



## Penetration Test Report

NLNet Imap-Codec

V 1.0

Amsterdam, May 30th, 2025

Confidential

## Document Properties

Client	NLNet Imap-Codec
Title	Penetration Test Report
Target	Imap-Codec
Version	1.0
Pentester	Neha Chriss
Authors	Neha Chriss, Marcus Bointon
Reviewed by	Marcus Bointon
Approved by	Melanie Rieback

## Version control

Version	Date	Author	Description
0.1	October 25th, 2024	Neha Chriss	Initial draft
0.2	March 12th, 2025	Marcus Bointon	Review
1.0	May 30th, 2025	Marcus Bointon	1.0

## Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Radically Open Security B.V. is registered at the trade register of the Dutch chamber of commerce under number 60628081.

# Table of Contents

<b>1</b>	<b>Executive Summary</b>	<b>4</b>
1.1	Introduction	4
1.2	Scope of work	4
1.3	Project objectives	4
1.4	Timeline	4
1.5	Results In A Nutshell	4
1.6	Summary of Findings	5
1.6.1	Findings by Threat Level	6
1.6.2	Findings by Type	6
1.7	Summary of Recommendations	6
<b>2</b>	<b>Methodology</b>	<b>8</b>
2.1	Planning	8
2.2	Risk Classification	10
<b>3</b>	<b>Fuzzing and Analysis</b>	<b>11</b>
<b>4</b>	<b>Findings</b>	<b>12</b>
4.1	CLN-009 — Fuzzing Efficiency Comparison – Command vs Command to Bytes and Back	12
4.2	CLN-010 — Mutation Strategy Comparison Across All Targets	19
4.3	CLN-012 — Recommended Dictionary – Response Bytes Target	22
4.4	CLN-013 — Recommended Dictionary – Command Bytes Target	24
<b>5</b>	<b>Future Work</b>	<b>26</b>
<b>6</b>	<b>Conclusion</b>	<b>27</b>
<b>Appendix 1</b>	<b>Testing team</b>	<b>28</b>

# 1 Executive Summary

## 1.1 Introduction

Between February 23, 2024 and March 15, 2024, Radically Open Security B.V. carried out a penetration test for NLNet Imap-Codec.

This report contains our findings as well as detailed explanations of exactly how ROS performed the penetration test.

## 1.2 Scope of work

The scope of the penetration test was limited to the following target:

- Imap-Codec

The scoped services are broken down as follows:

- Evaluate and benchmark Fuzz Testing of Imap-Codec's fuzz targets: 5 days
- **Total effort: 5 days**

## 1.3 Project objectives

ROS will perform an evaluation of the fuzz test configuration for the imap-codec code base with Imap-Codec in order to assess the efficiency of the library's fuzz targets. To do so ROS will review code and run fuzz tests of Imap-Codec in an attempt to identify vulnerabilities and failure modes.

## 1.4 Timeline

The security audit took place between February 23, 2024 and March 15, 2024.

## 1.5 Results In A Nutshell

We discovered 4 Moderate-severity issues during this penetration test.

During this crystal-box engagement we discovered no panics and/or faults in imap-codec code. However, we did observe that imap-codec should further leverage structure-aware fuzzing.

Our research also found that the optimal duration for fuzzing imap-codec targets is between 4-7 days, for all targets. This would indicate that the current method of launching fuzz operations via GitHub actions CI with a 3-hour duration may be

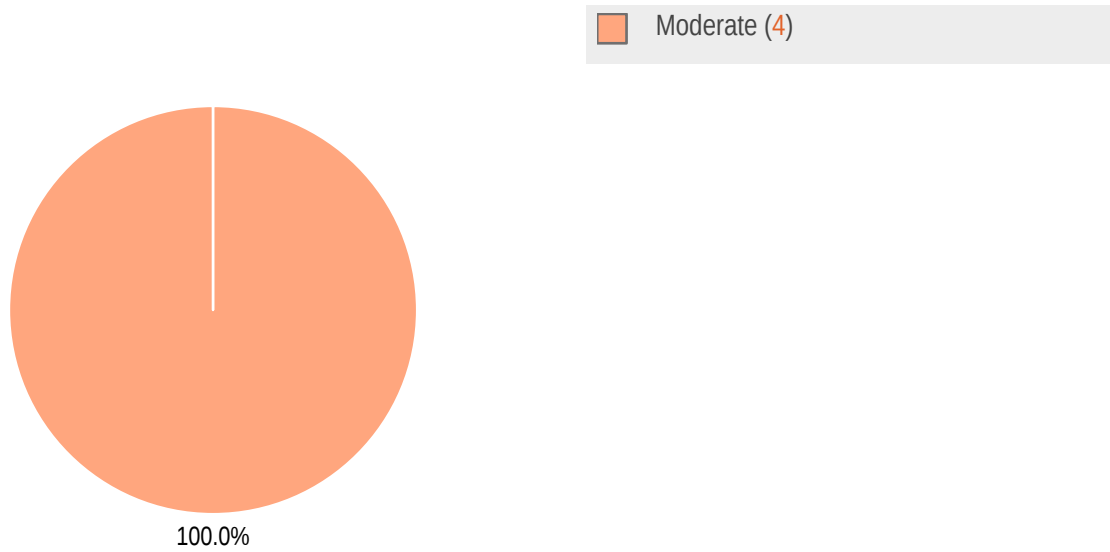
insufficient. Our recommendations include seeking a fuzzing platform that can stress test imap-codec parsers for longer periods.

These issues may prevent the imap-codec project from capturing valid vulnerabilities and preventing serious security bugs.

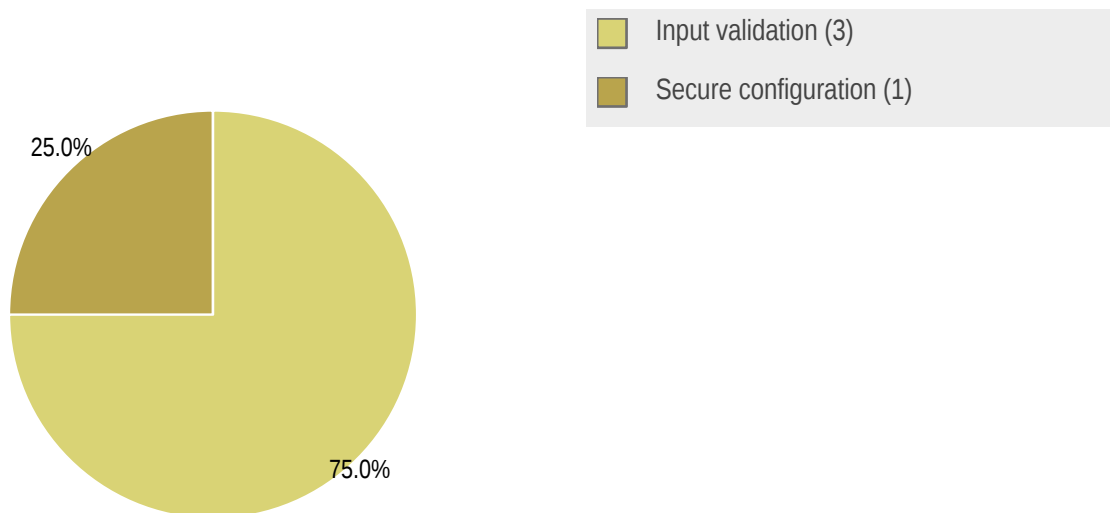
## 1.6 Summary of Findings

ID	Type	Description	Threat level
CLN-009	Input Validation	A comparison between Command and Command to Bytes and Back fuzzing targets reveals fuzzing trends and most effective mutation strategies.	Moderate
CLN-010	Secure Configuration	Our analysis of imap-codec's fuzz coverage reveals the Response/Command targets achieve the highest raw coverage, while the most efficient fuzzing actually occurs in smaller targets like Authenticate. The to_bytes variants generally perform better, suggesting that byte-level mutations are more effective for discovering new paths in the codebase.	Moderate
CLN-012	Input Validation	After a completed fuzzing run, cargo fuzz/libfuzzer recommends a set of strings for further dictionary-guided fuzzing.	Moderate
CLN-013	Input Validation	After a completed fuzz run, cargo fuzz/libfuzzer recommends a set of strings for further dictionary-guided fuzzing.	Moderate

### 1.6.1 Findings by Threat Level



### 1.6.2 Findings by Type



## 1.7 Summary of Recommendations

ID	Type	Recommendation
CLN-009	Input Validation	<ul style="list-style-type: none"><li>Favour structural mutations.</li></ul>

CLN-010	Secure Configuration	<ul style="list-style-type: none"><li>• Byte-level combined with CopyPart mutations is the most effective approach.</li><li>• Optimal fuzzing would take 4-7 days to achieve &gt;90% coverage.</li></ul>
CLN-012	Input Validation	<ul style="list-style-type: none"><li>• Use the recommended strings.</li></ul>
CLN-013	Input Validation	<ul style="list-style-type: none"><li>• Review code that handles escaping, control characters, binary data, and path resolution.</li></ul>

## 2 Methodology

### 2.1 Planning

We first evaluated IMAP-Codec's fuzzing strategies for their configuration and target types. Due to resource limitations of running in a localized environment, we implemented a containerized approach to run targets independently while collecting logs for later analysis.

The rust-based *cargo fuzz* cli wrapper for libfuzzer was used to execute imap-codec's fuzz targets.

#### Fuzz Target Review

We reviewed Imap-Codec fuzz targets for functionality and effectiveness. Fuzz targets are split into two groups:

#### Parsing Test Targets

These targets focus on testing parsing routines, ensuring parsers don't panic or crash and can handle malformed input. These targets take random input and attempt to parse it. If parsing succeeds:

- The parsed object is serialized.
- The serialized output is parsed again.
- The results are compared to ensure they match.

#### Misuse-Resistance Test Targets

These targets test for API safety and defense against misuse. They use the *Arbitrary* trait which produces a structured instance of the target type. These instances must be parseable and valid.

The misuse-resistance targets should catch all relevant cases in which it is possible to create invalid messages objects through the public API.

#### Fuzzing Platform

Each fuzz target was run in a dedicated AWS fargate container with 4vCPUs and 8 GB Mem allocated.

#### Data Analysis

Fuzz logs were processed and analyzed offline, and are in the standard libfuzzer format. Each log line contains a set of both core and metrics fields. The focus was on analyzing coverage time, CPU time per iteration, memory utilization, and any CRASH or PANIC logs generated from the fuzz iteration. Mutation strategies were compared for each fuzz run and compared across the corresponding class of fuzz targets.

Core fields include:

- **TARGET:** The fuzzing target being tested (e.g., command, greeting, response)
- **RUN\_ID:** A unique hexadecimal identifier for the fuzzing run
- **ITERATION:** The current fuzzing iteration number (e.g., #34, #52)
- **STATUS:** Current status of the fuzzing (e.g., NEW, INITED)
- **CRASH:** If a crash was detected, the crash information is displayed



- **TIME:** The time taken to run the fuzzing target

Metrics fields include:

- **cov:** Coverage count (number of code paths covered)
- **ft:** Features count (number of code features discovered)
- **corp:** Corpus statistics in format count/sizeKb (e.g., `2/3b` means 2 test cases totaling 3 bytes)
- **lim:** Current size limit for generated inputs
- **exec/s:** Executions per second
- **rss:** Resident Set Size (memory usage) in megabytes
- **L:** Length range of test cases (e.g., `L: 1/2` means current/max length)

Mutation Strategy Information (MS): The logs include mutation strategies applied to generate new test cases, shown as `MS: [count] [operations]`.

Common operations include:

- **CopyPart:** Mutate by adding or reusing parts of valid target structures
- **CrossOver:** Combines multiple test cases/valid fuzz target structures
- **ChangeBit:** Modifies individual bits
- **EraseBytes:** Removes existing bytes
- **ChangeByte:** Mutates data by changing 1 bit
- **InsertBytes:** Inserts a byte
- **InsertRepeatedBytes:** Adds repeated byte sequences, useful for finding buffer related issues.
- **ChangeBinInt:** Modifies binary integers
- **ShuffleBytes:** Mutates data by erasing bytes
- **PersAutoDict:** Dictionary-based mutations
- **ChangeASCIIInt:** Modifies ASCII Integers

Special Statuses:

- **INIT:** Initial state when fuzzing starts
- **NEW:** New interesting test case found (increased coverage or features)
- **REDUCE:** New interesting test case found (increased coverage or features)

Note that our analysis also covered mutation strategy effectiveness wherever those mutation strategies were implemented, including when used in combination — as in `MS: 3 ChangeBit-ChangeBit-InsertByte-` — where the operation ChangeBit is applied twice, followed by InsertByte, in sequence.

## 2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see <http://www.pentest-standard.org/index.php/Reporting>.

These categories are:

- **Extreme**  
Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.
- **High**  
High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**  
Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**  
Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**  
Low risk of security controls being compromised with measurable negative impacts as a result.

## 3 Fuzzing and Analysis

We were able to gain information about the software and infrastructure through the following automated scans. Any relevant scan output will be referred to in the findings.

- Cargo Fuzz – <https://github.com/rust-fuzz/cargo-fuzz>
- K8s – <https://kubernetes.io/>
- Pandas – <https://pandas.pydata.org/>

## 4 Findings

We have identified the following issues:

### 4.1 CLN-009 — Fuzzing Efficiency Comparison – Command vs Command to Bytes and Back

**Vulnerability ID:** CLN-009

**Vulnerability type:** Input Validation

**Threat level:** Moderate

#### Description:

A comparison between Command and Command to Bytes and Back fuzzing targets reveals fuzzing trends and most effective mutation strategies.

#### Technical description:

##### Command Fuzz Target Summary

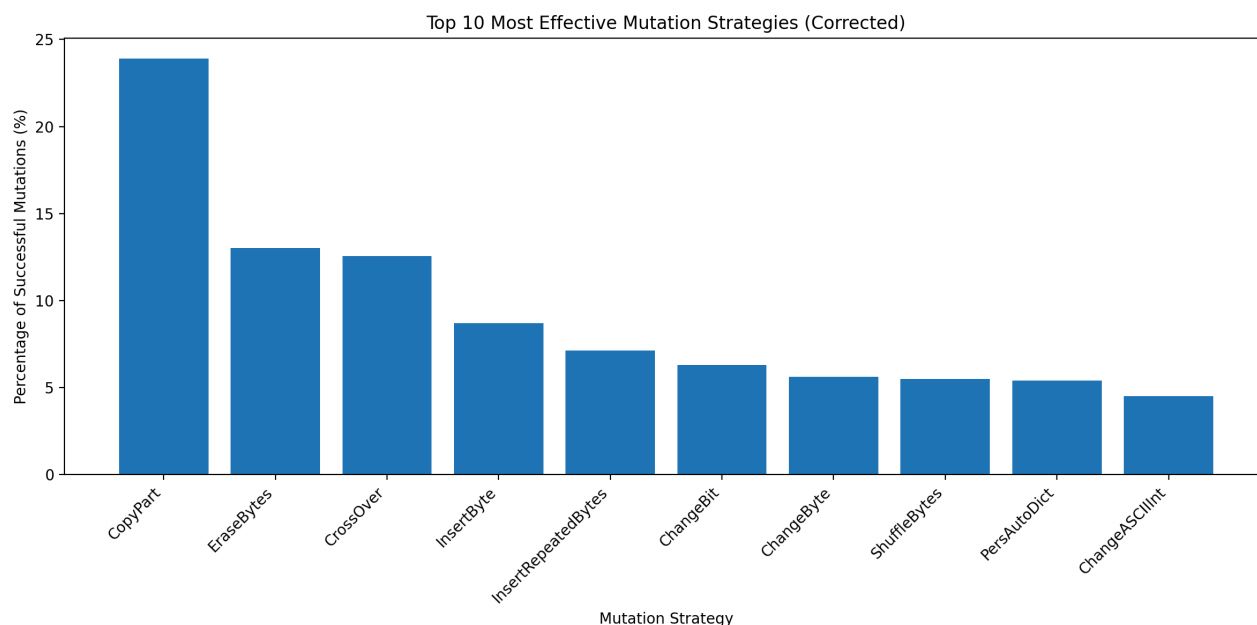
Like many of the imap-codec fuzzing targets, the Command target is most productive in the early phases of its run.

The CopyPart mutation strategy was the most successful strategy based on the run, suggesting that reusing valid IMAP command structures was highly effective. EraseBytes and CrossOver were tied second for most effective strategies. The success of Erasebytes would indicate that removing parts of commands leads to interesting edge cases, while CrossOver would indicate combining valid commands led to significant coverage increase.

Operation	Count	Percentage
CopyPart	701	23.89
EraseBytes	382	13.02
CrossOver	368	12.54
InsertByte	255	8.69
InsertRepeatedBytes	209	7.12
ChangeBit	185	6.31
ChangeByte	165	5.62
ShuffleBytes	161	5.49

PersAutoDict	158	5.39
ChangeASCIInt	132	4.50

A side by side comparison for the fuzzing strategies:



For the Command fuzz target, structural mutations (CopyPart, EraseBytes, CrossOver) were significantly more effective than byte-level mutations for finding new paths in the IMAP command parser. This makes sense given the structured nature of IMAP commands, where maintaining syntactic validity while exploring edge cases is important. On the other hand, the bulk of the byte level mutations were less effective.

Mutation Strategies from most effective to least effective:

1. Structural mutations (CopyPart, EraseBytes, CrossOver) - ~49% of successful findings
2. Byte-level insertions and modifications - ~20% combined
3. Advanced dictionary and ASCII mutations - ~10%

## Coverage Growth

As far as coverage growth, the early phase was extremely productive, finding 97% of the total coverage. This indicates rapid initial growth of test cases that reveal new paths and features.

Final coverage: Final coverage: 5,114

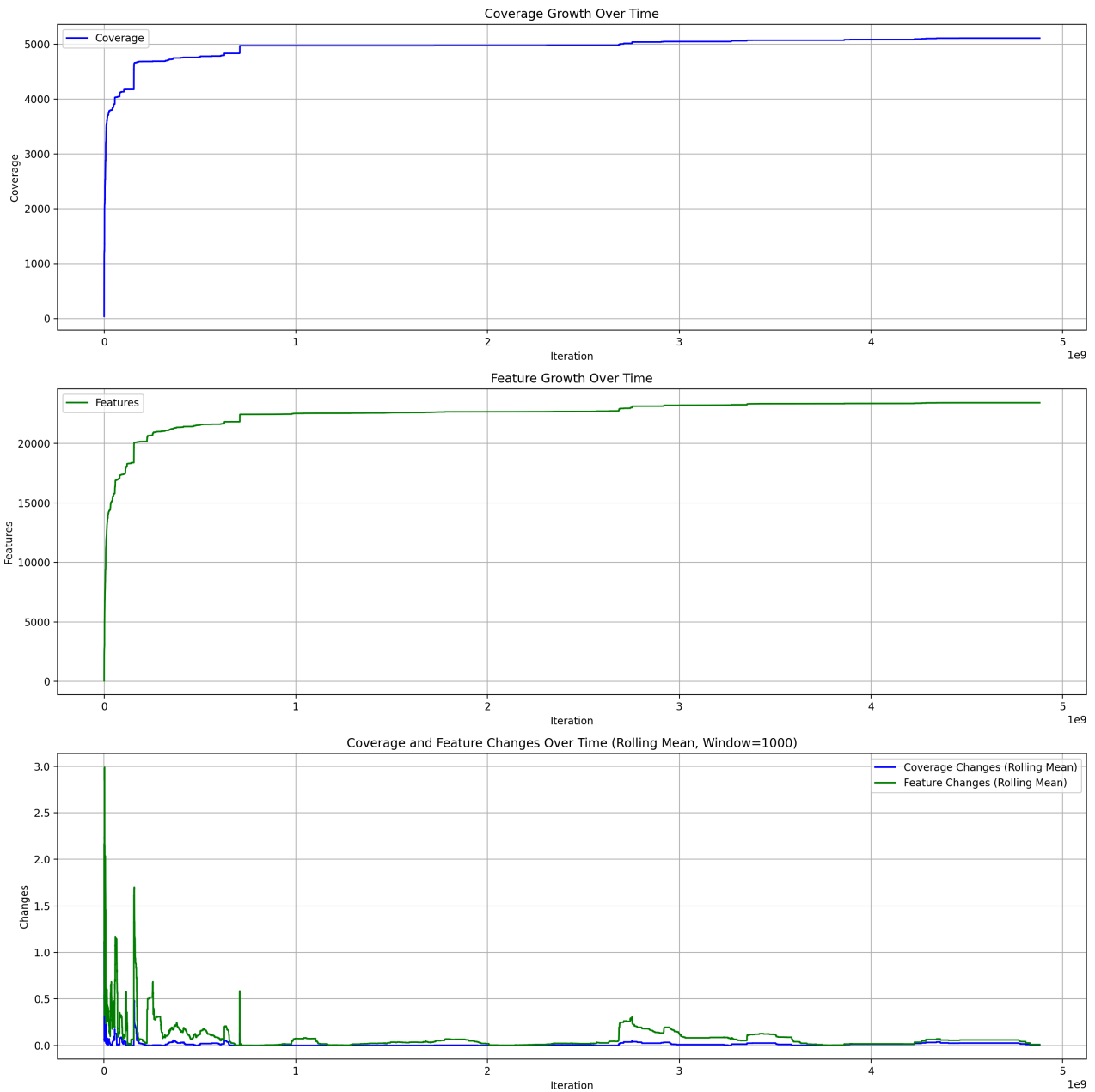
Total coverage increase: Total coverage increase: 5,069

Feature Discovery:

**Initial features:** 46

**Final features:** 23,434

**Total features increase: 23,388**



**Coverage Growth Statistics** Number of coverage-increasing events: 958 Average coverage increase when found: 5.29  
Largest single coverage increase: 417

**Feature Growth Statistics** Number of feature-increasing events: 3,924 Average feature increase when found: 5.96  
Largest single feature increase: 468

### Coverage Growth by Phase

Early phase gain: 4,930 paths Middle phase gain: 76 paths Late phase gain: 63 paths

Growth Pattern: Rapid initial growth in both coverage and features Steady middle phase with continued discoveries Late phase showing diminishing returns but still finding new paths Discovery Events: 958 coverage-increasing events 3,924

feature-increasing events Large jumps occurred (max 417 new paths at once) Most Productive Phase: The early phase was extremely productive, finding 97% of the total coverage

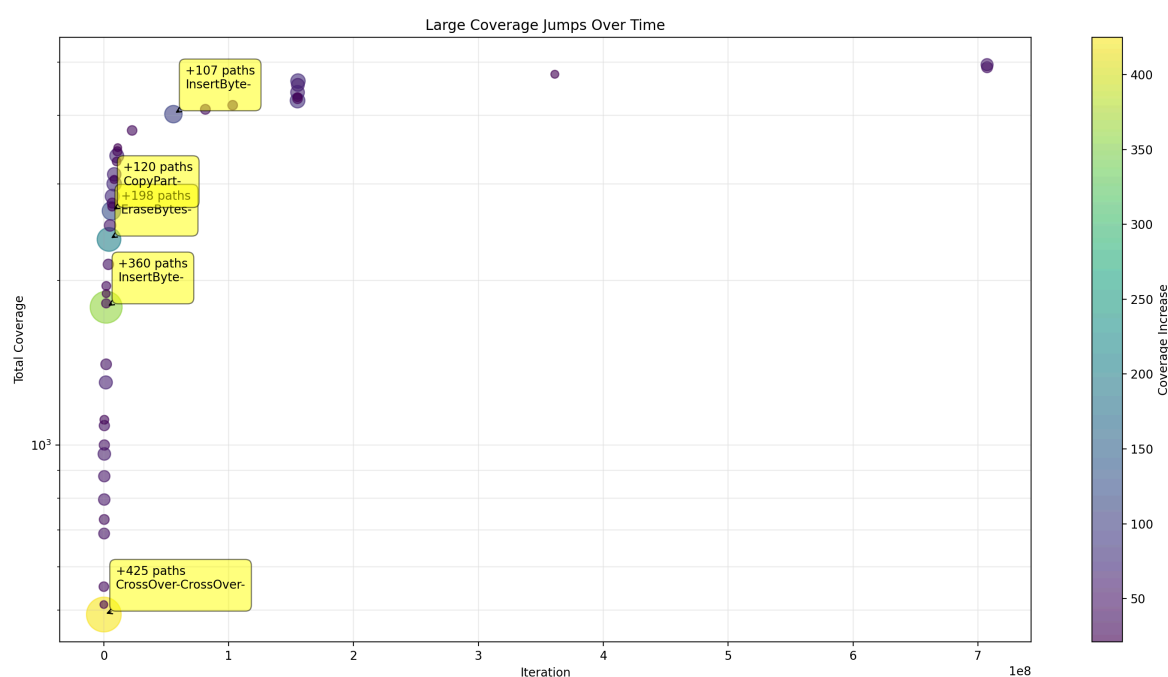
**Discovery Events:** 958 coverage-increasing events, 3,924 feature-increasing events Large jumps occurred (max 417 new paths at once)

**Efficiency:** Coverage increased by 11,264% (45 to 5,114 paths) Features increased by 50,943% (46 to 23,434 features)

## Coverage Jumps

Analyzing the most significant coverage jumps shows how different mutation sequences affect code coverage over iterations. The most effective mutation (largest jump) in this fuzz run was at iteration 1,285, where CrossOver-CrossOver- increased coverage by 425 paths. Most significant discoveries happened in early phase (41 jumps vs 3 jumps in middle/late phases). CrossOver and CopyPart were the most effective mutations for large coverage increases. The frequency of large jumps decreased over time, but even late-stage jumps found 70+ new paths.

Average time between large jumps was about 259K iterations, showing consistent but spreading-out discoveries.



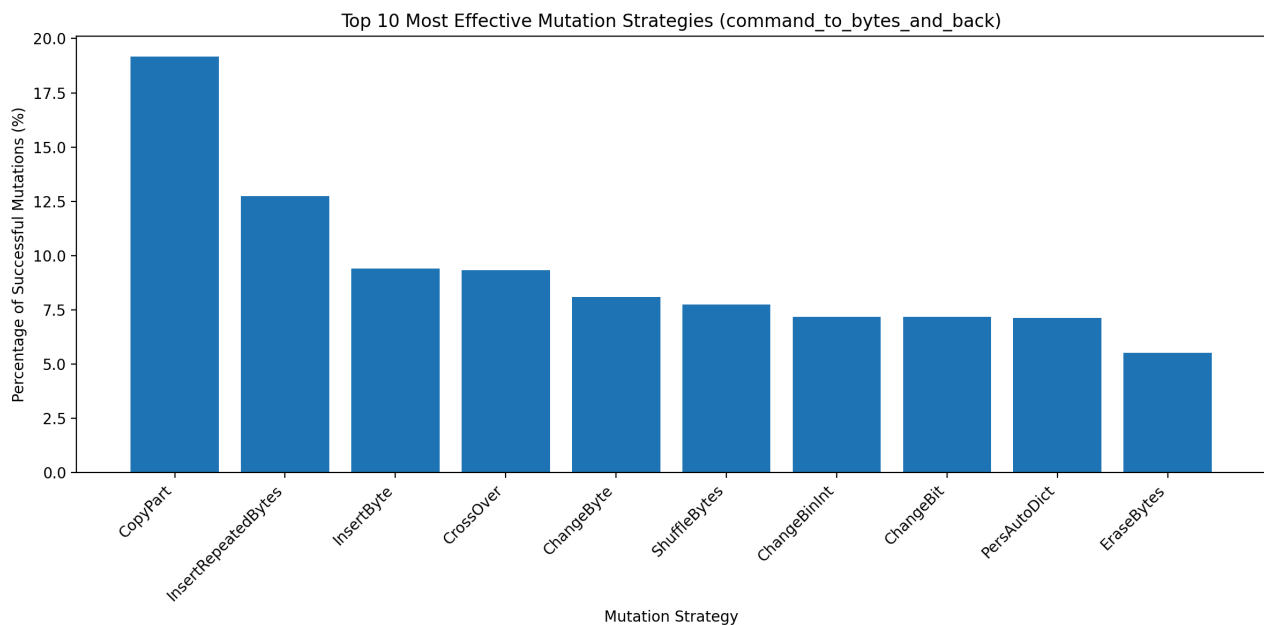
Iteration	Coverage Increase	Total Coverage	Mutation Sequence
1,285	425	491	CrossOver-CrossOver-
1,849,779	360	1,786	InsertByte-
4,136,825	198	2,374	EraseBytes-
6,082,474	120	2,677	CopyPart-
55,699,922	107	4,019	InsertByte-
155,127,759	76	4,255	ChangeBit-

8,259,272	74	3,000	PersAutoDict
155,477,345	73	4,619	EraseBytes-
6,721,886	69	2,846	ChangeByte-CMP
10,341,579	68	3,374	InsertRepeatedBytes-

## Command to Bytes and Back

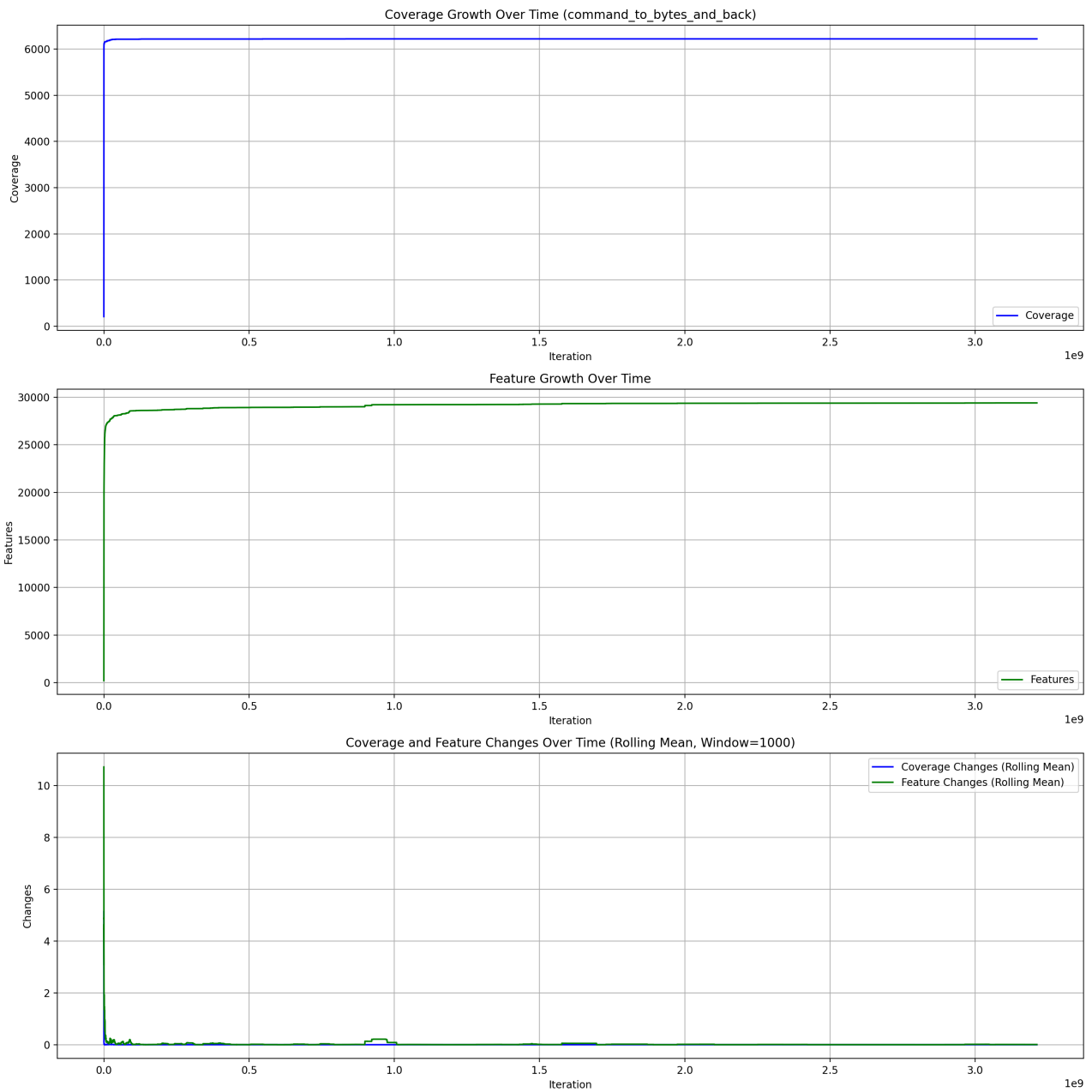
Taking the above analysis, we can quickly plot and compare the effectiveness of the "Bytes and Back" Command target (`command_to_bytes_and_back.rs`). In short, the results indicate that the `command_to_bytes_and_back` target is more amenable to fuzzing. Including results that demonstrate faster initial discovery, higher overall coverage, more efficient mutation strategies, and a better feature discovery rate.

We see similar results to the Command target in regard to Mutation Strategies:

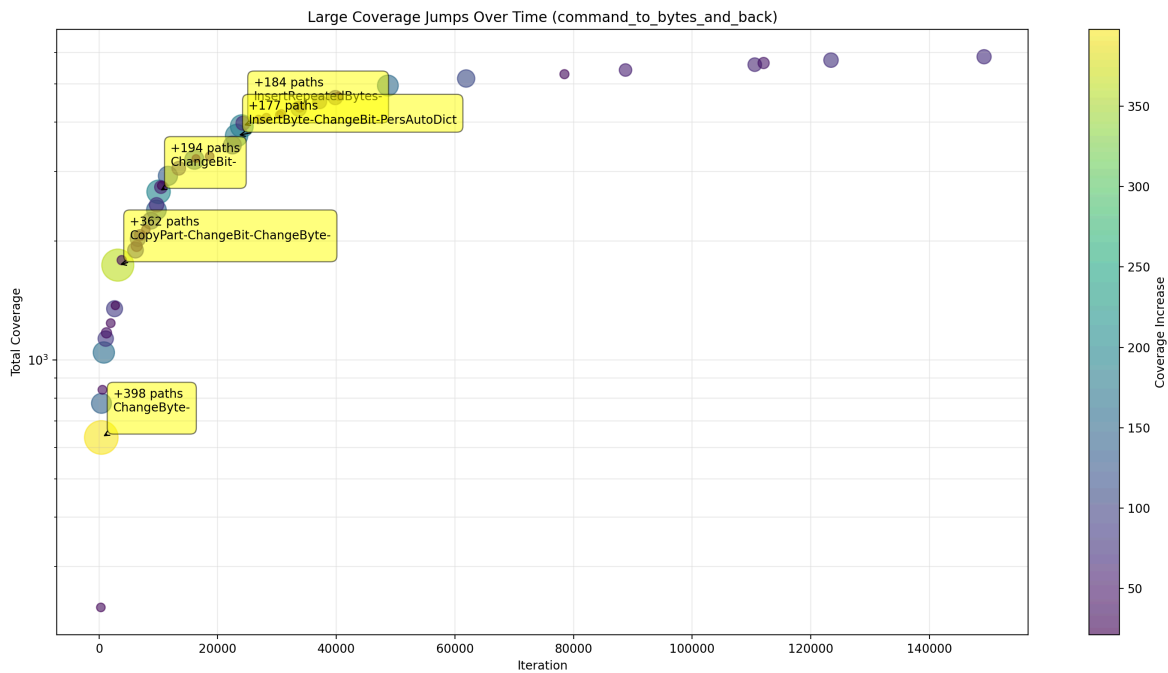


Coverage Visualization:





Coverage Jumps Visualization:



Key Differences from Command Target:

**Initial State** Started with higher initial coverage (208 vs 45) Higher initial features (209 vs 46)

**Growth Pattern** More aggressive early growth All significant gains occurred in early phase Smaller individual jumps (top jump of 398 paths vs previous 425)

**Mutation Effectiveness** More balanced distribution among top mutation strategies

CopyPart was most effective (19.16% of successful mutations) InsertRepeatedBytes second most effective (12.74%)

**Coverage Jumps** Similar number of large jumps (49 vs 44) More concentrated in early phase (42 early jumps) Faster discovery rate (median 2,674 iterations between jumps vs previous 1.6M)

**Final Results** Higher final coverage (6,220 vs 5,114) Higher feature count (29,407 vs 23,434) Faster convergence to final coverage

Recommendation:

- Favour structural mutations.

## 4.2 CLN-010 — Mutation Strategy Comparison Across All Targets

**Vulnerability ID:** CLN-010

**Vulnerability type:** Secure Configuration

**Threat level:** Moderate

### Description:

Our analysis of imap-codec's fuzz coverage reveals the Response/Command targets achieve the highest raw coverage, while the most efficient fuzzing actually occurs in smaller targets like Authenticate. The `to_bytes` variants generally perform better, suggesting that byte-level mutations are more effective for discovering new paths in the codebase.

### Technical description:

Structural mutations (CopyPart, CrossOver) were significantly more effective than byte-level mutations (save EraseBytes) for finding new paths in the IMAP APIs and parsers. This makes sense given the structured nature of IMAP APIs and the protocol, where maintaining syntactic validity while exploring edge cases supports effective discovery of edge cases.

The top performer among mutation strategies is CopyPart which was the most consistent in finding new paths in parsers and APIs and increasing coverage as a result. Loosely speaking, the structural variant mutation strategies would be most effective at discovering logical bugs and byte variants better at finding memory corruption issues, in line with the imap-codec project's distinction of these fuzz targets into

### Top Performers

#### CopyPart Mutation Strategy

- Consistently 15-24% success rate
- Dominant across all targets
- Most reliable mutation type

#### Bytes-based Mutations

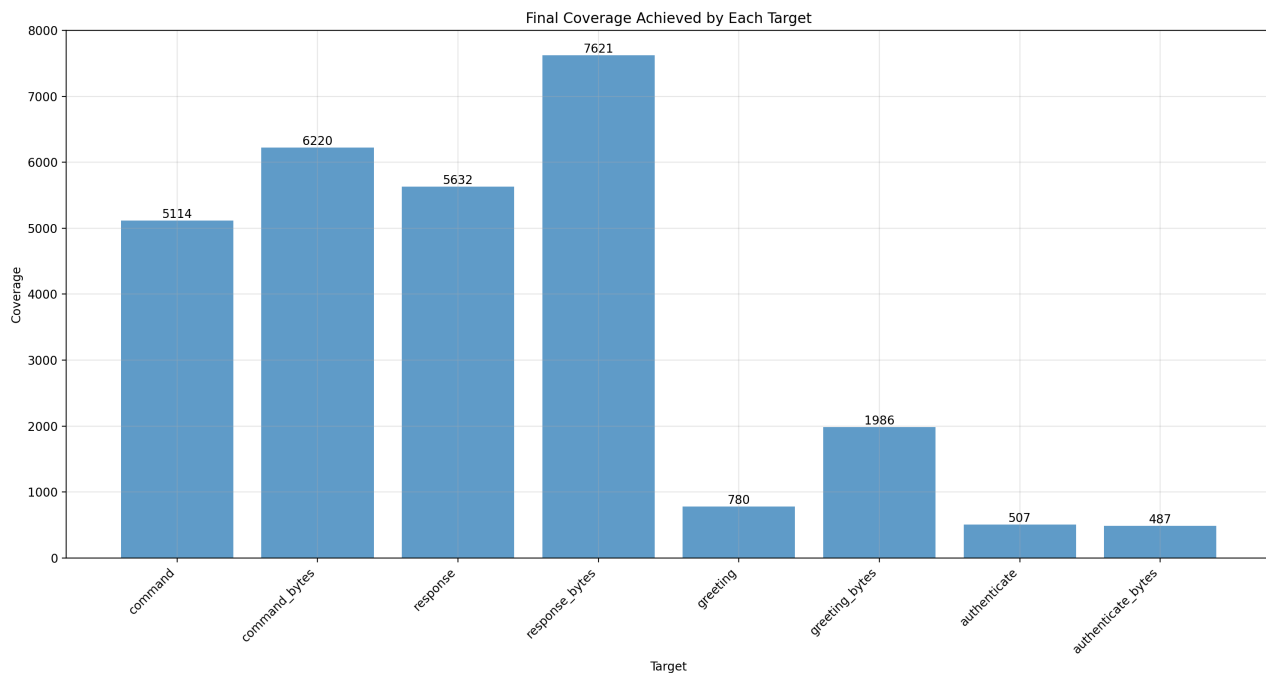
- ShuffleBytes + CrossOver: ~26% combined
- More effective in larger targets
- Better feature discovery rate

### Overall:

- Response/Command targets achieve the highest coverage (5-7K)

- Greeting targets show moderate coverage (0.7-2K)
- Authenticate targets have the lowest coverage (0.4-0.5K)
- CopyPart is consistently the top mutation across all targets
- Bytes variants generally show higher coverage except for authenticate

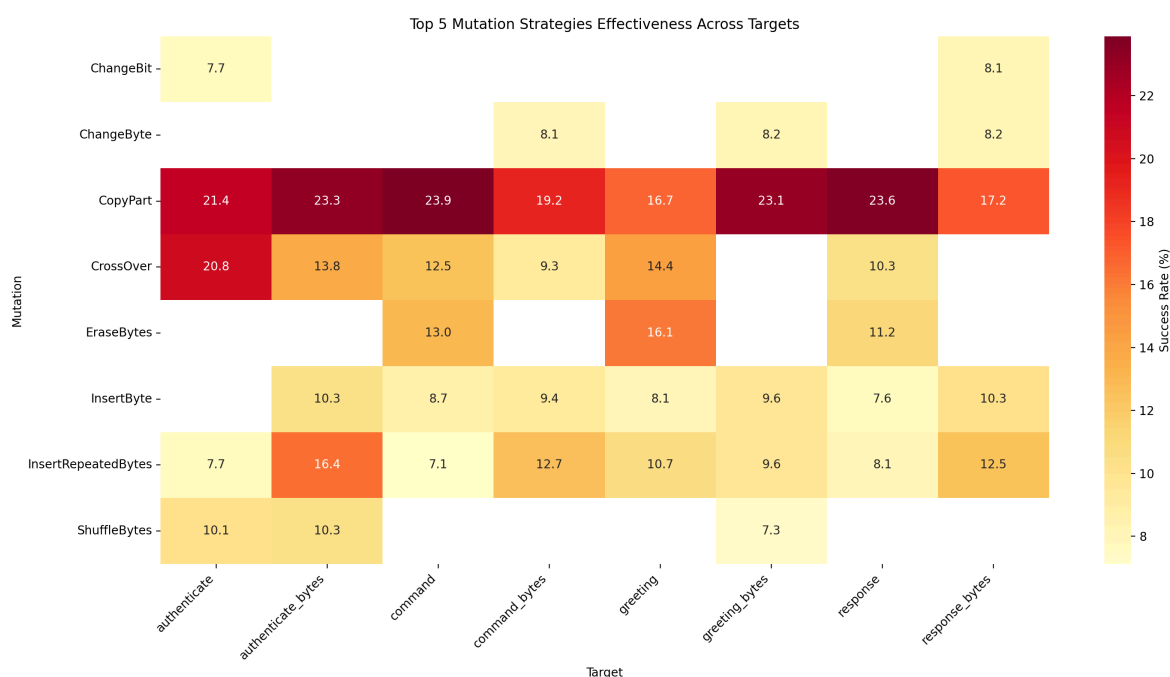
### Coverage Comparison Across Non-Byte vs Byte Targets



### Bytes vs Non-bytes Patterns:

- Bytes variants generally show higher final coverage
- Exception: Authenticate where base variant performs better
- Bytes variants show more feature discovery

The following heat map demonstrates the general distribution of mutation strategies and code coverage growth across all available targets:



## Impact:

### Growth Patterns:

- Response\_bytes shows highest final coverage (~7K)
- Command\_bytes follows with (~6K)
- Authenticate targets show lowest but most efficient growth

### Efficiency Leaders:

- Authenticate\_bytes: Highest coverage efficiency (2.213 per iteration)
- Response\_bytes: Best feature discovery rate (0.875 per iteration)
- Command targets: Most consistent growth pattern

### Significant Jumps:

- Response: Most coverage jumps (~350+)
- Response: Highest feature jumps (~800+)
- Authenticate: Fewest jumps but highest efficiency

## Recommendation:

The most time, coverage and findings efficient configuration for cargo fuzz should include byte-level mutation combined with the CopyPart mutation strategy. It may be necessary to run a dedicated fuzz operation with these specific settings to fully take advantage of CPU and memory resources.

The optimal length of fuzzing would be 4-7 days to achieve 90%+ mature coverage for the large code bases - command and response targets.

## 4.3 CLN-012 — Recommended Dictionary – Response Bytes Target

**Vulnerability ID:** CLN-012

**Vulnerability type:** Input Validation

**Threat level:** Moderate

### Description:

After a completed fuzzing run, cargo fuzz/libfuzzer recommends a set of strings for further dictionary-guided fuzzing.

### Technical description:

```
"\003\000\000\000\000\000\000\000" # Uses: 87
"\001\000\000\003" # Uses: 84
"+A{" # Uses: 92
"\377\377\377\377" # Uses: 61
"_A67" # Uses: 84
"\377\377\377\377\377\377\377\377" # Uses: 63
"NT" # Uses: 91
"\\\\" # Uses: 86
"\004\000\000\000\000\000\000\000" # Uses: 79
"\000\000\000\000" # Uses: 89
"\001\000\000\000\000\000\000\002" # Uses: 93
"\001\011" # Uses: 91
"<=HT" # Uses: 80
"binary" # Uses: 79
"\000\000\000\012" # Uses: 72
"\\000" # Uses: 79
"hlsube" # Uses: 63
"\001\000\000\000\000\000\000\000" # Uses: 60
"\n\221\014\315/\000\000" # Uses: 79
"\000\000\000\000\000\000\000\000-" # Uses: 81
"\"000\000\000" # Uses: 76
"kmq " # Uses: 85
"\001\000" # Uses: 75
"\014\000\000\000\000\000\000\000" # Uses: 60
"\001\000\000\000\000\000\000\000=" # Uses: 78
"xilr&s==" # Uses: 69
"-__nnose" # Uses: 51
```

```

"auth" # Uses: 70
"\305\377\377\377" # Uses: 77
"marked" # Uses: 71
"\377\006" # Uses: 74
"\010\000\000\000\000\000\000\000" # Uses: 68
"[\000\000\000" # Uses: 59
"\000\000\000\000\000\000\000\000" # Uses: 71
"\006UUUUUU" # Uses: 64
"\377\377\377\377\377\377\001" # Uses: 60
"bscriber" # Uses: 61
"=\000\000\000\000\000\000\000" # Uses: 63
"\001\000\000\000\000\000\000\010" # Uses: 68
"inbox" # Uses: 66
"sort" # Uses: 56
"E " # Uses: 80
"==w==-_2nnosb" # Uses: 68
"\\\"" # Uses: 66
"ccce" # Uses: 66
"\377\377\377\377\377\377\377\004" # Uses: 59
"\013\000\000\000" # Uses: 56
"\001\000\000\000" # Uses: 48
"\376\357\010\000" # Uses: 67
"\001\000\000\000\000\000\000\032" # Uses: 67
"unselect" # Uses: 48
"\000\000\000\000\000\000\000\004" # Uses: 32
"sasl-ir" # Uses: 50
"\377\377\377\016\377W\333\335" # Uses: 46
"= o*y" # Uses: 32
"9:::" # Uses: 42
"EN" # Uses: 35
"idle" # Uses: 27
"\377\377\377\377\377\377\377\177" # Uses: 34
"\022\000\000\000\000\000\000\000"

```

## Impact:

The suggested cases here are for:

- IMAP Command/Response Keywords – e.g. `binary`, `auth`, `idle`, `sasl-ir`
- Boundary Value Tests – e.g. `\377\377\377\377\377\377\377\377`
- Base64 and Encoding Tests – e.g. `"x!lr&s=="`, `"==w==-_2nnosb"`
- Length and Size Tests – e.g. `\004\000\000\000\000\000\000\000`

## Recommendation:

Use the recommended strings. They are designed to test:

- UTF-8 parsing robustness
- Buffer overflow prevention
- Escape sequence handling
- Command/response keyword recognition

- Base64 decoding
- Length field processing

## 4.4 CLN-013 — Recommended Dictionary – Command Bytes Target

**Vulnerability ID:** CLN-013

**Vulnerability type:** Input Validation

**Threat level:** Moderate

### Description:

After a completed fuzz run, cargo fuzz/libfuzzer recommends a set of strings for further dictionary-guided fuzzing.

### Technical description:

```
##### Recommended dictionary. #####
"\001\006" # Uses: 281
"\000" # Uses: 330
"\\\000" # Uses: 323
"\001\000\342\004" # Uses: 311
"\000\000/\314\262\216ET" # Uses: 283
"5\000\000\000\000\000\000\000" # Uses: 281
"\\\000" # Uses: 311
"\000\000\000" # Uses: 281
"\000\000\000\000\000\000\000\000" # Uses: 283
"\005\000\000\000" # Uses: 307
"\001\000" # Uses: 302
"\377\377\377L" # Uses: 275
"\000\000\000\000" # Uses: 258
"\001\001" # Uses: 305
"\002\000\000\000\000\000\000\000" # Uses: 271
"\001T" # Uses: 280
"\001\033" # Uses: 297
"\001\000\000\000\000\000\000\000\012" # Uses: 264
"\001\001Cp" # Uses: 296
"\000\000\000\363" # Uses: 304
"\001\000\000@\000\001\323\320" # Uses: 278
"s " # Uses: 292
"\377\377\377S" # Uses: 308
"PU" # Uses: 275
"\015\000\000\000\000\000\000\000" # Uses: 297
"\001\000\000\000\000\000\000\000" # Uses: 262
"0\003\000\000\000\000\000\000" # Uses: 237
"\377\377\377\377\377\377\377\010" # Uses: 256
"x " # Uses: 274
"# " # Uses: 250
"\012\000" # Uses: 272
```



"yycam" # Uses: 238

## Impact:

Based on the provided dictionary and codebase context, these string values represent different types of test cases for IMAP protocol parsing, particularly focusing on edge cases and potential vulnerabilities. These strings represent:

- Escape Sequences – e.g. `\\` – double backslash escape
- Null Byte Tests – e.g. `\000\000\000\000\000\000\000\000` – null
- Control Character Tests – e.g. `\012\000`; – newline character
- File Path Separators – e.g. `\000\000/\314\262\216ET`
- Binary Data – e.g. `\377\377\377\377\377\377\377\010`

## Recommendation:

- Review code that handles escaping, control characters, binary data, and path resolution.

## 5 Future Work

- **Retest of findings**

When mitigations for the vulnerabilities described in this report have been deployed, a repeat test should be performed to ensure that they are effective and have not introduced other security problems.

- **Regular security assessments**

Security is an ongoing process and not a product, so we advise undertaking regular security assessments and penetration tests, ideally prior to every major release or every quarter.

- **Structure Aware Fuzzing Target for Tags**

After validating the default fuzz targets, ROS produced a functional reference for structure aware fuzzing using imap-codec's 'tags' parser as a target. A refactor of the greetings, command, and auth fuzz targets would result in a more comprehensive coverage of the code base.

## 6 Conclusion

In this fuzzing test we evaluated several different fuzzing strategies for the IMAP codec, but discovered no panics and/or faults in imap-codec code.

Overall, we found the most effective approach is to use structural modifications rather than byte alterations or string substitutions. The structural approach provided a far higher incidence of interesting results, indicating that this strategy is the most worthwhile to follow in tests run against the source code.

The byte and string substitution approaches still produced interesting results, but with a far lower hit rate, suggesting that the codec is more robust against this kind of malformed input.

We also observed that the current fuzzing approach, time-limited by GitHub capacity limits, is probably insufficient, and we recommend increasing the time allocated for fuzz tests to 4–7 days.

We recommend using these findings to build more effective fuzzing targets, and to invest resources in long-term fuzzing runs, to enhance the long-term robustness of the IMAP codec.

Finally, we want to emphasize that security is a process that must be continuously evaluated and improved – this penetration test is just a one-time snapshot. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this pentest report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report

## Appendix 1 Testing team

Neha Chriss	Neha Chriss is a Senior Security Engineer and a classic builder/breaker, with over two decades in infrastructure and product security. Her recent work has been testing payments and banking stacks, but she has hacked various types of systems, from SmartTVs and DRM to the GSN (Global Seismographic Network) and national energy management systems.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.