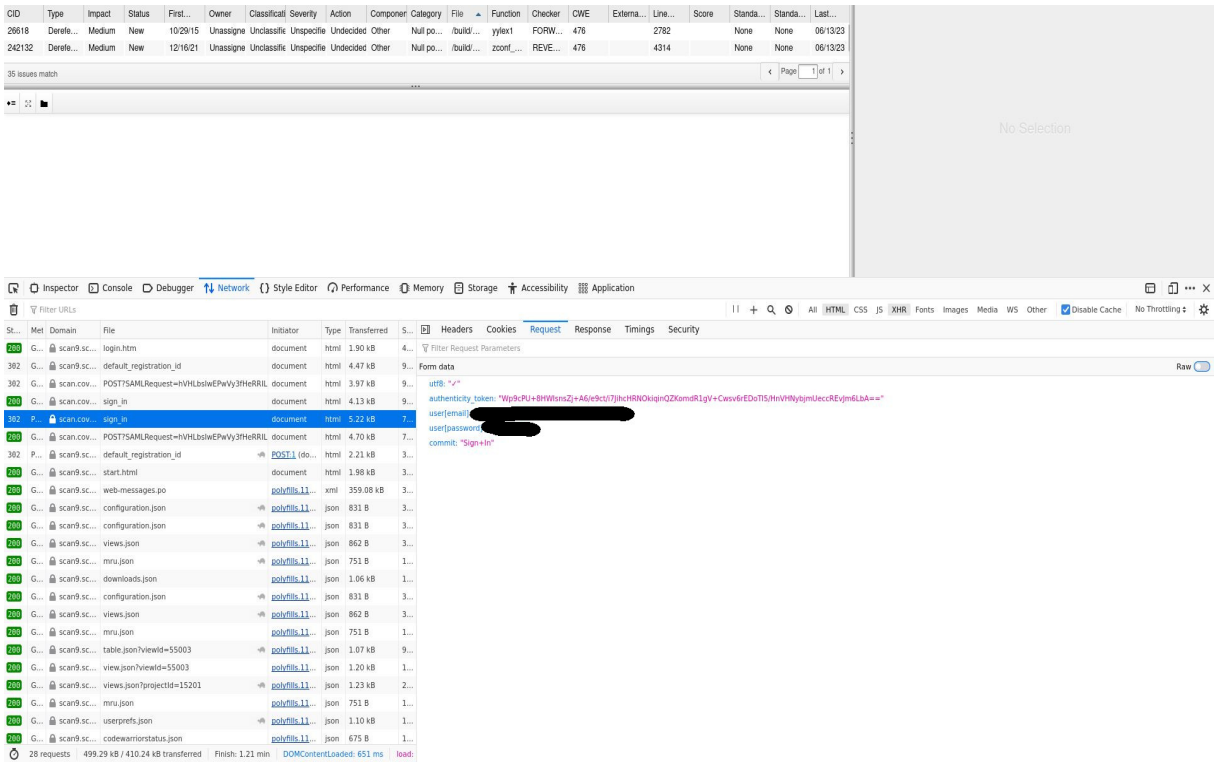

Figure 22: The request body of the same dialogue in step 8 contains the "authenticity_token" acquired during step 6 together with the user credentials.

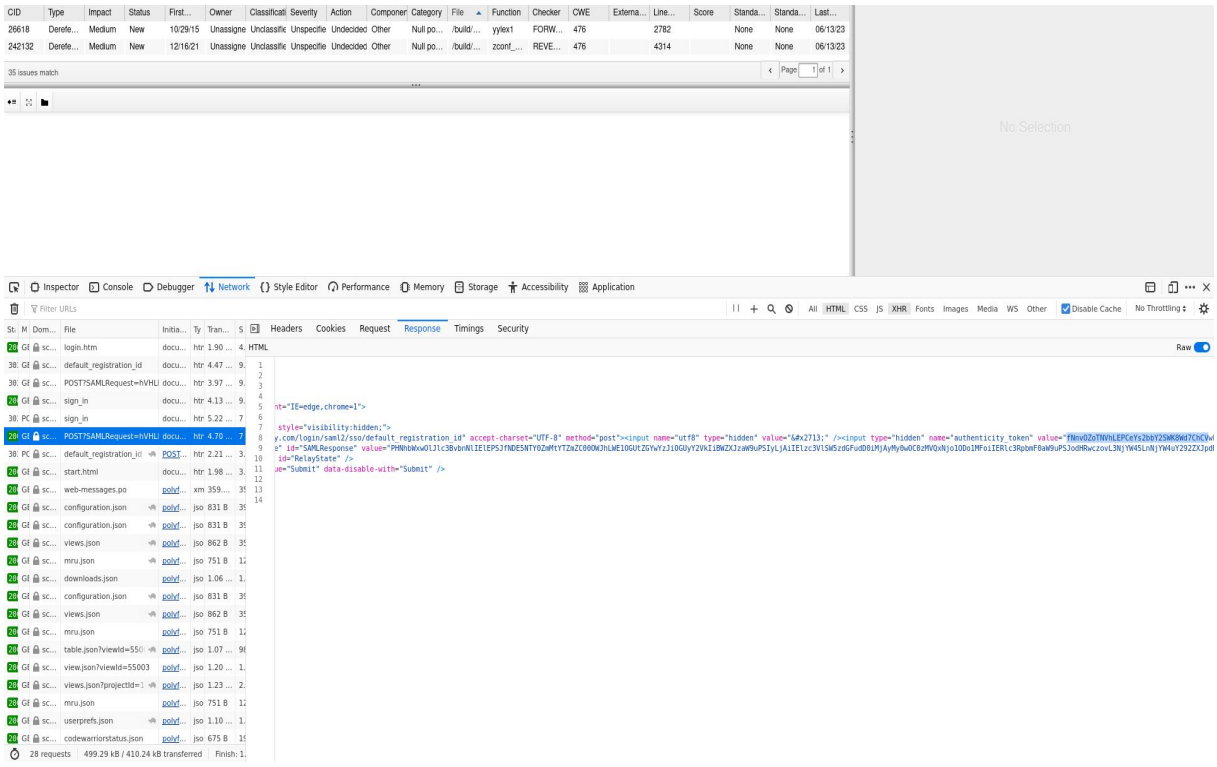

Figure 23: The second SAML dialogue behaves as an authorization acceptance, using the same SAML request token given in step 5. In the response contents we need to extract the "authenticity_token" and the "SAMLResponse" value.

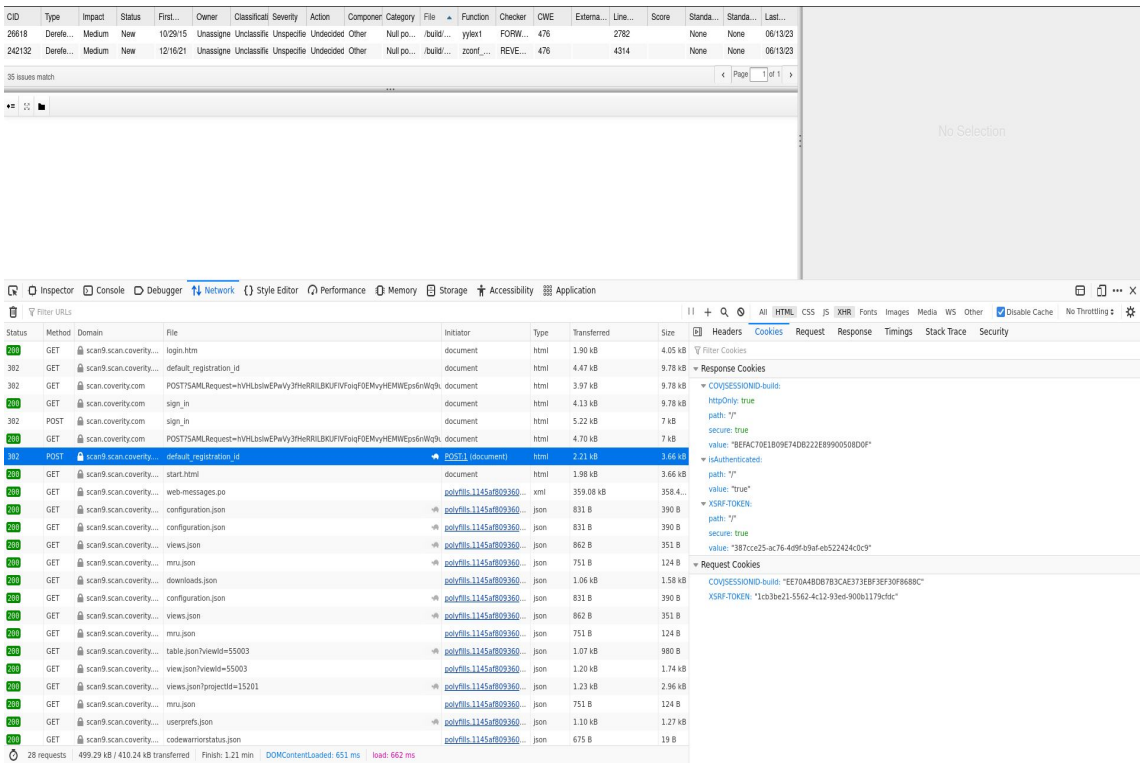

Figure 24: Afterwards, a second registration process begins in which we send the "COVJSESSIONID-build" and "XSRF" cookies gathered from step 1. In exchange, we receive updated instances of those cookies.

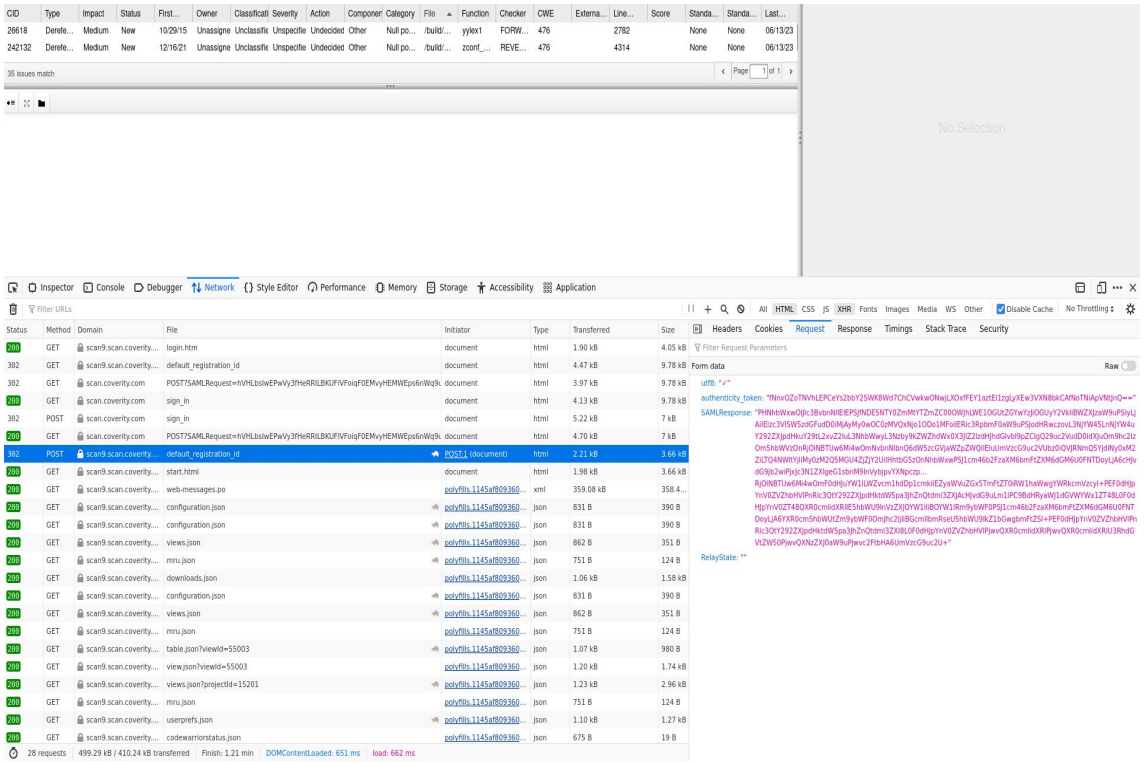

Figure 25: This screenshot represents same dialogue from previous step but highlighting the "SAMLResponse" and "authenticity_token" acquired from step 10 that must be sent to the server.