CS412 Fuzzing Lab Demo Report

May 15, 2025

Abstract

This report presents an analysis of fuzzing techniques applied to the libxml2 library using OSS-Fuzz. We compare coverage between seeded and unseeded fuzzing, identify significant uncovered regions, and develop a custom harness and improve an existing to target them. Additionally, we triage a real-world vulnerability (CVE-2024-25062), providing a detailed root cause and security analysis.

1 Introduction

Students:

- Patrick Pataky, 345302
- Hajar Mjoun, 310755
- Derya Cögendez, 310427
- Valentin Schneeberger, 310272

For the fuzzing lab we chose the libxml2 project https://gitlab.gnome.org/GNOME/libxml2, a XML toolkit implemented in C.

2 Part 1: Setup and Coverage Comparison Methodology

All files and scripts for this part are located in the `/part1` directory of the project repository. This section outlines how seeded and unseeded fuzzing were conducted using OSS-Fuzz and explains how line coverage differences were extracted and analyzed.

2.1 Directory Contents and Scripts

- `report/` — Contains HTML coverage reports for both seeded and unseeded runs, along with the comparison output.
- `run.w_corpus.sh` — Runs the XML fuzzer using the default OSS-Fuzz seed corpus.
- `run.w_o_corpus.sh` — Runs the XML fuzzer with the seed corpus manually removed.
- `coverage_comparison.py` — Python script to compare line-level coverage across all source files.
- `coverage/coverage_comparison_results.txt` — Output of the comparison script, showing which lines were covered only in one setup or the other.

2.2 Build and Execution Instructions

All scripts should be executed from within the `part1` directory.

To build the fuzzing image:

```
python3 ../infra/helper.py build_image libxml2
```

To run the seeded XML fuzzer:

```
sh run.w_corpus.sh
```

To run the unseeded XML fuzzer:

```
sh run.w_o_corpus.sh
```

To get the coverage unseeded

```
python3 ../infra/helper.py build_fuzzers --sanitizer coverage libxml2

python3 ../infra/helper.py coverage libxml2 --corpus-dir build/out/part1NoSeed
    / --fuzz-target xml
```

To get the coverage seeded

```
python3 ../infra/helper.py build_fuzzers --sanitizer coverage libxml2

python3 ../infra/helper.py coverage libxml2 --corpus-dir build/out/part1Seed/
    --fuzz-target xml
```

Note: In order to run the harness without any seed, the unseeded script deletes the default seed corpus from the build directory before execution.

2.3 Coverage Comparison Script

The file `coverage_comparison.py` parses the HTML coverage reports generated by OSS-Fuzz. It identifies uncovered lines (i.e., those marked in red) and compares them across the seeded and unseeded reports.

Diff and Script Snapshot: A complete diff showing the script and supporting files is available at `part1/report.diff`. An excerpt of the core logic is shown below for reference:

```
def extract_red_lines(html_path):
    ...
    if code_td.find("span", class_="red"):
        uncovered_lines.add(line_number)
```

This logic is used to extract uncovered lines from each file, and the script then computes the set differences between the two runs to identify lines uniquely covered or uncovered in each.

Comparison results are saved in:

```
report/coverage_comparison_results.txt
```

These results form the basis for the analysis in the following section, detailing qualitative differences in code paths reached by seeded vs. unseeded fuzzing. Fuzzing `libxml2` using the OSS-Fuzz-provided XML fuzzer for 4 hours with and without the default seed corpus resulted in the following coverage:

- **19.00%** total line coverage without seed
- **19.28%** total line coverage with seed

While the total coverage difference is marginal, seeded fuzzing led to unique code paths being exercised, the specifics are discussed later. This section explains why some code was covered in both setups and why certain lines were only covered with (or without) seed input.

Coverage Type	Description
Both setups	Early-stage parsing, error fallback logic, and basic structural checks.
Only with seed	Namespaces, URI parsing, encoding detection, DTD and SAX2 processing — all requiring valid structured XML.
Only without seed	Frequent malformed input paths: error reporting, recovery code, and edge-case fallbacks.

Table 1: Qualitative coverage differences between seeded and unseeded fuzzing.

2.4 Commonly Covered Code

Several core parsing routines were covered in both runs due to the nature of the inputs:

- Even without a seed, libFuzzer can generate minimal XML-like structures, enabling coverage of early-stage parsing logic such as tag detection, buffer handling, and entry-point initialization.
- Many error-handling and fallback routines are triggered by both malformed and partially correct inputs, contributing to shared coverage in files such as `parser.c`, `xmlIO.c`, and `xmlstring.c`.

2.5 Coverage Unique to Unseeded Fuzzing

Lines such as those in `error.c` (793–804) and `parser.c` (e.g., 1301–1373, 5630–5652) were covered only without the seed corpus. These lines correspond to:

- Error-reporting mechanisms triggered by invalid XML syntax.
- Entity mishandling and encoding failure paths.
- Structural violations like unmatched tags or invalid characters.

Justification: The absence of a seed increases the likelihood of malformed inputs, which in turn exercise robust error handling and defensive parsing logic in `libxml2`.

2.6 Coverage Unique to Seeded Fuzzing

The seed corpus enabled coverage of deeper semantic processing paths, such as:

- `valid.c`: Namespace validation (lines 2561–2575) and document structure checking.
- `uri.c`: URI normalization and encoding-specific logic (e.g., lines 97–100, 1204–1217).
- `SAX2.c`, `encoding.c`, `parserInternals.c`: SAX2 event handling, character set detection, and internal scanning routines.

Justification: These routines require well-structured XML input to be triggered. Without a seed corpus, libFuzzer struggles to generate such inputs randomly due to the syntactic complexity of XML. Seeded inputs guide execution into high-level semantic checks that are otherwise inaccessible.

2.7 Summary

Although the overall difference in percentage coverage is minor, seeded fuzzing enables deeper semantic exploration of the input space, reaching functionality that is unlikely to be covered by chance, as described in Table 1. This makes it a crucial complement to unseeded exploration.

3 Part 2

In this part, we based our analysis and assumptions on the coverage report from the oss-fuzz introspector: https://introspector.oss-fuzz.com/project-profile?project=libxml2. This reports a line coverage of 69.25%, as of the version of the 6th of May. Most of the code that isn't covered is deprecated or not used, so there aren't many useful parts left for the fuzzer to test. However, we still managed to identify two code regions that should be covered by the fuzzer.

3.1 Uncovered region 1

The `catalog.c` file implements functions handling XML catalogs, which are lookup mechanisms used to map external identifiers (like URIs, etc.) to local resources or alternative locations. This file is barely covered by existing fuzzing harnesses. This is because the catalog functionalities as implemented in the fuzzer work only if we tell the program that the file is a catalog. It requires setting the `"--catalogs"` option in `xmllint.c`, and would otherwise not parse catalogs by default. As shown in lint.c, the fuzzer doesn't test the catalog feature because it requires handling a different file structure. We would need a separate harness specifically designed to load and handle catalog files. Since no harness calls catalog functions or tries to set catalogs, the functions in `catalog.c` cannot be reached. Another reason is that catalogs are meant to be stored on disk, and accessing disk during fuzzing slows down the entire process. Finally, some unit tests are covering basic features and corner cases for most of functions in this file.

`Catalog.c` file contains multiple functions, but we selected some that are significant and should be covered for different reasons:

- `xmlFreeCatalogEntry, xmlFreeCatalogEntryList, xmlFreeCatalog`: Those functions handle memory and more specifically are responsible for deallocating memory, namely entries, list or the entire catalog. Improper handling of this process can result in memory leaks or double-free errors, potentially leading to vulnerabilities. These functions are even more important to fuzz, as they are called multiple times throughout the file, and also called by the API.
- `xmlCatalogXMLResolve, xmlCatalogXMLResolveURI, xmlCatalogListXMLResolve`: These functions have some medium complexity, respectively 50, 32 and 26, which is significant compared to other uncovered functions. This means the function has many possible paths, making it more likely to have a bug. These functions are called multiple times throughout the file, which is explained by the fact that they handle the main functionality of the file, which makes them significant. Bugs in this part of the code can lead to incorrect resource resolution.
- `xmlParseSGMLCatalogName, xmlParseSGMLCatalog`: These functions are parsing entries or catalogs and can thus be prone to bugs coming from malformed inputs.It is crucial that these functions handle edge cases properly and are fuzzed to detect any bugs that could potentially be exploited. Moreover, these functions have also medium complexity.
- `xmlACatalogAdd, xmlACatalogRemove, xmlCatalogAdd, xmlCatalogRemove`: These functions add and remove entries in the catalog, which involves managing memory and internal data structures. If there are bugs in these functions, it can lead to memory corruption or security vulnerabilities like invalid entries. Thus, if there is a bug in those functions it may impact the whole functionality of the catalog, even more when we know that these functions are called by the API and are not internal functions.

The catalog.c file has a line coverage of only 4.5%, with all existing harnesses. The significant uncovered region composed of the functions mentioned above has a total of lines, distributed as follows:

- `xmlFreeCatalogEntry, xmlFreeCatalogEntryList, xmlFreeCatalog`: 43 lines
- `xmlParseSGMLCatalogName, xmlParseSGMLCatalog`: 200 lines
- `xmlCatalogXMLResolve, xmlCatalogXMLResolveURI, xmlCatalogListXMLResolve`: 385 lines
- `xmlACatalogAdd, xmlACatalogRemove, xmlCatalogAdd, xmlCatalogRemove`: 77 lines

This code region has a total of 705/2287 lines, which means that the uncovered region is approximately 30.8% of `catalog.c`, which is a third of the file.

3.2 Uncovered region 2

The `xmlXPathNodeSetContains` function in `xpath.c` is part of the XPath implementation and is used to determine whether a specific node exists within an XPath node set, a node and a node set as

input. XPath, or XML Path Language, is a language designed to navigate and query parts of an XML document. This language has path expressions which are similar to file paths in a system but for parts of an XML document. Thus, XPath is an important feature of the library, and its implementation should be covered entirely by the fuzzer. However, the `xmlXPathNodeSetContains` is not covered by the existing fuzzers at all. This is due to the fact that the existing harness `fuzz/xpath.c` evaluates arbitrary expressions but does not generate any XPath that involves node set comparisons, such as union or equality operations, which explains why `xmlXPathNodeSetContains` is not called during fuzzing. This function is also not called directly in the harness. Moreover, it is not covered by the `xmllint.c` harness because it requires setting the "–xpath" option in xmllint.c, and would otherwise not parse xpath by default. As shown in `lint.c`, the fuzzer doesn't test the xpath feature because it requires the use of the resource loader, and it is not done by `lint.c`. This is why they implemented a specific harness for the XPath language implementation.

We identified two reasons why we think that `xmlXPathNodeSetContains` is important to be covered by fuzzers:

- This function is called several times in `xpath.c` file and also in `c14n.c`, which is explained by its use when evaluating widely used operations in XPath expressions, such as union, set comparisons, difference and intersection. Even if it is a small utility function, it can have a big impact when using the library, and even a small error or bug in this function can lead to unexpected behavior, such as giving the wrong results, which affects negatively the whole purpose of using XPath. As we can see on this example below, which is part of the function `xmlXPathDifference`, if `xmlXPathNodeSetContains` returns the wrong value it can falsify the result of the function, as it may add the same node to the set and contain twice the same value. Depending on the XML file, it can have some important consequences.

```
1    if (!xmlXPathNodeSetContains(nodes2, cur)) {
2            if (xmlXPathNodeSetAddUnique(ret, cur) < 0) {
3                xmlXPathFreeNodeSet(ret);
4                return(NULL);
5        }
6    }
```

  It could also lead to crashes or memory corruption.

- Since the function processes input and is often used as a condition, it plays a key role in identifying enforcing bound correctness and evaluating specific situations. It is crucial that the function is correct, as it can help detect corrupted node sets and discard them from being used by the rest of the XPath implementation. However, if there is a bug it can allow a malicious adversary to bypass filters and maybe access region of code or memory that should be protected.

The function consists of 24 lines, which is 0.35% of the whole `xpath.c` file. It represents 0.4% of all the functions in the file.

4 Part 3: Improving the Existing Fuzzers

4.1 Uncovered Region 1

As the catalog implementation was not covered at all by the existing fuzzers, we decided to create a new harness, testing its API. For that, we took inspiration of the api.c harness, located in:

```
1 oss-fuzz-libxml2/projects/libxml2/libxml2/fuzz/api.c
```

This harness implements an efficient virtual machine architecture that interprets fuzzer-generated data as an instruction set that is able to cover the libxml2 library's API functions. The idea is the following: each input byte typically represents an operation code that is mapped to a specific API function call, and uses registers to store arguments and results. It uses rotating fixed-size register arrays for each basic types and libxml2 objects (integers, strings, nodes, and documents), allowing to have complex call sequences with minimal input bytes.

Following the same mechanism, we developped a very basic harness for testing exclusively a sample of the exposed catalog functions, as mentionned in Section 3.1. The small size of catalog functions targeted in this harness is simply due to time constraints. Nonetheless, we believe the chosen subset provides sufficient new coverage to meet the objectives of this lab.

4.1.1 Why a virtual machine?

The virtual machine technique has multiple advantages.

First, as each byte can trigger a completly different API call, it allows to express very complex test cases with very few input bytes.

Then, the idea of using registers is to add some state persistency. Indeed, each operation can be influenced by a previous one, which allows to test interactions between the functions.

Finally, it fits very well the catalog API, as it is designed to be run in sequence (create a catalog, and then insert/resolve multiple entries).

4.1.2 Virtual machine architectural details

The virtual machine implemented in our catalog harness uses three types of rotating registers:

- Integer registers (64 slots): Store operations return values and use for numeric parameters
- String registers (64 slots): Store string values for passing to catalog functions
- Catalog registers (16 slots): Store catalog object pointers

Each register type has its own index counter that increments in a circular manner. This way, the harness always stores results in the newest register (at index 0), refers to older results as arguments to subsequent operations, and manages memory by cleaning old registers when they are overwritten.

4.1.3 Logic flow

Algorithm 1 Virtual Machine Execution Flow

1: **Initialization**: Set up the virtual machine and initialize the `libxml2` library.
2: **input processing**: Read the first 4 bytes to determine the failure injection point (e.g., force a `malloc` failure).
3: **for all** remaining bytes in the input **do**
4: - Map the current byte to an operation (i.e., a tested function).
5: - Execute the function using inputs from registers.
6: - Store the result in the appropriate register.
7: - Perform memory cleanup if needed.
8: **end for**
9: **Cleanup**: Free all allocated resources.

This logic flows (see Algorithm 1) allows for complex combinations of inputs, where each operation can influence subsequent operations through the register contents.

4.1.4 Description of the scripts

The following shows the format of the scripts for this part:

```
$ tree

part3/
+-- improve1/
|   +-- coverage_improve1/
|   |-- run.improve1.sh
|   |-- catalog.patch
|-- patch.sh
```

Where:

- `improve1/run.improve1.sh` modifies the library via `improve1/catalog.patch`, builds & runs the new harness `catalog`, for 3 times 4 hours. Its result can be observed under `improve1/coverage_improve1/`.
- `patch.sh` is used to apply or reset the patch file.

This first region was never covered, thus no base coverage report are relevant for our use-case. The scripts are meant to be executed directly from their folders:

```
$ cd part3/improve1
$ chmod +x run.improve1.sh && ./run.improve1.sh
```

4.1.5 Implementation issue

Unfortunately, constructing a strong harness is a hard task, and requires a strong knowledge of the tested library. With the current harness implementation, a `Memory Leak` is detected after a few minutes of running it (using ASan), specifically after calling the `xmlCatalogAdd` function. We didn't investigate too much the origin of the failure to focus on more interesting bugs. To bypass it, the memory leak detection were simply disabled for this harness.

In addition, `catalog.c` inherently needs I/O operations, as catalogs in XML are meant to be stored on device, in a persistent way. This limits the harness performance by a lot (up to $\approx 1k - 10k$ slower than without using I/O). We believe it's the main reason why no harness for this function has been implemented.

4.1.6 Discussion of the achieved coverage

Prior to this lab, the introspector [Goo23] indicates that the `catalog.c` file, with no harness targetting specifically its functions, yielded only 4.5% of line coverage. After introducing our targeted harness, we obtaines 41.5% line coverage, and 58.1 % function coverage. Publicly accessibles functions from `catalog.c` that were directly tested by the new harness are:

- xmlNewCatalog (100 % coverage)
- xmlLoadCatalog (80 % coverage)
- xmlCatalogAdd (50 % coverage)
- xmlCatalogRemove (100 % coverage)
- xmlCatalogResolve (100 % coverage)
- xmlCatalogResolveSystem (100 % coverage)
- xmlCatalogResolvePublic (100 % coverage)
- xmlCatalogResolveURI (100 % coverage)

Those functions called the following internal functions:

- xmlFreeCatalogEntry (44 % coverage)
- xmlFreeCatalogEntryList (100 % coverage)
- xmlFreeCatalog (88 % coverage)
- xmlParseSGMLCatalogName (42 % coverage)
- xmlParseSGMLCatalog (13 % coverage)
- xmlCatalogXMLResolve (71 % coverage)
- xmlCatalogXMLResolveURI (78 % coverage)

- xmlCatalogListXMLResolve (100 % coverage)
- xmlACatalogAdd (40 % coverage)
- xmlACatalogRemove (69 % coverage)

All together, a solid 41.5 % of lines could be covered, with 58.1 % functions reached of `catalog.c`, which is a strong improvement.

4.1.7 Opportunities for further improvement

1. **Test the remaining functions:** As only a small subset of functions in `catalog.c` were tested, it would be easy to simply add the entire API into the virtual machine.
2. **Seeded inputs:** Catalog entries require well-formed XML/SGML fragments (e.g., entries beginning with '<' and ending with '>'). Providing well crafted seed files would lead the fuzzer to a deeper parsing.
3. **External catalog files:** Many resolution functions read on-disk catalogs (e.g., `/etc/xml/catalog`). By supplying custom complex catalog files, similarly to the seeds, we could reach the remaining lines of the covered functions.
4. **Dynamic error injection:** Extending the harness to vary failure-injection points around I/O functions (i.e., not just with `malloc` as it is already done currently) could reveal the error handling paths and cleanup routines.

4.1.8 Extra: Minor bug discovery

As a side note, we managed to find a bug within the library, in particular in the function `catalog.c:xmlCatalogNormalizePublic`. Here is the faulty part of the function:

```
// ...
// assert(pubID != NULL);
ret = xmlStrdup(pubID);

q = ret;
white = 0;
for (p = pubID;*p != 0;p++) {
    if (xmlIsBlank_ch(*p)) {
        if (q != ret)
            white = 1;
    } else {
        if (white) {
            *(q++) = 0x20;
            white = 0;
        }
        *(q++) = *p;
    }
}
*q = 0;

return(ret);
```

`xmlStrdup` implements the standard `strdup` function, thus it copies the NULL terminated string into a newly allocated memory region.

The bug arise when `xmlStrdup` fails (due to a `malloc` failure), the pointer `q` takes the value NULL, but can still be accessed later without any checks (`*(q++)` and `*q`). This error produces a fatal crash in the program (`Segmentation Fault`). The fix consists in checking the return value of `xmlStrdup`, and exit if it failed.

4.2 Uncovered Region 2

The second region we targeted was the XPath node set functionality in libxml2, specifically the `xmlXPathNodeSetContains` function which is used to check if a node is contained in an XPath node set.

We modified the existing api fuzzer located at:

```
oss-fuzz-libxml2/projects/libxml2/libxml2/fuzz/api.c
```

As mentionned in the previous Section (4.1), the api fuzzer implements a virtual machine architecture that interprets fuzzer-generated data as an instruction set to test `libxml2`'s API functions. Our goal was to ensure to test further the XPath node set, currently poorly tested.

4.2.1 Implementation details

Our improvement involves two main components:

- **Node set population:** Adding functionality to populate the node set whenever a new node is created or stored in a register.
- **Node set maintenance:** Ensuring nodes are removed from the set when they are freed to avoid issues with dangling pointers.

We added a new register to the virtual machine, representing a Node set, of type:

```
// projects/libxml2/libxml2/include/libxml/xpath.h

typedef struct _xmlNodeSet xmlNodeSet;
struct _xmlNodeSet {
    int nodeNr;
    int nodeMax;
    xmlNodePtr *nodeTab;
};
```

We implemented two helper functions:

```
/**
 * Add `node` to the node set, if the
 * set is not full.
**/
static void addNodeToSet(xmlNodePtr node);

/**
 * Removes `node` from the node set, if the
 * node is present in the set.
**/
static void removeNodeFromSet(xmlNodePtr node);
```

We called these functions in strategic locations within the api fuzzer:

- Each time a new node is added in the node register, via the function `setNode`.
- Each time a node is removed from the node register, via the function `dropNode`.

We also added initialization of the node set data structure in the beginning of the `LLVMFuzzerTestOneInput` function.

4.2.2 Randomization strategy

To ensure diversity in the node set content, we used randomization when deciding whether to add a node to the set:

```
// static void addNodeToSet(xmlNodePtr node)

if (node == NULL || (xmlFuzzReadInt(1) % 3 != 0))
    return;
```

This means only about 1/3 of nodes will be added to the set, which allows testing both positive and negative results when calling xmlXPathNodeSetContains. This value is purely arbitrary, and could be tuned to get a better coverage.

4.2.3 Description of the scripts

Similarly as in Section 4.1.4, the following shows the format of the scripts for the second uncovered region:

```
$ tree

part3/
+-- coverage_noimprove/
+-- improve2/
|   +-- coverage_improve2/
|   |-- run.improve2.sh
|   |-- api.patch
|-- patch.sh
```

Where:

- `coverage_noimprove/` contains the html report (accessible under `linux/index.html`) on the harness `api`, ran for 3 times 4 hours. This harness was not modified at all.
- `improve2/run.improve2.sh` modifies the library via `improve2/api.patch`, builds & runs the modified `api` harness, for 3 times 4 hours. Its result can again be observed under `improve2/coverage_improve2/`.
- `patch.sh` is used to apply or reset the patch file.

The scripts are meant to be executed directly from their folders:

```
$ cd part3/improve2
$ chmod +x run.improve2.sh && ./run.improve2.sh
```

4.2.4 Discussion of the achieved coverage

The coverage improvement from our modification is significant. After our changes:

- The `xmlXPathNodeSetContains` function is now properly called.
- Line coverage for the function increased from 0% to 50%.
- The function is tested with both positive cases (nodes that exist in the set) and negative cases (nodes that don't exist in the set).

Unfortunately, this function also works with `xmlNsPtr`, whch is not implemented at all in this harness. This is the reason why the coverage is stuck at 50%.

4.2.5 Opportunities for further improvement

As this function is fairly simple in its core, further improvement would be to include more functions from `xpath.h` in the harness `api.c`, as a considerable number of functions (52 as of the 6th May 2025) is poorly tested in this file, and this harness seems the most adequate to test all functions related to XML nodes, due to the already existing infrastucture regarding nodes.

5 Part4

We discovered a bug using our new fuzzing harness (see Section 4.1.8). However, this was a straightforward null pointer dereference, so we opted to triage a different, more interesting bug. In this section, we analyze CVE-2024-25062[Cor24], a use-after-free vulnerability with potentially serious security implications. It is a recent vulnerability affecting versions up to `v2.11.7` and `v2.12.5` of the `libxml2` library. Older versions remain unpatched, so any program relying on these is still vulnerable.

To reproduce the bug, we used resources from the original issue. Inside the `part4/` directory, we provide a `Dockerfile` that builds `libxml2 v2.11.6` and runs `xmllint` with the necessary flags on a crafted XML file that triggers the use-after-free. The script `run.poc.sh` builds and executes the container to demonstrate the crash using this unpatched version. All analysis in this section is based on `v2.11.6`.

5.1 Root Cause Analysis

This bug occurs only when both the `--xinclude` and `--valid` flags are enabled while streaming a file. It results from a series of subtle issues that arise in a specific parsing context.

We use the minimized test case from the original Gitlab issue[Pro23a] to demonstrate the vulnerability. The XML file `reproduce_bug.xml` (included in `part4/`) looks like this:

Listing 1: `reproduce_bug.xml`

```
<!DOCTYPE doc [
    <!ELEMENT doc (elem)>
]>
<doc xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include href="404.xml">
        <xi:include href="404.xml">
            <xi:fallback>
                <xi:fallback>
                    <fb1/>
                </xi:fallback>
            </xi:fallback>
        </xi:include>
    </xi:include>
</doc>
```

The issue is triggered by running:

```
xmllint --stream --xinclude --valid reproduce_bug.xml
```

This command leads to the call of `streamFile` in `xmllint.c`, which uses `xmlTextReaderRead` in `xmlreader.c` in a loop to parse the file and then validates the nodes with `xmlTextReaderIsValid`.

Problems begin when the outer `<xi:include href="404.xml">` element is processed and the file `404.xml` is not found. The call to `xmlXIncludeProcessNode` fails and returns `-1`, but this return value is not checked:

Listing 2: Excerpt from `xmlTextReaderRead` in `xmlreader.c`

```
...
/*
 * expand that node and process it
 */
if (xmlTextReaderExpand(reader) == NULL)
    return -1;
xmlXIncludeProcessNode(reader->xincctxt, reader->node);              <--------
}
if ((reader->node != NULL) && (reader->node->type == XML_XINCLUDE_START)) {
    reader->in_xinclude++;
    goto get_next_node;
}
```

```
1469 if ((reader->node != NULL) && (reader->node->type == XML_XINCLUDE_END)) {
1470     reader->in_xinclude--;
1471     goto get_next_node;
1472 }
```

The reader continues as if nothing went wrong. Eventually, it finishes reading the document and begins freeing nodes via `xmlTextReaderFreeNode`. During backtracking, it reaches a point where the following condition is met, and a second call to `xmlXIncludeProcessNode` is made:

Listing 3: Excerpt from `xmlTextReaderRead` in `xmlreader.c`

```
#ifdef LIBXML_XINCLUDE_ENABLED
    /*
     * Handle XInclude if asked for
     */
    if ((reader->xinclude) && (reader->in_xinclude == 0) &&
        (reader->node != NULL) &&
        (reader->node->type == XML_ELEMENT_NODE) &&
        (reader->node->ns != NULL) &&
        ((xmlStrEqual(reader->node->ns->href, XINCLUDE_NS)) ||
         (xmlStrEqual(reader->node->ns->href, XINCLUDE_OLD_NS)))) {
    ...
    xmlXIncludeProcessNode(reader->xincctxt, reader->node);
```

This second call reprocesses nodes that were already freed, reintroducing dangling pointers into the document. Because the parser is now backtracking, it starts from the inner `xi:include`, and this time the fallback is processed successfully.

Eventually, the process reaches `xmlTextReaderValidatePop`, and since the `--valid` flag is enabled, it attempts to validate the node structure — at which point the use-after-free is triggered by dereferencing freed memory.

5.2 Security Implications

This vulnerability results in a heap use-after-free (UAF) in libxml2's XML streaming logic. Use-after-free vulnerabilities are a serious memory safety bug, which can lead to denial-of-service, memory corruption, or in some cases, arbitrary code execution.

5.2.1 Triggerability

The bug only occurs when XML is processed using the `xmlTextReader` API with both `--xinclude` and `--valid` flags enabled. These flags would need to be enabled, but their usage is realistic in applications that validate XML documents.

Libxml2 is not just a standalone tool. It is also used as a dependency in many programs, including GNOME, Python modueles (such as `lxml`, `libxslt`) [Sta23, Pro23b], and many other libraries that parse XML[con23]. Any one of these applications that perform XML parsing on user inputs with options could be vulnerable to this issue. Additionally, since XInclude allows references to external files, attackers can trigger the bug remotely by submitting specially crafted XML documents to trigger the bug remotely.

5.2.2 Exploitability

The use-after-free affects internal structures that are allocated on the heap. These structures may contain user-controlled data, such as XML element names and attributes, since these are directly taken from the parsed document.

This is particularly dangerous because if the attacker can influence the content of these freed structures and reclaim their memory with controlled data, the application may subsequently interpret that data as valid.

Depending on heap layout, the bug could allow:

- **Denial of service** — immediate and deterministic crashes upon dereferencing freed memory if ASAN is enabled or reused memory has issues.

- **Information Leak** — if the freed memory is reallocated, stale data may be exposed.
- **Arbitrary code execution** — if attacker can allocate certain values on the heap, the use after-free can lead to overwriting function pointers or control structures

Although exploitation is non-trivial and would require heap layout manipulation, it is entirely possible. The vulnerability arises in a parser operating on attacker-supplied input, which significantly increases the risk.

Furthermore, the use-after-free is silent during normal execution and does not cause a crash unless ASAN is enabled. Without ASAN, the application continues execution with corrupted memory, which makes the issue harder to detect and more dangerous. Without ASAN, the application continues execution with corrupted memory, making the issue both harder to detect and more dangerous. Even when ASAN is enabled, a sufficiently knowledgeable attacker could potentially evade detection by adjusting memory accesses to avoid touching redzones or shadow memory regions, bypassing ASAN's checks.

5.2.3 Fixes

Although this bug has already been patched, we will talk about this fix and why it works.

The fix adds a missing check of the return value from `xmlXIncludeProcessNode`. Previously, this function could return an error (`-1`), but the caller did not verify the result and continued processing. The maintainer proposed the following fix:

Listing 4: Proposed fix to `xmlTextReaderRead` in `xmlreader.c`

```
-        xmlXIncludeProcessNode(reader->xincctxt, reader->node);
+        /*
+         * If the XInclude expansion fails, the XInclude node and its
+         * children will be left in the document unmodified. We shouldn't
+         * try to expand nested XIncludes, so better stop on error.
+         */
+        if (xmlXIncludeProcessNode(reader->xincctxt, reader->node) < 0) {
+            reader->mode = XML_TEXTREADER_MODE_ERROR;
+            reader->state = XML_TEXTREADER_ERROR;
+            return(-1);
+        }
```

The patch ensures that if the function fails, parsing is halted or error handling is correctly invoked, preventing the program from continuing in an invalid state.

Additionally, condition that enabled the second call to `xmlXIncludeProcessNode` was modified to check for backtracking.

Listing 5: Proposed fix to `xmlTextReaderRead` in `xmlreader.c`

```
1428   * Handle XInclude if asked for
1429      */
1430     if ((reader->xinclude) && (reader->in_xinclude == 0) &&
1431 +       (reader->state != XML_TEXTREADER_BACKTRACK) &&
1432         (reader->node != NULL) &&
1433         (reader->node->type == XML_ELEMENT_NODE) &&
1434         (reader->node->ns != NULL) &&
```

This prevents the freed structures being put back into the document during backtracking, and therefore preventing the use-after-free bug.

5.2.4 Conclusion

This bug is remotely triggerable in specific but realistic configurations and occurs in widely used software. While exploitation is non-trivial, the vulnerability causes a real and reproducible memory corruption. Given the nature of the vulnerability, its remote triggerability, and the widespread adoption of libxml2, we consider this bug as high severity. The severity of the bug is supported by the assignment of a CVE, the embargo prior to public disclosure, and the fact that Apple received a $5,000 bug bounty in recognition of the discovery.

References

[con23] Wikipedia contributors. libxml2 — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Libxml2, 2023.

[Cor24] MITRE Corporation. Cve-2024-25062. https://www.cve.org/CVERecord?id=CVE-2024-25062, 2024.

[Goo23] Google. oss-fuzz-introspector. https://introspector.oss-fuzz.com/, 2023.

[Pro23a] GNOME Project. libxml2 issue #604. https://gitlab.gnome.org/GNOME/libxml2/-/issues/604, 2023.

[Pro23b] GNOME Project. Python bindings — libxslt wiki. https://gitlab.gnome.org/GNOME/libxslt/-/wikis/Python-bindings, 2023.

[Sta23] Stack Overflow users. Python packages depending on libxml2 and libxslt. https://stackoverflow.com/questions/1365075/python-packages-depending-on-libxml2-and-libxslt, 2023.