NAME

libcurl-security - security considerations when using libcurl

Security

The libcurl project takes security seriously. The library is written with caution and precautions are taken to mitigate many kinds of risks encountered while operating with potentially malicious servers on the Internet. It is a powerful library, however, which allows application writers to make trade-offs between ease of writing and exposure to potential risky operations. If used the right way, you can use libcurl to transfer data pretty safely.

Many applications are used in closed networks where users and servers can (possibly) be trusted, but many others are used on arbitrary servers and are fed input from potentially untrusted users. Following is a discussion about some risks in the ways in which applications commonly use libcurl and potential mitigations of those risks. It is not comprehensive, but shows classes of attacks that robust applications should consider. The Common Weakness Enumeration project at https://cwe.mitre.org/ is a good reference for many of these and similar types of weaknesses of which application writers should be aware.

Command Lines

If you use a command line tool (such as curl) that uses libcurl, and you give options to the tool on the command line those options can get read by other users of your system when they use *ps* or other tools to list currently running processes.

To avoid these problems, never feed sensitive things to programs using command line options. Write them to a protected file and use the -K option to avoid this.

.netrc

.netrc is a pretty handy file/feature that allows you to login quickly and automatically to frequently visited sites. The file contains passwords in clear text and is a real security risk. In some cases, your .netrc is also stored in a home directory that is NFS mounted or used on another network based file system, so the clear text password flies through your network every time anyone reads that file.

For applications that enable .netrc use, a user who manage to set the right URL might then be possible to pass on passwords.

To avoid these problems, do not use .netrc files and never store passwords in plain text anywhere.

Clear Text Passwords

Many of the protocols libcurl supports send name and password unencrypted as clear text (HTTP Basic authentication, FTP, TELNET etc). It is easy for anyone on your network or a network nearby yours to

just fire up a network analyzer tool and eavesdrop on your passwords. Do not let the fact that HTTP Basic uses base64 encoded passwords fool you. They may not look readable at a first glance, but they are easily "deciphered" by anyone within seconds.

To avoid this problem, use an authentication mechanism or other protocol that does not let snoopers see your password: Digest, CRAM-MD5, Kerberos, SPNEGO or NTLM authentication. Or even better: use authenticated protocols that protect the entire connection and everything sent over it.

Unauthenticated Connections

Protocols that do not have any form of cryptographic authentication cannot with any certainty know that they communicate with the right remote server.

If your application is using a fixed scheme or fixed host name, it is not safe as long as the connection is unauthenticated. There can be a man-in-the-middle or in fact the whole server might have been replaced by an evil actor.

Unauthenticated protocols are unsafe. The data that comes back to curl may have been injected by an attacker. The data that curl sends might be modified before it reaches the intended server. If it even reaches the intended server at all.

Remedies:

Restrict operations to authenticated transfers

Use authenticated protocols protected with HTTPS or SSH.

Make sure the server’s certificate etc is verified

Never ever switch off certificate verification.

Redirects

The *CURLOPT_FOLLOWLOCATION(3)* option automatically follows HTTP redirects sent by a remote server. These redirects can refer to any kind of URL, not just HTTP. libcurl restricts the protocols allowed to be used in redirects for security reasons: only HTTP, HTTPS, FTP and FTPS are enabled by default. Applications may opt to restrict that set further.

A redirect to a file: URL would cause the libcurl to read (or write) arbitrary files from the local filesystem. If the application returns the data back to the user (as would happen in some kinds of CGI scripts), an attacker could leverage this to read otherwise forbidden data (e.g. **file://localhost/etc/passwd**).

If authentication credentials are stored in the ~/.netrc file, or Kerberos is in use, any other URL type

(not just file:) that requires authentication is also at risk. A redirect such as ftp://some-internal-server/private-file would then return data even when the server is password protected.

In the same way, if an unencrypted SSH private key has been configured for the user running the libcurl application, SCP: or SFTP: URLs could access password or private-key protected resources, e.g. **sftp://user@some-internal-server/etc/passwd**

The *CURLOPT_REDIR_PROTOCOLS(3)* and *CURLOPT_NETRC(3)* options can be used to mitigate against this kind of attack.

A redirect can also specify a location available only on the machine running libcurl, including servers hidden behind a firewall from the attacker. e.g. http://127.0.0.1/ or http://intranet/delete-stuff.cgi?delete=all or tftp://bootp-server/pc-config-data

Applications can mitigate against this by disabling *CURLOPT_FOLLOWLOCATION(3)* and handling redirects itself, sanitizing URLs as necessary. Alternately, an app could leave *CURLOPT_FOLLOWLOCATION(3)* enabled but set *CURLOPT_REDIR_PROTOCOLS(3)* and install a *CURLOPT_OPENSOCKETFUNCTION(3)* or *CURLOPT_PREREQFUNCTION(3)* callback function in which addresses are sanitized before use.

CRLF in Headers

For all options in libcurl which specify headers, including but not limited to *CURLOPT_HTTPHEADER(3)*, *CURLOPT_PROXYHEADER(3)*, *CURLOPT_COOKIE(3)*, *CURLOPT_USERAGENT(3)*, *CURLOPT_REFERER(3)* and *CURLOPT_RANGE(3)*, libcurl sends the headers as-is and does not apply any special sanitation or normalization to them.

If you allow untrusted user input into these options without sanitizing CRLF sequences in them, someone malicious may be able to modify the request in a way you did not intend such as injecting new headers.

Local Resources

A user who can control the DNS server of a domain being passed in within a URL can change the address of the host to a local, private address which a server-side libcurl-using application could then use. e.g. the innocuous URL **http://fuzzybunnies.example.com/** could actually resolve to the IP address of a server behind a firewall, such as 127.0.0.1 or 10.1.2.3. Applications can mitigate against this by setting a *CURLOPT_OPENSOCKETFUNCTION(3)* or *CURLOPT_PREREQFUNCTION(3)* and checking the address before a connection.

All the malicious scenarios regarding redirected URLs apply just as well to non-redirected URLs, if the user is allowed to specify an arbitrary URL that could point to a private resource. For example, a web

app providing a translation service might happily translate **file://localhost/etc/passwd** and display the result. Applications can mitigate against this with the *CURLOPT_PROTOCOLS(3)* option as well as by similar mitigation techniques for redirections.

A malicious FTP server could in response to the PASV command return an IP address and port number for a server local to the app running libcurl but behind a firewall. Applications can mitigate against this by using the *CURLOPT_FTP_SKIP_PASV_IP(3)* option or *CURLOPT_FTPPORT(3)*.

Local servers sometimes assume local access comes from friends and trusted users. An application that expects https://example.com/file_to_read that and instead gets http://192.168.0.1/my_router_config might print a file that would otherwise be protected by the firewall.

Allowing your application to connect to local hosts, be it the same machine that runs the application or a machine on the same local network, might be possible to exploit by an attacker who then perhaps can "port-scan" the particular hosts - depending on how the application and servers acts.

IPv4 Addresses

Some users might be tempted to filter access to local resources or similar based on numerical IPv4 addresses used in URLs. This is a bad and error-prone idea because of the many different ways a numerical IPv4 address can be specified and libcurl accepts: one to four dot-separated fields using one of or a mix of decimal, octal or hexadecimal encoding.

IPv6 Addresses

libcurl handles IPv6 addresses transparently and just as easily as IPv4 addresses. That means that a sanitizing function that filters out addresses like 127.0.0.1 is not sufficient - the equivalent IPv6 addresses **::1**, **::**, **0:00::0:1**, **::127.0.0.1** and **::ffff:7f00:1** supplied somehow by an attacker would all bypass a naive filter and could allow access to undesired local resources. IPv6 also has special address blocks like link-local and site-local that generally should not be accessed by a server-side libcurl-using application. A poorly configured firewall installed in a data center, organization or server may also be configured to limit IPv4 connections but leave IPv6 connections wide open. In some cases, setting *CURLOPT_IPRESOLVE(3)* to CURL_IPRESOLVE_V4 can be used to limit resolved addresses to IPv4 only and bypass these issues.

Uploads

When uploading, a redirect can cause a local (or remote) file to be overwritten. Applications must not allow any unsanitized URL to be passed in for uploads. Also, *CURLOPT_FOLLOWLOCATION(3)* should not be used on uploads. Instead, the applications should consider handling redirects itself, sanitizing each URL first.

Authentication

Use of *CURLOPT_UNRESTRICTED_AUTH(3)* could cause authentication information to be sent to an unknown second server. Applications can mitigate against this by disabling *CURLOPT_FOLLOWLOCATION(3)* and handling redirects itself, sanitizing where necessary.

Use of the CURLAUTH_ANY option to *CURLOPT_HTTPAUTH(3)* could result in user name and password being sent in clear text to an HTTP server. Instead, use CURLAUTH_ANYSAFE which ensures that the password is encrypted over the network, or else fail the request.

Use of the CURLUSESSL_TRY option to *CURLOPT_USE_SSL(3)* could result in user name and password being sent in clear text to an FTP server. Instead, use CURLUSESSL_CONTROL to ensure that an encrypted connection is used or else fail the request.

Cookies

If cookies are enabled and cached, then a user could craft a URL which performs some malicious action to a site whose authentication is already stored in a cookie. e.g. http://mail.example.com/delete-stuff.cgi?delete=all Applications can mitigate against this by disabling cookies or clearing them between requests.

Dangerous SCP URLs

SCP URLs can contain raw commands within the scp: URL, which is a side effect of how the SCP protocol is designed. e.g.

```
scp://user:pass@host/a;date >/tmp/test;
```

Applications must not allow unsanitized SCP: URLs to be passed in for downloads.

file://

By default curl and libcurl support file:// URLs. Such a URL is always an access, or attempted access, to a local resource. If your application wants to avoid that, keep control of what URLs to use and/or prevent curl/libcurl from using the protocol.

By default, libcurl prohibits redirects to file:// URLs.

Warning: file:// on Windows

The Windows operating system tries automatically, and without any way for applications to disable it, to establish a connection to another host over the network and access it (over SMB or other protocols), if only the correct file path is accessed.

When first realizing this, the curl team tried to filter out such attempts in order to protect applications for inadvertent probes of for example internal networks etc. This resulted in CVE-2019-15601 and the associated security fix.

However, we have since been made aware of the fact that the previous fix was far from adequate as there are several other ways to accomplish more or less the same thing: accessing a remote host over the network instead of the local file system.

The conclusion we have come to is that this is a weakness or feature in the Windows operating system itself, that we as an application cannot safely protect users against. It would just be a whack-a-mole race we do not want to participate in. There are too many ways to do it and there is no knob we can use to turn off the practice.

If you use curl or libcurl on Windows (any version), disable the use of the FILE protocol in curl or be prepared that accesses to a range of "magic paths" potentially make your system access other hosts on your network. curl cannot protect you against this.

What if the user can set the URL

Applications may find it tempting to let users set the URL that it can work on. That is probably fine, but opens up for mischief and trickery that you as an application author may want to address or take precautions against.

If your curl-using script allow a custom URL do you also, perhaps unintentionally, allow the user to pass other options to the curl command line if creative use of special characters are applied?

If the user can set the URL, the user can also specify the scheme part to other protocols that you did not intend for users to use and perhaps did not consider. curl supports over 20 different URL schemes. "http://" might be what you thought, "ftp://" or "imap://" might be what the user gives your application. Also, cross-protocol operations might be done by using a particular scheme in the URL but point to a server doing a different protocol on a non-standard port.

Remedies:

Use --proto

curl command lines can use *--proto* to limit what URL schemes it accepts

Use CURLOPT_PROTOCOLS

libcurl programs can use *CURLOPT_PROTOCOLS(3)* to limit what URL schemes it accepts

consider not allowing the user to set the full URL

Maybe just let the user provide data for parts of it? Or maybe filter input to only allow specific choices?

RFC 3986 vs WHATWG URL

curl supports URLs mostly according to how they are defined in RFC 3986, and has done so since the beginning.

Web browsers mostly adhere to the WHATWG URL Specification.

This deviance makes some URLs copied between browsers (or returned over HTTP for redirection) and curl not work the same way. It can also cause problems if an application parses URLs differently from libcurl and makes different assumptions about a link. This can mislead users into getting the wrong thing, connecting to the wrong host or otherwise not working identically.

Within an application, this can be mitigated by always using the *curl_url(3)* API to parse URLs, ensuring that they are parsed the same way as within libcurl itself.

FTP uses two connections

When performing an FTP transfer, two TCP connections are used: one for setting up the transfer and one for the actual data.

FTP is not only unauthenticated, but the setting up of the second transfer is also a weak spot. The second connection to use for data, is either setup with the PORT/EPRT command that makes the server connect back to the client on the given IP+PORT, or with PASV/EPSV that makes the server setup a port to listen to and tells the client to connect to a given IP+PORT.

Again, unauthenticated means that the connection might be meddled with by a man-in-the-middle or that there is a malicious server pretending to be the right one.

A malicious FTP server can respond to PASV commands with the IP+PORT of a totally different machine. Perhaps even a third party host, and when there are many clients trying to connect to that third party, it could create a Distributed Denial-Of-Service attack out of it. If the client makes an upload operation, it can make the client send the data to another site. If the attacker can affect what data the client uploads, it can be made to work as a HTTP request and then the client could be made to issue HTTP requests to third party hosts.

An attacker that manages to control curl’s command line options can tell curl to send an FTP PORT command to ask the server to connect to a third party host instead of back to curl.

The fact that FTP uses two connections makes it vulnerable in a way that is hard to avoid.

Denial of Service

A malicious server could cause libcurl to effectively hang by sending data slowly, or even no data at all but just keeping the TCP connection open. This could effectively result in a denial-of-service attack.

The *CURLOPT_TIMEOUT(3)* and/or *CURLOPT_LOW_SPEED_LIMIT(3)* options can be used to mitigate against this.

A malicious server could cause libcurl to download an infinite amount of data, potentially causing all of memory or disk to be filled. Setting the *CURLOPT_MAXFILESIZE_LARGE(3)* option is not sufficient to guard against this. Instead, applications should monitor the amount of data received within the write or progress callback and abort once the limit is reached.

A malicious HTTP server could cause an infinite redirection loop, causing a denial-of-service. This can be mitigated by using the *CURLOPT_MAXREDIRS(3)* option.

Arbitrary Headers

User-supplied data must be sanitized when used in options like *CURLOPT_USERAGENT(3)*, *CURLOPT_HTTPHEADER(3)*, *CURLOPT_POSTFIELDS(3)* and others that are used to generate structured data. Characters like embedded carriage returns or ampersands could allow the user to create additional headers or fields that could cause malicious transactions.

Server-supplied Names

A server can supply data which the application may, in some cases, use as a file name. The curl command-line tool does this with *--remote-header-name*, using the Content-disposition: header to generate a file name. An application could also use *CURLINFO_EFFECTIVE_URL(3)* to generate a file name from a server-supplied redirect URL. Special care must be taken to sanitize such names to avoid the possibility of a malicious server supplying one like **"/etc/passwd"**, **"\autoexec.bat"**, **"prn:"** or even **".bashrc"**.

Server Certificates

A secure application should never use the *CURLOPT_SSL_VERIFYPEER(3)* option to disable certificate validation. There are numerous attacks that are enabled by applications that fail to properly validate server TLS/SSL certificates, thus enabling a malicious server to spoof a legitimate one. HTTPS without validated certificates is potentially as insecure as a plain HTTP connection.

Showing What You Do

Relatedly, be aware that in situations when you have problems with libcurl and ask someone for help, everything you reveal in order to get best possible help might also impose certain security related risks. Host names, user names, paths, operating system specifics, etc. (not to mention passwords of course) may in fact be used by intruders to gain additional information of a potential target.

Be sure to limit access to application logs if they could hold private or security-related data. Besides the obvious candidates like user names and passwords, things like URLs, cookies or even file names could also hold sensitive data.

To avoid this problem, you must of course use your common sense. Often, you can just edit out the sensitive data or just search/replace your true information with faked data.

setuid applications using libcurl

libcurl-using applications that set the 'setuid' bit to run with elevated or modified rights also implicitly give that extra power to libcurl and this should only be done after careful considerations.

Giving setuid powers to the application means that libcurl can save files using those new rights (if for example the 'SSLKEYLOGFILE' environment variable is set). Also: if the application wants these powers to read or manage secrets that the user is otherwise not able to view (like credentials for a login etc), it should be noted that libcurl still might understand proxy environment variables that allow the user to redirect libcurl operations to use a proxy controlled by the user.

File descriptors, fork and NTLM

An application that uses libcurl and invokes *fork()* gets all file descriptors duplicated in the child process, including the ones libcurl created.

libcurl itself uses *fork()* and *execl()* if told to use the **CURLAUTH_NTLM_WB** authentication method which then invokes the helper command in a child process with file descriptors duplicated. Make sure that only the trusted and reliable helper program is invoked!

Secrets in memory

When applications pass user names, passwords or other sensitive data to libcurl to be used for upcoming transfers, those secrets are kept around as-is in memory. In many cases they are stored in the heap for as long as the handle itself for which the options are set.

If an attacker can access the heap, like maybe by reading swap space or via a core dump file, such data might be accessible.

Further, when eventually closing a handle and the secrets are no longer needed, libcurl does not explicitly clear memory before freeing it, so credentials may be left in freed data.

Saving files

libcurl cannot protect against attacks where an attacker has write access to the same directory where libcurl is directed to save files.

Report Security Problems

Should you detect or just suspect a security problem in libcurl or curl, contact the project curl security team immediately. See https://curl.se/dev/secprocess.html for details.