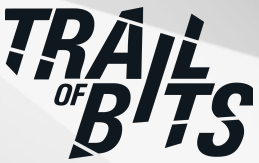

cURL HTTP/3 Components

Security Assessment

February 22, 2024

Prepared for:
Daniel Stenberg, cURL
Organized by Open Source Technology Improvement Fund, Inc.

Prepared by: **Vasco Franco, Emilio López, and Spencer Michaels**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits **1**
Notices and Remarks **2**
Table of Contents **3**
Project Summary **4**
Executive Summary **5**
Project Goals **7**
Project Targets **8**
Project Coverage **9**
Fuzzing Coverage Assessment **10**
Assessment Overview 10
HTTP/1 and HTTP/2 11
HTTP3 11
BUFQ Implementation 13
Strategic Fuzzing Recommendations 14
Codebase Maturity Evaluation **16**
Summary of Findings **18**
Detailed Findings **19**
1. OSS-Fuzz coverage silently dropped significantly 19
2. curl_fuzzer is ineffective 20
A. Vulnerability Categories **21**
B. Code Maturity Categories **23**
C. Dolev-Yao TLS Fuzzing Using tlspuffin **25**

Project Summary

Contact Information

The following project manager was associated with this project:

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

Anders Helsing, Engineering Director, Application Security
anders.helsing@trailofbits.com

The following consultants were associated with this project:

Vasco Franco, Consultant
vasco.franco@trailofbits.com

Emilio López, Consultant
emilio.lopez@trailofbits.com

Spencer Michaels, Consultant
spencer.michaels@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
December 8, 2023	Pre-project kickoff call
December 21, 2023	Status update meeting #1
January 4, 2024	Delivery of report draft; report readout meeting
February 22, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

The Open Source Technology Improvement Fund engaged Trail of Bits to review the security of cURL's newly added HTTP/3 components.

A team of three consultants conducted the review from December 8 to 26, 2023, for a total of six engineer-weeks of effort. Our testing efforts focused on components recently added to cURL to support HTTP/3, as well as cURL's fuzz tests implemented for said components. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. In addition, we both modified existing fuzz tests and wrote additional tests to increase fuzzing coverage. The scope of this audit included only code directly related to HTTP/3 functionality within cURL itself—notably, excluding the internals of third-party libraries such as `ngtcp2` and `nghttp3` that cURL calls out to for lower-level HTTP/3 operations.

Observations and Impact

cURL's HTTP/3 components are implemented fairly robustly, making heavy use of preexisting primitives common to much of the rest of the cURL codebase (e.g., `bufq` and `dynbuf`). In effect, the components within the scope of this audit largely comprise an intermediate layer that lightly handles incoming data in order to pass it on to third-party libraries for lower-level processing, maintaining some associated state meanwhile. We did not identify any memory safety, data handling, or state maintenance issues in cURL's HTTP3 components; however, we did identify regressions and gaps in cURL's fuzz tests that have caused recent versions of cURL to suffer considerably in terms of fuzzing coverage.

It should be noted that the scope of the code reviewed within this audit is relatively narrow. In particular, while we audited cURL's *use* of the third-party libraries `ngtcp2`, `nghttp3`, `quiche`, and `msh3` to implement HTTP/3 functionality, we did not investigate the internals of those libraries—which is where the majority of the low-level parsing and data transformation necessitated by the HTTP/3 protocol occurs. The fuzz tests we implemented did involve those library internals, insofar as they invoked code paths that called them internally, but they were not targeted directly. We recommend conducting additional audits targeted at the internals of those libraries, especially `ngtcp2` and `nghttp3`, which are currently the cURL developers' main focus for HTTP/3 support.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the cURL development team take the following steps going forward:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Implement measures to detect regressions in fuzzing coverage.** Significant reductions in coverage should be promptly identified and addressed. This is particularly relevant when we consider that OSS-Fuzz is fuzzing cURL continuously; any changes that make harnesses ineffective will negate the benefit of continuous fuzzing.
- **Conduct additional security audits of the `ngtcp2`, `nghttp3`, `quiche`, and `msh3` HTTP/3 libraries employed by cURL, and implement fuzz tests that cover them.** Much of the lower-level data processing involved in parsing the HTTP/3 protocol occurs in these libraries, rather than in cURL's codebase directly.
- **Consider alternatives to decouple or stub out encryption from the QUIC implementation.** A very limited amount of code paths can be explored currently in the HTTP/3 implementation, as a traditional fuzzer is not able to produce valid encrypted traffic. Including a way to be able to fuzz HTTP/3 and HTTPS in plaintext would enhance the fuzzability of the protocols. This will require coordinated work with the third-party libraries implementing HTTP/3.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	0
Informational	2
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Configuration	2

Project Goals

The engagement was scoped to provide a security assessment of cURL's new HTTP/3 components. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there any logic errors within the HTTP/3 components that could result in reaching an inconsistent state, given ill-formatted inputs?
- Are there any aspects of the HTTP/3 specification with which cURL's implementation does not comply, especially areas where the HTTP/2 and HTTP/3 specifications differ substantially?
- Are there any circumstances in which cURL could mismanage its underlying UDP components?
- Does cURL use its underlying HTTP/3 libraries (e.g., `ngtcp2`) in unsafe ways?
- Does cURL have sufficient fuzz test coverage on its core components?
- What code paths within the HTTP/3 components are most likely to benefit from additional fuzz tests?

Project Targets

The engagement involved a review and testing of the targets listed below.

cURL

Repository	https://github.com/curl/curl
Version	`ede2e812c22fd42527acffdbafd98ee90eaa0dbe`
Type	Library and CLI binary
Platform	Native

cURL fuzzer for OSS-Fuzz

Repository	https://github.com/curl/curl-fuzzer
Version	`f67fa1000e8dbc2f9f0189f8669bec9816d5a2f3`
Type	Fuzzing harnesses and scripts
Platform	x86 and x86_64

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Manual code review and static analysis of cURL's HTTP/3-related components, with a particular focus on code paths involving the `ngtcp2` back end.
- Analysis of existing fuzz test coverage for HTTP/3-related functionality, and implementation of additional fuzz tests.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Given our engineers' relative unfamiliarity with the details of the HTTP/3 specification, compared to the cURL developers themselves, our ability to identify protocol-level issues such as spec noncompliance was limited.

Fuzzing Coverage Assessment

As part of this engagement, Trail of Bits reviewed the cURL project's fuzz tests and their coverage, with the aim of improving their depth and coverage of the HTTP3 implementation. The libcurl library is continuously fuzzed by OSS-Fuzz, an initiative for fuzzing open-source software, using scripts and harnesses from the `curl-fuzzer` repository.

Assessment Overview

As a first step, we reviewed the coverage currently achieved by the fuzzing harnesses in the repository, based on the seed cases. We also reviewed coverage reports from OSS-Fuzz. These reports showed a significant decrease in coverage compared to last year (see finding TOB-CURLH3-1 for further context). We also observed nil coverage of the code implementing HTTP/3. This was expected, as OSS-Fuzz does not currently build cURL with HTTP/3 support.

During the first week of the engagement, we investigated the root cause of the drop in coverage and provided a pull request on the `curl-fuzzer` repository to resolve the issue. Once it was merged and a few days passed, we saw the coverage rise in general to levels similar to those observed in November 2022. Once we had a good baseline to reference, we reviewed the coverage in more detail.

To start covering HTTP/3 code paths, we then made changes to the `curl-fuzzer` repository to build cURL with HTTP/3 support. This necessitated adaptations in the build scripts to build and install a compatible TLS library as well as one or more libraries implementing the QUIC and HTTP/3 protocols. After discussions with the cURL development team, we selected `QuicTLS`, `ngtcp2`, and `nghttp3` as the most suitable build combination. We wrote scripts to download and build these libraries as part of the `curl-fuzzer` repository, and to enable HTTP3 support in cURL. Following that work, we also needed to improve the `curl_fuzzer` harness, so that it performed adequately with a datagram-based protocol like HTTP/3.

We also identified a common code path from cURL, BUFQ, which had some indirect coverage but was not being directly tested. This module manages memory buffers and is used for both HTTP2 and HTTP3 cURL implementations, so we opted to write a standalone harness for it.

A summary of the harness improvements and new harnesses can be found below. In the short term, we recommend including these modifications as part of the OSS-Fuzz cURL harness suite. Long term, we recommend working with the community to make the dependencies more fuzzing-friendly, and improving the HTTP/3 harness further to achieve higher coverage.

Fuzzing harness changes	
Harness	**Description**
curl_fuzzer_http	Resolved coverage drop due to build misconfiguration
curl_fuzzer_http3	Fuzzing HTTP/3 with ngtcp2, nghttp3 and quictls
curl_fuzzer_bufq	Fuzzing BUFQ buffer management

HTTP/1 and HTTP/2

Rationale

The current harnesses have support for fuzzing HTTP/1 and HTTP/2 protocols. Both of these protocols work over TCP connections, unlike HTTP/3, which is built over UDP datagrams.

The fuzzing coverage at the time of starting this engagement was significantly reduced due to an issue in the build scripts (TOB-CURLH3-1) that resulted in cURL being built without SSL support, which was not expected nor supported by the harness.

Harness

On this occasion, we did not change the harness, but we provided a pull request to fix the build scripts issue. Once it was merged, we monitored OSS-Fuzz coverage levels. The coverage levels recovered within a few days and nearly reached the levels it used to have before the issue was introduced.

Future work

As mentioned in TOB-CURLH3-1, we recommend frequently monitoring the harnesses for errors in build and execution, as well as the resulting coverage levels. These issues should be addressed promptly, as running a harness that cannot progress meaningfully is unlikely to provide the project with any benefit, while potentially giving a false sense of security.

HTTP3

Rationale

cURL supports HTTP/3 with multiple QUIC implementations and TLS back ends. The use of HTTP/3 in the public internet has grown lately, as reported by Cloudflare and W3Techs. However, the current fuzzing coverage did not show any coverage for the relevant code implementing HTTP/3. This is also explained by the current harness build scripts not enabling HTTP/3 support in cURL.

Harness

We asked the cURL team for a recommendation of the most mature build combination for cURL HTTP/3 support. The team recommended building cURL with `QuicTLS`, `ngtcp2`, and `nghttp3`:

- `QuicTLS` is a fork of OpenSSL which adds QUIC-related API.
- `ngtcp2` uses QuicTLS to provide QUIC.
- `nghttp3` implements HTTP/3 on top of QUIC.

We therefore wrote scripts to download and build these libraries as part of the `curl-fuzzer` repository, and to enable HTTP3 support in cURL.

Following that work, we also needed to improve the `curl_fuzzer` base harness. The harness was built with TCP-based protocols in mind and uses a `SOCK_STREAM` socket to allow a libcurl client to receive random data packets from the fuzzer, which acts as a server. This works well for connection-based protocols like older HTTP and HTTPS versions, but HTTP3 is built upon UDP datagrams. We therefore had to allow the harness to use a `SOCK_DGRAM` socket, which is meant for datagram-based communication, like the UDP datagrams used in HTTP/3. We also discovered that several code paths in dependencies and cURL itself assumed that the socket had the address family `AF_INET`, which is used for IP addressing. These code paths therefore did not work correctly when provided a socket with address family `AF_UNIX`, like the one used in the fuzzing harness. As a result, we also needed to patch some of the third-party libraries.

Once these changes were implemented, we executed the harness for several days with address sanitizer (ASan) enabled, but it did not find any failures. Using the OSS-Fuzz coverage calculation and reporting feature, we observed coverage in the `vquic` module (30% line coverage, 42% function coverage) and in `ngtcp2` (15% line coverage, 27% function coverage), but did not observe any coverage of the `nghttp3` library code. We suspect that, as the HTTP/3 protocol itself is significantly intertwined with TLS, the encryption makes it hard for a fuzzer to progress to the point where data can be decoded and parsed meaningfully.

Future work

To achieve end-to-end testing of HTTP/3, we recommend working with the developers of the TLS, QUIC, and HTTP/3 libraries to identify opportunities to make the code more fuzzing-friendly. For instance, making encryption optional and stubbing out TLS, and adding support for a wider variety of datagram sockets, would facilitate fuzz testing and make it more effective.

Trail of Bits is developing tlspuffin, a custom fuzzer for TLS 1.3 capable of decrypting TLS messages and fuzzing the plaintext behind the ciphertext. This tool could also facilitate work on fuzzing HTTP/3 communications.

BUFQ Implementation

Rationale

cURL has an internal module named `bufq` that manages input/output buffers and is used by several protocol implementations, including WebSockets, HTTP/2, and all three HTTP/3 implementations. While the current fuzzing coverage showed the module had some indirect coverage, not all functions were covered, and there was no harness directly exercising the functionality. Additionally, managing memory buffers can be error-prone, which makes it a good target for fuzzing.

Harness

We implemented a harness that receives a TLV (Type-Length-Value) encoded buffer containing a set of parameters and operations, decodes it, and follows its instructions to allocate a `bufq`, read, write, skip, and otherwise operate on the data in the buffer. Any data read from the buffer is checked to ensure that it matches the written data. The buffer length is also checked to ensure that no bytes are lost. We executed this harness for over a week with ASan enabled, but it did not find any failures.

Future work

Some functions remain uncovered—namely `Curl_bufq_write_pass`, `Curl_bufq_is_full`, and `Curl_bufq_space`. We recommend enhancing the harness suite to exercise these functions as well. The harness could also benefit from becoming structure-aware to improve efficiency; for the sake of time and code reuse during the engagement, it was written based on the existing TLV handling code.

Strategic Fuzzing Recommendations

We recommend the following general changes to improve the coverage and efficiency of cURL's fuzzing setup. These recommendations follow from our observations in both the 2022 and 2023 cURL fuzzing assessments:

- **Add dictionaries for other protocols to libFuzzer and OSS-Fuzz.** Adding a dictionary with common words greatly improves the efficiency of fuzzing in certain cases, such as text-based protocols. A dictionary can initially be populated by extracting relevant strings from header files or manual pages, by using AFL++'s AUTODICTIONARY feature, or by running the binary through the `strings` command. If the protocol is well-known, tools such as ChatGPT can also be prompted to produce a dictionary. The fuzzing chapter of our testing handbook provides an example of such a prompt.
- **Ensure that all build configurations (e.g., non-OpenSSL builds, quiche, msh3) are covered by the fuzz tests.**
- **Add a round-trip fuzzing harness for every encoder/decoder pair.** This will ensure that the encoding and decoding processes work as expected and that data is not corrupted or otherwise modified.
- **Implement structure-aware fuzzing.** `curl-fuzzer` currently uses a type-length-value (TLV) format for inputs in order to encode various types and components of requests and responses. However, as libFuzzer is not aware of the TLV structure, many of the mutations it generates are invalid at the TLV-unpacking stage and have to be discarded by `curl-fuzzer`. This reduces fuzzing efficiency. In accordance with Google's recommendation above, we recommend implementing structure-aware fuzzing by adding a custom mutator that ensures that the fuzzer always receives a valid input. There is an open pull request from 2019 to add such a mutator, but its current status is unclear.
- **Cover `argv` fuzzing.** Fuzzing the curl binary with different options can be useful to discover issues in the command-line tool. This can be achieved using the `argv-fuzz-inl.h` header from the AFL++ project to build the arguments array from standard input in cURL. Also, consider adding a dictionary with possible options and protocols to the fuzzer based on the source code or cURL's manual.

To improve the coverage of HTTP/3 in particular, we suggest the following actions:

- **Work with the dependency library developers to improve the external libraries and make them fuzz-friendly.** Successful end-to-end fuzzing of HTTP/3 communications will require coordination and collaboration between cURL and other actors, such as TLS library developers and HTTP/3 library developers. Some

fuzzing-specific features may need to be developed, like support for non-UDP sockets or encryption-less connections.

- **Work with the dependency library developers to improve their own fuzzing.** While we did not review the state of fuzzing of any third-party library during this engagement, fuzzing the standalone libraries may prove easier than trying to fuzz the full vertical integration with cURL. Having these libraries covered by OSS-Fuzz would indirectly help improve the maturity of the resulting cURL builds.
- **Implement a mechanism to be able to fuzz encrypted protocols in plaintext.** Having a way to mock encryption operations in cURL to allow fuzzers to operate in cleartext will benefit not just HTTP/3, but HTTPS and other encrypted protocols as well. This could be implemented by either mocking the TLS implementation, or by an approach similar to tlspuffin (see appendix C: Dolev-Yao TLS Fuzzing Using tlspuffin).
- **Implement differential fuzzing harnesses to compare HTTP/3 implementations.** Building libcurl with different HTTP/3 back ends, testing the same input on the different builds, and comparing the obtained results can be a good way to detect differences in behavior and handling of the protocol among libraries.
- **Separate the HTTP/3 harness into its own implementation, to more easily account for the connectionless nature of UDP.**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Critical arithmetic operations are present in cURL's HTTP/3 code in the form of determining data lengths, buffer positions, etc. In all noted cases, such values are computed using appropriately sized types and bounds-checked where necessary.	**Satisfactory**
Auditing	cURL's HTTP/3 code issues a reasonable number of warnings, errors, and debug messages for critical events and operations.	**Satisfactory**
Authentication / Access Controls	cURL's HTTP/3 code does not implement authentication or access controls.	**Not Applicable**
Complexity Management	cURL's HTTP/3 code is well-organized according to discrete functionality implemented, backing libraries invoked, and so on.	**Satisfactory**
Configuration	cURL makes reasonably standard use of the third-party libraries (e.g., `ngtcp2`) implementing its lower-level HTTP/3 functionality.	**Satisfactory**
Cryptography and Key Management	cURL's HTTP/3 code does not handle key material. cURL relies on well audited third-party libraries such as BoringSSL, GnuTLS, and WolfSSL to perform cryptographic operations.	**Not Applicable**
Data Handling	cURL's HTTP/3 code mostly consists of passing incoming data to underlying libraries such as `ngtcp2`, with relatively little parsing or processing. Where it is necessary to interpret or transform this data before passing it along, such operations are accompanied by	**Satisfactory**

	appropriate error checks and safety measures.	
Documentation	cURL's new HTTP/3 features are somewhat sparsely documented compared to older functionality. While the basics are covered, details are not necessarily covered in depth.	**Moderate**
Maintenance	cURL's HTTP/3 code is updated together with the rest of the application, a monolithic binary, and needs no separate provisions to update itself.	**Not Applicable**
Memory Safety and Error Handling	cURL's HTTP/3 code engages in relatively little direct memory management, instead relying on prewritten `alloc`/`init` and `free` functions for common primitives such as `bufq` and `dynbuf`. Array accesses are appropriately bounded, potentially null pointers checked, and so on.	**Strong**
Testing and Verification	At the time of the audit, cURL had some functionality-oriented tests for HTTP/3 features, but had no fuzzing or security-oriented tests.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	OSS-Fuzz coverage silently dropped significantly	Configuration	**Informational**
2	curl_fuzzer is ineffective	Configuration	**Informational**

Detailed Findings

1. OSS-Fuzz coverage silently dropped significantly	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Configuration	Finding ID: TOB-CURLH3-1
Target: curl_fuzzer repository	

Description

Between November 30, 2022 and December 1, 2022, the fuzzing coverage for cURL in OSS-Fuzz dropped significantly. By the end of November, cURL had over 50% line coverage and over 67% function coverage; however, in December, cURL fuzz runs reflected a low 6.62% line coverage and 10.18% function coverage.

Reviewing build logs and Git change history, we observed that this occurred after an OpenSSL version upgrade. The new OpenSSL version started installing the `libssl.a` static library on a different directory, `lib64`, instead of the traditional `lib` folder. The cURL fuzz scripts did not expect nor support this alternate location and therefore built cURL without SSL support, which broke several expectations in the fuzzing harnesses.

This significant loss of coverage went undetected for over a year, as we observed that the coverage had not recovered by the time we started this engagement in December 2023.

The Trail of Bits team submitted a pull request to the `curl_fuzzer` repository to fix the issue. Once it was merged, we observed the coverage started to increase again starting on December 15. By December 20, 2023, coverage was up again and near the November 2022 values, with a 48.83% line coverage and 65.73% function coverage of cURL code.

Recommendations

Short term, frequently monitor coverage changes over time, especially after changes are merged in the `curl_fuzzer` repository. If a regression is identified, act as needed to resolve it and restore the fuzzing functionality. Consider modifying the harnesses to immediately fail if an operation that is supposed to always work, such as setting a static cURL option, fails.

Long term, implement an automated system to monitor coverage changes in OSS-Fuzz and alert the maintainers if significant changes are detected. Integrate tests in the `curl_fuzzer` CI to compare corpus coverage before and after changes, in order to detect regressions earlier on.

2. curl_fuzzer is ineffective

Severity: **Informational**	Difficulty: **Undetermined**
Type: Configuration	Finding ID: TOB-CURLH3-2
Target: `curl_fuzzer/curl_fuzzer.cc`	

Description

The `curl_fuzzer` harness displays significantly worse coverage than other similar harnesses like `curl_fuzzer_http`. Upon inspecting the harness code and coverage logs, we observed that the harness consistently fails to set the allowed protocols list, as highlighted in figure 2.1.

This list is overly broad, and contains protocols that cURL is not built to support, causing the `setopt` call to fail every time. The harness cannot proceed beyond this point and therefore does not achieve any interesting coverage.

```
int fuzz_set_allowed_protocols(FUZZ_DATA *fuzz)
{
  int rc = 0;
  const char *allowed_protocols = "";

#ifdef FUZZ_PROTOCOLS_ALL
  /* Do not allow telnet currently as it accepts input from stdin. */
  allowed_protocols =
    "dict,file,ftp,ftps,gopher,gophers,http,https,imap,imaps,"
    "ldap,ldaps,mqtt,pop3,pop3s,rtmp,rtmpe,rtmps,rtmpt,rtmpte,rtmpts,"
    "rtsp,scp,sftp,smb,smbs,smtp,smtps,tftp";
#endif
  /* (...) */
  FTRY(curl_easy_setopt(fuzz->easy, CURLOPT_PROTOCOLS_STR, allowed_protocols));

EXIT_LABEL:
  return rc;
}
```

Figure 2.1: The fuzzer harness fails to configure the allowed protocols (`curl-fuzzer/curl_fuzzer.cc#505-577`)

Recommendations

Short term, adjust the `allowed_protocols` list so that it contains only protocols supported by the cURL build under test.

Long term, review the existing harnesses as time passes and cURL features change to ensure that they are still exercising code paths as expected.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Dolev-Yao TLS Fuzzing Using `tlspuffin`

Since 2022, Trail of Bits has been researching stateful fuzzing of cryptographic protocols. The project started in 2021 as a research project at Inria Nancy (LORIA) in France. This research culminated in a paper on the Dolev-Yao (DY) fuzzing approach, which will be published at 2024 IEEE S&P. The corresponding fuzzer is called tlspuffin.

The current TLS fuzzer in projects such as OpenSSL essentially fuzzes only the client/server hello messages, as they are the only messages in TLS 1.3 that are not encrypted. It is unlikely that the fuzzer triggers interesting states beyond the first message. This is where the idea of DY fuzzing comes into play. In the 1980s, the formal methods community identified and mathematically defined the DY model. It allows us to reason about cryptographic protocols on a logical and structural level. To fuzz a protocol specifically on a structural level, a DY fuzzer injects, omits, and modifies encrypted TLS messages. The fuzzer is capable of decrypting TLS messages and modifying individual fields. Using this approach, the tlspuffin fuzzer has discovered several CVEs of medium severity in wolfSSL.

The tlspuffin fuzzer is also capable of detecting logical security flaws. This class of bug usually does not result in a crash or memory corruption that would be detectable by AddressSanitizer. The current version of tlspuffin is capable of detecting issues like authentication bypasses, where a server or client can impersonate another one.

The tlspuffin fuzzer is continuously improved, and development is ongoing. For example, a new feature promises to add classical bit-level fuzzing capabilities to tlspuffin. As already mentioned, tlspuffin works on a more structural level and does not flip single bits in its current version. However, it makes perfect sense to combine both approaches. This feature is expected to be released later this year.